TU Delft

Delft University of Technology

Delft Students On Software Architecture: DESOSA 2020

van Deursen, A.; Boone, C.C.

Publication date
2020

Document Version
Final published version

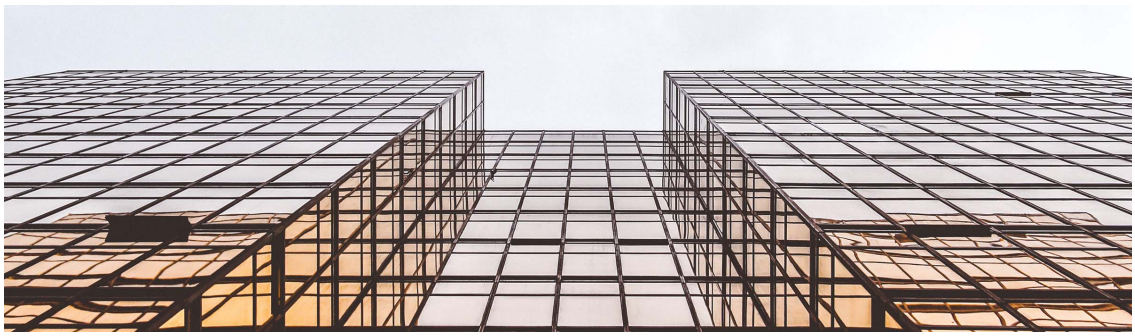
Citation (APA)
van Deursen, A., & Boone, C. C. (Eds.) (2020). *Delft Students On Software Architecture: DESOSA 2020*. (2020 ed.) Delft University of Technology. https://desosa.nl

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.



Delft Students on Software Architecture

DESOSA 2020

Edited by Casper Boone and Arie van Deursen

April 2020, version 1.0

Contents

1 DESOSA 2020: About these Essays **1**
- 1.1 Recurring Themes . . . 1
- 1.2 Highlights . . . 4
- 1.3 Studying in Times of Corona . . . 6
- 1.4 Further Reading . . . 7
- 1.5 Copyright, License and Availability . . . 9

2 Ansible **11**
- 2.1 Team . . . 11
- 2.2 Current and future vision of Ansible . . . 12
- 2.3 Views on Ansible's Architecture . . . 20
- 2.4 Ansible paying off their technical debt . . . 23
- 2.5 Does Conway's Law Apply to Ansible? . . . 29

3 ArduPilot **35**
- 3.1 Team Introduction . . . 36
- 3.2 Uplink to ArduPilot . . . 36
- 3.3 What Makes ArduPilot Soar - An Architectural Overview . . . 42
- 3.4 Under the Hood of ArduPilot: Software Quality and Improvements . . . 50
- 3.5 Standardize, Abstract, Overcome: Exploring Variability in ArduPilot . . . 59

4 Blender **69**
- 4.1 Team . . . 70
- 4.2 Blender in Perspective . . . 70
- 4.3 Architecting Blender . . . 77
- 4.4 Blender Behind the Scenes . . . 84
- 4.5 Blender's Variability . . . 91

5 Bokeh **97**
- 5.1 Team . . . 98
- 5.2 Plotting Bokeh, an Analysis to its Present and Future . . . 98
- 5.3 Plotting Bokeh, an Analysis of its Architectural Variables . . . 102
- 5.4 Plotting Bokeh, an Analysis of its Quality . . . 114
- 5.5 Plotting Bokeh, an Analysis of its Collaboration . . . 126

6 Docker Compose **139**

6.1 The Architectural Journey . . . 139
6.2 Team . . . 139
6.3 What is Docker Compose, and why does it matter? . . . 139
6.4 Relevant Architectural Views . . . 147
6.5 Quality and Technical Debt . . . 155
6.6 System and People Collaboration in Docker Compose . . . 170

7 ESLint 179
7.1 The vision behind ESLint's success . . . 179
7.2 Behind the ESLint Architecture . . . 185
7.3 ESLint's variability management . . . 193

8 Gatsby 199
8.1 Usage . . . 199
8.2 The great Gatsby and its vision . . . 199
8.3 Gatsby Through the Eyes of... . . . 203
8.4 Gatsby in Debt . . . 207
8.5 To Gatsby and Beyond! . . . 212

9 GitLab 217
9.1 Team Introduction . . . 217
9.2 GitLab: A single application for the entire DevOps lifecycle . . . 217
9.3 Architecting for everyone's contribution . . . 220
9.4 Staying on the Rails: A Quality Assessment . . . 225
9.5 Towards a greener DevOps lifecycle . . . 230

10 Ludwig 235
10.1 The Team . . . 237
10.2 The Vision of Ludwig . . . 237
10.3 Ludwig - Connecting the Vision to Architecture . . . 243
10.4 Ludwig's Code Quality and Tests . . . 251
10.5 Variability Analysis of Ludwig . . . 257

11 Material UI 263
11.1 About the authors . . . 263
11.2 Materialising Material-UI . . . 264
11.3 Dissecting Material-UI . . . 268
11.4 Qualifying Material-UI . . . 272
11.5 Varying Material-UI . . . 277

12 Meteor 281
12.1 Why investigate Meteor? . . . 281
12.2 Who are we? . . . 282
12.3 More information . . . 282
12.4 The vision which makes Meteor shine . . . 282
12.5 The architecture behind Meteor's impact . . . 286
12.6 Comets and commits: the software quality of Meteor . . . 291
12.7 Exploring the Atmosphere: the variability of Meteor . . . 296

13 Micronaut **303**
13.1 The Team . . . 304
13.2 Micronaut - The Product Vision . . . 304
13.3 Micronaut - From Vision to architecture . . . 307
13.4 Micronaut - Quality and Technical Debt . . . 314
13.5 Micronaut - A perfect Microservice framework? . . . 328

14 MuseScore **337**
14.1 Team . . . 338
14.2 MuseScore: Road to reducing paper use in music industry . . . 341
14.3 MuseScore: Views on development . . . 346
14.4 MuseScore: Cost of music . . . 352
14.5 MuseScore: Architecting for accessibility and usability . . . 358

15 Mypy **365**
15.1 Authors . . . 365
15.2 The vision behind mypy . . . 366
15.3 Architecture of mypy . . . 369
15.4 Checking the typechecker mypy . . . 373
15.5 The configuration of mypy . . . 379
15.6 Contributor Workflow - an optimization analysis . . . 386

16 Next.js **391**
16.1 Next.js: Back to the Future . . . 391
16.2 Architecting Next.js . . . 397
16.3 Can Next.js Stay Ahead: A Case of Quality Control . . . 402

17 NumPy **411**
17.1 About the team . . . 411
17.2 NumPy: its Goals, Stakeholders, Use and Future . . . 412
17.3 NumPy: Awesome Architecting for Amazing Arrays . . . 418
17.4 NumPy: Software Quality by the Numbers . . . 424
17.5 NumPy: Carefully Crafting Components through Collaboration . . . 436

18 Open edX **441**
18.1 Product Vision . . . 442
18.2 From Vision to Architecture . . . 448
18.3 Quality and Technical Debt . . . 452
18.4 Variability Analysis . . . 457

19 openpilot **463**
19.1 Product Vision: Make Driving Chill . . . 463
19.2 From Vision To Architecture: How to use openpilot and live . . . 469
19.3 When an autonomous car is in the garage . . . 475
19.4 How even a single product can vary . . . 482

20 OpenRCT2 **487**
20.1 Team . . . 487

20.2 OpenRCT2, Porting RollerCoaster Tycoon into 2020 . . . 488
20.3 The architecture of architecting Rollercoasters . . . 492
20.4 Rollercoaster (tycoon) should not crash . . . 499
20.5 Developing fun . . . 505
20.6 On OpenRCT2's Frontlines . . . 511

21 RIOT **515**
21.1 About us . . . 516
21.2 RIOT: The future of IoT . . . 516
21.3 RIOT: From Vision to Architecture . . . 520
21.4 RIOT: Quality and Technical Debt . . . 526
21.5 RIOT: The power of variability . . . 532

22 Ripple **537**
22.1 About Us . . . 538
22.2 Surfing through Ripple: A Grand Tour . . . 538
22.3 From Words to Actions: How Ripple Gets it done . . . 546
22.4 Assessing Ripple . . . 554
22.5 Here To Stay : How Ripple Achieves Sustainable Development . . . 563

23 Scikit-learn **571**
23.1 About us . . . 571
23.2 Scikit-learn, what does it want to be? . . . 572
23.3 From Vision to Architecture . . . 578
23.4 Scikit-learn's plan to safeguard its quality . . . 583
23.5 How does scikit-learn balance usability with variability? . . . 589

24 Sentry **599**
24.1 Putting Sentry into Context . . . 599
24.2 The Architecture Powering Sentry . . . 606
24.3 Monitoring Sentry's Software Quality . . . 614
24.4 Supporting Every Language: Sentry's SDKs . . . 621

25 Signal for Android **627**
25.1 Who we are . . . 628
25.2 A clear vision or mixed signals? . . . 628
25.3 Rome wasn't built in a Signal day . . . 631
25.4 Reducing noise to improve Signal quality . . . 637
25.5 Smoke Signals or secure crypto? . . . 643

26 Solidity **651**
26.1 What is Solidity? . . . 651
26.2 Meet the Team . . . 653
26.3 Solidity: The Product Vision . . . 654
26.4 Solidity: From vision to architecture . . . 658
26.5 Solidity: Solid code or not? . . . 669
26.6 Solidity and its variability . . . 676

27 spaCy **681**
27.1 The Team . . . 682
27.2 spaCy: For All Things NLP . . . 682
27.3 From Vision to Architecture . . . 688
27.4 Quality and Technical Debt . . . 693
27.5 Dependencies and Modular Software Design . . . 700

28 Spyder **705**
28.1 Your friendly neighborhood SPYDER . . . 706
28.2 How Spyder Knits its Architecture from Vision . . . 718
28.3 Analyzing the Quality of Spyder . . . 727

29 TensorFlow **735**
29.1 The Team . . . 736
29.2 TensorFlow: making machine learning manageable . . . 736
29.3 Embedding Vision into Architecture . . . 742
29.4 Combining quality and collaboration in TensorFlow . . . 749

Chapter 1

DESOSA 2020: About these Essays

Arie van Deursen and Casper Boone

Delft Students on Software Architecture (DESOSA) is a collection of technical essays in which students explore the software architecture of 28 different open source systems. These essays are available as a series of blog posts for easy browsing, besides being available in ebook form as pdf and epub.

These essays were written as part of a master level course on Software Architecture that took place in the spring of 2020. The essays emerged as they were written, with most of them posted in March / April 2020.

In total over 120 students worked in teams of four, studying over 30 different systems. The course is open by design, with students not just learning from the teacher, but primarily from each other and from the wisdom of the open source communities. The course builds on ideas in open learning from (the late) Erik Duval, and on Amy Brown and Greg Wilson's book series addressing the Architecture of Open Source Applications[1].

Students could decide for themselves whether to publish their essays, or whether to keep them visible to other students only. The large majority (28 out of 30) of teams decided to make their work public, with the result visible here.

1.1 Recurring Themes

Students were free to pick any open source system they liked, provided the system was sufficiently complex and active (a few pull requests per day, and actually used). Driven by their interest, they selected systems from a wide variety of domains, such as data science (Ludwig, NumPy, Bokeh), web frameworks (Gatsby, Material-UI), autonomous driving and flying (Ardupilot, openpilot), games (OpenRCT2), media (Musescore, Blender), the Internet-of-Things (RIOT, Micronaut), and many others.

Students were asked to write four different essays, which ended up as four separately readable blog posts, as well as four sections in the ebook chapter devoted to their system.

[1] Amy Brown and Greg Wilson (editors). The Architecture of Open Source Applications. Volumes 1-2, 2012. aosabook.org.

1.1.1 Product Vision

The starting point for all essays was a look at the future: What is the vision underlying the system, what is the system's roadmap, who are the people that have a stake in the system, and in what context does the system operate.

To do this, many teams made use of Coplien and Bjørnvig's *Lean Architecture*[2]. This book emphasizes the need to discern an *end user mental model*, reflecting the *expectations* people have when using the system. These expectations are often domain-specific, corresponding to the way a user understands the application domain. As an example, team scikit-learn explains how *estimators*, *predictors* and *transformers* are core to *any* machine learning approach that is to be supported by scikit-learn. Likewise, team RIOT explains how the embedded nature of IoT devices shapes the expectations of RIOT users.

Coplien and Bjørnvig also argue for the need to conduct a stakeholder analysis, distinguishing end users, the business, customers, domain experts and developers (see, e.g., Ardupilot). The student teams identified these stakeholders, sometimes also falling back to the stakeholder categories provided by Rozanski and Woods in their book *Software Systems Architecture*[3].

1.1.2 From Vision to Architecture

With the vision in place, the next essay takes a look under the hood, looking at the underlying architecture.

To that end, architectural *views* played a keyrole. Many students relied on Philippe Kruchten's "4+1" model, distinguishing a logical, development, process, and physical view, as well as scenarios connecting them together[4]. Other teams relied on the seven viewpoints proposed by Rozanski and Woods [5], or benefited from the Arc42 templates for documentation and communication of software and system architectures.

Here, the students also looked at the various architectural *styles* used. The descriptions of the 28 systems serve to illustrate a wide variety of architectural styles, as well as common combinations of styles. Some examples include:

Style	Example Systems
Micro-kernel	RIOT
Micro-services	Micronaut
Inversion-of-Control	Micronaut
Layering	Material-UI, Ardupilot, Signal-Android
Event-based, Pub-Sub	Ardupilot, Bokeh, Sentry, openpilot
Model-View-Controller	Blender, GitLab, MuseScore, OpenRCT2, Signal-Android
Model-Template-View	Sentry
Flux Dataflow	Sentry
Client-server	Bokeh, Signal-Android
Type-based abstraction	Ludwig
Pipe-and-Filter	MyPy

[2] Jim Coplien and Gertrud Bjørnvig. Lean Architecture. Wiley, 2010.

[3] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

[4] Philippe Kruchten. The 4+1 View Model of architecture. IEEE Software 12(6), 1995. (doi, preprint, wikipedia)

[5] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

Style	Example Systems
Plugins	RIOT, Spyder

1.1.3 Quality and Technical Debt

Software architecture is not just about design in a greenfield setting: Architecture becomes especially challenging under the pressure of continuous change and evolution.

In their third essay, students investigate the processes in place to safeguard high quality evolution. For example, students look at the continuous integration processes, reviewing practices, and the test adequacy criteria used (such as code coverage).

Another aspect taken into account is what the quality of the system is in relation to the earlier established roadmap: Which components will be affected by this roadmap, and to what extent are these affected components ready for this change? If not, what refactoring would be needed to address the technical debt currently present in the system?

For this essay, students made use of various tools. Many students relied on Sigrid, a code analysis tool offered by the Amsterdam-based Software Improvement Group. This tool offers insight in component dependencies, and offers rating of the a system's maintainability using seven factors [6]. Another tool many students used to assess quality is SonarQube.

1.1.4 Deepening the Analysis

For the final essay, students could pick a topic of choice, from one of these categories:

- **Variability analysis:** Most systems can be configured in many different ways, at compile time, installation time, or dynamically at run time. A substantial body of research in *managing* such variability exists, giving rise to so-called "product lines": Software systems that can be configured to a series of different software products[7]. Many students looked at their system from a variability perspective, for example when studying Ardupilot, openpilot, or NumPy.
- **Socio-Technical Congruence:** Conway's law suggests that organizational structure is reflected in system design. This gives rise to the notion of *socio-technical congruence*[8]: An assessment of how social interactions and software component interactions are "congruent". Several teams studied this congruence based on module dependencies, developers working on certain modules, and developer interactions (in, e.g., issues or reviewer comments). For example, team Bokeh used the SIG component analysis and contrasted this with developer communications obtained via the GitHub API, and team NumPy contrasted developer communication between maintainers of tightly and loosely coupled components.
- **Green computing:** An often overlooked quality attribute of software system is its energy consumption. Several students analyzed this for their system, including GitLab exlporing how to make the DevOps life cycle greener, and cryptocurrency Ripple.

[6] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A Practical Model for Measuring Maintainability. 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, IEEE, pp 30-39. (pdf)

[7] Sven Apel, Don Batory, Christian Kästner, Gunter Saake. Feature-oriented software product lines. Springer-Verlag, 2013. (url)

[8] Cataldo, Herbsleb, and Carley. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. ICSE 2008. https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf

- **Other:** Lastly, teams could come up with proposals of their own to study aspects specific to their system. For example, for OpenRCT2, the open version of the well-known game Rollercoaster Tycoon, the students investigate what it means to optimize "fun" in a game. Likewise, the Micronaut team specifically studied how Micronaut supports the creation of micro-service applications, the Signal-Android team looked at the specific way in which privacy and security is ensured, and the MuseScore team explored the usability and accessability aspects of its music scoring application.

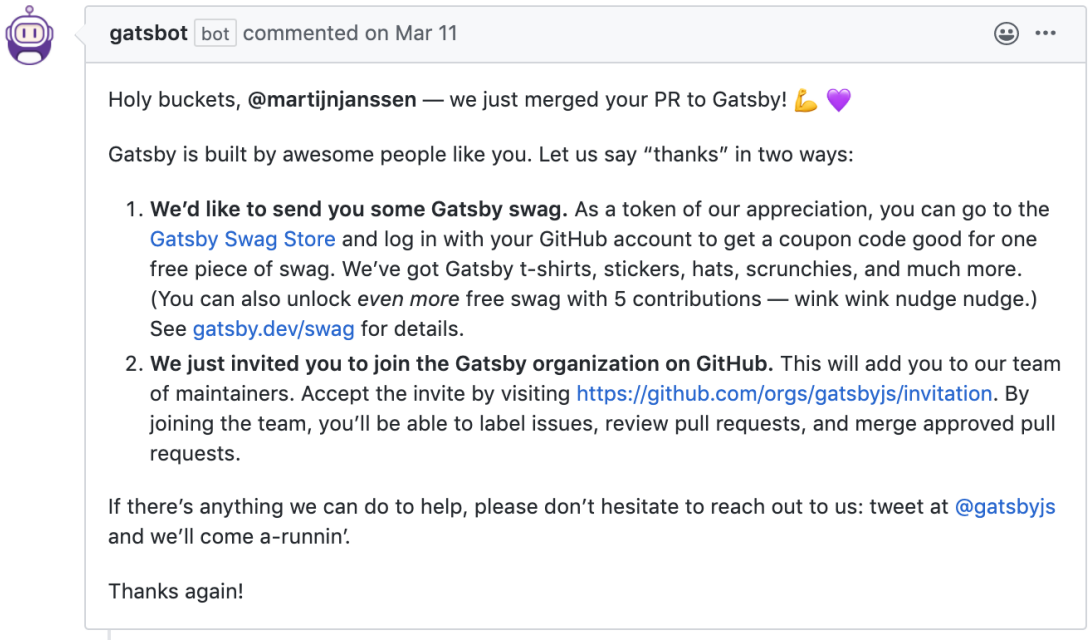


Figure 1.1: Appreciation for merged code contributions

1.1.5 Code Contributions

In order to get exposure to the source code, students made a total of 121 pull requests to the 30 open source systems, of which 71 were merged, 36 are still open at the time of writing, and the remaining 14 were closed (rejected). In this way, students interacted with senior developers, fixed issues, improved documentation, provided localizations, and learned about project policies concerning code quality, refactoring, testing, feature prioritiziation, git usage, and about the importance of keeping changes small and focused.

For each team, the overview page of their system contains a short status update of all pull requests made, with links to the actual pull requests.

We are grateful to all open source developers who patiently interacted with all students.

1.2 Highlights

As part of the course, students studied and learned from each other's material and gave lots of feedback to each other (in fact, measured in words the feedback exceeded the actual essays). The feedback also included

various Likert-scale answers. Based on all this, we feature some popular projects:

- Sentry, a system for triaging, analyzing and resolving errors and bugs.
- Next.js, a framework to help programmers build fast and modern web applications on top of React and Node
- NumPy, the de facto standard library for efficient numerical data structures such as matrices and arrays in Python
- Micronaut, a modern, JVM-based, full stack microservices framework designed for building modular, easily testable microservice applications.
- Gatsby, the free and open source framework based on React 1 that helps developers build blazing fast websites and apps
- Bokeh, an interactive visualization library for the creation of rich interactive plots, dashboards and data applications in the browser
- openpilot, a driving system aimed at achieving partial automation (level 2), letting the software control both steering and acceleration, while the human is still responsible for environment monitoring and fallback performance.



Figure 1.2: Pre-Corona: First lecture with 120 students in a room fitting 140, February 2020

1.3 Studying in Times of Corona

The course started mid February 2020, before any case of Corona (COVID-19) had hit The Netherlands. With 120 participants we enjoyed lectures in a hall with a capacity of 140.

Right from the start, the course had, by design, an international setup, embracing online opportunities where relevant. For example, in our second lecture, we had Grady Booch in an online connection from Hawaii, in an *Ask-me-Anything* session. Students had watched some of Booch's keynotes on software architecture as preparation. In the course of 30 minutes they fired a battery of questions to Grady Booch, who used his decades of experience architecting dozens of widely used systems to explain many principles of software architecture.



Figure 1.3: Pre-Corona: Ask-Me-Anything over Skype with Grady Booch, in TU Delft lecture hall, February 2020

Half-way the course, mid March, because of Corona, The Netherlands went into lockdown, all students and staff had to study and work from home, and all education had to be fully online. Two (invited, industry) lectures had to be cancelled, but luckily we were able to organize one via Twitch. Steffan Norberhuis, TU Delft alumnus, cloud architect, and DevOps consultant, took up the challenge and managed to switch his lecture to a completely virtual setting with just 24 hour notice.

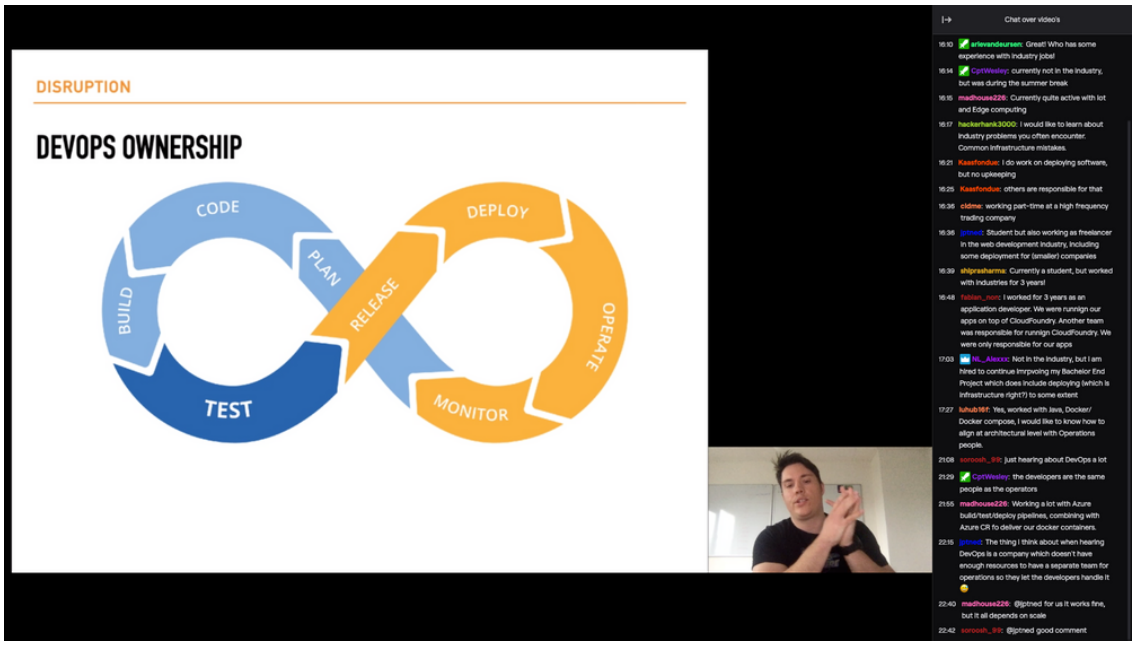


Figure 1.4: In-Corona: Steffan Norberhuis on Architecting for Operations, teaching via Twitch, March 2020

We originally had planned to conclude the course with a full day of posters, presentations, and interaction. Given Corona, instead of presentations, the students created videos of 5-10 minutes summarizing what they did, with some great creative results. Many of the teams made these videos publicly available as well (see, e.g., Signal-Android, Material-UI, or openpilot), which serve as a great introduction to the work the students did.

Instead of in-person presentations, we watched the videos together in chat rooms setup using Discord. The 30 teams were split into six separate sessions of five teams each. For each team, 20 minutes were allocated: 10 minutes to watch the actual video on YouTube, and 10 minutes for Q&A. Students asked questions asynchronously over the chat, and the presenting team answered using voice. This setting worked very well, also because the students were extremely participatory, and very much interested in each other's work.

For all students participating in the course, the impact of Corona was substantial. They had to stay home, and could not meet other students. Some international students had to travel back last minute to their home country. Some students got sick, some developed COVID-19, yet fortunately all got better again. Many students had family or friends affected by the disease; some lost their loved ones. All students had to rethink their immediate and long term future.

It was under these circumstances, that the students wrote most of the essays presented in this collection.

1.4 Further Reading

Earlier editions of DESOSA appeared as online books, in the years 2015, 2016, 2017, 2018, and 2019.

More information about the course can be found on:

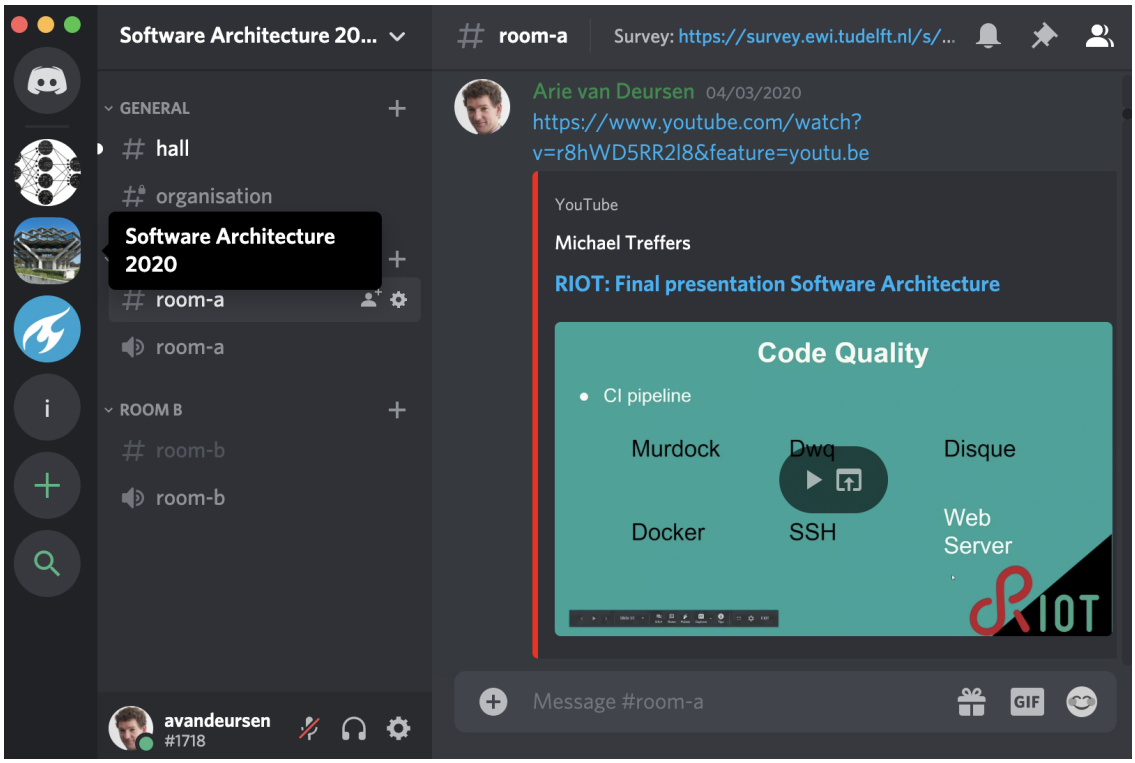


Figure 1.5: In-Corona: Final presentations over Discord, with Q&A over chat, April 2020

1. Arie van Deursen, Maurício Aniche, Joop Aué, Rogier Slag, Michael de Jong, Alex Nederlof, Eric Bouwers. A Collaborative Approach to Teach Software Architecture. 48th ACM Technical Symposium on Computer Science Education (SIGCSE), 2017.[9]
2. Arie van Deursen, Alex Nederlof, and Eric Bouwers. Teaching Software Architecture: with GitHub! avandeursen.com, December 2013.[10]

The schedule, slides, and assignment of the 2020 edition are available online.

1.5 Copyright, License and Availability

The copyright of the chapters is with the authors of the posts. All posts are licensed under the Creative Commons Attribution Share Alike 4.0 International License:

> *Reuse of the material is permitted, provided adequate attribution (such as a link to the essay on the DESOSA site) is included. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.*

Cover photo by Alex Wong, Unsplash.

[9] Arie van Deursen, Maurício Aniche, Joop Aué, Rogier Slag, Michael de Jong, Alex Nederlof, Eric Bouwers. A Collaborative Approach to Teach Software Architecture. 48th ACM Technical Symposium on Computer Science Education (SIGCSE), 2017 (preprint).

[10] Arie van Deursen, Alex Nederlof, and Eric Bouwers. Teaching Software Architecture: with GitHub! avandeursen.com, December 2013.

Chapter 2

Ansible



Ansible is a universal language, unraveling the mystery of how work gets done. Turn tough tasks into repeatable playbooks. Roll out enterprise-wide protocols with the push of a button.

Ansible is a really nice system that makes a lot of peoples' lives easier. It has a good community - both directly in the form of frequent updates, chat groups and mailing lists and indirectly in the form of shared configurations. This means it should be easy to really dig into the system, how it is used and how it is architected. It is also written in Python, which is a definite bonus for our team.

2.1 Team

- Rutger van den Berg
- Dirk van Bokkem
- Sjoerd van den Bos
- Naqib Zarin

2.2 Current and future vision of Ansible

Ansible is in the top 10 of the largest and most popular open-source projects on Github[1]. To understand the vision that makes Ansible so successful, we take a look into the end-user mental model as well as the people and companies involved through a stakeholders analysis. Next to that, we unravel Ansible's following plans with the future roadmap.

2.2.1 Project goals

Before Ansible, system administrators were likely to use a number of ad-hoc scripts to accomplish simple tasks. They were probably run directly on the target machine, with different administrators achieving the same task in different ways. The system administrators can create high quality scripts that can be run repeatedly on machines with varying initial states. As a consequence, mistakes are easily made and a lot of time is wasted.

Enter Ansible.

Ansible aims to provide an easy to use automation tool that can perform any number of common tasks without compromising the security of the machine it runs on. Instead of writing complicated scripts, the user simply specifies the desired state of the target machine, and Ansible takes care of the rest.

Better yet, because of the community behind Ansible, many tasks a system administrator might need to perform on their machines have probably already been specified by others and shared on Ansible Galaxy.

2.2.2 End user mental model

Ansible's end users are primarily system administrators. They expect that Ansible can be used to easily perform a wide variety of tasks on (remote) servers. They can use it to deploy a new version of some software package on a number of servers, update some firewall settings or gather various log files. An example use case of Ansible could be to start a webserver. The server host is retrieved from the inventory, followed by some tasks. For example for an Apache webserver, the tasks could be[2]:

```
tasks:
- name: ensure apache is at the latest version
  yum:
    name: httpd
    state: latest
- name: write the apache config file
  template:
    src: /srv/httpd.j2
    dest: /etc/httpd.conf
  notify:
  - restart apache
- name: ensure apache is running
  service:
    name: httpd
    state: started
```

[1] https://octoverse.github.com/

[2] https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html

The system administrators need the system to just work on servers configured in any number of ways. Some of which might not contain any services or tooling, could be hidden behind (layers off) firewalls or may not be online at any given time.

When run, Ansible is expected to connect to a (set of) machine(s), upload the tasks it needs to accomplish, run the tasks, clean up after itself, and leave the system in the desired state. Regardless of the initial state of the machine or how often the task is run, the end result should always be that the desired state is achieved.

2.2.3 Simple. Powerful. Agentless.

Ansible aims to be a simple but powerful tool. Every member of an IT team should be able to use it because of the human-readable language used in writing automation tasks. Despite its simplicity, Ansible is a powerful tool as mostly any automation task that is needed for a software system can be run within an Ansible playbook.

Ansible has an agentless architecture, which means that instead of having to install software on all the remote machines, Ansible can directly access the machines and execute tasks using the built-in services of those machines. For example, for Linux and Unix machines SSH is used to connect to the machine, while for Windows hosts Windows Remote Management is used[3].

Ansible provides the following capabilities[4]:

- **Provisioning**; setting up the environment and resources
- **Configuration management**; managing configuration files
- **Application deployment**; deploying the application with human-readable playbooks
- **Continuous delivery**; creating a CI/CD pipeline
- **Security automation**; using modules, roles and playbooks to automate security in the system
- **Orchestration**; combining all different configurations and automation tasks into one (orchestrated) whole

2.2.4 Stakeholders Analysis

To know which actors will influence the decision-making process, a stakeholders analysis is required. A stakeholder in software architecture can be a person, group, or entity that has interest in or concerns about the realization of the architecture[5]. The categorization proposed by Rozanski and Woods is used to elaborate on the different stakeholders for Ansible. Note that we added testers to the developer section since every new piece of code needs to be tested and documented and must be backwards compatible. After we have identified the main stakeholders we prioritize them through a Power/Interest Grid.

2.2.4.1 Stakeholder classes

Acquirers
Rozanski and Woods describe “acquirers” as the group that oversees the procurement of the system[6]. Typically, they include sponsors and senior management. Ansible was founded by Ziouani and DeHaan. In

[3] https://www.ansible.com/hubfs/pdfs/Benefits-of-Agentless-WhitePaper.pdf
[4] https://www.ansible.com/use-cases
[5] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[6] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

2015 they sold the company (including its 50 employees) to Red Hat Inc[7]. Therefore, we conclude that their senior management is the same as that of Red Hat Inc.

Assessors
Ansible has a Contributor License Agreement where it states that with any sort of contribution (documentation or code) the contributor grants a full, complete, irrevocable copyright license to all users and developers of the project, present and future, pursuant to the license of the project[8]. Every PR still needs to be assessed. Ansible gives assessors a lot of power and responsibility. In most of the times, the assessors are the module leaders and maintainers.

Communicators
Communicators explain the system to other stakeholders[9]. There are several mailing lists, Internet Relay Chat (IRC) channels and working groups[10]. Each working group has one or more leaders that developers need to contact first. Moreover, Ansible has release managers who need to coordinate between developers, contributors, the community, Ansible documentation team and Ansible Tower team[11].

Developers
With almost 5000 contributors, Ansible is one of the largest open source projects[12]. In order to manage such a large project smoothly, they created many different roles and groups. However, on their website they have a list of 39 individuals who have generally been contributing in significant ways to the Ansible community for some time[13]. The top 10 contributors in terms of commits can be found here. Most of these contributors are also the leaders of a module.

Maintainers
Maintainers manage the evolution of the system once it is operational[14]. For Ansible it holds that the person that contributes a new module to the ansible/ansible repository is the maintainer of that for that module once it is merged[15]. Maintainers have the authority to accept, reject, or revise pull requests on their module. A full list of maintainers can be found in BOTMETA.yml[16].

Suppliers
According to Rozanski and Woods, suppliers build and/or supply the hardware, software, or infrastructure on which the system will run, or possibly they provide specialized staff for system development or operation. Ansible is almost 100% written in Python. Only a very small percentage of code is written in C# and JavaScript. Although Python is free software, one could see Python Software Foundation as an important supplier.

Support Staff
Ansible offers support through communication channels (Github and Freenode). Since the project is very large, most of the support team are specialized in their own field. Next to the fact that Ansible has a lot

[7] https://fortune.com/2015/10/16/red-hat-acquires-ansible/
[8] https://docs.ansible.com/ansible/latest/community/contributor_license_agreement.html#contributor-license-agreement
[9] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[10] https://docs.ansible.com/ansible/latest/community/communication.html
[11] https://docs.ansible.com/ansible/latest/community/release_managers.html#release-managers
[12] https://github.com/ansible/ansible
[13] https://docs.ansible.com/ansible/latest/community/committer_guidelines.html
[14] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[15] https://docs.ansible.com/ansible/latest/community/maintainers.html
[16] https://github.com/ansible/ansible/blob/devel/.github/BOTMETA.yml

of documentation they also offer developers and clients to get in touch with them via Support Cases, Live Chats, Call or Email[17]. Finally, they offer support through social media via Twitter, YouTube or Facebook.

System Administrators
Rozanski and Woods describe system administrators as the people who run the system once it has been deployed[18]. Note that this group of people can be omitted since Ansible is an open source product and therefore has no system administrators.

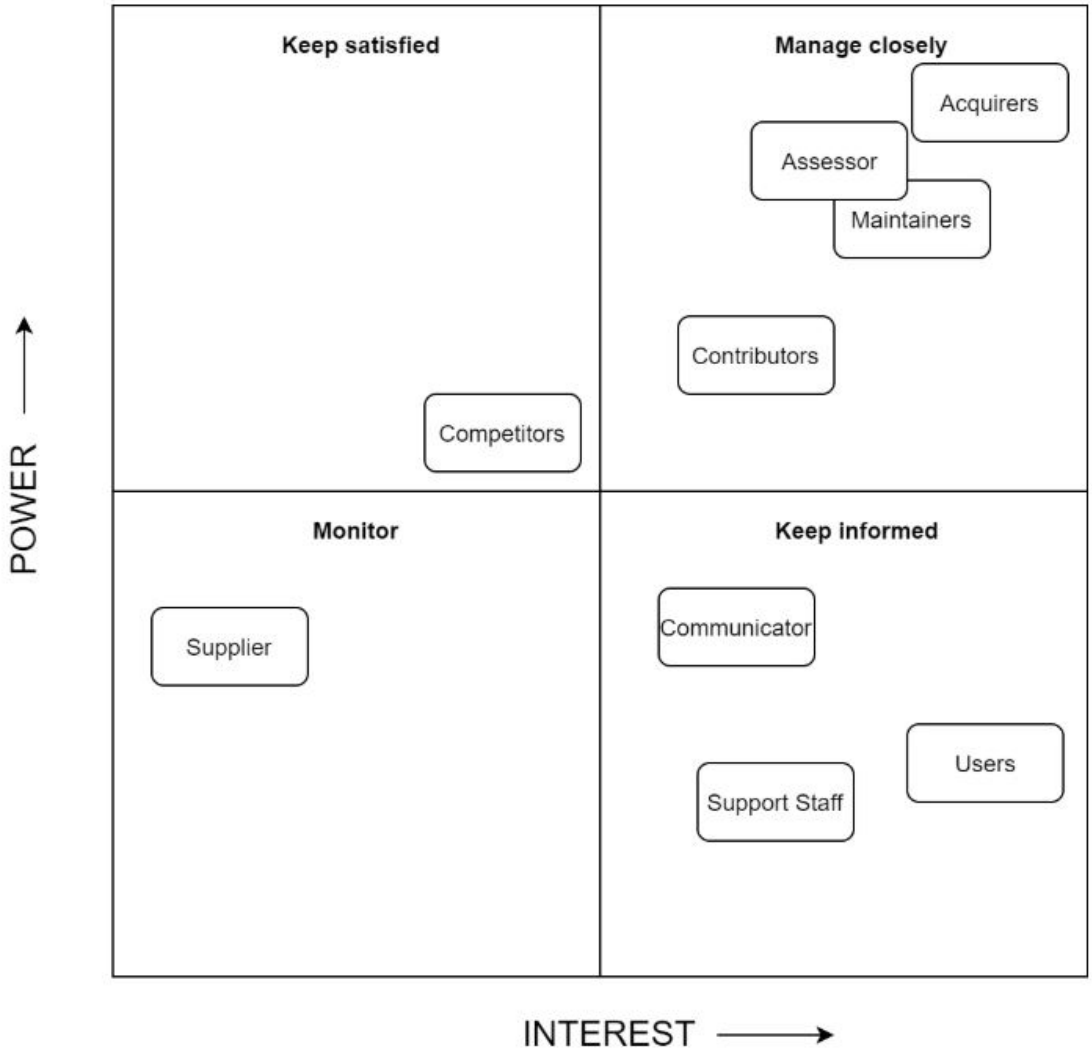
Users
Ansible has a lot of users. They include Jupyter Networks, Apple, NASA, and many more. Most of the users are companies who have a complex IT set up and need Ansible for their configuration management and application deployment.

Competitors
One important stakeholder class we added to this section is Competitors. Ansible's competitors include Octopus Deploy, Chef, Rudder, SaltStack, and many more small players.

[17] https://access.redhat.com/products/ansible-tower-red-hat/?extIdCarryOver=true&sc_cid=701f2000001OH7YAAW#support

[18] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

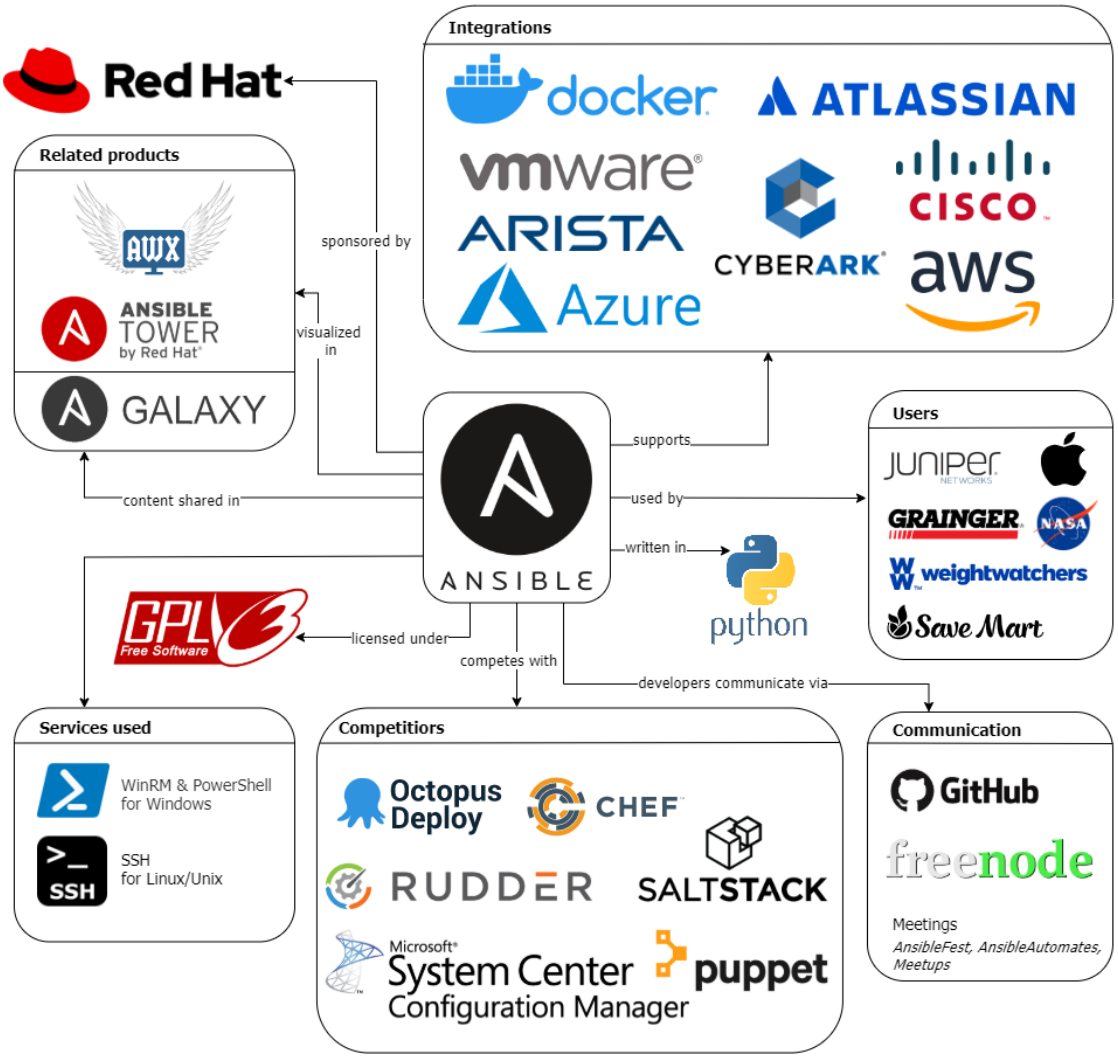
2.2.4.2 Power/Interest Grid



2.2.5 Context

After the stakeholder analysis, we can create an overview of the context of Ansible that describes "the relationships, dependencies, and interactions between the system and its environment" [19]. The best way to give this overview is to visualize it.

[19] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.



In the context view we can see a couple of boxes, which represent the groups of entities that Ansible interacts with. As mentioned in Stakeholders Analysis, Ansible is used by many companies. A few of them can be seen in the Users box[20]. Ansible also has integrations for a lot of systems, such as docker, aws, and vmware[21]. In the top left corner, we see the sponsor RedHat and its related products that can be seen as an extension of Ansible, licensed under GPLv3. Both Ansible Tower and AWX are user interfaces for Ansible, the latter being web-based. Ansible Galaxy provides an online collection of content created by the Ansible community. Worth to mention are the main services WinRM and SSH used by Ansible to access machines. The competitors are also already mentioned in Stakeholders Analysis, as well as Ansible being written in python. Communication between developers is done mainly via Github, IRC channels through freenode and

[20] https://www.ansible.com/blog/enterprise-ansible
[21] https://www.ansible.com/integrations

meetings such as AnsibleFest.

2.2.6 Roadmap

The Ansible team develops a roadmap for each major and minor Ansible release. The latest roadmap shows current work; older roadmaps provide a history of the project. Roadmaps are not published for subminor versions. Team and community feedback is incorporated in each roadmap. Each roadmap is a best guess, based on the Ansible team's experience, requests from the community, and feedback from the community. The roadmap is published both to give an idea as to what is coming and for seeking feedback from the community, making it an iterative process.

The official documentation yields the following roadmap:

Release Schedule

Expected

PRs must be raised well in advance of the dates below to have a chance of being included in this Ansible release.

Note

There is no Alpha phase in 2.10.

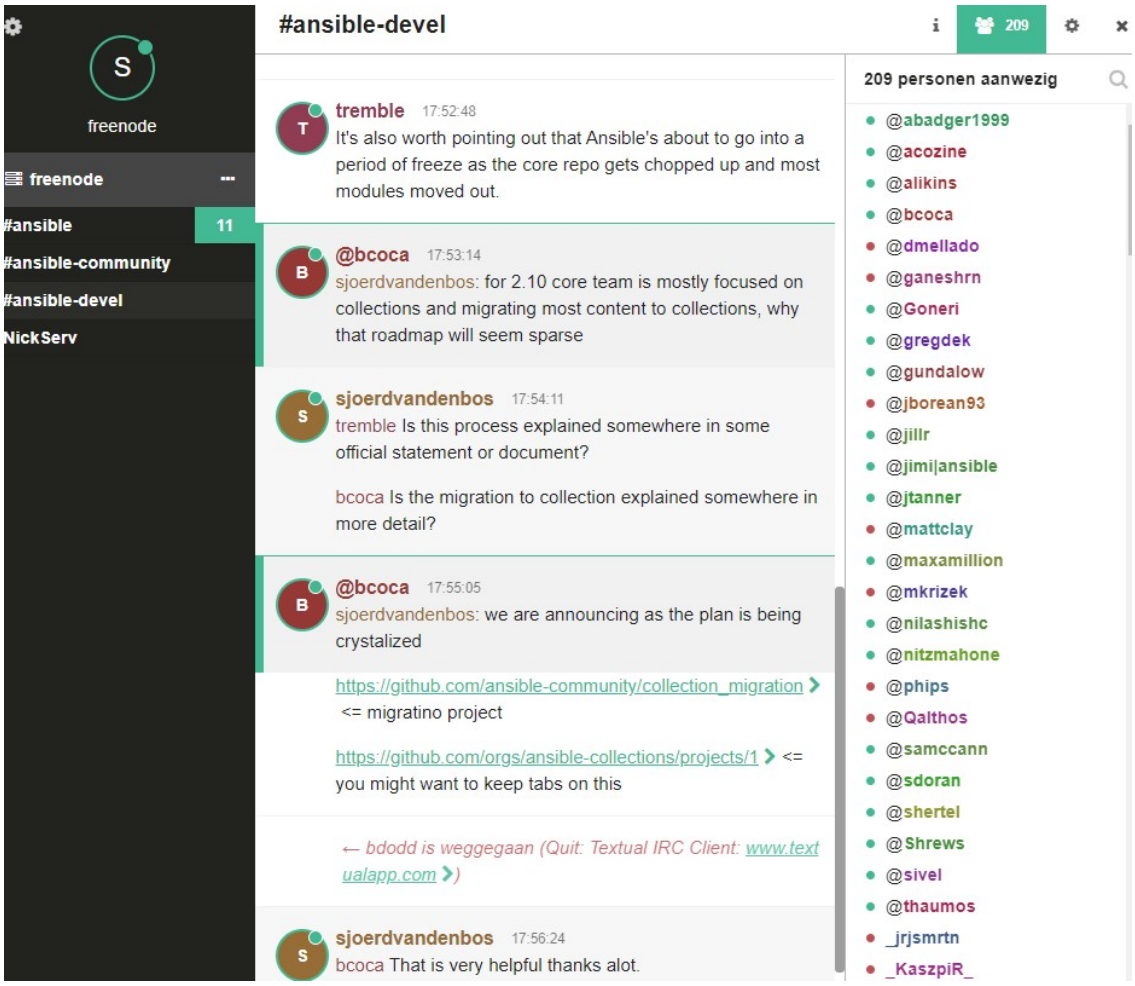
- TBD 2020-??-?? Beta 1 **Feature freeze** No new functionality (including modules/plugins) to any code
- 2020-??-?? Release Candidate 1
- 2020-??-?? Release Candidate 2 if needed)
- 2020-??-?? Release

Release Manager

TBD

Temporarily, Matt Davis (@nitzmahone) or Matt Clay (@mattclay) on IRC or github.

With the official documentation yielding a lack of information, more informal methods of communication can be consulted. To this end the Ansible channels on webchat.freenode.net can be consulted.



The meaning of 'feature freeze' can be found on github.com/ansible-collections. Here a timeline has been documented with 3 bullet points:

- *2nd March 2020*, we will freeze the devel branch using protected branches, and we will create the temp-2.10-develbranch from devel. This date marks the end of merging non-base plugin/module PRs into ansible/ansible.
- *9th March 2020*, we will perform the initial migration against temp-2.10-devel, and we will do our initial testing of the components.
- *23rd March 2020*, we intend to unfreeze devel and merge temp-2.10-devel back into devel. From that point on, devel for ansible/ansible will be for the ansible-base project only.
- TBC, community.general accepts new Pull Requests (PRs).
- TBC, the ansible package has been updated to include the Community Collections.
- TBC, alpha, beta, RC, Release dates for Ansible 2.10

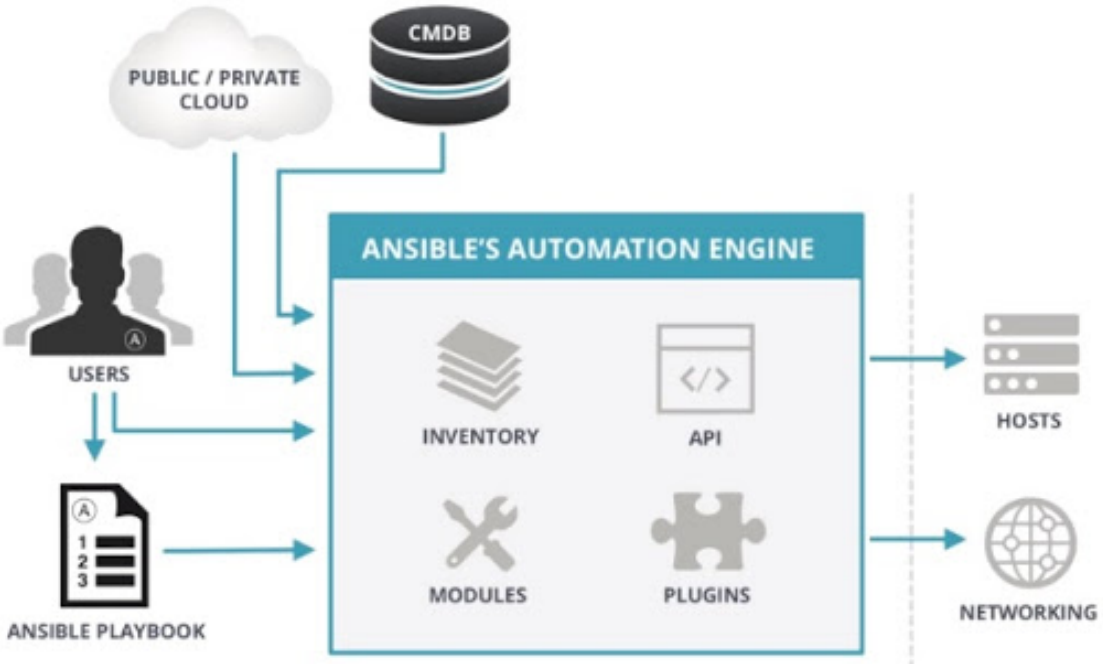
2.3 Views on Ansible's Architecture

2.3.1 Relevant Architectural Views

Software Architecture deals with abstraction, (de)composition, style and aesthetics[22]. In order to describe the architecture of Ansible, we use a model composed of multiple views. Views can be seen as perspectives of an architecture. To be able to properly display the architecture of a system, Kruchten came up with 5 different views. We use this so-called "4+1"-view model to give more insight into Ansible. In the next subsections we will cover each view in more detail.

2.3.1.1 The Logical View

This view is closely related to the structure of classes and how they interact with one another. The logical view is not meant to be a detailed visualization of the structure and interaction of all classes and objects, but is rather meant to give an overview of the key abstractions that are made in order for the system to fulfill its primary functional requirements. Ansible is written in Python and after inspection it is clear that developers consistently make use of Python classes and inheritance to apply abstraction. This makes the logical view a very prominent view.



The figure above[23] shows what the logical view of Ansible is. As explained in our previous post, Ansible is agentless. This means that there is just one node, the control/user node, and no other software is needed to be installed on remote machines to make them manageable[24]. Moreover, it has a push-based architecture which means that the control node only has to write down configurations (also knows as Playbooks) written in YAML and push them to the host machines through SSH.

[22] https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf
[23] http://www.findoutthat.com/ansible-architecture/
[24] https://www.ansible.com/hubfs/pdfs/Benefits-of-Agentless-WhitePaper.pdf

2.3.1.2 Process View

The process view visualizes the general program flow. The process view illustrates the chronological order of tasks that a system can run. This gives the reader an idea of the dependencies between different processes and also whether certain processes are synchronous or asynchronous. There are many different kinds of processes the consumer may want to use that are (not) reliant on each other. In general, there is a large variety of processes that can be run. The dependencies depend on the tasks. Moreover, Ansible can decide on the fly whether some processes can be run asynchronous or not. Ansible is also intelligent in the sense that if a certain task has already been implemented, then Ansible simply ignores and removes them to limit overkill. This all makes the designing of such a process view difficult, because it really depends on what the user aims to achieve.

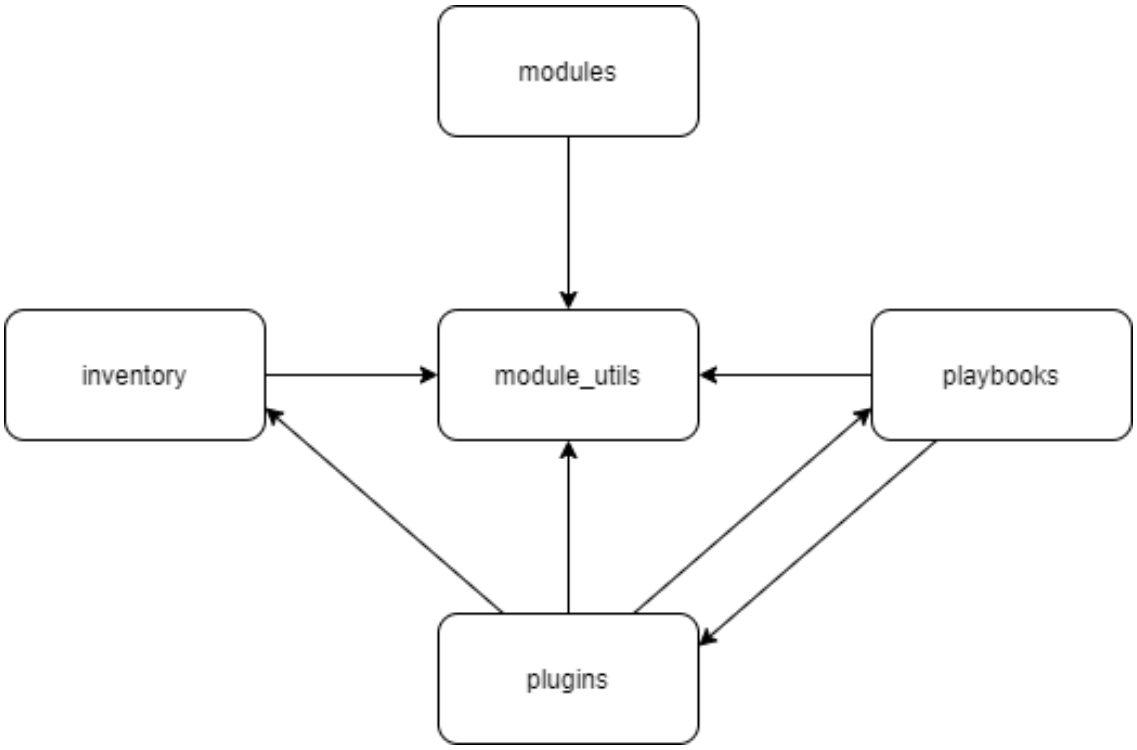
Once Ansible is installed on the control node, the user will typically need two types of configuration files: Host Inventory and Playbook. The Host Inventory contains the IP-addresses of the host machines that a user wants to manage. Inventories can be static or dynamic, or even a combination of them, and Ansible is not limited to a single inventory[25]. Playbooks contain plays and plays contain tasks. Tasks make use of different (combinatory) modules that need to be executed on the host machines. Tasks are made up of a name, a module reference, module arguments, and task control keywords[26]. In order to execute the task, SSH connections with the remote host are created. The first connection creates a temporary directory and is then closed. The second connection is opened to write out the task object from memory into a file within the temporary directory that was just created. After this is closed, Ansible opens a third connection to execute the module and afterwards delete the temporary directory and all its contents. Once the task is executed, the results are captured in JSON format, which Ansible will parse and handle appropriately. This is useful, because its users can trace back what happened in case of problems. If a task has an asynchronous control, Ansible will close the third connection before the module is complete, and SSH back in to the host to check the status of the task after a prescribed period until the module is complete or a prescribed timeout has been reached.

2.3.1.3 Development View

The development view visualizes how the system is divided into packages or modules. Connectors show how the modules depend on each other. This facilitates the allocation of work teams and in agreement with Conway's law, also the organizational structure. The importance of this view becomes apparent when one realizes that this project has thousands of contributors. The modularity of the system limits the overhead for contributors and maintainers since changes in one module of the software have a very limited effect on the rest of the system.

Most of the work Ansible does is contained in *Modules*. These are independent reusable scripts that can be pushed to managed nodes to accomplish some task. Any code that would be used in multiple modules can be found in a *Module Utility*[27]. Together with *plugins*, *inventory*, and *playbooks* they form the core of Ansible, which can be seen in the source code as well. A simple overview of the core components of Ansible can be seen in the following diagram.

[25] https://www.ansible.com/hubfs/-2016-ebooks/Packt_-*Mastering_Ansible*-_Excerpt.pdf
[26] https://www.ansible.com/hubfs/-2016-ebooks/Packt_-*Mastering_Ansible*-_Excerpt.pdf
[27] https://docs.ansible.com/ansible/latest/dev_guide/overview_architecture.html



The largest part of Ansible actually resides in the *modules* folder. Since there are a lot of modules, they are not included in the diagram. Most of the development for Ansible happens here.

2.3.1.3.1 Contributing to Ansible Through the docs of Ansible a lot of coding standards are communicated to the developers, such as where to place certain modules in the system structure, as well as naming conventions which can be found here.

2.3.1.3.2 Testing Developers are expected to test their code and show this in their PR, which will make it more likely that their code will be reviewed and merged. Tests are written in pytest.

2.3.1.4 Physical View

This view focuses on the mapping between software and hardware. For Ansible to be flexible it needs to be able to map its tasks to a wide variety of host systems. Some of these systems are their own abstraction on top of hardware, like the cloud systems on AWS. The overlap between all these cases do not make for an interesting view; Ansible simply moves to the target device(s), performs it's tasks and then cleans up after itself.

2.3.1.5 Deployment view

Ansible at its core involves two or more machines. One control node and a set of one or more managed nodes [28].

[28] https://docs.ansible.com/ansible/latest/network/getting_started/basic_concepts.html

The control node can be any machine that runs Python 2 or 3, except for Windows machines. [29] A windows machine can still be used if Windows Subsystem for Linux is available [30]. Ansible must be installed on this machine, which can be done using most package managers. Alternatively it can be installed through pip.

The managed nodes can be any machine that runs Windows, Linux or BSD, has a Python environment and provides SSH access[31] [32]. Other connection methods are available via plugins. This includes both remote access methods such as unix sockets, kubernetes pods and http(s) as well as various local methods such as chroot environments, lxc containers and BSD jails[33].

2.3.1.6 Scenarios

The scenarios are a combination of the previous four views, making it redundant, but still very useful. The scenarios can be used during prototyping and evaluation. Scenarios can be easily deduced from use cases and can also be used as a way to validate the system. Since Ansible has a set of very distinct use cases, the scenarios are a relevant part of the views. However, for Ansible, there are too many possible scenarios and they can not be ordered in level of importance. It really depends on whether a user wants to deploy, configure or orchestrate. More concretely, Ansible has over 750 modules that can be imported in Playbooks[34]. Ansible is therefore too flexible and large to capture most important scenarios.

2.3.2 Non-functional properties, and how potential trade-offs between them have been resolved.

Ansible has a number of different use cases, and therefore different requirements per use case. There is however overlap. The most important non-functional-requirement is the declarative nature of Ansible.

Declarative programming means that the developer only writes down how it wants the final state of the system to be. This is in contrast to the way developers and system administrators have to perform tasks without Ansible. For every different kind of system a developer would need to write different code to get a similar application running, i.e. the developer has to specify all steps that a certain system needs to take to make an application running and these steps differ per system.

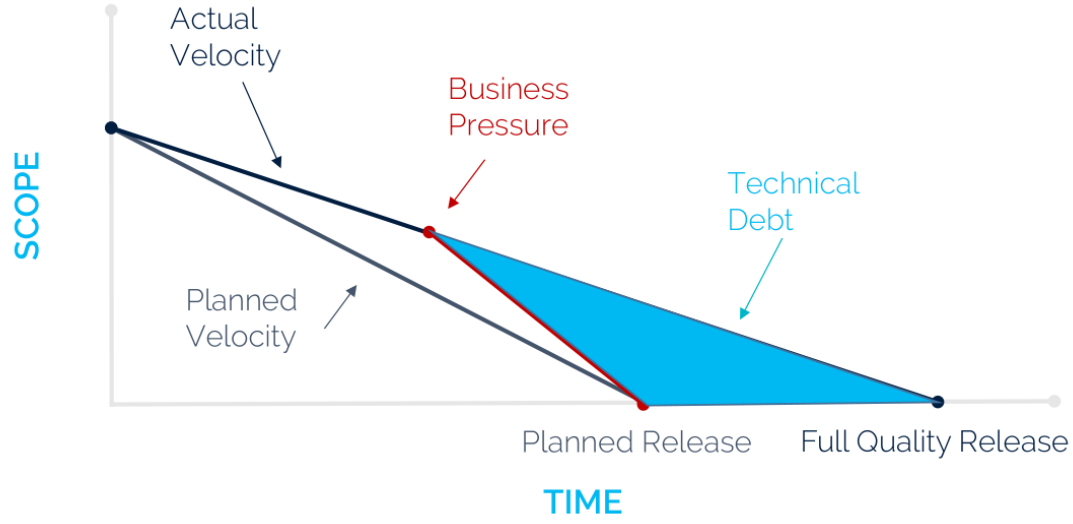
In summary, the trade-offs made between declarative and imperative programming is that the declarative offers ease-of-use and consistency through automation, while imperative programming offers more control to the developer. Ansible is strictly declarative.

2.4 Ansible paying off their technical debt

Now that we have discussed Ansible's architecture, it is time to dive deeper into the implementation details. Like any other software system, Ansible is prone to the build up of deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system[35]. This deficit in quality is often caused by business pressure on the tech team to meet the milestones. This results in quick and dirty solutions to increase the velocity (see figure below). Interestingly, the Ansible core team has been working on decreasing

[29] https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html#control-node-requirements
[30] https://www.jeffgeerling.com/blog/2017/using-ansible-through-windows-10s-subsystem-linux
[31] https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html#managed-node-requirements
[32] https://docs.ansible.com/ansible/latest/user_guide/connection_details.html#
[33] https://docs.ansible.com/ansible/latest/plugins/connection.html#plugin-list
[34] https://docs.ansible.com/ansible/latest/modules/list_of_all_modules.html
[35] https://martinfowler.com/bliki/TechnicalDebt.html

their technical debt, by migrating all the modules to so-called Collections, which will be laid out in more detail in this essay. Similarly, a major refactoring of the code was done in Ansible 2.0 to pay off the technical debt that had accumulated. To analyze the current technical debt of Ansible, we will first examine the development process. Then we give an analysis of the code quality and finally we will discuss how Ansible is planning to deal with this debt in the future milestones.



2.4.1 Process

At a high level, Ansible always works towards a release. Each release has a roadmap containing desired features which is tracked through a github project. A release has several deadlines. First comes the feature freeze, after which no new features will be added and focus shifts to removing bugs. Then comes the first Release Candidate (RC). This version will be released, but might not be stable. New RC's are released until no more bugs are found. At that point the release is finalized and a stable release is made available to the general public. This process ensures that releases contain no untested code.

At a lower level, development takes place using pull requests (PRs). Each pull request is examined by a bot which will add various tags indicating the type of pull request, which part of the project is touched by the PR, the maintainer of the files touched, and the targeted release[36]. This makes it easier for maintainers to keep a good overview of all open PRs.

2.4.1.1 Continuous integration

Ansible uses Shippable for continuous integration (CI). Each PR is checked against a large number of existing automated tests. Each test is run multiple times in various environments. This includes different Python versions and different operating systems[37]. Ansible uses several types of tests:

Compile Tests[38]. These check the code against the syntax of a variety of Python versions.

[36] https://docs.ansible.com/ansible/latest/community/development_process.html
[37] https://github.com/ansible/ansible/blob/devel/shippable.yml
[38] https://docs.ansible.com/ansible/latest/dev_guide/testing_compile.html#testing-compile

Sanity Tests[39]. This includes various scripts and tools that run static analysis on the code, primarily to enforce coding standards and requirements.

Unit tests[40]. These tests are isolated tests run against an individual library or module.

Integration tests[41]. These are functional tests defined as Ansible playbooks. They cover a variety of functionality including installing and removing packages and network functionality. Since some of these tests can cause destructive behavior, they are usually run in Docker containers.

Additionally, Ansible uses codecov.io[42] to generate code coverage reports. These reports give a decent metric on how well parts of the system have been tested.

2.4.1.2 Process evaluation

Ansible has a fairly comprehensive development process, with a good number of checks. However, some things could be improved. For example, while a code coverage report is generated, users have to specifically navigate to the coverage website to see it. This means that the effects of a PR on coverage are not easily visible. A summary could be added to PRs.

Further, despite the well documented PR lifecycle, many PRs are merged without any clear review (e.g. PR68429, PR68407, and PR68200). The fact that this is possible likely makes it easier to implement small changes, but it is not clear where the line is between PRs that need to be reviewed and PRs that don't. It is also not clear why certain PRs are not merged (see for example PR68083, PR67931, and PR67893). It looks like the core team also discusses PRs outside github.

When examining the coverage report a few things come up. First, the powershell code is barely tested at all. Especially when compared to the Python code. This might be because of the fact that powershell code is primarily used for very simple scripts while Python is generally used as a more full featured language. So developers are not used to writing tests for powershell. Second, coverage for individual modules varies greatly. This likely has to do with the fact that the (community) maintainer of each module is somewhat responsible for the testing standards.

2.4.2 Code quality

The development process is usually reflected in the actual code, which is why we will analyze the code quality. Code quality of products created by software engineers and technical debt are very related. According to Philippe Kruchten there are about 5 types of technical debt: architectural, documentation, technological gap, code level and testing[43]. For this essay we will stick to the latter two. In order to measure the software quality of Ansible we use SonarQube. Not only because it is one of the most popular world known solutions for enterprise software quality measurements, it is also open source and free. It applies software quality heuristics like code volume, amount of bugs, and code smells. Since in the last month Ansible has been migrating their code to multiple repositories to improve the code quality, we will focus on the repository both before and after the migration and make a comparison.

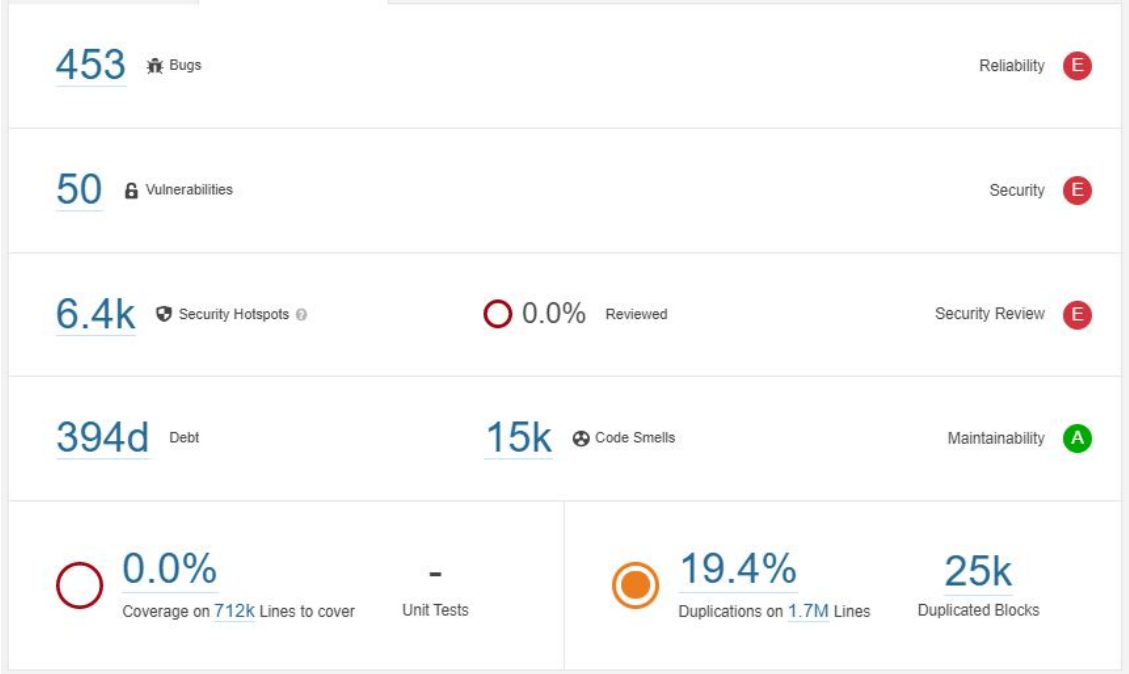
Notes: SonarQube uses letter ratings ranging from A (good) to E (bad). Vulnerabilities and Security Hotspots will not be addressed, since this is somewhat related to technical debt, but it is not the topic we

[39] https://docs.ansible.com/ansible/latest/dev_guide/testing/sanity/index.html#all-sanity-tests
[40] https://docs.ansible.com/ansible/latest/dev_guide/testing_units.html#testing-units
[41] https://docs.ansible.com/ansible/latest/dev_guide/testing_integration.html#testing-integration
[42] https://codecov.io/gh/ansible/ansible/
[43] Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. Ieee software, 29(6), 18-21.

want to dive in here. Test coverage will also not be discussed here, as SonarQube was not able to display this.

2.4.2.1 SonarQube - before migration

Code volume | Firstly, to be able to compare the results of SonarQube with the overall codebase, we need to assess the size. Ansible (before migration) has a total of 1.7 million lines written in Python and 13.000 in XML. Next to that, a relatively small amount is written in HTML, JavaScript, CSS, and Go. As we have seen already in our previous essay, with 1.3 million lines of code, `lib/ansible` is the largest folder in Ansible.



Bugs | SonarQube found 453 bugs, of which 39 are minor, 131 major and the remaining 283 bugs are so-called blockers. At first sight, 453 bugs on 1.7 million lines of code seems pretty good. Why then does SonarQube give the lowest possible reliability score? The answer lies in the blocker-bugs. These bugs have a "high probability to impact the behavior of the application in production"[44]. After examining the bugs, it becomes clear that almost all are caused by symlinks.

For example, in `lib/ansible/modules/source_control/gitlab/` a symlink to the file `gitlab_hook.py`is made, but SonarQube sees it as a python file and thinks a variable is used before it is defined. Another found blocker-bug is that of a mismatch between the number of arguments passed to a function and its parameters. However, the functionality here is implemented for backwards compatibility, as is documented in the code. By identifying and removing these false positives from the analysis, the blocker-bugs were reduced to three.

Some of the bugs in the *major* category are also questionable. Such as a redirecting web page missing a `<!DOCTYPE>` and `<title>` tag. It's sole purpose is to redirect to another page, making the missing tags acceptable. However, quite a few copying errors were found in the major bugs list, such as an `or` operator on two identical expressions.

[44] https://docs.sonarqube.org/latest/user-guide/issues/

Removing all the false positives resulted in 170 bugs in total.

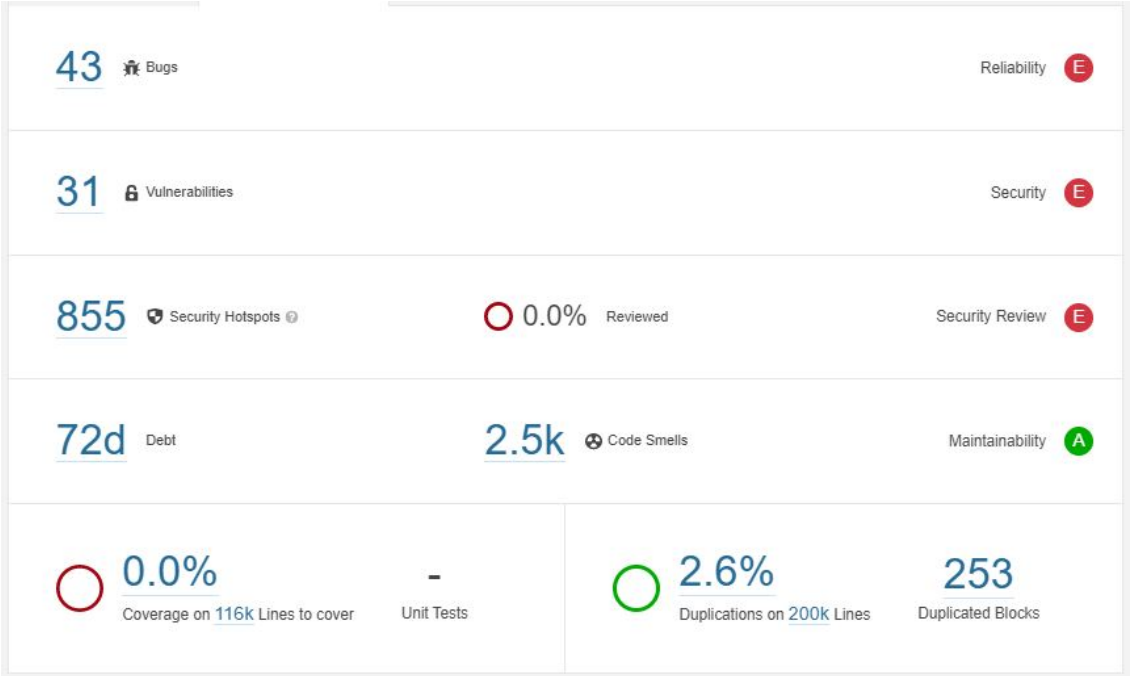
Code smells & Maintainability | Ansible scores an A for maintainability, while there are 15.000 code smells found. SonarQube estimates 394 days in technical debt, to fix these issues. Why does Ansible deserve the best maintainability score? Rating A means that the remediation cost is less than 5% of the time that has already gone into the application[45]. Time here is translated to the "cost to develop 1 line of code * Number of lines of code", where one line of code costs 0.06 days. Considering the size of Ansible, the reason for Ansible's high maintainability score becomes clear; 15.000 code smells on 1.7 million lines of code is not that much and code smells in general for a project with almost 5000 contributors is perhaps unavoidable.

Duplication | According to SonarQube, there was 19.4% duplication of code, which is a fairly high amount. However, this duplication may be the result of certain tools that create lines automatically, as can be seen in our pull request. Using such tools may thus increase the amount of duplicate code, but in the case of Ansible, the repository is well maintained and files like these are never touched. Still, many code blocks are found by SonarQube that are present in multiple files, which is a point of improvement. Some of these code blocks could be moved to the *modules_utils* folder, where a lot of utility functions of different modules already reside.

2.4.2.2 SonarQube - after migration

Code volume | The impact of the migration can be seen directly from the size of the new Ansible repository; with a little more than 1.5 million lines of code that were removed, Ansible core now has "only" 200.000 lines left. These lines are spread out over the corresponding Ansible Collections[46], resulting in a cleaner Ansible core repository. The decrease of lines of code has significantly lowered the amount of bugs. Of course, the bugs are only migrated, but so are the responsibilities. We can now focus on analyzing Ansible's core code.

[45] https://docs.sonarqube.org/latest/user-guide/metric-definitions/
[46] https://github.com/ansible-collections



Bugs | Some of the symlink false positives are still present in the new analysis. Removing these results in a total of 33 bugs; a big improvement coming from 170!

Code smells & Maintainability | As we can see from the maintainability rating, the amount of code smells has decreased proportionally with the amount of code moved. Ansible still has an A-rating, but now only needs around 72 days to fix their code smells. Some of these code smells include commented out code, unused variables, and `FIXME` comments.

Duplication | Code duplication has decreased greatly, as was expected. Many modules used the same functionalities and thus had duplicate code. By moving each of these modules to their own Collection repository, the analysis tool finds less duplications in the core code. In some way, this decrease in reusable code can be seen as a loss. Ansible could however still provide some utility Collections, but using these is up to the Collection creators and maintainers.

2.4.2.3 SonarQube - improvements

To summarize, we give an overview of the improvements found by SonarQube for Ansible before and after the migration:

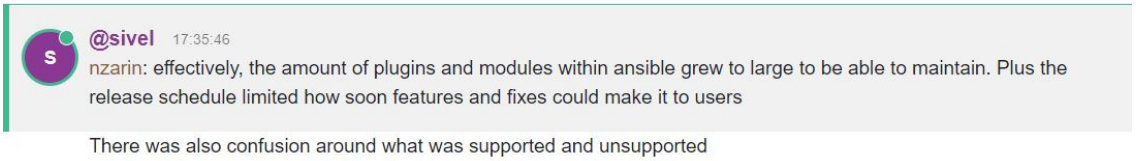
	Before migration	After migration
Bugs	170	33
Code smells	15.000	2.500
Technical debt	394 days	72 days
Code duplication	19.4%	2.6%

Logically, since a lot of code has moved, many bugs have moved as well. Still, the Ansible core repository is greatly improved. Some points of improvement remain:

- remove identical expressions (e.g. in `or` operators)
- add missing attributes on webpage (e.g. add `title` to `<iframe>`)
- implement functionality where `AnsibleError("Option not implemented yet")` is raised or a `FIXME` comment is placed
- review the duplicate code and remove where possible

2.4.3 Technical debt

As stated in the introduction of this essay, the technical debt hinders the ability to modify and extend a system. For Ansible it was the right time to pay off their technical debt as it became too large to maintain properly (see conversation on freenode below).



Our assessment of the code quality (before and after the migration) makes it is clear that the changes made in the migration are in line with paying off technical debt. The software quality is improved, and with that the debt is partly paid off. However, the technical debt cannot only be measured by programming aspects of software delivery. It is important to look at the full development lifecycle. This includes evaluating PRs. As we have stated in our process section, this is a point of improvement. That being said, it is not very clear what Ansible's internal approach is to handle technical debt. We suggest they use the 7-steps approach by Jean-Louis Letouzey. We like this approach as it includes almost all our points of improvement.

2.5 Does Conway's Law Apply to Ansible?

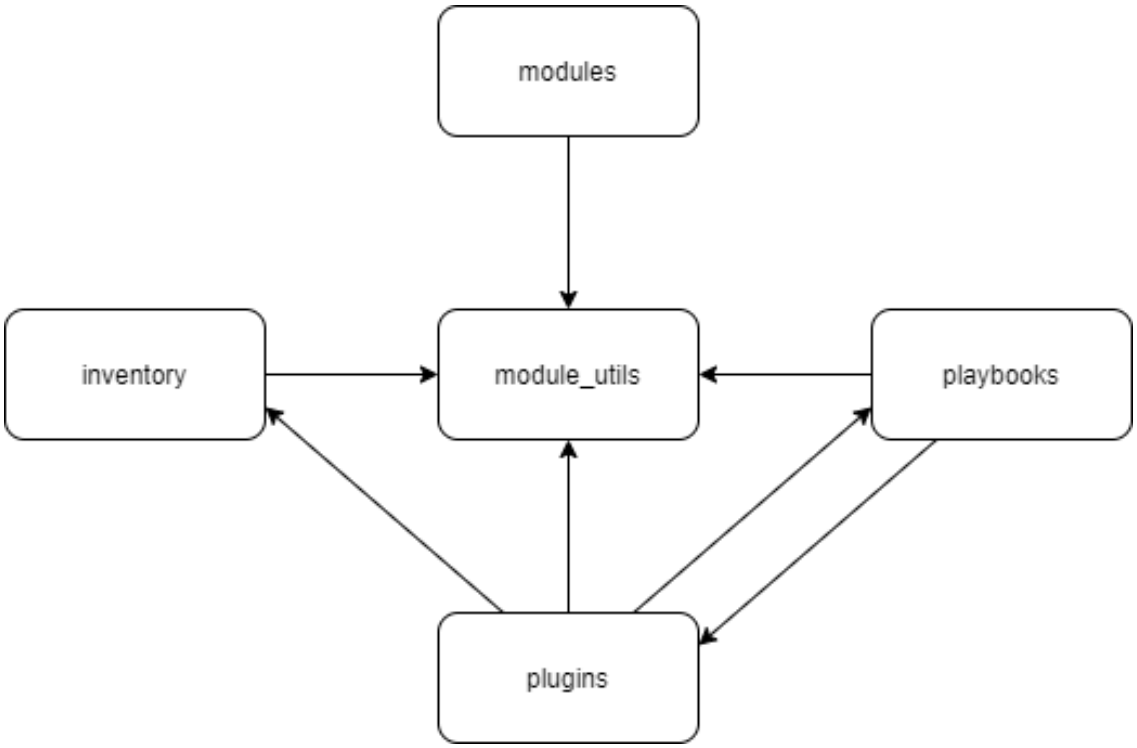
In this post the structure of Ansible will be analyzed. First the structure of the codebase will be discussed. This will be done on the component-level. The most important components will be brought to attention and their mutual coupling will be touched upon. This will be followed up by an in-depth look at the communication and social structure of the development community of Ansible.

With both strucures described, the relation between them will be discussed. The contrast and congruence will be made clear. This will be done in context of Conway's law. This law states that the structure of a system will reflect the social boundaries of the organisation(s) that produced it, across which communication is deemed difficult.

During the past month Ansible has undergone radical changes in the way its development is managed. This change also influenced the structure of the codebase which in turn changed the degree to which Ansible adheres to the concept of Conway's law. Therefore the effects of the migration will be explained in the conclusion.

2.5.1 Components coupling

For analyzing the components coupling within the ansible repository, we will look back at our findings in our second essay. Here we looked at the Development View, to understand the relations of the different components in Ansible.



As explained before, most components reside in the modules folder. This is where more than 750 modules[47] are implemented, but they are loosely coupled. Here the module maintainers work on a specific part of the code without bogging down the core code.

However, the core code itself is very tightly coupled. We see this in the bottom four components of the image. These are some of the most important components , and they are mainly maintained by the core development team. As we have discussed earlier, the ansible repository became too large to maintain, which is why the playbooks, roles, plugins, and modules that belong to one platform are migrated to one collection. Later in this essay, we will see how this affects the communication and development components.

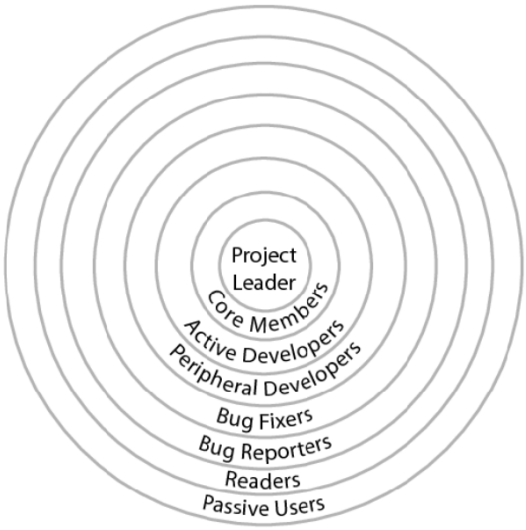
2.5.2 Communication developers

Ansible is a very large project with many contributors. It is interesting to know how they organize the communication. In this section we investigate the commmunication structure of Ansible's developers. Note that we do no talk about other employees of Ansible (such as promotion team) because we aim to investigate whether the code structure and communication structure are mirrored.

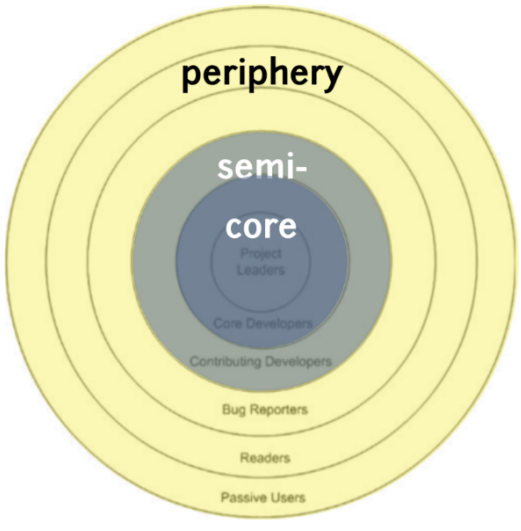
[47] https://www.ansible.com/overview/how-ansible-works

2.5.2.1 Onion model

The community structure of an open-source model can be described with the onion model[48]. In this model, there are three main layers: the core, the semi-core, and the periphery. The core includes the project leaders and the core developers. Contributing developers belong to the semi-core, which can be further divided into active- and peripheral developers, following the definition by Nakakoji et al.[49]. Lastly, there is the periphery layer that includes the remainder of the open-source community, such as bug fixers and reporters.



Seven layer community structure from Nakakoiji et al. (2002)



The Onion Model of an open source community from Amrit, C. A., & van Hillegersberg, J. (2010)

Core | The project leader of Ansible is the RedHat company. Here the top-level decisions and future plans for Ansible are made. But the other part of the core layer has an even greater influence on the decisions; the core ansible development team. The core team oversees the total project and directs the total project towards the next release candidate. Following ansible's github page, there are 62 members of the ansible core team, but examining the pull requests shows a group of 10 active maintainers:

- sivel
- mattclay
- abadger
- jborean93
- relrod
- bcoca
- acozine
- mkrizek
- samccann
- nitzmahone

[48] Amrit, C. A., & van Hillegersberg, J. (2010). Exploring the impact of socio-technical core-periphery structures in open source software development. Journal of information technology, 25(2), 216-229. https://doi.org/10.1057/jit.2010.7, https://doi.org/10.1057/jit.2010.

[49] Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., and Ye, Y. "Evolution patterns of open-source software systems and communities," in: Proceedings of the International Workshop on Principles of Software Evolution, 2002, pp. 76-85.

These members approve, merge, and close pull requests on a daily basis.

Note: there is no official documentation on the core team, the team of active maintainers may be bigger as this list is made by examining a random subset of recent pull requests

In the code base as it was before the "migration" (see our previous essay), this core team was responsible for reviewing all pull requests. As the pull request backlog of ansible was increasing, these reviews became more and more just an approval or disapproval after a module maintainer had already reviewed the code. Since the migration, the responsibility of merging the new code now lies completely with the collection maintainers.

It is not entirely clear how the core team communicates. Ansible has many IRC channels on Freenode. The 'problem' with Freenode is that we cannot trace back communication history. If a user goes offline and online again, he or she will not be albe to see communications that he or she missed. Anisble's core team also does not have location where they work together in the same office. For example, core team member Abadger is location in California, US while Akasurde works from Maharashtra, India. Red Hat has many offices all around the world. Additionally, it seems that the core team have private communication channels. After doing an extensive research, we have not managed to find out how many private channels there are and what sort decisions are made there.

Semi-core | This brings us to the semi-core layer; consisting of active- and peripheral developers. The active developers are the collection maintainers. They have full responsibility for their collection, but will follow and adjust to the decisions made by the core team. The collection maintainers lead working groups that work on a specific part of ansible. Such as the *community.windows* working group. On their github wiki page, we see a list of 8 collection maintainers. Two of them are part of the core team as they are employed by Red Hat. It can be concluded that the core team operate both in the core layer as in the in the semi-core layer.

System	Window	Solaris	macOS	HP UX	BSD	AIX
Jborean93						
nitzmahone						
bcoca						
akasurde						
kyleabenson						
chris-short						
wtcross						

Ansible has many working groups. The figure above shows which members of the core team (rows) are active in a specific working group (columns). Note that these are only the working groups that work on the 'system' component of Ansible. Green indicates that the member is either a leader or a reviewer in that group. Every working group has its own meeting schedule on Freenode. Also note that bcoca is active in many working groups. This indicates that there is also an hierachy within the core team.

People that work on a collection but are not maintainers, are the peripheral developers. They implement features that are decided on by the maintainers. However, the peripheral developers can participate in the decision making and the working group as a whole sets out the future plans. The working group wiki holds a list of ideas and a progress tracker.

Periphery | Lastly, the outer layer of the "open-source onion" consists of the remaining members of the ansible community that do not make any big decisions or are responsible for any part. A large part of this

layer is made up of bug reporters. The bug reporters are mainly the users of ansible, that come across bugs while writing their playbooks. They fill in the issue template in the corresponding part of the ansible or ansible-collections repository, with the file in question and steps to reproduce the bug. These bugs are then picked up and fixed by peripheral developers of the corresponding collection or by bug fixers in the peripheral layer, that incidentally fix bugs.

This layer communicates mostly through PRs. Ansible is well-documented and not much communication is needed with the core team. The communication relationship between this layer and the semi-core layer is a 'A reports to B' relationship. Developers in this layer do not have much authority and every proposal has to be approved.

2.5.3 Contrast code and community

In section 2 we saw that some core team members are active in many groups. This limits opportunities and as uncertainty increases, the amount of information that must be processed by decision makers also increases. This forms a bottleneck for the development of Ansible, which can lead to pressure to meet the deadlines for releases. This pressure can for example lead to increased technical debt or burnouts of the team members. This can be very harmful, since Ansible heavily relies on its core team.

Ansible must have been aware of this danger and decided to act upon it. We see in this blog post that the migration is meant to remove the core maintainers (which mainly consists of core team members) as a bottleneck to the community module and plugin development.

Looking at this from the perspective of Conway's law, one can conclude that the codebase did not match the organisation structure. To be more precise, the division of the development community in so-called "working groups" was not reflected back in the codebase. What happened in the migration is that from the perspective of the codebase a clearer distinction was created between the core code and the collections. The relation between the core code and the collections now very accurately matches the relation between the core team and the working groups, that consist of contributors.

Chapter 3

ArduPilot



Figure 3.1: ArduPilot

ArduPilot is an open source autopilot software consisting of a collection of well-established autopilot software for several vehicle types from drones to submarines, with over 1 million installs in vehicles. The community ranges from hobbyists to large institutions and groups such as NASA, Boeing, and Intel. It also features data-logging, analysis and simulation tools.

The ArduPilot software suite is made up of:

- **ArduCopter**: Multicopter UAV controller that can transform a range of helicopters and multirorot aircrafts into autonomous flying vehicles. It provides different flight modes from manual to automatic ones.
- **ArduPlane**: provides "fixed wing aircrafts" with full autonomous capabilities.
- **ArduRover**: software used for guiding ground vehicles and boats and provides acces to fully autonomous systems and capabilities leveraged by a mission planning software or pre-recorded events.
- **ArduSub**: software used for remotely operated underwater vehicles (ROVs) and autonomous underwater vehicles (AUVs) providing: depth and heading hold, autonomous navigation, etc.
- **Antenna Tracker**: firmware which calculates the position of a remote vehicle using its own GPS position and GPS telemetry from a vehicle running Copter, Rover or Plane. It then uses this information to aim a directional antenna at the vehicle.

Each software runs on a wide range of embedded hardware consisting of a microprocessor connected to some sort of peripheral sensors used for navigation. Some case studies relate to:

- **Aerial Mapping with drones**
- **VTOL Search & Rescue**
- **Agricultural automatic robot tractors**
- **Underwater exploration for coral reefs, oil pipe inspections**

More information can be found in our posts (below) or on the official website: https://www.ardupilot.org/

3.1 Team Introduction

- **Levent Gungen** is an MSc Embedded Systems student with an Electrical Engineering background, but a leniency towards software as well as writing.
- **Phu Nguyen** is an MSc Embedded Systems student with an Electrical Engineering background at TU Delft and is interested in smaller devices and IoT.
- **Serge Saaybi** is an MSc Embedded Systems student with a Computer and Communications Engineering background and two years of technology consulting experience prior to the master.
- **Arthur Breurkes** is a MSc Computer Science student with a Computer Science background at TU Delft, and is interested in data science.

3.2 Uplink to ArduPilot

As understood from its motto "Versatile, Trusted, Open", ArduPilot aims to be an ideal autopilot software platform. It presents the quality of being versatile, by supporting all kinds of vehicles and numerous sensors, components, and communication systems. It provides several features such as variable levels of automation, and simulations. ArduPilot is further described as "trusted" based on being reliable thanks to extensive use and testing (over 1 million installs), as well as being transparent, secure, and privacy-compliant. This is enabled by its third major point: open source.



Figure 3.2: ArduPilot Logo

As an open source project started by professionals, the software is openly accessible and has a large and strong community that continuously improves it to stay on the cutting edge. It aims to serve the industry, academia, and hobbyists alike, as it already does with users such as NASA and Boeing [1].

[1] https://ardupilot.org/about

We present below an overview of the overall system' vision, which would serve as an introduction to anyone planning on diving deeper into ArduPilot and its capabilities.

3.2.1 The end-user mental model

In general terms, users view ArduPilot as an autopilot entity that is expected to function essentially like a regular pilot, who manages flight objectives and parameters with respect to given instructions. It is expected to take care of tasks such as stabilization, navigation, sensor reading, regular communication, or any other supported tasks presented to end-users through the interface.

ArduPilot does not have a default graphical user interface. It is viewed via user interfaces of different supported programs called "ground stations" which may lead to slightly different mental models. The most popular ground station application is Mission Planner. The program provides a graphical user interface where users can use point-and-click controls for interactions such as vehicle configuration, tracking vehicles on maps, waypoint planning, and interfacing with simulations. A snapshot of the Mission Planner interface is given below, where waypoints are set for a circular flight[2].

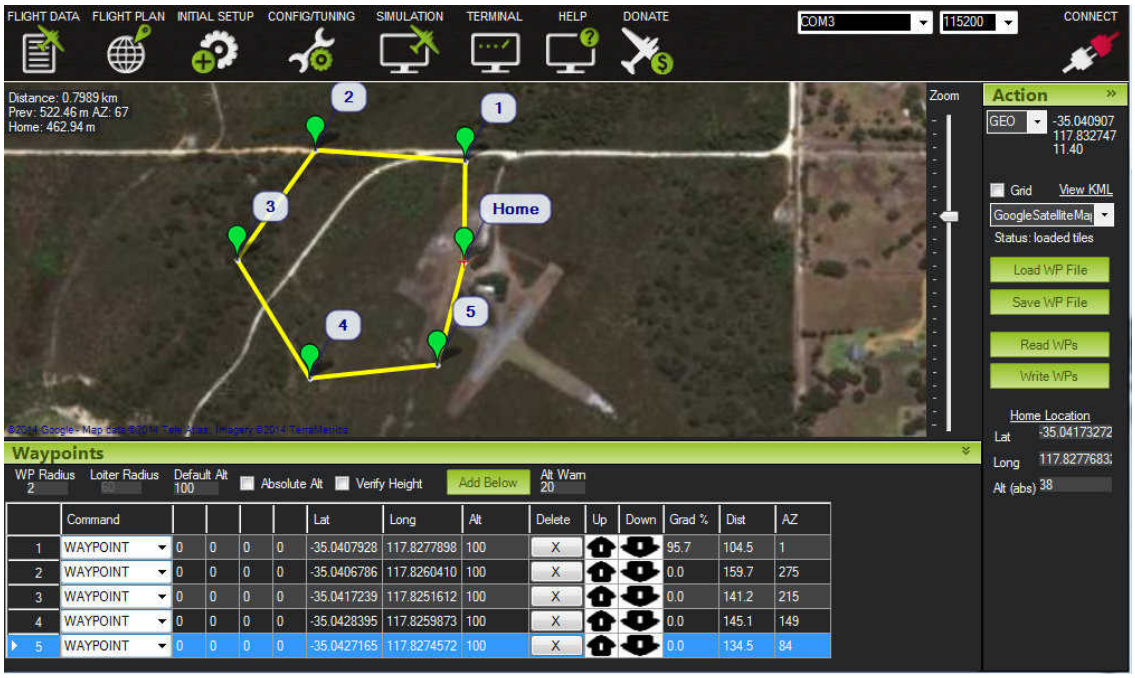


Figure 3.3: Mission Planner snapshot

Any supported ground station program on a user's computer interacts with ArduPilot on a vehicle via serial messages, with an open source protocol called MAVLink. Similarly, ArduPilot can interface via MAVLink with applications made using the DroneKit framework [3].

[2] https://ardupilot.org/copter/docs/common-choosing-a-ground-station.html
[3] https://ardupilot.org/dev/docs/learning-ardupilot-introduction.html

3.2.2 The current and future context

The autopilot software is used in different kinds of vehicles. Besides drones and rovers, ArduPilot is also used by autonomous planes and helicopters, as well as underwater vehicles such as submarines and boats.

Drones used to map out certain areas leverage the software to fly a predetermined route, without needing human interactions. Combined with IR or multispectral cameras, drones can provide information in order to help improve the yield from crops or monitor the current state of forests. ArduPilot can also be used in an agricultural setting to automate the route for a tractor, and for automatic harvesting, and saving the farmers' money on wages. Another case relates to inspecting dangerous areas, where drones are used instead of humans, to observe the area of interest in a safe manner. Moreover, crowded areas can be monitored by drones to look out for threatening situations, etc.

ArduPilot is also a leading platform for small or big autonomous underwater vehicles, ROVs (remotely operated vehicles) and AUVs (autonomous underwater vehicles), made possible by ArduSub. The latter is used for inspections, research and adventuring. These vehicles equipped with sensors help researchers solve ocean-related problems such as the impact of chemical spills or help capture invasive species that are posing a major threat to the overall health of marine ecosystems [4].

3.2.3 ArduPilot's characteristics and features

The system scope and requirements of ArduPilot relate to the main responsibilities, that is, the critical capabilities that it will be required to provide and what it is supposed to do. ArduPilot can be leveraged for developing a large system of robotic vehicles able to perform a multitude of tasks. ArduPilot pioneers in the field of Linux based autopilot hardware, rapidly lowering the barrier for working on drones and encouraging innovation the field of Unmanned Vehicles (UV).

- Copter is a flexible and customizable advanced open source system providing multicopters and helicopters with various flight modes, and offering these vehicles both manual and autonomous capabilities [5].
- The Plane firmware provides full autonomous capabilities for any fixed-wing aircraft including vertical take-off and landing (VTOL) fixed-wing aircraft that hover and cruise in different configurations [6].
- Rover is an open source autopilot for guiding ground vehicles and boats. It can run fully autonomous missions that are defined using mission planning software or pre-recorded by the driver during a manual run [7].
- ArduSub is a fully fledged UV solution for guiding underwater vehicles (ROVs) and autonomous underwater vehicles (AUVs) [8].
- The antenna tracker tracks a vehicle's location in order to properly align a directional antenna [9].

The figure below presents the key capabilities and requirements provided by each ArduPilot vehicle type best represented by the firmware components.

[4] https://ardupilot.org/index.php/casestudies
[5] https://ardupilot.org/copter/index.html
[6] https://ardupilot.org/plane/index.html
[7] https://ardupilot.org/rover/index.html
[8] http://www.ardusub.com
[9] https://ardupilot.org/antennatracker/index.html

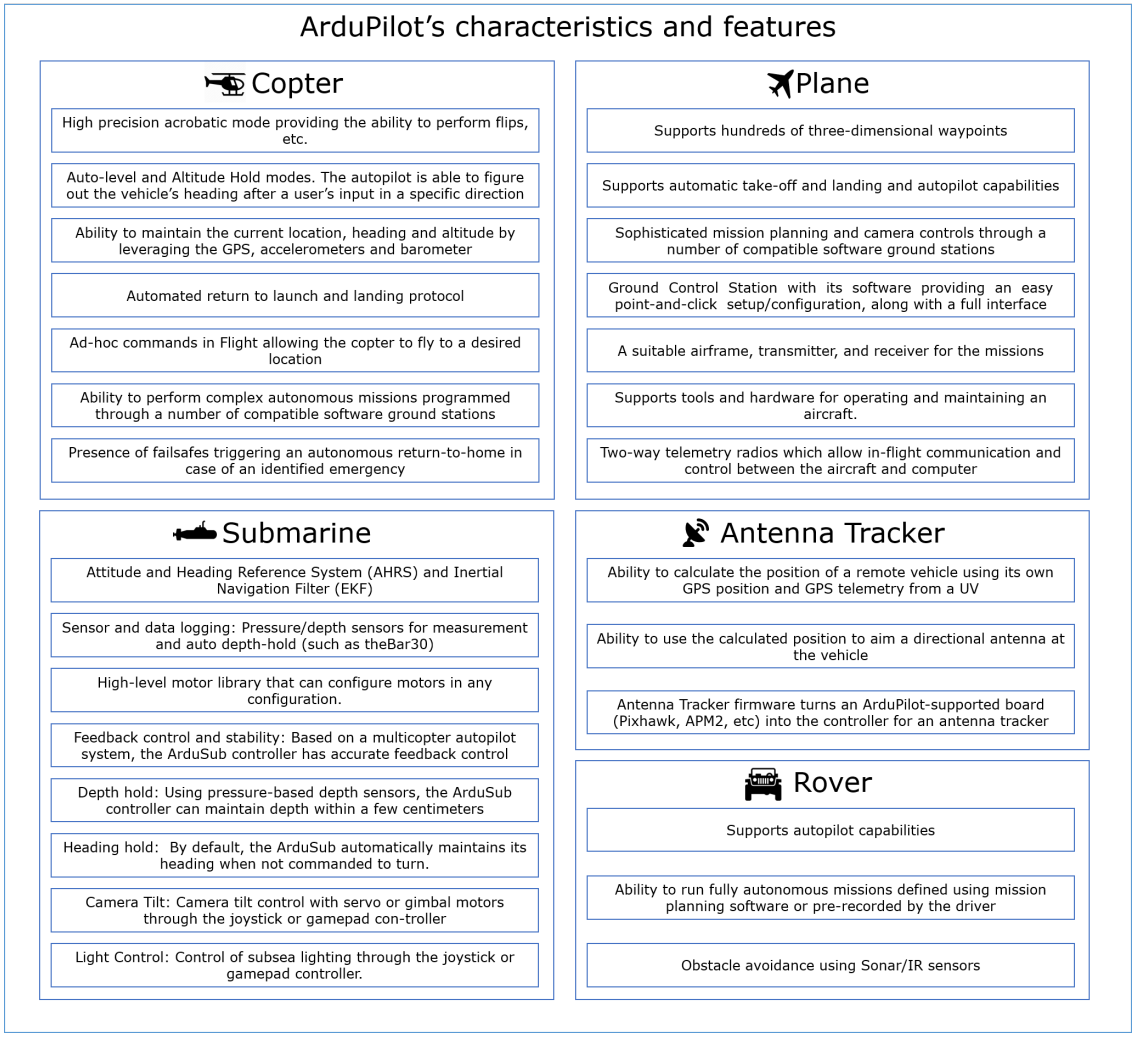


Figure 3.4: ArduPilot's key capabilities

3.2.4 Stakeholders

Following the book "Lean Architecture: For Agile Software Development" by Gertrud Bjørnvig and Jim Coplien [10], we identified five major stakeholders area to be further discussed below. These roles represent those of a lean organization focusing on the value pipeline rather than individuality.

- **End users** are the people that are actually using the software suite, which is supposed to be up to their expectations in terms of functionalities, and thus meet their technical needs. Systems can cater to a multitude of end users, therefore providing different value streams. The identified end users for ArduPilot are:
  - The hobbyists: individuals interested in UVs, experimenting with the different kinds of vehicles and systems provided by ArduPilot as part of a personal project or initiative
  - Commercial users: The software suite is installed in aircraft from many OEM UAV companies,such as 3DR, jDrones, PrecisionHawk, AgEagle and Kespry
  - Academics: Several large institutions use ArduPilot either for projects, experimenting with UVs, etc.
  - Enterprises: NASA, Intel and Insitu/Boeing using ArduPilot for testing purposes
- The **business** plays a role in the provision of the software and its functionalities to the user, ensuring some sort of Return On Investment. Investing in ArduPilot gives them the possibility to get a seat on the board and to participate in the decision making process for new functionalities and design decisions. We highlight below some of the companies investing into ArduPilot [11]. A more comprehensive list can be found on the website.
  - Carbonix
  - ProfiCNC
  - mRobotics.io
  - jDrones
  - EAMS lab
  - emlid
  - altiuas
- **Customers** treat the software as a commodity that passes through their systems. They are the ones performing the purchasing transactions. Since ArduPilot is an open source project where there is no actual "customer". However, business partners listed above may also be considered as customers since they get a seat at the "ArduPilot Advisory Board" from which they can influence how funds are spent, and can therefore request new features based on their needs as Customers - features which they will incorporate in their end products.
- **Domain experts** hold a special architectural role in the development of the system in question. They have integrated the perspectives of multiple end user communities and companies and other stakeholders into the forms that underlie the best systems. The main roles are firmware leads (plane, copter, rover, etc.), Bug master, IT and HW lead, etc. [12].

Name	Function
Andrew Tridgell	Plane Lead
Francisco Ferreira	Bug Master
Jani Hirvinen	IT & HW Lead
Jacob Walser	Submarine Maintainer

[10] http://www.leansoftwarearchitecture.com
[11] https://ardupilot.org/about/Partners
[12] https://ardupilot.org/index.php/about/team

Name	Function
Lucas De Marchi	HAL Linux maintainerx
Michael du Breuil	GPS Maintainer
Michael Oborne	MissionPlanner Lead
Peter Barker	DataFlash & Tools Maintainer
Randy Mackay	Copter & Rover Lead
Tom Pittenger	Plane Co-Lead
Bill Geyer	Traditional Heli Co-Maintainer
Chris Olson	Traditional Heli Co-Maintainer
Leonard Hall	Copter Control & Navigation Maintainer
Matt Lawrence	Solo Maintainer
Paul Riseborough	EKF Maintainer
Pierre Kancir	Copter & Rover SITL Maintainer
Bill Bonney	APMPlanner
Craig Elder	Founders of the ArduPilot Initiative
David Bussenschutt	ArduPilot software developer
Grant Morphett	Rover Co-Lead
Philip Rowse	HW Lead

- **Developers** are responsible for the construction of the system. The software suite is also used for testing and development by several large institutions and corporations such as NASA, Intel and Insitu/Boeing, as well as countless colleges and universities around the world. The top 6 developers are identified from the main contributors page on GitHub based on their contributions to the master (excluding merge commits) from Sep 4, 2011 – Mar 1, 2020 [13]
  - Andrew Tridgell (“@tridge”)
  - AerialRobotics Australia Pty Ltd
  - Randy Mackay (“@rmackay9”)- Japan Drones
  - Peter Barker (“@peterbarker”)
  - Lucas De Marchi ("@lucasdemarchi)- Intel
  - @WickedShell (name not provided)
  - Paul Riseborough (“@priseborough”)- GNC Solutions Pty Ltd

3.2.5 Roadmap and future work

Regarding the future, former technical community manager for Dronecode (who leads software teams in ArduPilot) Craig Elders says the following about the roadmap for ArduPilot in 2016:

> “Some people would say we have no road map, and that is a good thing because we are not limiting it to our own imaginations. We are allowing people to do something new and something creative with it.” [14]

Each year, a new roadmap is developed by the ArduPilot team. This roadmap is not meant to guarantee when features will be added, but instead offers people a means to collaborate with others that share similar interests. The roadmap for 2020 will be made during the ArduPilot Developer Conference from March 27th

[13] https://github.com/ArduPilot/ardupilot/graphs/contributors
[14] https://www.interdrone.com/news/surveying-mapping/craig-elder-talks-the-future-of-ardupilot-outside-of-dronecode

to 29th this year. New functionalities for vehicles are added every year, as well as additional support for more hardware and OS [15].

The roadmap for 2019/2020 is illustrated below, with the main architectural changes and additions to be made to ArduPilot. The different streams are ran in parallel with different developers acting as point of contact.

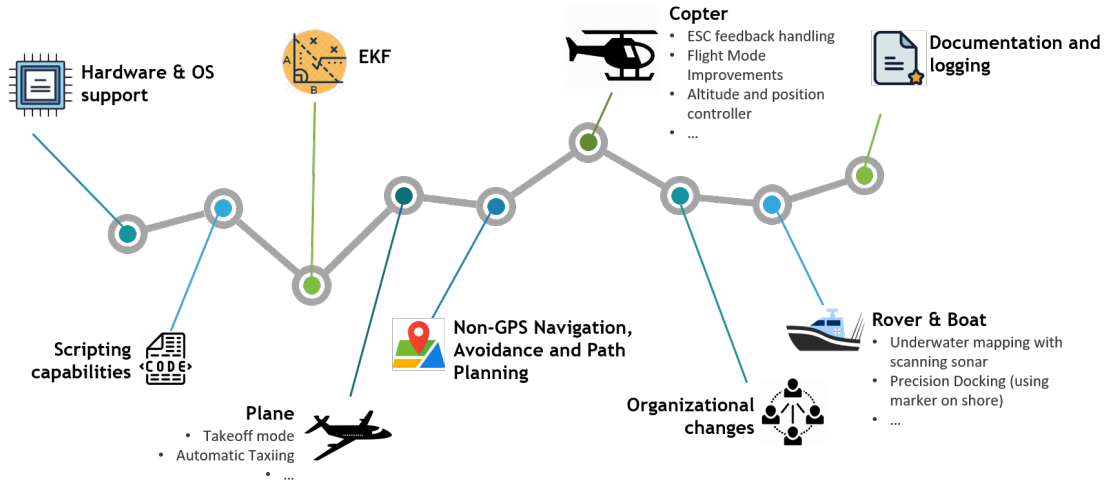


Figure 3.5: 2019/2020 Roadmap

3.3 What Makes ArduPilot Soar - An Architectural Overview

Having introduced ArduPilot as a whole in our first post, in this second one we look into its components. For ArduPilot, having a sound architecture is important because the software needs to manage various sensors and actuators while operating all kinds of drones like helicopters, planes, rovers, or even submarines. As exemplified by the popular "4+1 View" model [16] of software architecture as well as various academic sources [17] [18], there are many ways to view the architecture, each with a different focus. In this post we analyze ArduPilot using some of these views. We focus on the different architectural elements of ArduPilot and the connections between them, and the design principles used by the software that allow it to soar.

3.3.1 Architectural Fundamentals

In terms of architectural views, we see from our research that the Development View, Deployment View, and Run Time View are relevant to the project, because the developers discuss topics about them at length in the documentation [19]. We will look at these views in detail after mentioning some fundamental aspects of the architecture.

[15] https://ardupilot.org/dev/docs/roadmap.html
[16] https://en.wikipedia.org/wiki/4%2B1_architectural_view_model
[17] Jim Coplien Gertrud Bjørnvig. Lean Architecture for Agile Software Development. Wiley, 2010
[18] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[19] https://ardupilot.org/dev/docs/learning-ardupilot-introduction.html

A key aspect of ArduPilot is it being designed with a multi-layered structure. It has three layers: vehicle-specific code, shared library code, and the hardware abstraction layer ("HAL"). Similarly, ArduPilot itself is a part of a larger multi-layer system, as later described when describing the module structure. Meanwhile, the codebase is split into five parts, with the three layers having their own parts. The other two parts are tools directories (like testing tools), and external support code (that runs alongside ArduPilot) [20]. This functionality-based division into layers and parts demonstrates the principle of the separation of concerns, which allows developers to manage each aspect of the software individually while treating the others as blackboxes [21].

Looking at the architecture of ArduPilot's code files, many of them imitate the iconic "setup()" and "loop()" structure of Arduino sketches[22] from which the software gets its name. However unlike Arduino sketches, ArduPilot's software is multi-threaded[23]. Also, because it deals with sensors and controllers, ArduPilot has an event-driven architecture, operating primarily using tasks and a task scheduler [24].

While it is normal for form to reflect function, we see that non-functional properties too shape ArduPilot's architecture, namely universal applicability and security:

Universal Applicability

ArduPilot aims to be universally applicable. This means that its developers want it to run on as many kinds of unmanned vehicles as possible. Although the development team hasn't explicitly stated it, this goal is arguably why ArduPilot has no first-party graphical user interface (GUI). If ArduPilot were to implement custom GUI's for the dozens of vehicles it supports[25] , it would have to shift much of the developers' efforts away from controlling the vehicles to designing several specific GUI's. Therefore, ArduPilot has users write their own GUI software or pick from supported external "ground stations" that serve this purpose[26], but in return the autopilot core gets much more development time.

Security

It is possible for vehicles to receive control messages from different ground stations than the one they were booted up on. While going through the code, we saw that a simple but effective security measure was put in place to prevent this: The vehicle would check whether the message received came from the ground station they were registered on, and would not execute the order otherwise. Thus we see that ArduPilot was designed to support one ground station at a time. This is good for security, but in return it makes users unable to send their vehicles very far away, where the control could be passed over to other ground stations.

3.3.2 Development View: ArduPilot as A Software Collection

Careful design decisions are needed when creating a development environment that can successfully realize ArduPilot. The development viewpoint relates to the code and modules structure and the building, testing, releasing, and configuration requirements standardized within ArduPilot [27].

Module Structure of ArduPilot

[20] https://ardupilot.org/dev/docs/learning-ardupilot-introduction.html
[21] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[22] https://www.arduino.cc/en/Guide/Introduction
[23] https://ardupilot.org/dev/docs/learning-ardupilot-threading.html
[24] https://github.com/ArduPilot/ardupilot/blob/05a0fe615b50fc2fa8dba11e541ba4f5740a1181/libraries/AP_Scheduler/AP_Scheduler.cpp
[25] https://ardupilot.org/copter/docs/common-all-vehicle-types.html
[26] https://ardupilot.org/copter/docs/common-choosing-a-ground-station.html
[27] Software Systems Architecture, Chapters 20 (development viewpoint)

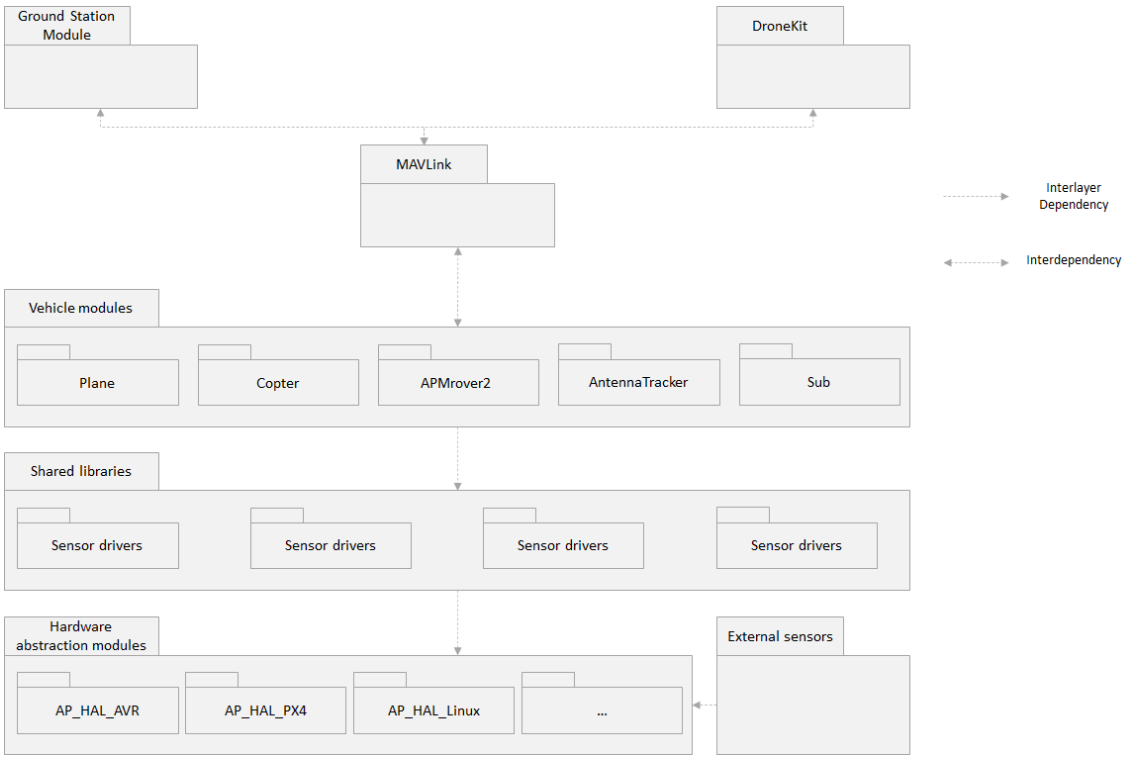


Figure 3.6: Module structure model for ArduPilot

ArduPilot's three layers can be seen in this structure, while communication (with MAVLink) and user interfacing (with Ground Station or DroneKit) can be seen as two other layers above it.

- Ground Station module: As a supported external software for ArduPilot, Ground Station runs on a ground-based computer, and communicates with an unmanned aerial vehicle (UAV). It displays real-time data on the UAV, and can be used for controlling it.
- DroneKit: As another supported external software, DroneKit helps create powerful apps for UAV's, running on a UAV's Companion Computer.
- MAVLink: An external software used by ArduPilot with its own protocols for communication with ground stations / companion computers.
- Vehicles Modules: The unique firmware for each vehicle type.
- Shared libraries modules: The code usable by multiple vehicles including core, sensors, and other libraries for various functions: control, navigation, etc.
- Hardware abstractions module: Renders ArduPilot portable to many different platforms and development boards.

ArduPilot's Source Code Structure

The overall structure of the ArduPilot directory is represented above [28]. It is designed to be simple to comprehend, as each folder is named after the module it represents. This layout also exhibits a separation of concerns, which simplifies development as previously explained.

Given the structure, we have vehicle code files in the folders APMrover2, ArduCopter, ArduPlane, and ArduSub. This code is largely `.cpp` files since ArduPilot is largely written in C++, while vehicle directory also contains a make.inc file which lists library dependencies for that vehicle. Meanwhile, the Antenna Tracker folder is home to the firmware responsible for controlling the Antenna Tracker which tracks a vehicle's location in order to properly align a directional antenna.

The Tools folder contains miscellaneous support tools, namely the AutoTest suite which regularly tests the vehicle code. The libraries and docs folders are respectively home to the shared libraries and the documentation for the four vehicle types and the AntennaTracker.

Build, Integration, and Test Approach of ArduPilot Developers

Building ArduPilot differs based on the platform it is meant to run from (i.e. Linux/Ubuntu, MacOS or Windows). The main steps are:

1. Setting up the Build Environment on the required platform
2. Building / Compiling, which supports two build systems, "waf" or "make", based on the board used
3. Building Mission Planner (which is an independent ground control station software used for Plane, Copter and Rover) [29]

For continuous integration, ArduPilot makes use of "Travis CI", an integration service used for building and testing software projects hosted on GitHub [30].

Thus, each integration can be directly verified by an automated build and automated tests allowing developers to quickly detect and locate errors. The `.travis.yml` file in the root directory specifies the programming language used (C++), the dependencies, building environment, testing, etc. [31]

[28] https://github.com/ArduPilot/ardupilot
[29] https://ardupilot.org/planner/docs/mission-planner-overview.html
[30] https://en.wikipedia.org/wiki/Travis_CI
[31] https://travis-ci.org/ArduPilot/ardupilot

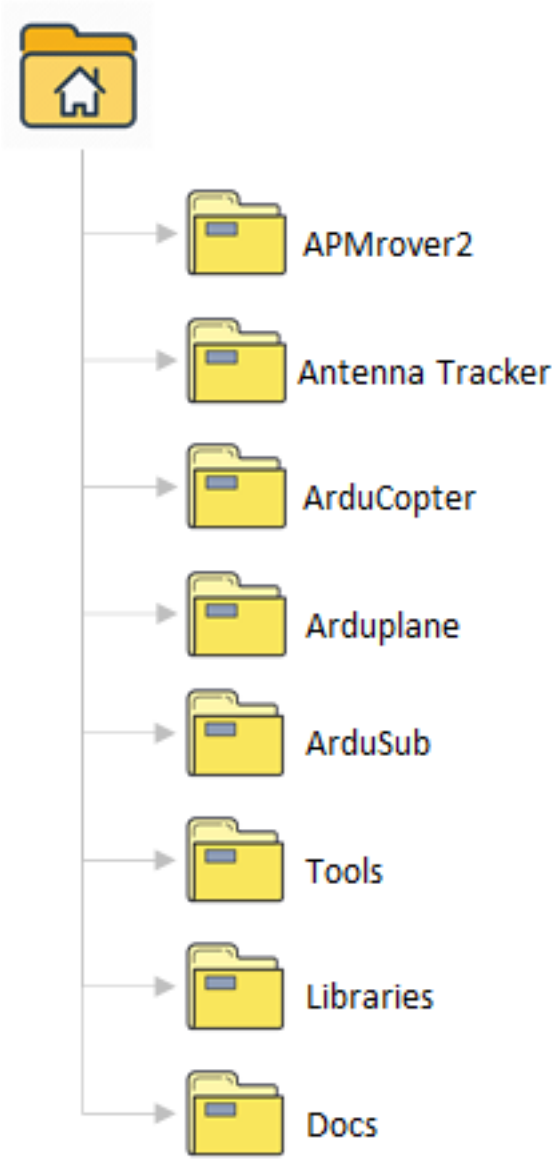


Figure 3.7: The home directory of the source code

As for testing, an autotest consisting of 28 tests can be run [32]. Its results and logs can be visualized and monitored from the project's root. Testing can be performed either on real hardware or in simulations that allow for safe testing of experimental code and settings. Some of the most commonly used simulators for ArduPilot testing are: SITL, Gazebo, RealFlight, and AirSim [33][34].

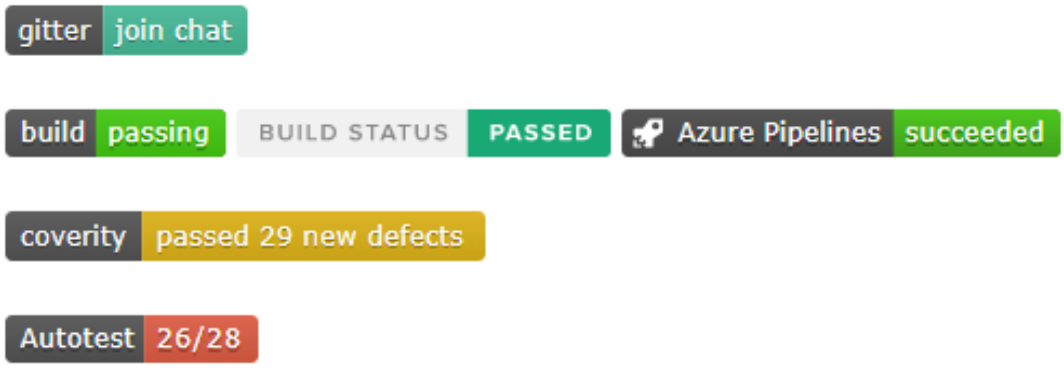


Figure 3.8: Autotest runs after each integration. The results can be checked along with their status from the GitHub root directory by accessing the links in the image above

The diagram below describes the overall build, integration and testing approach followed by ArduPilot developers:

ArduPilot's Release Process

A clear process for releasing ArduPilot is available on the website [35]. We have summarized it here:

1. Alpha Testing: AutoTester runs these tests after a commit is done
2. Releasing Beta Versions
   - Create a new release branch or switch to an existing release branch in the GitHub repository
   - Pull from the master
   - Update release details such as version, release notes and tags
   - Make sure Mission Planner displays correctly the new versions added
   - Announce the availability of a new version to the beta testers, who test in simulations or on hardware
3. Releasing Stable Versions
   - After weeks / months of beta testing, and no more unexplained crashes, prepare to release the stable version
   - Discuss the go-no-go decision on a stable release on the preceding weekly development call
   - Release the stable version similarly to the beta one and add the "stable" version tag: i.e. add the ArduCopter-stable and ArduCopter-stable-heli tags in the case of a version relating to ArduCopter
   - Create an additional tag with the patch release number

[32] https://github.com/ArduPilot/ardupilot
[33] https://autotest.ardupilot.org/
[34] https://ardupilot.org/dev/docs/simulation-2.html
[35] https://ardupilot.org/dev/docs/release-procedures.html

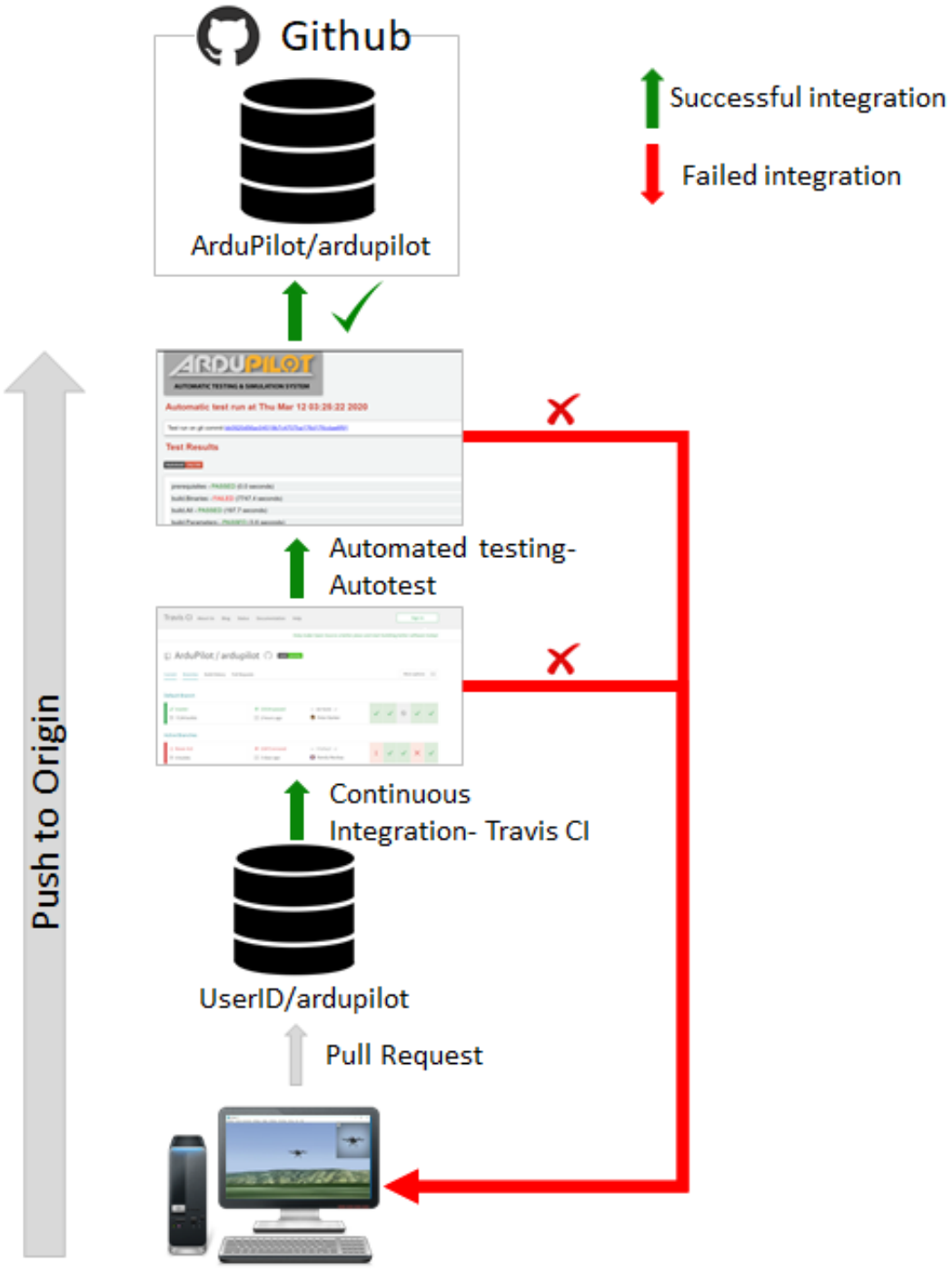


Figure 3.9: Build, Integration, and Test Approach

- Make an announcement on ArduPilot's forums and on ArduPilot's Facebook page. Mission Planner will also show a pop-up informing users that a new version is available.

Configuration Management

Currently, ArduPilot does not seem to have an official configuration management process that keeps a record of any parameter changes. However, GitHub can be used instead for easy configuration management: one can check the changes in the code and quickly switch to different versions or settings. Hence there is no need to manually do things like maintaining backups.

3.3.3 Run Time View: Connecting the Modules

ArduPilot has to take care of different scenarios while it runs. The run time view specifies how the modules work together to realize certain key scenarios. Think of starting up a drone and configuring it, then controlling and moving it as you want, and then returning the vehicle back to the ground control. We will describe the first two of these scenarios in further detail, assuming the vehicle is a drone copter. In the first scenario, we talk about the dependencies between the process, the libraries and the HAL to realize the initialization. In the other scenario we will talk about the dependency between certain actions during autonomous flight mode.

Starting up

For dronecopter running ArduCopter, the function `Copter::init_ardupilot` takes care of all necessary initialization before taking off. Surprisingly, this function initially assumes that the copter is in mid-air. Later on, the system determines if the copter is actually in the air or on the ground. Only at a ground start, the copter will perform a calibration to ensure correct values on its devices. During the startup phase, several modules are initialized which include: * Board: setting the correct time, scheduler delay and SD Card settings * Battery: clearing out the cell voltages and turning on battery monitoring * Radio channels: setting values for the flight control parameters of roll, pitch, yaw and throttle, and the default dead zones * Sensors: initializing the GPS and the compass * Motors: settings the loop and update rate

The figure below shows how some of these initializations depend on the shared library layer and HAL layer in order to do an operation.

Automated Control

A copter on the ground is cool, but making it fly is a lot more impressive. How about making it fly on its own? Even better! The autopilot control of the copter is done in the `mode_auto.cpp` file. The file has its own initialization for the automated controller and contains 10 different implementations such as taking off, waypoint navigation, and landing. The function `ModeAuto::run()` ensures that the correct auto controller is selected, and `ModeAuto::run_autopilot()` handles the decision making process as well as the non-navigation related commands. Each type of action has its own verifying operation, which returns true if the action has completed. Only when the verify function returns true, a new action is allowed.

3.3.4 Deployment View: Realizing ArduPilot

Since ArduPilot supports four different types of vehicles, an end user can implement the software based on the type of vehicle they want to deploy. The software is commonly used by commercial companies seeking professional performance levels [36], or by hobbyists who strive to augment their toys with custom features

[36] https://bluerobotics.com/

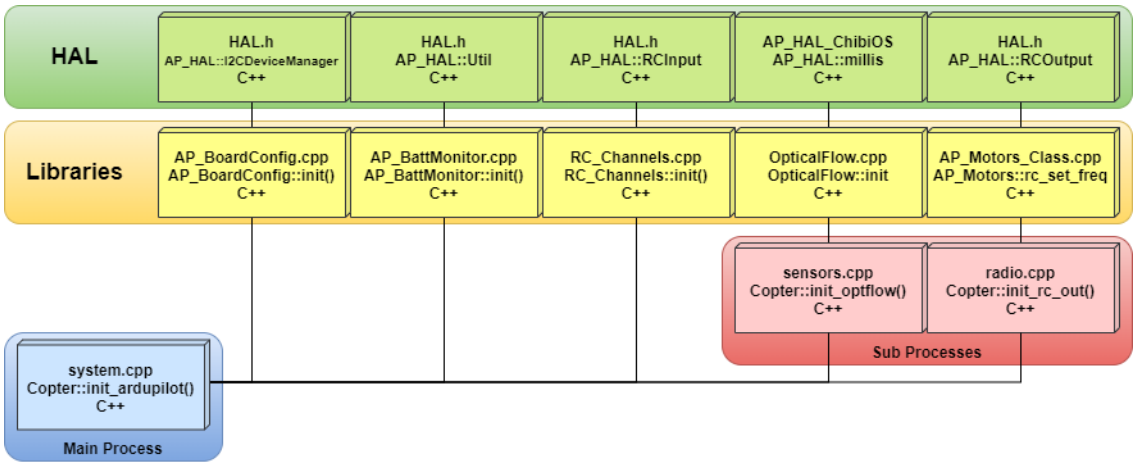


Figure 3.10: Small overview of the initialization

in a more affordable way [37]. The ArduPilot software can be implemented on various dedicated control boards, such as the Pixhawk and the Mateksys. As well as on the more popular Raspberry Pi boards with a Navio2 HAT. ArduPilot runs on Linux, and is in the process of supporting ChibiOS too [38][39]. Vehicles that are supported by ArduPilot are listed on their website [40].

3.3.5 Conclusion

In this post we examined the architecture of ArduPilot both in general and from multiple views. Having three different views with different focuses enables the separation of concerns. This allows developers to work on specific aspects of the project without having to consider all aspects at once [41]. Meanwhile, the three views come together to paint a wholistic picture of the project.

3.4 Under the Hood of ArduPilot: Software Quality and Improvements

Having gone through the vision and key architectural aspects of ArduPilot in our previous essays, we will focus in this post on the software quality. Initially we look at the continuous integration and test processes. Then we examine recent coding activities and their alignment with the roadmap. Finally, we assess the code quality and maintainability, as well as the technical debt.

[37] https://ardupilot.org/copter/docs/common-fpv-first-person-view.html
[38] https://ardupilot.org/copter/docs/common-loading-firmware-onto-chibios-only-boards.html
[39] https://diydrones.com/profiles/blogs/plane-3-9-0-stable-released
[40] https://ardupilot.org/copter/docs/common-all-vehicle-types.html
[41] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

3.4.1 Continuous Integration: ArduPilot's Journey

Continuous Integration (CI) is a way of developing a software application requiring the developers to continuously integrate their updated code in a shared repository several times during a certain predetermined period of time (could be several times a day, etc.). Pull Requests are thus verified by an automatic build, allowing teams to figure out and detect problems early on, and locate them more easily.

Concerning ArduPilot, the developers follow a certain set of practices and processes ensuring the continuous integration of ArduPilot, such as maintaining a single source repository on GitHub, automating the build, testing and deploying process, testing in a different branch than the master, and ensuring the latest version of the code is the one being worked on. A large part of these steps are automated using the Travis CI, Semaphore CI, AutoTest ArduPilot and Azure DevOps, which are launched automatically at each pull request.

The latter procedures are maintained by ensuring that developers follow the practices below before submitting patches to the Master (consists of building and testing ArduPilot):

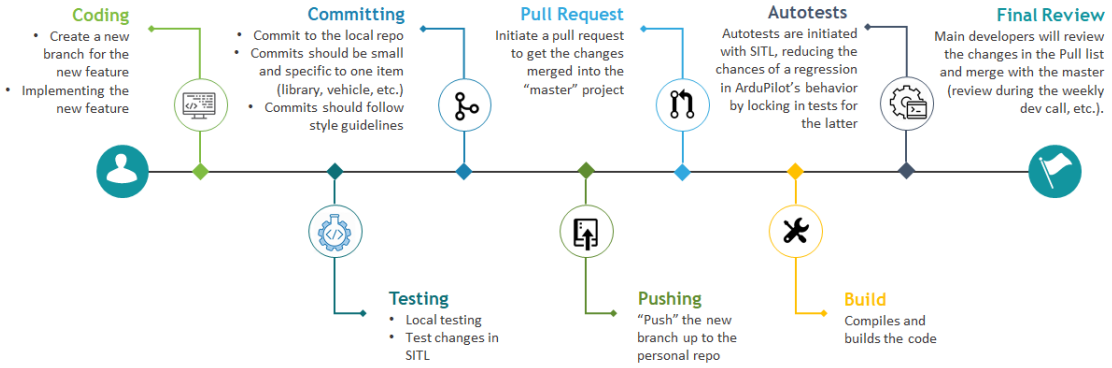


Figure 3.11: Continuous Integration process

3.4.2 Pre-flight Checks: ArduPilot's Test Process

As mentioned in our previous essay, the test process for ArduPilot is in summary[42]:

1. Alpha Testing: Run by AutoTest
   - After most commits done to the project's master branch, AutoTest runs standard tests in a simulation.
2. Beta Testing: Beta testers conduct tests in simulations or on hardware.
   - Beta versions are defined, released, and announced by developers (example).
   - Beta testers report bugs.
   - After weeks of testing, developers discuss the go-no-go decision for a stable release on development calls
3. Stable Releases: Prepared versions are released and announced.
   - Forums are used for further bug reports and related testing discussion.

[42] https://ardupilot.org/dev/docs/release-procedures.html?highlight=release

3.4.2.1 Simulating for Success

ArduPilot's "Software in the Loop (SITL)" Simulator is an integral part of its testing process. It is a build of the autopilot/flight code that uses an ordinary C++ compiler and offers several debugging tools. When ArduPilot is running in SITL, the sensor data comes from a model of the craft in a flight simulator, while the autopilot treats it the same as real world data[43].

SITL can simulate many vehicles and devices including multi-rotor or fixed wing aircraft, ground vehicles, underwater vehicles, support devices like antenna trackers, and optional sensors. In addition, new simulated vehicle types or sensors can be added by following the steps[44].

ArduPilot's versatile design makes it indifferent to the platform used. This includes virtual platforms in SITL such that ArduPilot behaves in a simulated environment exactly as it would in the real world [45]. This eliminates most issues about reproducing bugs or about the software behaving differently in different environments. Hence successful simulation tests, with sufficient coverage of the code and use cases, can be considered as credible and reliable.

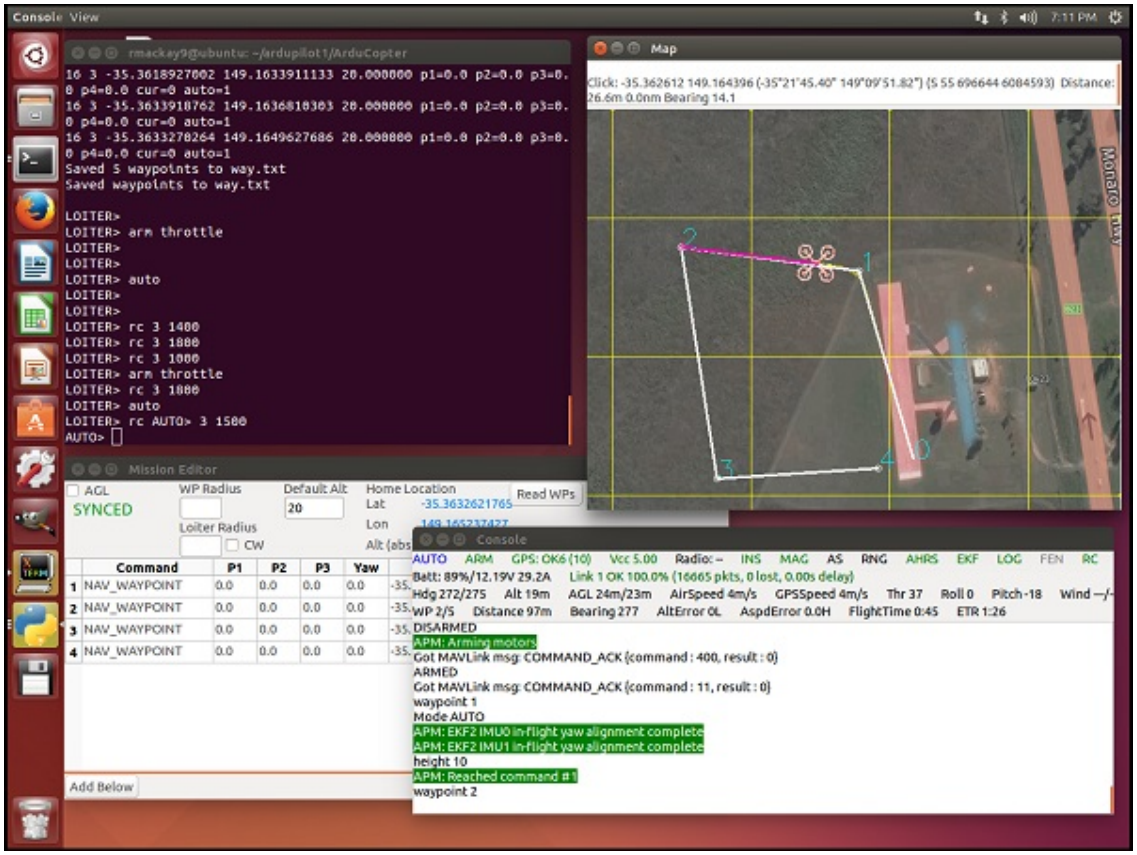


Figure 3.12: An example screenshot of using SITL

[43] https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html
[44] https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html
[45] https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html

3.4.2.2 AutoTest-ing

The AutoTest suite is an ArduPilot tool that is used to quickly create and run repeatable tests in SITL. It saves developers time that would be taken for manually preparing and running complex tests. This allows running several tests with small, controlled differences in order to isolate conditions causing bugs. Furthermore, the bug-causing conditions can be made into a repeatable test. The prepared test can be sent to developers to work on fixing that specific bug. Hence "test-driven-development" proceeds[46].

Apart from being used locally, the AutoTest suite runs automatically on ArduPilot's AutoTest server, which automatically tests most commits made to the master branch with respect to 28 standard tests for catching the most frequent bugs. Debugging aside, developers are also encouraged to prepare or use the repeatable tests to clearly demonstrate how their patches make a difference in flight behavior[47].

3.4.2.3 Developer Testing

During the Beta testing, and in response to bug reports after releases, developers of the community use both simulators and real vehicles for testing. Effective communication between developers is made possible via public forums (such as the ArduPilot forums and GitHub issues page) where issues and the work done about them can be logged and tracked easily. Furthermore, the developer chat page and the weekly development call provide an environments for real time discussions of bugs by senior developers[48]. Hence selected bugs can be identified, addressed, and solved efficiently by the community.

3.4.2.4 Code Coverage

To analyze code coverage, ArduPilot developers use the open source tool LCOV, an extension of GCC's GCOV [49]. The tool reports how many times each line of the code in the repository was executed in a given test. It can be used to verify whether the intended executions happened. ArduPilot developers use it for some tests (but not all) and keep the generated reports online. This way, they are able to verify sufficient code coverage of successful tests.

3.4.3 Architectural Activity: Diving Into ArduPilot's Code History

Over the past years, people have worked on different components within ArduPilot. In this section we look at their recent merge history.

3.4.3.1 Recent Focus

We mentioned in our last essay that the code structure has 3 types of modules: vehicle modules, shared libraries, and hardware abstraction layer modules (HAL). Digging into each individual commit results in the figure shown below. By far, the most commits are done on the libraries and the fewest are done on vehicle codes. At the end of last year, ArduPilot introduced experimental support for Lua scripting [50] [51]. Only recently they included the math and string library for Lua, which explains the high number of code changes. Also included in the figure is *Tools*. Lately, there have been many contributions to the AutoTest, which

[46] https://ardupilot.org/dev/docs/the-ardupilot-autotest-framework.html
[47] https://ardupilot.org/dev/docs/the-ardupilot-autotest-framework.html
[48] https://ardupilot.org/ardupilot/docs/common-contact-us.html
[49] https://github.com/ArduPilot/ardupilot/blob/194998d6314082f7d51a64d7fc967316ad766a05/Tools/scripts/run-coverage.sh
[50] https://github.com/ArduPilot/ardupilot/blob/master/ArduPlane/release-notes.txt
[51] https://ardupilot.org/copter/docs/common-lua-scripts.html

explains the high number of code changes. Interestingly, the HAL has more commits than Tools, but fewer changes in code, which indicates that the HAL module is worked on actively, but only small changes are made each time.

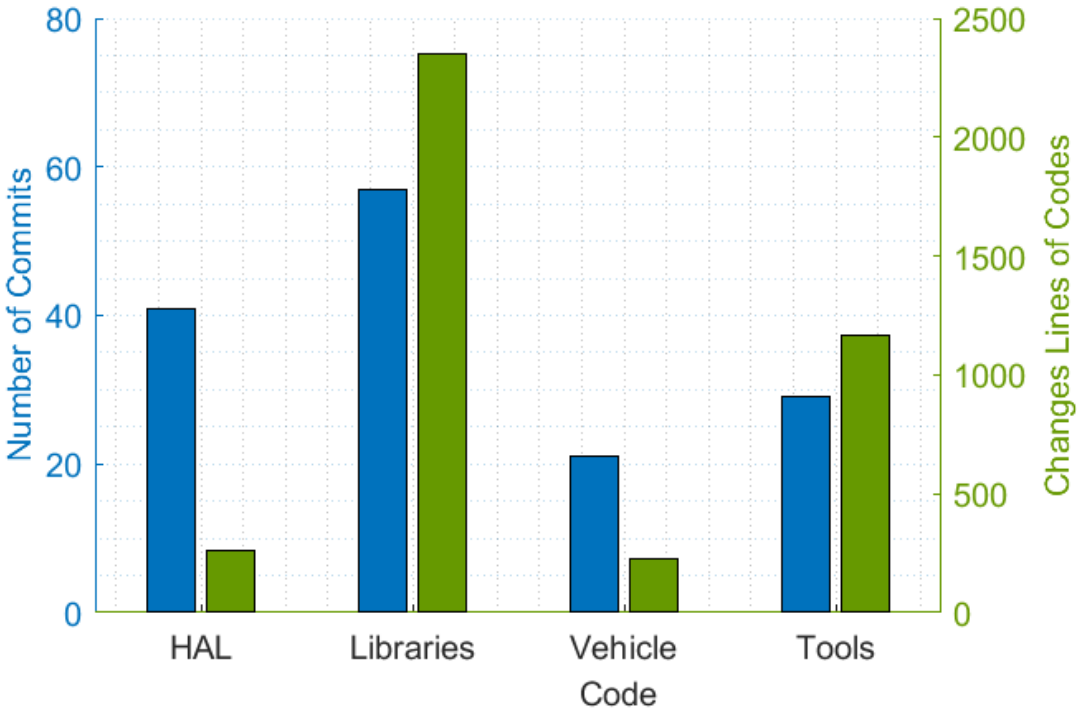


Figure 3.13: Merge history February 19th - March 23th

A further look into the different vehicle codes show that ArduCopter is the most active in terms of code changes, and the least active is ArduSub.

3.4.3.2 Implementing the Roadmap

ArduPilot has a roadmap of features they want to add for 2019 and beyond. An overview of some of the implemented features in the release versions are shown in the figure below.

The figure shows 4 features that are implemented. As the features do not consist of only a single function, but rather quite a lot of code and files, only some files are showed that are related to these features. As can be seen, the *CAN ecosystem ramp-up* mainly focuses on the HAL module, the implementation of Lua is part of the shared libraries, and both features for the copter and plane are part of the vehicle code module.

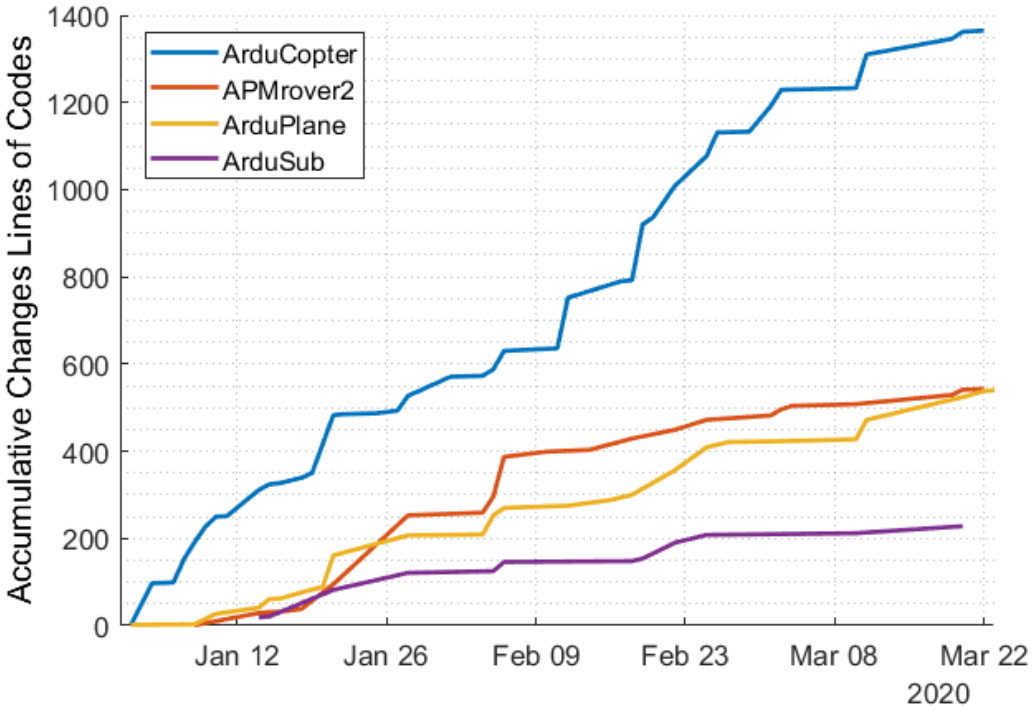


Figure 3.14: Merge history different vehicle codes in 2020

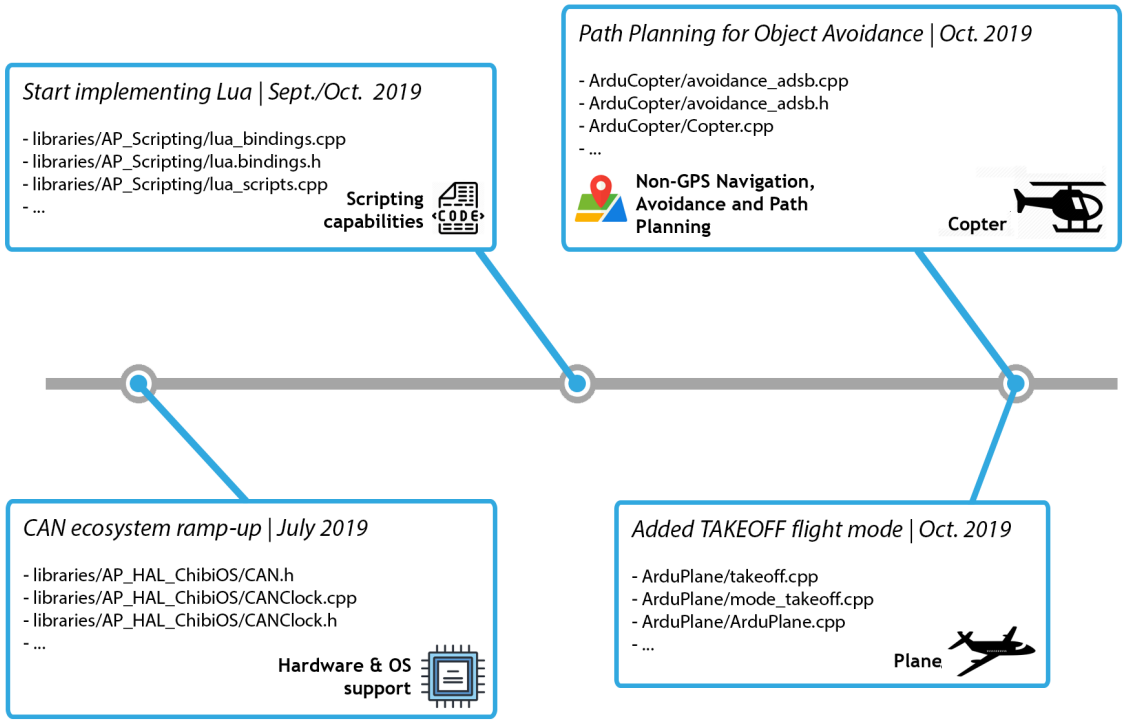


Figure 3.15: Mapping the roadmap onto actual files

3.4.4 Does the Code Look Any Good?

Code quality is arguably one of the most important aspects of a successful project [52]. It goes hand in hand with qualities like reliability, maintainability, and efficiency. With approximately 40.000 lines of code, ArduPilot can be considered a relatively large project. And for large projects, we could say that maintainability is extra important. Think about all the extra effort that has to go into modifying/adding features or fixing bugs.

After analyzing the code in the ArduPilot project, we concluded that the developers struggle with two things concerning maintainability: unit size and complexity. In other words, methods are overly large and overly complex, indicating that they might not all have just one responsibility. This makes it difficult for the developers to pinpoint exactly where bugs happen or where to start creating new features. Since the main components are where most of the changes in code happen, this can cause a lot of issues.

In our first pull request for this course, we have addressed the unit size issue for one of the `APMRover2` files. A method that was over 300 lines long, has now been separated into smaller methods, of which each has its own responsibility. Arguably some of the newly introduced methods could have been smaller. However, we chose not to do so because it would split the method's responsibility over multiple methods. Contributors quickly responded to the pull request, one of them saying:

> **khancyr**: *I like the idea as it will be simplier to deal with those functions (and use a static analyzer).*

From this response we can conclude that (some of) the developers might not have had maintainability in mind at all times while writing the code, but that their eyes have been opened.

3.4.5 Technical Debt: What the Numbers Say

Technical debt is a measure quantifying the consequences of having deliberately done "something wrong" in order to quickly deliver a product to market, focusing on short term benefit rather than a sustainable design strategy. In this section, we will analyze the technical debt within ArduPilot in terms of the identified parameters, tools, and methods used. We used SonarQube's [53] community version, which assesses HTML, CSS, and python files. However, much of ArduPilot's code is in C++ and the latter's analysis is supported only in the paid version of SonarQube. The assessment is given below.

We also used the static analysis from SIG [54] to analyze ArduPilot's debt. For each of the metrics identified below, we followed the "SIG/TUViT method" and identified thresholds allowing us to qualitatively address them [55] [56] [57].

3.4.5.1 Code Duplication

Measures the degree of duplication in ArduPilot's code. SIG provided a score of 3.1/5.0 on this metric. Analyzing the results per file, we notice that the average percentage of duplication is of 12% for the different vehicles codes. Putting this into perspective, we find it appropriate to lower the weight of this metric in the

[52] https://medium.com/emblatech/the-importance-of-code-quality-ac7afa598c0d
[53] https://www.sonarqube.org/
[54] https://sigrid-says.com/maintainability/tudelft/ardupilot/components
[55] Sigrid manual - https://sigrid-says.com/softwaremonitor/tudelft-ardupilot/docs/Sigrid_User_Manual_20191224.pdf
[56] Test Code Quality and Its Relation to Issue Handling Performance- http://resolver.tudelft.nl/uuid:3b6e5a90-d338-4c78-8c84-9c78598568bf
[57] SIG/TÜViT Evaluation Criteria Trusted Product Maintainability - SIG

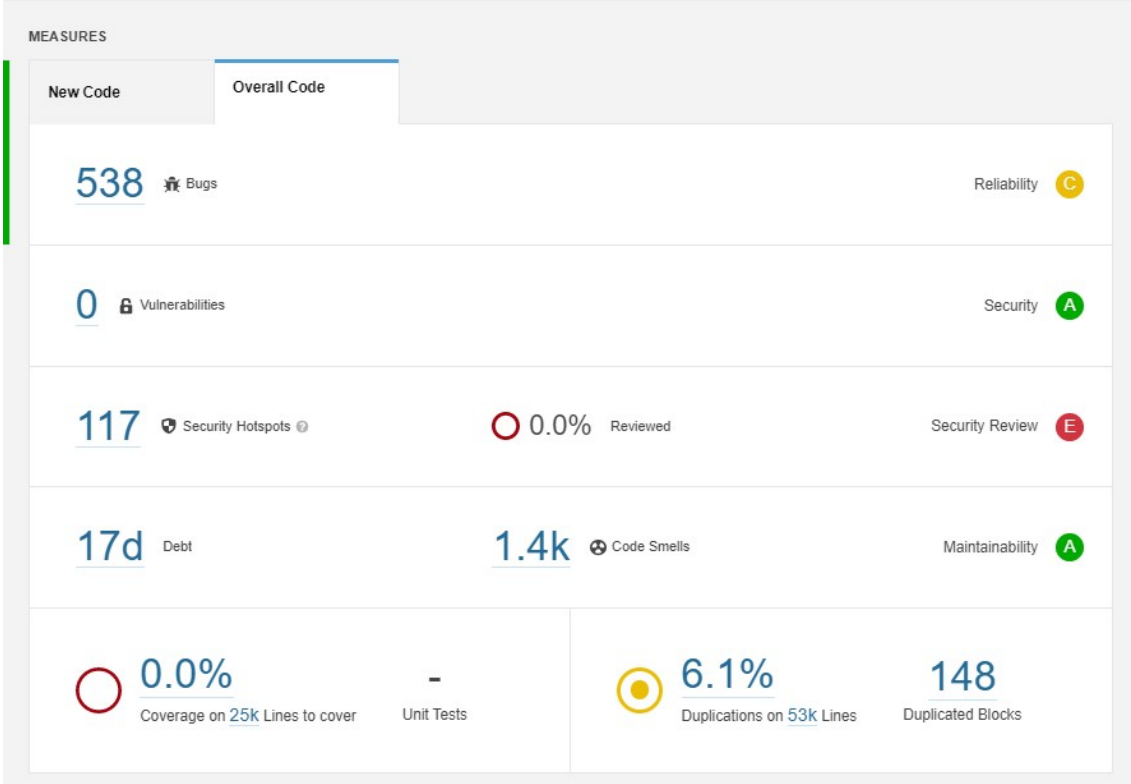


Figure 3.16: SonarQube analysis

debt calculation, as it is dependent upon the specific architecture of ArduPilot. The latter caters for a large variety of board models which may be rather similar requiring a somewhat similar code and common modes.

3.4.5.2 Unit Size and Complexity

Relates to the distribution of the number of lines of code over the units of the source code, and to the complexity of the system. Using SIG, Unit size is given a score of 1.7/5. This makes the code harder to understand and more difficult for developers to identify issues or areas of the code they need to modify, leading to a moderate to high debt. As for the complexity, analyzing the McCabe index from the Files view of SIG, we notice that nearly 25% of the objects have a very complex architecture, making it harder to test and maintain.

3.4.5.3 Module Coupling and Dependency Analysis

Quantifies the chance that a change of the code in one place propagates to other places in the system. This negatively impacts the modifiability of the source code, along with its modularity and testability. Major issues appear to be in the ArduCopter `mode.cpp`, the libraries files, and the `system.cpp` class of ArduSub and AntennaTracker.

3.4.5.4 Testing Coverage

In the latest LCOV report, about 36% of the functions and 45% of the lines were not covered in the tests[58]. This suggests incomplete test coverage, although key files like `Copter.cpp` had a proper coverage. Incomplete coverage risks the quality of the project because uncovered code can break behavior and cause regression issues without the error sources being properly identified.

3.4.5.5 Therefore...

Based on the above insights we got from SIG, and on our analysis, we notice that most of the issues originate from the libraries which are being called and used by the different vehicles and interfaces, and are probably accounting for most of the technical debt in the system. The major debt seems to be specifically originating from large unit sizes and complex modules.

3.5 Standardize, Abstract, Overcome: Exploring Variability in ArduPilot

For our last post on ArduPilot, we will be diving into the variability found in the project. We examine aspects like the architectural decisions such as hardware abstraction. A more in-depth analysis of the architecture of ArduPilot can be found in our second post.

Variability in a software allows having different features to support different needs, like different environments or complexity levels. It can be expressed at several points along the development, such as offering different versions of the software or different configurations within it[59].

[58] https://firmware.ardupilot.org/coverage/index.html
[59] https://se.ewi.tudelft.nl/delftswa/2020/slides/tudelft-architecture-spl2020.pdf

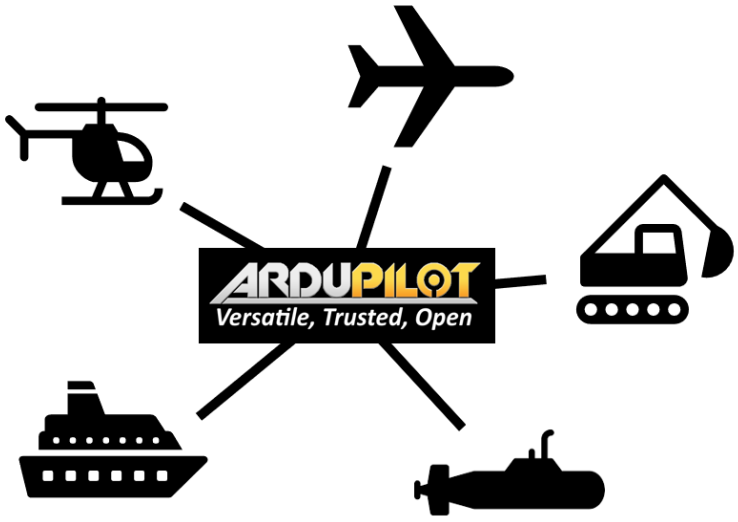


Figure 3.17: ArduPilot supports multiple vehicle types

3.5.1 Variability in ArduPilot

Let us first explore what kind of variability exists in ArduPilot, by looking at which features vary and then summarizing it in a feature model. Looking back to our analysis of its vision and features in our first post, we see that variability is central to ArduPilot because one of its principles is being versatile, which translates to supporting variability to meet anyone's needs.

3.5.1.1 Which Features Vary and Who Benefits from It?

Immediately we see how ArduPilot features various vehicle types (copters, planes, rovers, boats, submarines) and supports many platforms (boards and operating systems). This provides end users the benefit of accessibility: everyone can use ArduPilot for essentially any vehicle project, using the board(s) of their preference. The inclusiveness is enriching also to developers in the drone business who seek to promote their hardware.

ArduPilot also exhibits variability by accommodating different sensor types using libraries, and any user interface program (or "ground station") that is compatible via MAVLink (an open source drone communication software project with its own protocols). This high flexibility caters to a wide range of end users, from hobbyists seeking a simple control interface, to researchers wanting to do specific measurements in water or in air, or to aerospace enterprises testing out new vehicle concepts. Everyone can use the interface and sensors that suit their needs. Meanwhile developers benefit from not having to maintain a single complex interface that needs to please everyone.

Given that the support for each individual hardware and software option is a feature, we see that ArduPilot has many variable features, which we demonstrate in our model.

3.5.1.2 Is Any Feature Combination Possible?

ArduPilot's extreme versatility may seem utopic at this point, but it is of course not possible to use every single feature at once. For example, helicopter firmware and applications will not work well for planes or the such, hence there being different codebases for different vehicles. As for subtler incompatibilities, due to the codebases developing independently, one vehicle's firmware may support a new board while the other firmware will not have it until its next release[60]. Nevertheless, thanks to ArduPilot's design meant to function like an abstract microcontroller[61], a vehicle designer can choose virtually any sensor or actuator combination on any vehicle.

3.5.1.3 How do we Summarize the Variability?

Taking our architectural analysis of ArduPilot as a basis, and by consulting relevant documentation pages (overview[62], sensors[63], boards[64], I/O[65]) and GitHub directories[66], we can visually express the variability found in ArduPilot with a "feature model", as shown below. However, ArduPilot's variability is present not only in its own implemented code, but also in the system around it. Hence we have the second feature model to depict variability in an ArduPilot system.

We used Feature IDE to make the models. To elaborate on the legends, only one feature from an "alternative group" can be selected. "Concrete features" refer to features directly implemented in the code while "abstract features" are not implemented as themselves, but are collective names to define groups of implemented features[67].

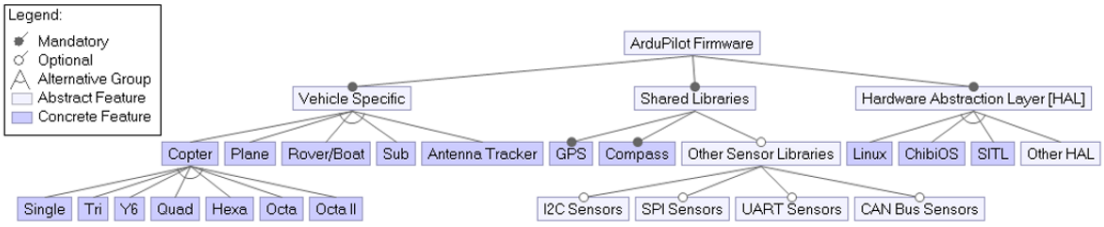


Figure 3.18: Feature Model of ArduPilot Firmware

3.5.2 Variability Management

Variability within ArduPilot is manageable via the detailed documentation available for each of the main stakeholders. Throughout this section, we gathered the sources of information relevant to each of the stakeholders identified in our first post, mainly: the business owners, developers, and users of ArduPilot. These sources can thus be utilized by them to run different configurations/products and adequately update ArduPilot.

[60] https://discuss.ardupilot.org/t/pixhawk-4-incompatible/33454
[61] https://ardupilot.org/copter/docs/common-flight-controller-io.html
[62] https://ardupilot.org/ardupilot/index.html
[63] https://ardupilot.org/dev/docs/code-overview-sensor-drivers.html
[64] https://ardupilot.org/copter/docs/common-choosing-a-flight-controller.html
[65] https://ardupilot.org/copter/docs/common-flight-controller-io.html
[66] https://github.com/ArduPilot/ardupilot/tree/master/libraries
[67] https://github.com/FeatureIDE/FeatureIDE/wiki/Feature-Diagram#abstract-feature

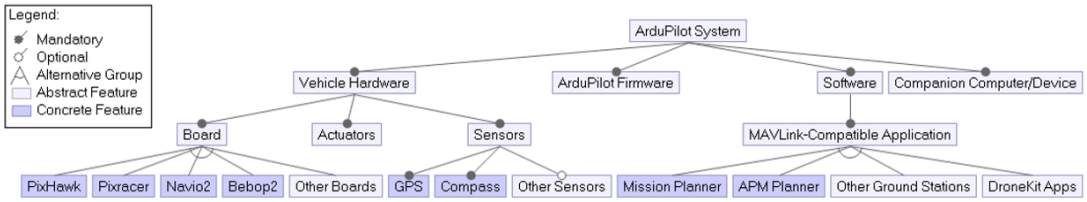


Figure 3.19: Feature Model for an ArduPilot System

1. The business owners of ArduPilot properly manage variability by having access to various information such as financial metrics and a platform to overview proposals for new functionalities. This platform acts a direct line of communication with the ArduPilot domain experts, and is used for adding new features, or update old ones. Below is an example of a discussion between one of the domain experts of ArduPilot and a business owner, in which the latter is assessing the funding needed for a new capability[68].

2. ArduPilot's developers have access to a multitude of documents relating to understanding and contributing to the code, testing, and debugging. Updating the code and adding new functionalities is thus made easier via the "Learning the code"[69] section, which provides the developers with an introduction into the different codes of the various vehicles: adding parameters, adding new modes for flying or driving, a view of the libraries, etc. Additionally, ArduPilot has a strong developer community which communicates via weekly development calls, chats or by asking questions on Gitter or Mumble[70] [71]. In order to decrease the complexity of variability management for the developers, ArduPilot offers an Autotest Framework[72] (based on ArduPilot's SITL architecture) which allows for the creation of repeatable tests to prevent regressions in ArduPilot's behaviour:

- It shortens the time spent for running the same scenario over and over again, and therefore renders the development process more efficient
- It allows to repeatedly replicate bad behaviour in ArduPilot, and assigns the specific tasks to each developer to handle it ("test-driven-development")
- It reduces the risks of a regression in ArduPilot's behaviour

3. Coming to ArduPilot's end-users, the latter have access to a thorough documentation describing the available features to be deployed, along with tutorials on how to correctly run ArduPilot on a vehicle, and leverage the tools available to control and monitor it. This documentation includes a detailed review of the different vehicles and tools available. Additionally, ArduPilot gives access to tutorials for running a project from start (setting up, flying/ driving, configuring the parameters, going through the logs, etc.) to end. Aiming to ease this variability management from a user's point of view, ArduPilot thus offers a "First Time Setup and Configuration"[73] for the different vehicles' implementation, explaining critical aspects as relating to installing the ground station software, running the autopilot, loading the firmware on the boards, connecting mission planner to Autopilot and finally configuring

[68] https://discuss.ardupilot.org/t/dronekit-python-rescue-project/48505
[69] https://ardupilot.org/dev/docs/learning-the-ardupilot-codebase.html
[70] https://gitter.im/ArduPilot/ardupilot
[71] https://ardupilot.org/dev/docs/ardupilot-mumble-server.html
[72] https://ardupilot.org/dev/docs/the-ardupilot-autotest-framework.html
[73] https://ardupilot.org/rover/docs/apmrover-setup.html

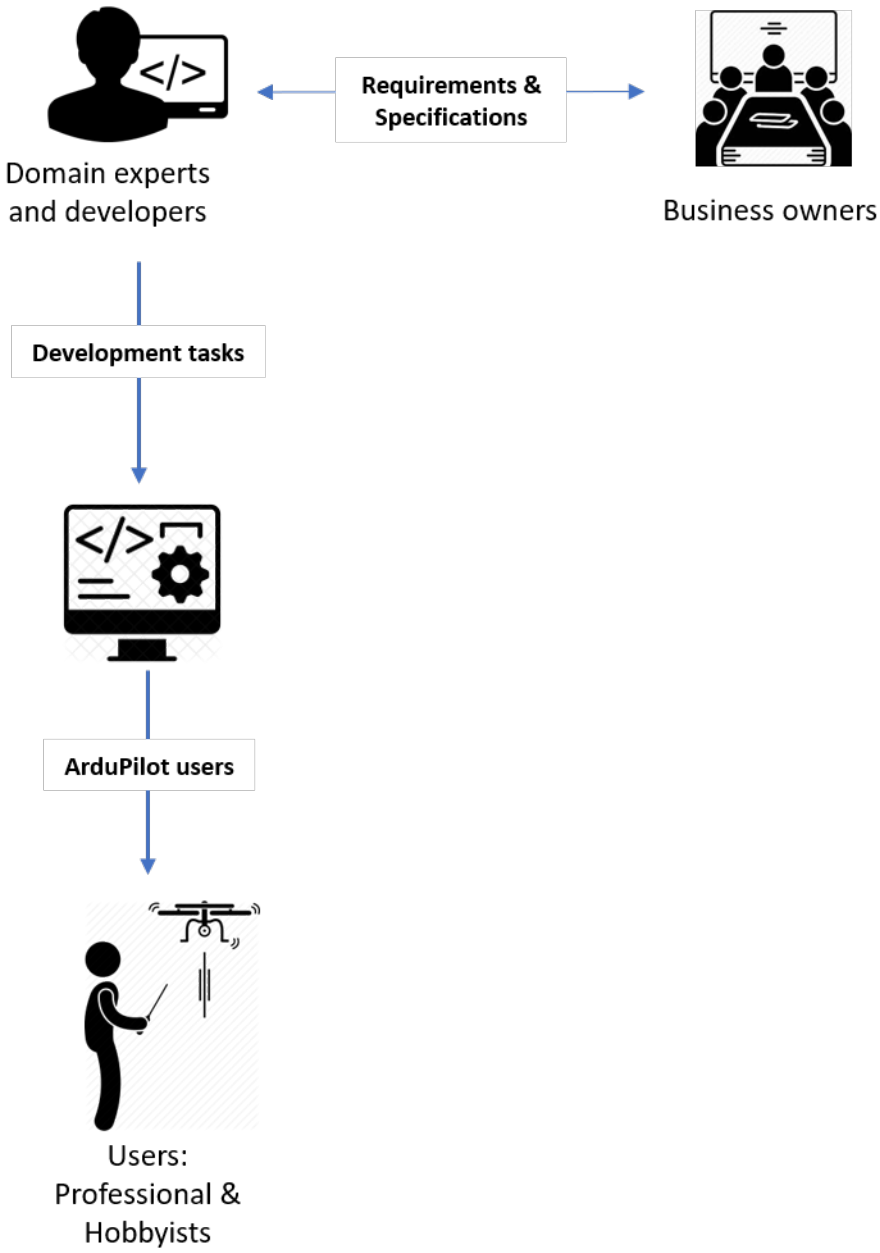


Figure 3.20: ArduPilot main stakeholders as identified in essay 1

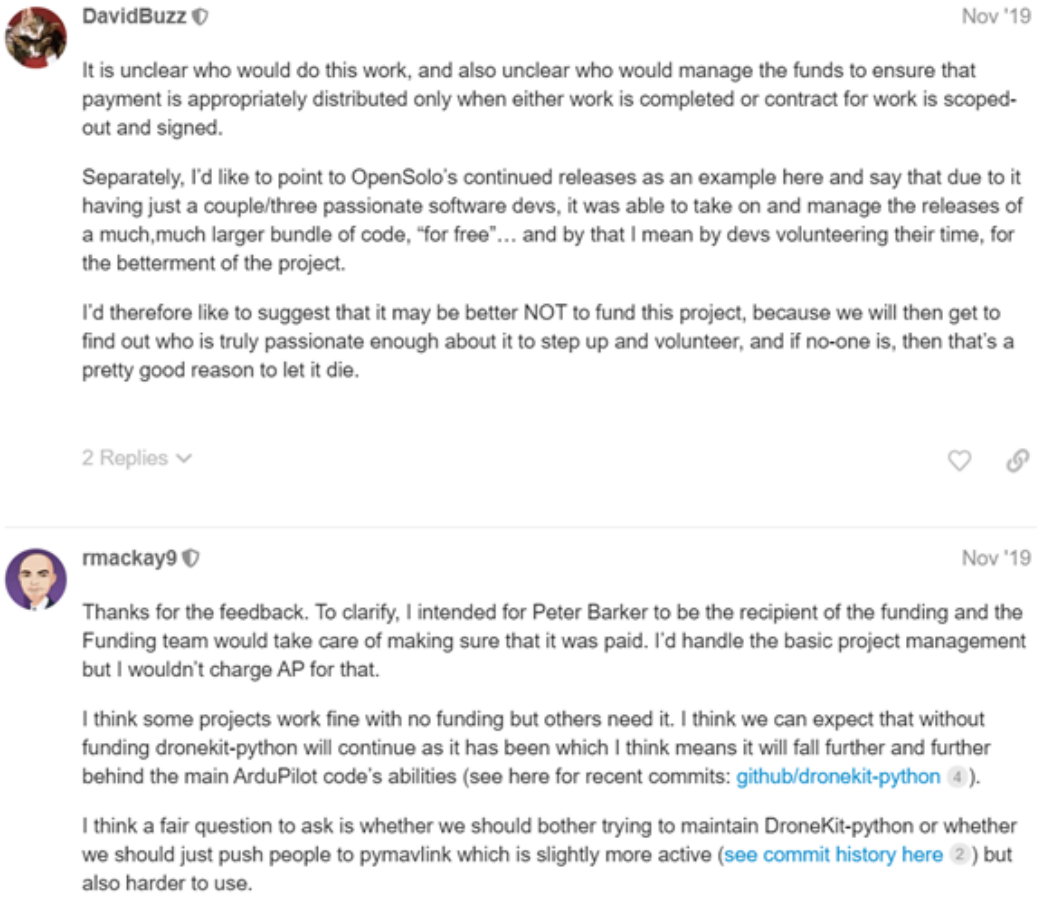


Figure 3.21: DroneKit-Python Rescue Project: interaction between a domain expert and a business admin

the overall system. A tutorial on the users' "first drive"[74] or "first flight" is also available depending on the vehicle. Furthermore, for Windows users using the ground station Mission Planner, graphical interfaces help to set up and configure vehicles[75].

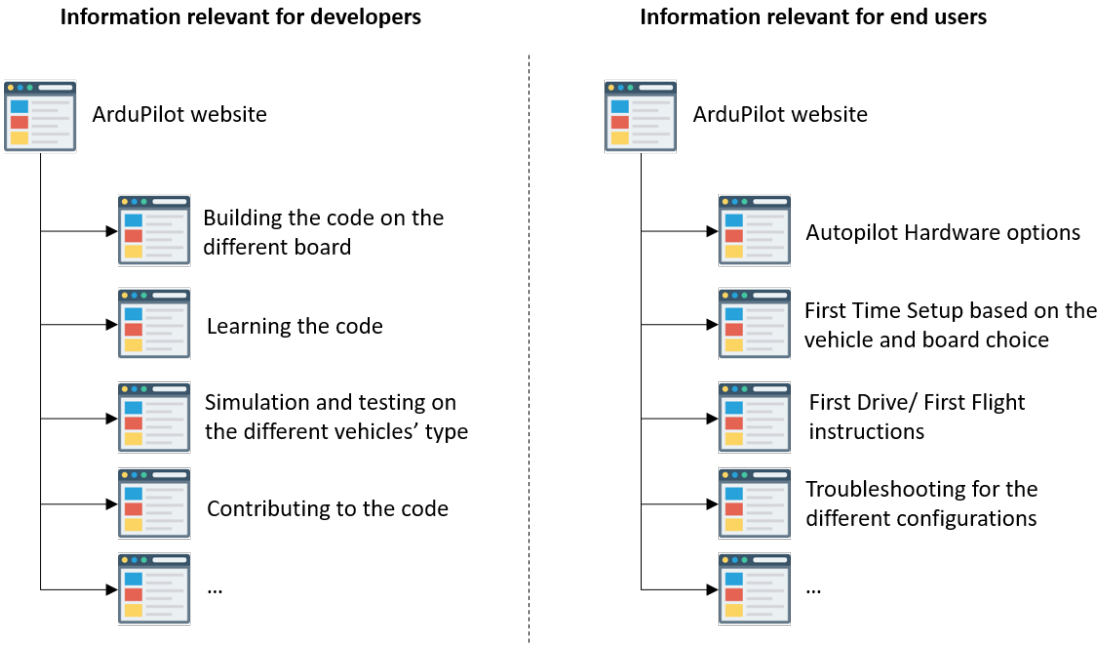


Figure 3.22: Main sources of information available for the ArduPilot's developers and end users

3.5.3 ArduPilot's Way of Implementing Variability

Having the different options within ArduPilot is cool, but it has to be implemented in some way or another. The most notable variability is the different vehicles it supports. As known, ArduPilot offers autopilot for various vehicle types, which means they have to deal with some of the vehicle specific features: a rover rides over land and a copter uses its rotor to fly. Another variability we will talk about are the operating systems ArduPilot support and how that is implemented.

3.5.3.1 Variability in Hardware: Vehicles

The users can utilize ArduPilot for their copter, or they may choose to use a plane, or any other supported vehicle type. Depending on the chosen vehicle, the user has to simply build the code using a specific command for the different vehicles. ArduPilot recommends users to use the build automation tool Waf to ease this building process[76] [77], which also serves as the mechanism for variability. Based on the command, Waf will compile the files specific for that vehicle. This enables the user to build different versions using the build environment mechanism. To keep everything clean, ArduPilot's codebase has different folders

[74] https://ardupilot.org/rover/docs/rover-first-drive.html
[75] https://ardupilot.org/planner/docs/common-loading-firmware-onto-pixhawk.html
[76] https://ardupilot.org/dev/docs/building-the-code.html
[77] https://waf.io/

for each vehicle type that only contain the code for that specific vehicle. This includes features that may be commonly shared among the vehicles, think of radio contact with the ground control. This results in very similar functions such as `do_aux_function_change_mode` to appear in all vehicle codes. ArduPilot most likely chose to do this to ensure consistency, as some functions related to the radio contact are vehicle specific (think of initialization). The number of duplicate functions is not high to warrant a shared library for it, which may make things more complicated.

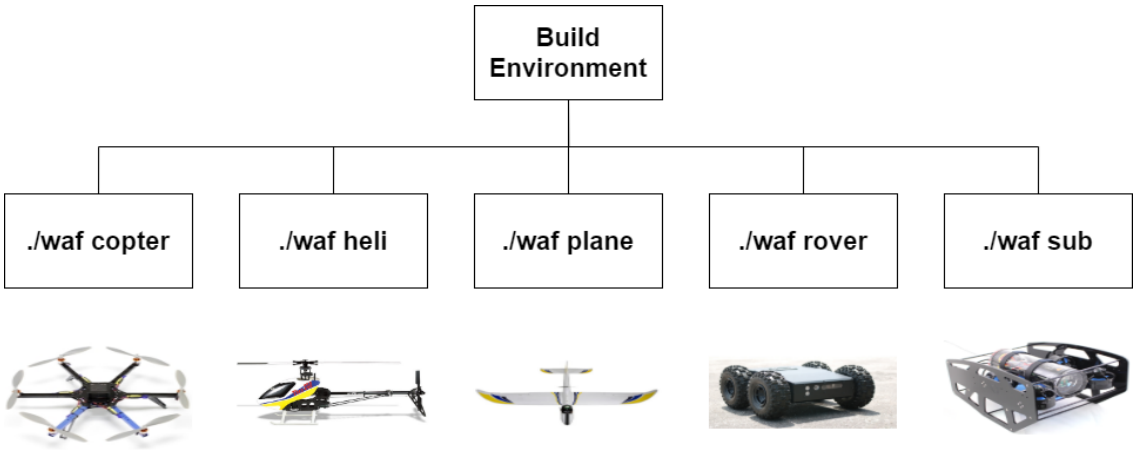


Figure 3.23: Different ways to build the code for various vehicles

As the vehicle code is chosen during the building phase, we can say that the variability is bound at compile-time. After building the code, it is not possible to change anything without recompiling the code again.

If in the future a new vehicle type is added, ArduPilot would create a new folder that will be dedicated to that vehicle type. When building the code, only the specific code in that folder will be used.

3.5.3.2 Variability in Platform: Operating Systems

ArduPilot mainly supports two operating systems. These are Linux and ChibiOS. ChibiOS can be used for boards that uses the STM32 microcontroller in order to ensure real-time actions[78]. Linux can be used for most other boards, including the STM32[79]. Choosing which OS you want to use is done similar to choosing which vehicle code is used. Shown in this figure below are some of the commands to select the OS wanted on the board.

The option for OS is bound after building this code. Therefore, the variability in OS is compile-time binding as well. If more operating systems are going to be supported in the future, ArduPilot will most likely implement it by adding a new folder to the libraries folder. Currently, both for Linux and ChibiOS, a folder exists called `AP_HAL_Linux` and `AP_HAL_ChibiOS`. In here, the hardware abstraction layer (HAL) is present. Adding support for another OS means that a HAL needs to be created for that specific OS that can act as the middleman between the different libraries and the external hardware (more info about the layers is found in our second post).

[78] http://chibios.org/dokuwiki/doku.php
[79] https://ardupilot.org/copter/docs/common-autopilots.html

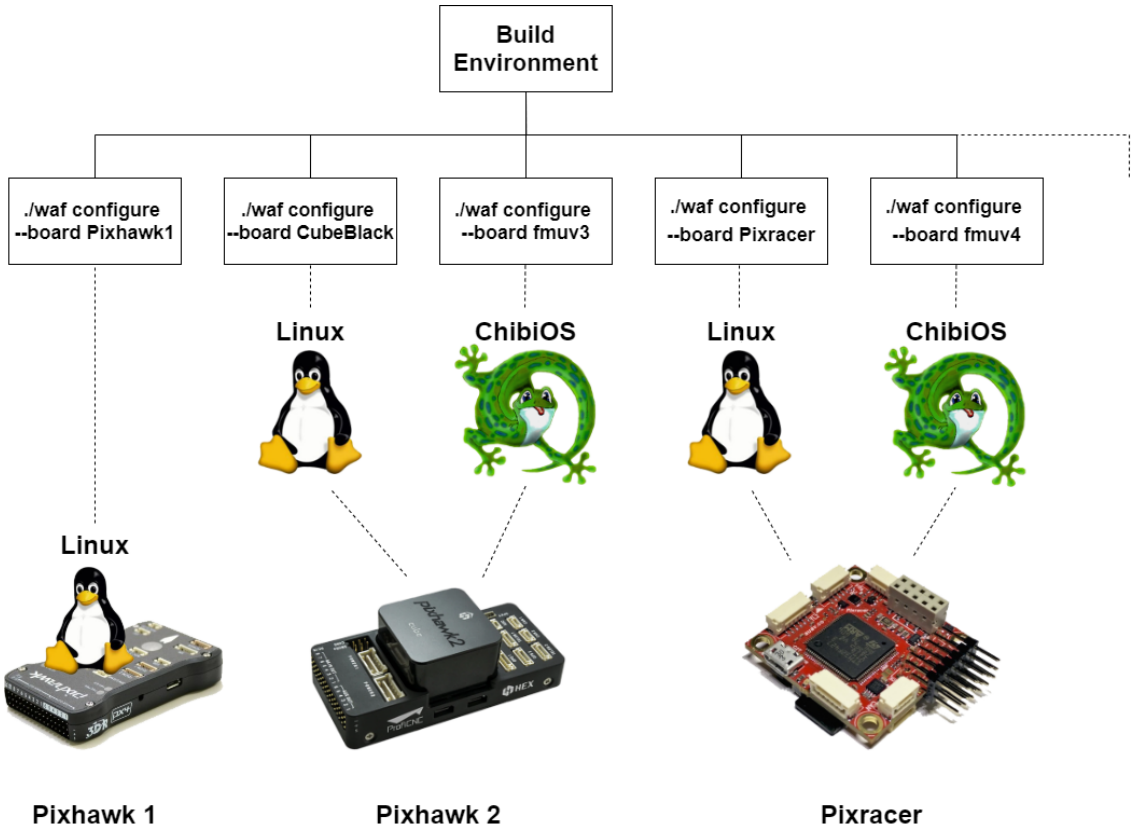


Figure 3.24: Some boards only support one OS, while others support multiple

3.5.4 Assessing ArduPilot's Variability Decisions

The way ArduPilot implemented the variability is well thought out. The way their architecture is built based on the 3 layers (vehicle code, shared libraries and HAL) makes it easy to extend one of the layers, without changing any of the 2 other layers. Structuring the different layers in different folders gives a good overview and it is easy for users to contribute to specific parts of the code. Maintaining a good overview is important for the contributors, because no one likes spaghetticode, right? When one of the layers is expanded, the compiler needs to know where the new files are located. The build environment needs to be adjusted to add a new command and the files that needs to be build[80]. Luckily, this is not hard to scale. All in all, ArduPilot seems to have thought about scalability when new features are added, which is very important with rapid developments.

The overall variability is managed through the detailed documentation[81] and the weekly developer meetings[82]. However, this only really covers the variability from the developers' side. Currently, the Mission Planner includes a tool that can load the firmware on the board for ArduRover, ArduPlane and ArduCopter[83]. Unfortunately, the Mission Planner only works for Windows. We would recommended ArduPilot to expand this tool for other operating systems such as Linux and macOS, and to include loading firmware for ArduSub.

As for developers, it would be nice to have a feature model given in the documentation as we have presented earlier in this post. It will help, especially the contributors that are new to ArduPilot, to have a clear understanding of all dependencies and how they are grouped.

[80] https://ardupilot.org/dev/docs/building-the-code.html
[81] https://ardupilot.org/ardupilot/index.html
[82] https://ardupilot.org/dev/docs/ardupilot-mumble-server.html
[83] https://ardupilot.org/planner/docs/mission-planner-overview.html

Chapter 4

Blender



Launched in 1994 and headquartered in Amsterdam, Blender is free and open source 3D computer graphics software, with the aim of supporting the entirety of the 3D pipeline. With Blender, users can model, sculpt, texture, and shade 3D objects, or meshes — vertices connected by edges. These objects can then be rendered to a still image, or rigged and animated for export to a video file. The render results can be manipulated further in Blender's image and video editors. Besides the three-dimensional space, Blender also supports 2D drawing and animation with its grease pencil tool. On top of this, Blender sports a Python scripting API. Blender itself is written in C, C++, and Python.

Blender founder Ton Roosendaal launched his own animation studio in 1989, called "NeoGeo" and which mainly focused on product visualization. According to Roosendaal, they could afford only one computer for animation work, but animation software costed another 30,000 guilders. He therefore decided to write his own in-house 3D tool during the evening hours, which eventually grew to become Blender. In 2002, after trying to market Blender, his investors got cold feet, and Roosendaal had to file for bankruptcy. To prevent his software — for which he naturally cared deeply at that point — from being taken hostage by the investors, Ton founded the Blender Foundation: through a "Free Blender" campaign, €100,000 were raised to continue Blender as an open source, community-driven project. The campaign succeeded and since then Blender is maintained under the GNU General Public License. It has grown immensely in popularity, with an estimated 1 to 3 million Blender users.[1]

In 2007, the Blender Foundation established an office, the Blender Institute.[2] Through the institute, the goals

[1] Blender, Blender by the Numbers. https://www.blender.org/press/blender-by-the-numbers-2019/. Last accessed on 2019-03-01.
[2] Blender, History. https://www.blender.org/foundation/history/. Last accessed on 2019-03-04.

of the Blender Foundation can be coordinated efficiently and core Blender developers are brought together. Every 1-2 years, an "open [creative] project", usually a short film (though a feature film is planned), is created by the Blender Institute to showcase Blender's abilities and stimulate its adaptation. At this time, 18 people are in full-time employment at the Blender Institute.

4.1 Team

The team consists of:

- Bertan Konuralp
- Jaap Heijligers
- Jason Xie
- Emiel Bos

4.2 Blender in Perspective

Blender aims to "build a free and open source complete 3D creation pipeline for artists and small teams, by publicly managed projects on blender.org".[3] That is, Blender wants to offer functionalies of the entire 3D creation pipeline. This "3D creation pipeline" refers to all the stages towards 3D pre-rendered or real-time imagery: modeling and/or sculpting, rigging, animation, simulation, rendering, compositing, motion tracking, and even video editing.[4]

In order to achieve this vision, Blender's development team is constantly working together with its contributors from all over the world to improve and meet the needs of its users, as well as catch up with its competitors. Towards this end goal, one recent major change was the UI overhaul in update 2.8, released in July 2019. Before the update, the UI of Blender used to be its most salient source of criticism, and this revamp has arguably brought Blender on par with its $1000-a-year-costing competitors[5] with regards to usability. Personally, we anticipate Blender to become the industry standard at some point, especially given its current rapid development momentum. The 2.8 update also included a brand new real-time rendering engine, called EEVEE. This engine complements Blender's older Cycles engine, a physically-based — which can more or less be interpreted as meaning "photorealistic" — path tracer. Where Cycles is slower but more realistic, EEVEE is much faster and more akin to a game engine renderer, in that it uses rasterization rather than path tracing. Another noteworthy function of Blender is its grease pencil tool, which allows for 2D drawing and animation. Since the 2.8 update, two more updates have released, each with a host of new features and improvements. This brings the current version to 2.82, as of writing.

We feel Blender to be a tremendous project, centered around the core principles of enabling creativity and sharing those creations with as little involvement of monetary considerations as possible. Blender enables small animation studios, game developers or artists to save thousands of dollars a year, while still having the functionality and ease-of-use they desire, with no strings attached.

Blender's most distinguishing property is that Blender is free for anyone to use. The license they use is GNU GPL v2.[6] They also wrote a blog post about Blender's freedom[7]. This means:

[3] Blender, About. https://www.blender.org/about/. Last accessed on 2020-02-20.
[4] Blender used to have its own game engine, but this was dropped as per Blender's 2.8 update.
[5] Autodesk Maya 2020 subscription. https://www.autodesk.com/products/maya/subscribe. Last accessed 2020-03-07.
[6] Blender, License. https://www.blender.org/about/license/
[7] Blender, Blender is Free Software. https://code.blender.org/2019/06/blender-is-free-software/

- Blender is free to use
- You can study the source code of Blender and modify it
- You can freely redistribute Blender and its modified versions
- Anything created using Blender remains the property of the creator
- Blender add-ons are also free to be shared

The rest of this essay will focus on putting Blender in perspective by analysing it from different angles. Firstly, it is important for people who are not familiar at all with Blender to see it from the view of an end user.

4.2.1 End users' mental model

3D computer graphics modelers are among the more complex applications, and booting up Blender for the first time can be a little daunting. For Blender, the user is met with the overhauled UI from update 2.8, which puts a lot more emphasis on intuitiveness and ease-of-use than before.

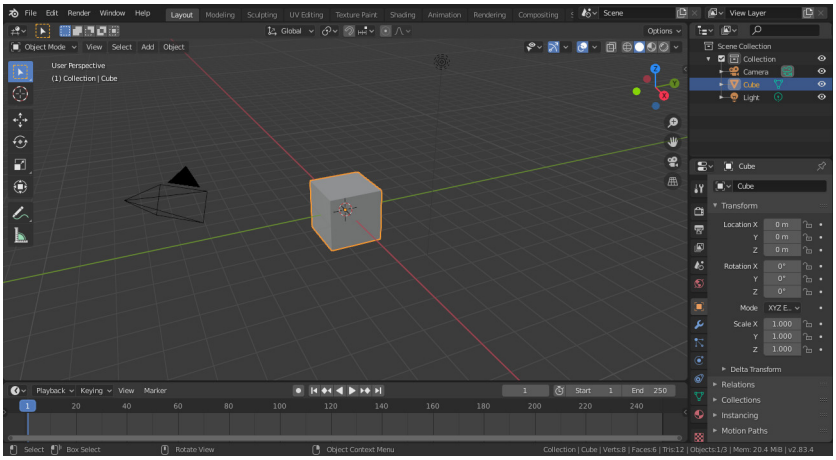


Figure 4.1: Blender's first interface on startup.

Analysing the system through the mental model of an end user entails segregating a project into its form; what the system is, as seen from the perspective of domain experts or managers, and its function; what the system does, its functionality as experienced by end users.

Considering the *form* of Blender, a first glance at the interface makes it clear that Blender concerns itself with three-dimensional space, as the majority of the screen consists of its 3D viewport, on startup at least. This suggests that Blender is about manipulating (objects in) this 3D space, and in a grander sense, about creating digital worlds and art. The many ways of how this manipulation is possible — the *functionality* of Blender — is outlined in the top bar:

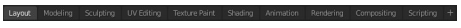


Figure 4.2: The top bar of Blender's interface, where the user can switch between workspaces.

This bar shows the different workspaces that Blender supports. A workspace is essentially a particular organization of the interface, suited for a specific task, or a specific part of the 3D pipeline. **Layout** facilitates

the composition of the scene, **Modeling** facilitates the manipulation of the objects — which is done by changing their mesh; **Sculpting** allows for manipulating meshes in a different, more organic way, which is not unlike physical sculpting; **UV Editing** is meant for creating and/or changing the UV maps, which determine how textures are applied to an object/mesh; **Texture Paint** allows painting on an object directly, rather than mapping images; **Shading** uses a graph-based node editor for changing the material — or the physical appearance — of objects; **Animation**'s workspace is rather self-explanatory; the **Rendering** workspace shows (intermediate) output of the render engine used; **Compositing** allows for image filtering and editing of the render result; and **Scripting** facilitates the use of Blender's Python API. The **+** button on the right supports even more workspaces: **Rendering**, **2D Animation**, **2D Full Canvas**, **Masking**, **Motion Tracking**, and **Video Editing**. It is clear that pretty much all aspects of 3D creative work is supported by Blender. They seem to try and make sure that the user never has to touch other software in his or her 3D creative work again.

Ultimately, Blender's form and function are pretty intertwined in the same way Blender's customers and end-users largely overlap. Blender doesn't get licensed by the management of big companies — concerned with the form — after which it is used by a totally different group of employees — concerned with the function. The people that "buy" Blender (i.e. download for free) are mostly the people that use it.

4.2.2 Stakeholders

Being more familiar with Blender, this section aims to identify Blender's stakeholders; all the people or businesses surrounding the Blender project and consider the projects value stream. The value stream of Blender is relatively straightforward, with a small core development team and a number of volunteer developers contributing to the software, which can then be used by the end user and/or customer (which in Blender's case are more or less synonymous). We'll treat every respective stakeholder area — as defined in Lean Architecture[8] — in turn.

4.2.2.1 Developers

Blender is developed and maintained by the Blender Foundation, which is managed by its chairman and "benevolent-dictator-for-life" Ton Roosendaal. He oversees everything to do with Blender. Most of the Blender team are its developers. For the management, there are two development coordinators; Dalai Felinto and Nathan Letwory, and a lead architect; Brecht van Lommel. There are also admins who take decisions when no agreement can be reached among contributors. The rest of the development team is divided into so-called "modules". Each module focuses on a part of the Blender software and is assigned an owner.

The owners' roles are to contribute actively to the project, decide over implementation and design issues, approve/reject requests and patches, fix bugs, test for release, document new features/changes and communicate with other module owners to discuss overlapping work. Then there are developers who contribute to the code on a regular basis. They are available for helping with bug fixing, reviews and documentation. They can also implement features, review contributions, listen to user reports and handle bug reports. The last position within modules are the rest of the members who can be developers or non-developers. Non-developers are usually artists who represent other stakeholders. All module members make sure the stakeholders agree with features and development plans. Developers also spend two days each week on fixing open issues through the Tracker Curfew.[9]

[8] Jim Coplien, Gertrud Bjørnvig. Lean Architecture for Agile Software Development. Wiley, 2010.
[9] Blender, Tracker Curfew https://code.blender.org/2019/12/tracker-curfew/

Modules	Owner
Development Management	-
User Interface	William Reynish
Modelling	Campbell Barton
Sculpt, Paint, Texture	Jeroen Bakker
Animation & Rigging	Sybren Stüvel
Grease Pencil	Antonio Vazquez
Eevee & Viewport	Clément Foucault
Render & Cycles	Brecht Van Lommel
Data, Assets & I/O	Bastien Montagne
Python & Add-ons	Campbell Barton
VFX & Video	Sergey Sharybin
Nodes & Physics	Sebastián Barschkis
Platforms, Builds & Tests	-

Figure 4.3: Blender modules and their respective owners

4.2.2.2 Business

Contrary to the development of Blender, Blender Foundation has a very small management team which is kept more private. However, information about their funding is shown in the diagram below.

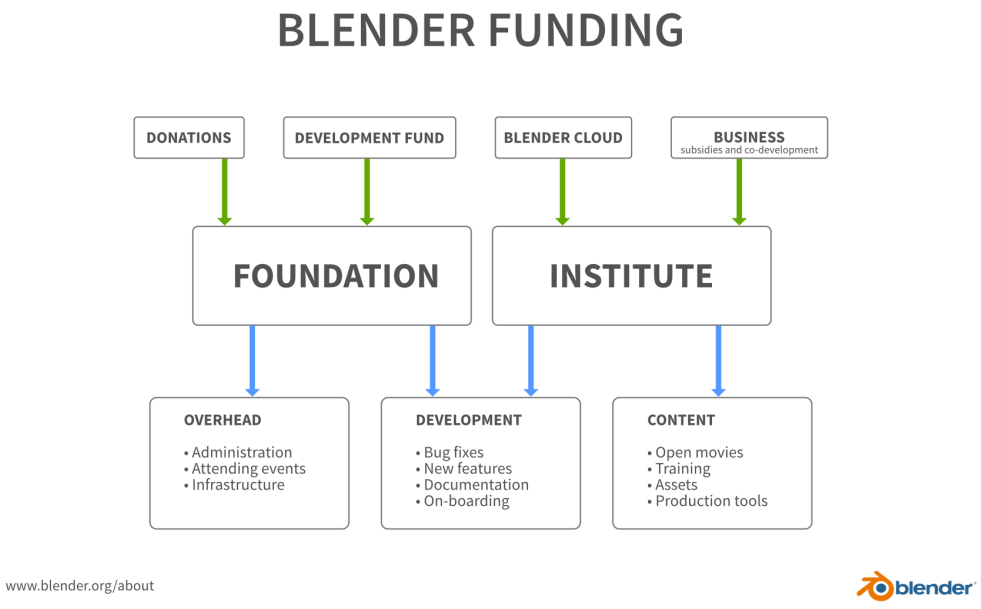


Figure 4.4: Blender financial and organizational structure.[10] Shared under CC BY-SA 3.0.

The Blender Foundation generates income through donations and through their Development Fund. The Development Fund is set up so that companies and individuals that use Blender can sponsor the project through monthly contributions. 3814 individuals and 35 corporates support Blender at the time of writing. Notable

[10] Blender, About. https://www.blender.org/about/. Last accessed on 2020-02-20.



corporates include:

The Blender Institute, on the other hand, generates income through their products: Blender Cloud (5,500 subscribers as of 11/2019, $9.90 pp/month [11]), and the Blender Store which sells various merchandise and books. All the funding contribute to the value stream of the business and business members.

4.2.2.3 End users/customers

This area encapsulates all the organizations and people that use Blender. These mainly consist of, but are certainly not limited to, small artists or creative companies. There are also YouTube channels devoted to Blender, the largest of which is Blender Guru [12] (we highly recommend this channel for its intuitive and engaging tutorials on Blender). The value stream for the end users/customers is pretty obvious; they get to benefit from using Blender for no cost whatsoever.

4.2.2.4 Domain experts

Most developers working on Blender, especially higher up in the hierarchy, are domain experts in some field of 3D rendering or modeling. As we already discussed development, we want to mention some of Blender's competitors. The main competitors for Blender are some other open source software but also many paid industry level 3D software. Some examples include:

[11] YouTube, Ton Roosendaal (Blender.org): "Het faillissement was het beste dat mij ooit had kunnen gebeuren".
[12] YouTube, Blender Guru's. https://www.youtube.com/channel/UCOKHwx1VCdgnxwbjyb9Iu1g. Last accessed on 2020-02-25.



4.2.3 System context

Blender is a relatively isolated and context-independent system. Anyone can simply download, install and use the software, on all major operating systems. Of course, the software could be used to generate assets for use in subsequent productions, e.g. 3D models and animations for use in games or movies, or product design for use in manufacturing, in which case it is part of workflow or business process, but this isn't necessarily so. If it is, then any software or person that uses Blender's output can be considered an external dependency. Blender, in turn, could use textures generated in another program (like Substance[13]), or obtained from the

[13] Substance 3D. https://www.substance3d.com/.

internet, in which case Blender is dependent on such external systems. Additionally, Blender offers a Python API with which add-ons can be made (and numerous have been made). Add-ons can offer a wealth of extra functionality. The scope definition can be summarized by all the steps in the 3D creation pipeline.

In a recent blog post, Blender discussed some big upcoming features for 2020 that are being worked on [14]. These include a new stable, reliable and flexible node based particle system, such a node based system for hair, faster animation playback, multiresolution meshes, and much more.

4.3 Architecting Blender

This essay will cover the software architecture of Blender, i.e. the structure and architectural patterns of Blender's codebase.

4.3.1 Architectural viewpoints

When architecting any software, it is important to look at the architecture from different angles or views, because trying to encapsulate all the viewpoints that every stakeholder has, in one diagram, becomes complicated. Therefore, common practice dictates creating multiple viewpoints, each of which can focus on a different aspect. In Rozanski and Woods' Software Systems Architecture[15], a viewpoint is defined as "a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views."

For the case of Blender, we have chosen to focus on the following views:

- Context viewpoint; aims to expand upon the system context and visually showcases the system scope, external entities, services and dependencies.
- Development viewpoint; is meant for the developers and illustrates the different components of the system as well as the code structure.
- Process viewpoint; focuses on the run time behaviour of Blender including what functions are being invoked and called.
- Deployment, or physical, viewpoint; takes a brief look at how Blender is deployed and built on different operating systems.
- Scenarios, or use cases; describes the use cases for Blender through sequences of interactions between objects and between processes.[16]

4.3.2 Architectural styles

When examining the architecture of Blender, it becomes clear to see that Blender uses a combination of two distinct architectural patterns, the main one being **model-view-controller** (MVC) pattern, even if not very apparant from the large codebase. Blender, by near definition of 3D computer graphics software, needs a GUI (view) for the end-user to be able to interact (controller) with the underlying logic (model). For this to be achieved, the code for each part needs to be, and is, separated in the architecture. When we zoom into the controller and view of Blender, we see that a **domain-driven design** lies underneath, because data are

[14] 2020 – Blender Big Projects https://code.blender.org/2020/01/2020-blender-big-projects/

[15] Nick Rozanski and Eoin Woods, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition. https://www.viewpoints-and-perspectives.info/

[16] Wikipedia, 4+1 architectural view model. https://en.wikipedia.org/wiki/4%2B1_architectural_view_model

divided into specific domains. In update 2.5, context structs were added into the code that clearly defined the data that each type of code can assume to be in its own context. The following figure gives an overview of the different contexts and modules of Blender.

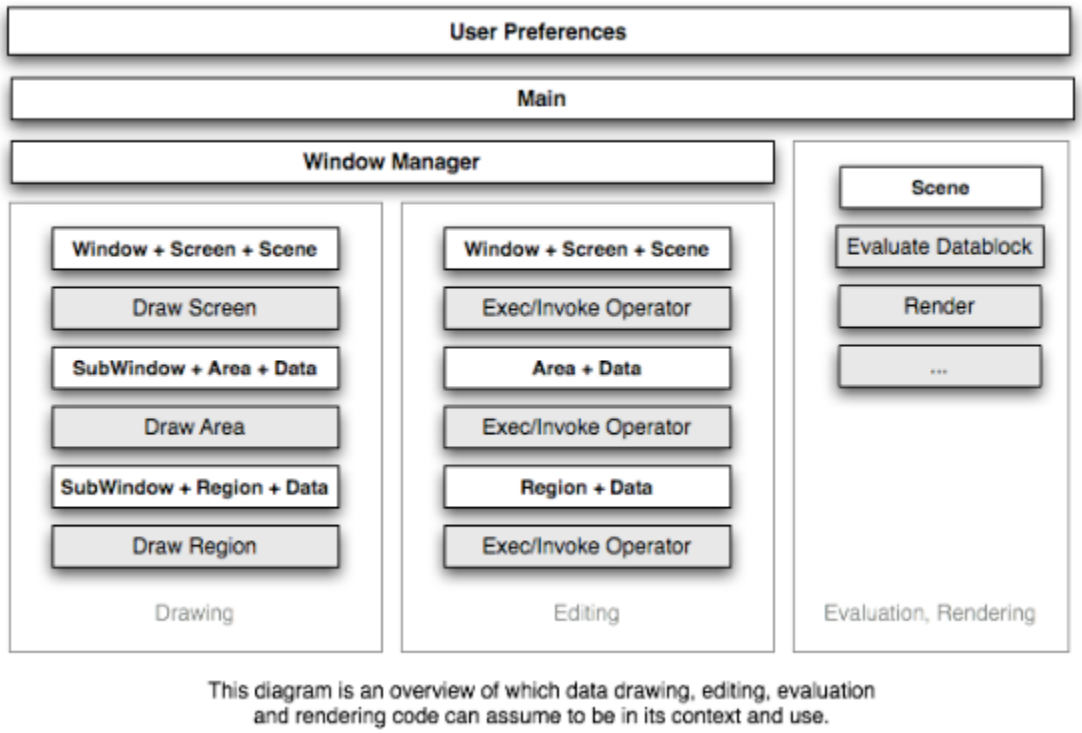


Figure 4.5: Context available for different parts of the code.[17] Shared under CC BY-SA 3.0.

The user preferences that sit on top of everything are the different preferences the end-user sets. The main is/are the `.blend` file(s) that is/are currently open. From there, there is a window manager which encapsulates drawing and editing code, which always works on three levels; screen, area or region, all containing certain data. The evaluation and rendering module should not have access to the three level structs, but should be run in the background. The only thing they can do is pass along a certain scene, for example to render it. The structure for the underlying logic is not very well documented yet. However, it is evident from the fact that the developmemt of Blender is divided into the modules mentioned in the previous blog post that all workspaces have their own module in the code architecture as well.

4.3.3 Context view

We present Blender's context in the following diagram, using what we have learned so far of its contextual scope.

[17] Blender wiki, Contexts. Last accessed 2020-03-19. https://wiki.blender.org/wiki/Source/Architecture/Context

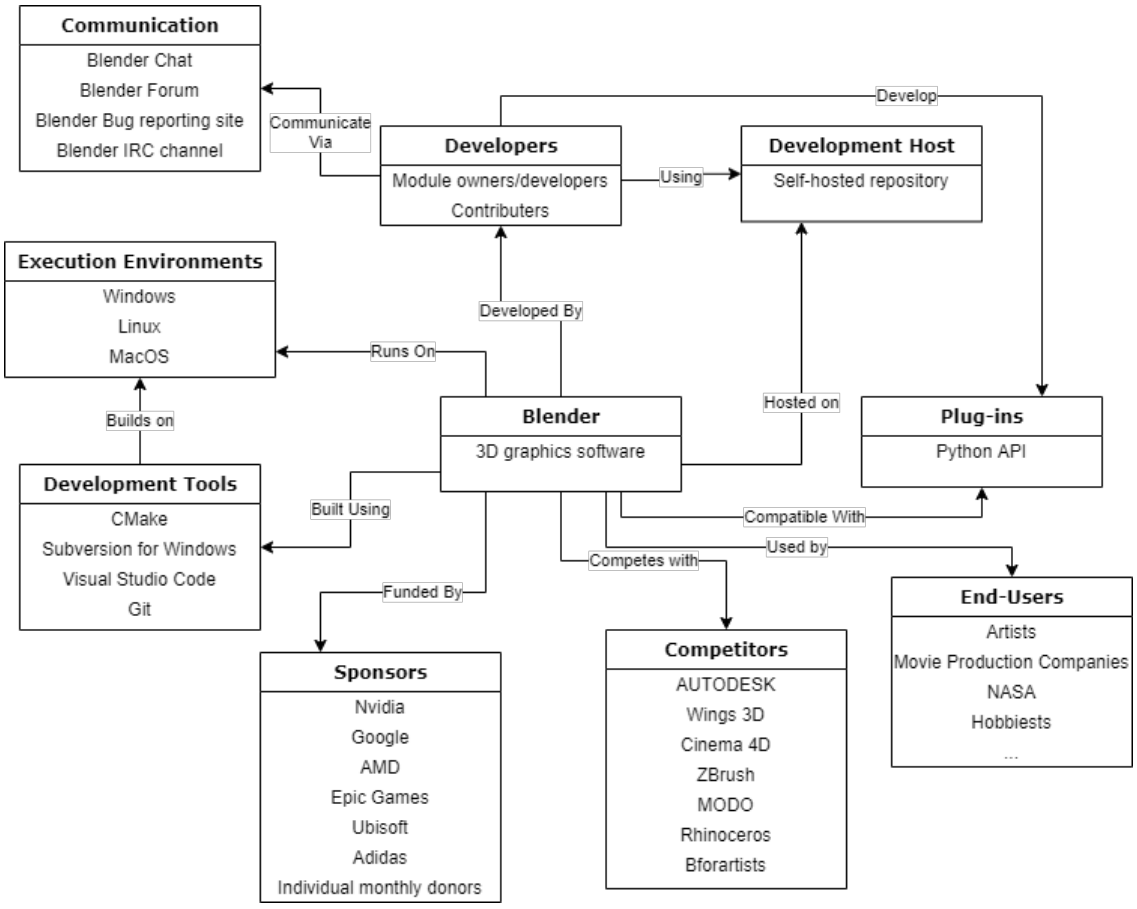


Figure 4.6: Context diagram.

4.3.4 Development view

This viewpoint looks at the architectural structure of the source code. Blender is a huge project, and we will therefore list only the most important components and parts. The full source code can be explored online at [18]. Besides code for document generation, platform-specific release files, tests, configuration files, and the build system, the root folder of Blender's source code contains three salient components: `source`, which holds Blender's main application code; `internal`, containing "source-code maintained by Blender developers but kept as isolated modules/libraries"; and `external`, containing (stand-alone) code libraries imported "because they aren't common system libraries we can rely on the system providing" [19]. A slightly outdated, but still insightful, overview of Blender's source code directory structure is given below:

`source`, in turn, contains two important components: `blender` and `creator`. The former contains the actual application code, while the latter contains the *main()* entry point of the application. The `blender` component consists of many subcomponents. The Blender kernel, i.e. the low-level data structure manipulation and memory allocation code, is located in its `blenkernel` component. `bmesh` comprises a mesh editing API, while `python` comprises the Python scripting API. According to the developer wiki, "most of the interesting code" is in the `editors` component, which contains all the code for Blender's graphical interface and its editors. A curious thing we noted was the fact that the Cycles rendering engine is located in *intern/cycles*, while the newer EEVEE renderer is in *source/blender/draw/engines/eevee*.

It is difficult to discern any kind of coherent architectural patterns from the dependency graph generated by the source code (figure 4). Furthermore, from the high-level ratings provided by SIG, it is apparent that Blender's codebase is rather complex, as the ratings are very low across the board with the exception of code duplication. We did not have access to SIG's Sigrid analysis results of Blender.

[Dependency graph of Blender's source code.]](images/blender/Dependency_graph.jpg)

4.3.5 Process view

This viewpoint analyzes the interoperation between components at run-time. Again, because Blender is a massive program and the numerous components are tightly coupled (see the dependency graph), we will start by treating the core loop of the program. When Blender is launched, it first executes its `main()` function in `source/creator/creator.c`, which initiates subsystems and runs Blender's event loop: `WM_main()`. This window manager continuously (`while(1)`):

- 1. gets events from GHOST (General Handy OS Toolkit), handles window events, and adds these to window-specific queues. GHOST is Blender's own GUI toolkit and windowing system, which handles peripheral input and provides access to the windows. GHOST then draws the GUI directly using OpenGL, so it doesn't rely on any OS-native windowing system. Python scripts constitute the most abstract layer of the Blender GUI chain, which issues the most general commands (e.g. "draw button") to GHOST and receives the most general commands ("the user clicked the button") from GHOST. GHOST replaced GLUT in order to make Blender cross-platform. Events are caused by user input.

- 2. handles window-specific events.

[18] Blender, Blender repository. https://developer.blender.org/diffusion/B/. Last accessed 2020-03-03.
[19] Blender Developer Wiki, Blender source code directories explained. https://wiki.blender.org/wiki/Source/File_Structure. Last accessed 2020-03-03.
[20] Blender Developer Wiki, Blender source code directories explained. https://wiki.blender.org/wiki/Source/File_Structure. Last accessed 2020-03-03.

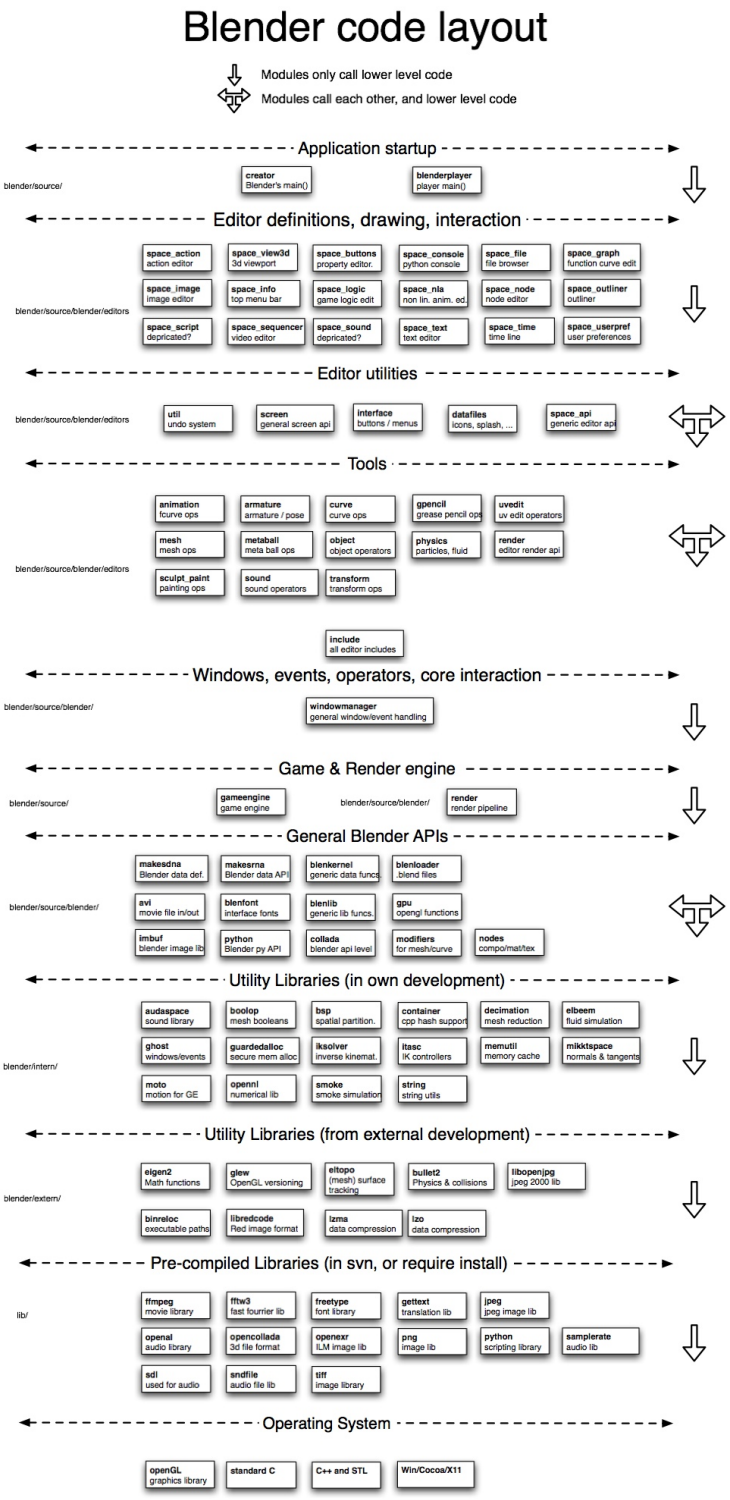


Figure 4.7: An overview of Blender's source code. From [20], shared under CC BY-SA 3.0.

- 3. handles and caches notifications (from other internal Blender systems) that events have left behind about changes
- 4. draws the cached changes to the screen

Like we said in step 1., "the UI layout is defined in Python, but the drawing is done in C, and some of the UI is defined in C too" [21]. To illustrate this, we consider adding a cube mesh. Python code (in `release/scripts/startup/bl_ui/space_view3d.py`) defines a button, `mesh.primitive_cube_add`, for adding a cube. During startup, the function `MESH_OT_primitive_cube_add` registers itself as an operator, with `add_primitive_cube_exec` as its callback function. When a user clicks the Add Cube button, Blender's own RNA library translates `mesh.primitive_cube_add` to the operator type `MESH_OT_primitive_cube_add`, which in turn calls its `add_primitive_cube_exec` function. What's RNA? Blender has two systems that handle data structures: DNA and RNA, names after the genetic nucleic acids. DNA defines how Blender's data — user settings, mesh data, object data, scene data, etc. — is serialized as `.blend` files, Blender's file format for projects. This ensures compatibility with old `.blend` files: a coded description of the layout of each DNA struct is also written to and read from disk, meaning that this layout does not have to stay fixed. RNA then is a "nicer", more abstract interface to DNA [22]. Though not much used in the C/C++, all Python code relies heavily on this data API.

4.3.6 Deployment view

Blender is a large software product with many dependencies, to be deployed to multiple runtime environments: Windows, macOS and Linux. To avoid version mismatching with the runtime's own libraries, all dependencies are packaged along with Blender. Even a Python interpreter is bundled in the final Blender build. Because of this, final package size is about 150MB, but third-party software requirements are eliminated for the runtime platforms.

To create a Blender build, `make package` can be run on each of the supported target platforms. An artifact archive including dependencies is then generated. These artifacts can be uploaded to Blender's website by official developers, such that end users can download and install the application. Official releases can be downloaded[23], as well as daily builds[24] and builds of experimental branches[25] for testing of cutting edge features. The package hosting environment and build bot are maintained by system administrators.

4.3.7 Scenarios view

The scenario we present here are meant to showcase the rendering of a scene to the end-users and the sequences of events that happen in the code. This will give an idea of how the code executes when an end-user is interacting with Blender.

Scenario: End-user switches between workspaces. In order to update the scene efficiently, Blender uses a dependency graph.[26] In short, the dependency graph makes it possible to only update what was dependent on the modified value and to not update anything which was not changed. Each workspace has its own window that the end-user can switch between. These windows own the dependency graphs and re-create it whenever workspaces are switched. The dependency graph holds pointers to the objects in a scene (DNA,

[21] Blender, Blender Dev FAQ. https://wiki.blender.org/wiki/Reference/FAQ. Last accessed 2020-03-14.
[22] Blender, RNA. https://wiki.blender.org/wiki/Source/Architecture/RNA. Last accessed 2020-03-15.
[23] Blender, Download. https://www.blender.org/download/
[24] Blender, Daily Builds. https://builder.blender.org/download/
[25] Blender, Experimental Branches. https://builder.blender.org/download/branches/
[26] Blender wiki, Blender 2.8: Dependency Graph. Last accessed 2020-03-19. https://wiki.blender.org/wiki/Source/Depsgraph

RNA) and their relationships (which object is owned by whom). Pointers are used in order to save time by not deep copying objects in a dependency graph when re-creating them.

4.3.8 Non-functional requirements

Functional requirements are requirements that describe what a system should *do* whereas non-functional requirements describe what a system should *be*.

4.3.8.1 Open source

Blender is an open source system. They host their own codebase alongside their website, wiki, and other services. Blender allows not only developers to contribute to the development of the application but also regular users can contribute. Users can request for features to be added or report an issue they encounter. Furthermore, anyone can join their chat rooms, or post threads on their developer forums. They also host weekly meetings in the developer chatrooms with an alternating schedule to accomodate for people living in different timezones.

4.3.8.2 Usability

Blender used to have a reputation for having an unintuitive user interface which can be especially daunting to beginners. One of the key points in the 2.80 update was to revamp the user interface to provide a better user experience. One example that many people found unintuitive is the fact that right click is used to select things by default. This was changed to left click in the 2.80 update alongside many other changes to the UI like providing a better way to organize your workflow with the addition of Workspaces. More widgets have been added.

4.3.8.3 Performance

Naturally, rendering scenes (either real-time or photorealistic) need to be as efficient as possible. Blender values the performance of the application as new patches (changes/commits) have to be evaluated for whether the have a negative impact on the performance.[27] Furthermore, Blender has a platform for collecting and displaying results of hardware and software test using their benchmark tool. These resulting benchmarks are used to compare differences in performances across different systems as well as to support the Blender development process.

4.3.8.4 Compatibility

Blender ensures forward *and* backward compatibility of their blend files. There is an exception from Blender version 2.80 onwards as files created in those versions of Blender are not always backwards compatible with (older verions)[https://developer.blender.org/T59120], because of the extensive amount of changes in Blender 2.80. Forwards compatibility is still ensured; Blender files created using older Blender versions can still be opened using the latest version of Blender. Compatibility is also part of their Quality Checklist.[28]

[27] Blender, Quality Checklist https://wiki.blender.org/wiki/Tools/CodeReview#Quality_Checklist. Last accessed 2020-03-19.
[28] Blender, Quality Checklist https://wiki.blender.org/wiki/Tools/CodeReview#Quality_Checklist. Last accessed 2020-03-19.

4.3.8.5 Security

Blender's blend files can be bundled with Pythons scripts, which poses a security risk. Blender has attempted to provide better security, but this has usually resulted in users voicing their discontent about discontent about driver constraints. A user expressed his worries in another thread at the fact that Python scripts are executed by default. Blender has taken measures in order to try and alleviate some of these concerns but security is not of high priority: "Blender does not attempt to achieve the same level of security as many other applications."[29]

In essence, Blender's non-functional priorities are ease-of-use and performance, the former in terms of GUI/lay-out, and the latter in terms of viewport responsiveness, rendering times, and speed of simulation calculations. There isn't any inherent trade-off between these properties; either of them can be optimized without impacting the other. Any other non-functional properties are less of a priority.

4.4 Blender Behind the Scenes

After having analyzed Blender's context in the first essay and its codebase architecture in the second, we take a look at the quality of the codebase in this third essay. Because Blender is already over 25 years old, and because of the complicated nature of the functionality it aims to deliver, we initially expect Blender to have a large amount of technical debt — the degree of inaccessibility, or, the number of "deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further."[30] In order to ensure a certain code quality, Blender maintains some rules, such as "no code gets in trunk without documentation" (meaning that contributors have to document new functionality in the Blender Documentation)[31], no new warnings are allowed, and code style has to remain consistent within files [32]. Numerous other guidelines can be found at [33].

Blender's release cycle is heavily tuned towards bug fixing [34]. Blender aims to release an update every three months, but the actual development of an update takes longer and overlaps with other updates, as shown in the figure below. Each update consists of five phases — Bcon1 through Bcon5 — and starts on the master branch. Bcon1 takes nine weeks and has developers focus mainly on new features and big changes. Bcon2 is four weeks, and is focused on improving, optimizing and stabilizing new and existing features, and only smaller changes and features are made here. From Bcon3 onward, the release gets its own branch, on which four weeks are spent on bug fixing[35] in Bcon3, one week is spent preparing the update in Bcon4, and the update is released in Bcon5. Also at the start of an update's Bcon3 phase, the succeeding update's Bcon1 phase starts. Developers spent 1-2 on update 1's Bcon3 phase, and the rest to update 2's Bcon1 phase. Therefore, approximately 60% - 70% of the time is spent on code improvement.

[29] Blender, How Does Blender Deal with Security? https://wiki.blender.org/wiki/Reference/FAQ#How_Does_Blender_Deal_with_Security.3F. Last accessed 2020-03-19.
[30] Martin Fowler. Technical Debt. https://www.martinfowler.com/bliki/TechnicalDebt.html.
[31] Blender Developer Wiki, New Developer Intro. https://wiki.blender.org/wiki/Developer_Intro/Overview.
[32] Blender Developer Wiki, Contributing Code. https://wiki.blender.org/wiki/Process/Contributing_Code.
[33] Blender Developer Wiki, For All Developers. https://wiki.blender.org/wiki/Developer_Intro/Committer.
[34] Blender Developer Wiki, Release Cycle. https://wiki.blender.org/wiki/Process/Release_Cycle.
[35] These fixes are naturally also merged with the master branch.

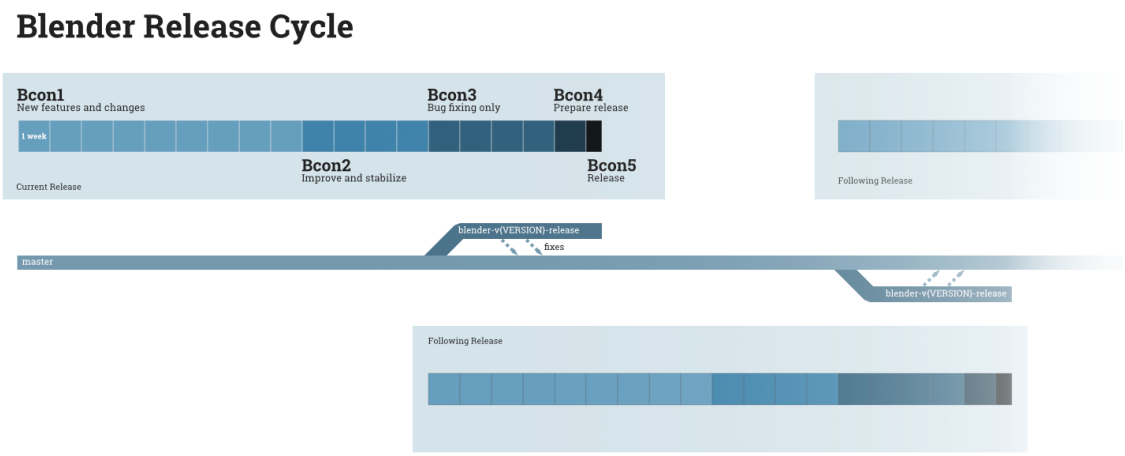


Figure 4.8: Blender's release cycle

4.4.1 Automated testing

Blender's testing pipeline is "a work in progress and isn't anywhere near full coverage"[36], and contributors are invited to add unit tests and increase coverage. Test coverage as a percentage of the codebase is not tracked, as the developers feel that there are "not enough tests to really be worth it at the moment. Focus is more on adding tests for things that we know are likely to break and lead to real bugs, for which such percentages are not always a good measure"[37]. They have some basic tests for the Python code (mainly UI and add-ons), and they use the Google Test unit testing library for the C/C++ (the majority of Blender's codebase). Tests for both these parts are called with CMake's CTest. Google Test needs to be enabled at build-time through a CMake build option (`WITH_GTESTS`), and creates testing executables which it then runs.[38] Blender publishes nightly unstable builds[39], and these are tested with the aforementioned test suites. However, since Blender lacks the resources to test and build every commit[40], we can't speak of Blender using a continuous integration or CI/CD approach.

The Python tests use Python's own `unittest` module. Python's tests can be run from the command line if Blender and Python are installed.[41] The list of Python tests is shown below.

Apart from these, there are exactly one hundred tests for the C/C++ code. 31 of these test `libmv` (the Blender-maintained multiview reconstruction library used for Blender's motion tracking[42]), 28 tests cover the Cycles rendering engine, 33 cover the various functions and types of the Blender Library (`BLI`), and the rest is meant for mesh operators and modifiers, and other I/O-functions, tools, and small internal libraries. Tests for the EEVEE and Blender Workbench rendering engines are disabled by default, because different

[36] Blender Developer Wiki, FAQ. https://wiki.blender.org/wiki/Reference/FAQ.
[37] Correspondence with Blender developer Brecht van Lommel.
[38] Blender Developer Wiki, GTest. https://wiki.blender.org/wiki/Tools/Tests/GTest.
[39] Blender, Download, Daily Builds. https://builder.blender.org/download/.
[40] Correspondence with Blender developers.
[41] Blender Developer Wiki, Python Tests. https://wiki.blender.org/wiki/Tools/Tests/Python.
[42] Blender, Libmv project. https://developer.blender.org/project/profile/59/.

```
emiel@Emielaptop:~/Documents/blender-git/build_linux$ ctest -N
Test project /home/emiel/Documents/blender-git/build_linux
  Test  #1: script_load_keymap
  Test  #2: script_load_addons
  Test  #3: script_load_modules
  Test  #4: script_bundled_modules
  Test  #5: script_pyapi_bpy_path
  Test  #6: script_pyapi_bpy_utils_units
  Test  #7: script_pyapi_mathutils
  Test  #8: script_pyapi_idprop
  Test  #9: script_pyapi_idprop_datablock
  Test #10: script_pyapi_prop_array
  Test #11: id_management
  Test #12: blendfile_io
  Test #13: blendfile_liblink
  Test #14: bmesh_bevel
  Test #15: bmesh_boolean
  Test #16: bmesh_split_faces
  Test #17: object_modifier_array
  Test #18: modifiers
  Test #19: constraints
  Test #20: operators
  Test #21: export_obj_cube
  Test #22: export_obj_nurbs
  Test #23: import_ply_cube
  Test #24: import_ply_bunny
  Test #25: import_ply_small_holes
  Test #26: export_ply_vertices
  Test #27: alembic_tests
  Test #28: script_alembic_io
  Test #29: ffmpeg

Total Tests: 29
```

Figure 4.9: Blender's Python tests.

GPUs yield different test results.[43] All tests are shown below.

```
Test   #1: libmv_predict_tracks_test        Test  #44: script_pyapi_bpy_path          Test  #87: alembic_tests
Test   #2: libmv_tracks_test                Test  #45: script_pyapi_bpy_utils_units  Test  #88: script_alembic_io
Test   #3: libmv_scoped_ptr_test            Test  #46: script_pyapi_mathutils         Test  #89: ffmpeg
Test   #4: libmv_vector_test                Test  #47: script_pyapi_idprop           Test  #90: BLI_array_test
Test   #5: libmv_array_nd_test              Test  #48: script_pyapi_idprop_datablock Test  #91: BLI_array_ref_test
Test   #6: libmv_convolve_test              Test  #49: script_pyapi_prop_array       Test  #92: BLI_array_store_test
Test   #7: libmv_image_test                 Test  #50: id_management                Test  #93: BLI_array_utils_test
Test   #8: libmv_sample_test                Test  #51: blendfile_io                 Test  #94: BLI_delaunay_2d_test
Test   #9: libmv_tuple_test                 Test  #52: blendfile_liblink            Test  #95: BLI_edgehash_test
Test  #10: libmv_euclidean_resection_test   Test  #53: bmesh_bevel                  Test  #96: BLI_expr_pylike_eval_test
Test  #11: libmv_fundamental_test           Test  #54: bmesh_boolean                Test  #97: BLI_ghash_test
Test  #12: libmv_homography_test            Test  #55: bmesh_split_faces            Test  #98: BLI_hash_mm2a_test
Test  #13: libmv_nviewtriangulation_test    Test  #56: object_modifier_array        Test  #99: BLI_heap_test
Test  #14: libmv_panography_test            Test  #57: modifiers                    Test #100: BLI_heap_simple_test
Test  #15: libmv_projection_test            Test  #58: constraints                  Test #101: BLI_index_range_test
Test  #16: libmv_resection_test             Test  #59: operators                    Test #102: BLI_kdopbvh_test
Test  #17: libmv_triangulation_test         Test  #60: export_obj_cube              Test #103: BLI_linklist_lockfree_test
Test  #18: libmv_dogleg_test                Test  #61: export_obj_nurbs             Test #104: BLI_listbase_test
Test  #19: libmv_function_derivative_test   Test  #62: import_ply_cube              Test #105: BLI_map_test
Test  #20: libmv_levenberg_marquardt_test   Test  #63: import_ply_bunny             Test #106: BLI_math_base_test
Test  #21: libmv_numeric_test               Test  #64: import_ply_small_holes       Test #107: BLI_math_color_test
Test  #22: libmv_poly_test                  Test  #65: export_ply_vertices          Test #108: BLI_math_geom_test
Test  #23: libmv_camera_intrinsics_test     Test  #66: cycles_bake                  Test #109: BLI_math_vector_test
Test  #24: libmv_detect_test                Test  #67: cycles_bsdf                  Test #110: BLI_memiter_test
Test  #25: libmv_intersect_test             Test  #68: cycles_denoise               Test #111: BLI_optional_test
Test  #26: libmv_keyframe_selection_test    Test  #69: cycles_denoise_animation     Test #112: BLI_path_util_test
Test  #27: libmv_modal_solver_test          Test  #70: cycles_displacement          Test #113: BLI_polyfill_2d_test
Test  #28: libmv_resect_test                Test  #71: cycles_hair                  Test #114: BLI_set_test
Test  #29: libmv_brute_region_tracker_test  Test  #72: cycles_image_colorspace      Test #115: BLI_stack_test
Test  #30: libmv_klt_region_tracker_test    Test  #73: cycles_image_data_types      Test #116: BLI_stack_cxx_test
Test  #31: libmv_pyramid_region_tracker_test Test  #74: cycles_image_mapping       Test #117: BLI_string_test
Test  #32: cycles_render_graph_finalize_test Test  #75: cycles_image_texture_limit Test #118: BLI_string_map_test
Test  #33: cycles_util_aligned_malloc_test  Test  #76: cycles_integrator            Test #119: BLI_string_ref_test
Test  #34: cycles_util_path_test            Test  #77: cycles_light                 Test #120: BLI_string_utf8_test
Test  #35: cycles_util_string_test          Test  #78: cycles_mesh                  Test #121: BLI_task_test
Test  #36: cycles_util_task_test            Test  #79: cycles_motion_blur           Test #122: BLI_vector_test
Test  #37: cycles_util_time_test            Test  #80: cycles_openvdb               Test #123: BLI_vector_set_test
Test  #38: cycles_util_avxf_avx_test        Test  #81: cycles_render_layer          Test #124: blenloader_test
Test  #39: cycles_util_avxf_avx2_test       Test  #82: cycles_reports               Test #125: guardedalloc_alignment_test
Test  #40: script_load_keymap               Test  #83: cycles_shader                Test #126: guardedalloc_overflow_test
Test  #41: script_load_addons               Test  #84: cycles_shadow_catcher        Test #127: bmesh_core_test
Test  #42: script_load_modules              Test  #85: cycles_sss                   Test #128: ffmpeg_test
Test  #43: script_bundled_modules           Test  #86: cycles_volume                Test #129: alembic_test
```

Figure 4.10: All Blender's tests.

Let's take a look at one GTest test and one Python test. The file `BLI_path_util_test.cc` (test #112) tests Blender's path utilities. The `#include "BLI_path_util.h"` indicates that this file tests `path_util.c`. Besides procedures for setting up and tearing down, the test file contains 24 unit tests, each of which tests one function `path_util.c`. Each such unit test, then, tests approximately 5-20 configurations, depending on the function under scrutiny. Ten unit tests investigate the `BLI_path_name_at_index` function, seven unit tests investigate `BLI_path_join`, and the functions `BLI_split_dirfile`, `BLI_path_frame_strip`, `BLI_path_extension_check`, `BLI_path_frame_check_chars`, `BLI_path_frame_range`, `BLI_path_frame_get`, and `BLI_path_extension` all get one unit test. However, `path_util.c` defines 51 functions, meaning upward of 40 functions aren't tested. The Python test file `bl_pyapi_mathutils.py` (test #46) tests the `source/blender/python/mathutils` component. It defines a "test case" (which is essentially the same as a unit test[44]) for the following datatypes: `Matrix`, `Vector`, `Quaternion`, and `KDTree`; and for the `tessellate_polygon(polyline)` function. However, this component also contains many other functions and object, like `EulerObject`, `Color`, all sorts of intersaction/collision tests, geometrical interpolation functions, and noise generators. Therefore, also this component is not fully covered.

[43] Blender Developer Wiki, Automated Tests. https://wiki.blender.org/wiki/Tools/Tests/Setup.

[44] Python 3 documentation, `unittest` — Unit testing framework. https://docs.python.org/3/library/unittest.html

4.4.2 Coding hotspots

Blender is a very large system. Files or directories where many commits or line changes are occuring are also called "coding hotspots". These coding hotspots can give one insight into which direction the system is developing. The latest stable version of Blender is version 2.82a, released on March 12th, 2020. Since then Blender has entered the Bcon2 stage of the 2.83 release cycle. The following table shows the number of commits, line insertions and deletions since March 12th for the top level directories.

Directory	Commits	Insertions	Deletions
source	290	135308	98723
release	52	279216	276144
intern	43	19900	13317
build_files	13	1167	865
tests	9	3712	378

At first glance it seems like the `source` and `release` directory seem to be the largest hotspots of coding activity, due to the number of commits and line changes respectively. In case of the `release` directory, this is due to the addition of a single file that contains more than 270 thousand lines. Further investigations in the `source` directory yields the following table.

Directory	Commits
source	290
source/blender	286
source/blender/editors	119
source/blender/blenkernel	93
source/blender/makesrna	53
source/blender/draw	48
source/blender/windowmanager	27
source/blender/makesdna	26
source/blender/blenloader	21
source/blender/depsgraph	17
source/blender/modifier	16
source/blender/gpu	13
source/blender/blenlin	12

The `editors` and `blenkernel` directories contain the most changes. `blenkernel` commits are mostly located in the `intern` subdirectory which contains all the source files. The `editors` contains more interesting subdirectories as each subdirectory located there is a feature. These distribution of code activity can be seen in the next table.

Directory	Commits
source/blender/editors/gpencil	27
source/blender/editors/editors	25
source/blender/editors/sculpt_paint	20

Directory	Commits
source/blender/editors/include	20
source/blender/editors/transform	17
source/blender/editors/object	16
source/blender/editors/animation	15
source/blender/editors/space_view3d	12
source/blender/editors/screen	11

As can be seen from the table, the `gpencil` and `sculpt_paint` directories are quite high on the list, meaning that there's still a lot to work going on for these features. These features also showed up in prior release notes and were highlighted in a (video)[https://www.youtube.com/watch?v=EfF2wDXalgU] posted on Blender's YouTube channel.

4.4.3 Roadmap activity

Blender does not have a global roadmap per release cycle due to their size. Instead, each module owner maintains a list of tasks. These lists contain tasks that are to be completed in a single release cycle as well as tasks that may span multiple release cycles. It is therefore rather difficult to keep track of what their goals are for a specific version. They do keep track of the changes that have been made in their (release notes)[https://wiki.blender.org/wiki/Reference/Release_Notes/2.83]. Blender has posted a roadmap for their release planning[45] as well as big projects that they would like to tackle before in the year of 2020 [46]. Blender 2.83 seems to revolve mainly around adding small sized features as well as bug fixes. Larger objectives, such as their upcoming **Everything Nodes** and **Asset Manager** projects, are projected to be included in later releases.

4.4.4 Technical debt

Searching through the forums of Blender at [47] and the Blender chat [48], there is little talk of technical debt amongst contributors. It is doubtful that many of them are actively worried about this during discussions, as Blender themselves hold code quality days [49] on each first Friday of each month. These days are fully dedicated on improving the Blender code quality. Any contributer who is interested is allowed to participate in the code quality days. The coordination is done online through the Blender chat. They usually work on:

- adding comments to explain code, giving overviews.
- renaming obscure functions and variables.
- split up big files into more organized smaller ones.
- Uupdate the wiki documentation.

A glance at Blender's development website[50] suggests that they share each task done during code quality days as separate tasks so that other contributors can chime in. In these tasks, it is possible to see discussions of testing new ideas, and code quality discussions between contributers. From these, we can conclude that

[45] Blender, Blender LTS and 3.0. https://code.blender.org/2020/02/release-planning-2020-2025/
[46] Blender, 2020 - Blender Big Projects. https://code.blender.org/2020/01/2020-blender-big-projects/
[47] Blender developer forum. https://devtalk.blender.org/c/blender/5.
[48] Blender developers chat. https://blender.chat/channel/blender-coders.
[49] Blender, Style guide, Code Quality Day. https://wiki.blender.org/wiki/Style_Guide/Code_Quality_Day.
[50] Blender, developer. https://developer.blender.org/.

Blender uses the code quality days in order to help reduce the technical debt and improve the code quality of Blender, rather than doing this per feature being worked on. This iterating through the code on a scheduled basis seems like a more organized way of bringing clarity to the project. Consequently, the code becomes easier to understand for new developers and saves time in implementing new features.

4.4.5 Automated code quality analysis

Static analysis can be used to generate an assessment of code quality. Popular projects for this purpose are SonarQube [51] and SIG [52]. We have set up SonarQube to do static analysis on the Blender project, as we only have little results from SIG's Sigrid, apart from the dependency graph. However, we did hear that — in the subset of components we submitted — there are 1401 cyclic dependencies, 380 indirect cyclic dependencies, and 45 bypassing layers.

The code of Blender mainly consists of C/C++ code. Unfortunately, the (free) community edition of SonarQube does not offer support for these languages. However, the UI and some tooling have been written in Python which is supported. Therefore, SonarQube could be run on Python and some markup languages within the Blender source code.

The analysis consists of 727 files and 101,000 lines of code, of which 98% is Python code. SonarQube lists 84 bugs, but many of these do not affect how Blender runs as a program. For example, there is a 10 years old HTML file that describes Blender's file format. This file contains many bugs and code smells: the 'Verdana' font is used which may not be present on all computers, and `<th>` (table header) elements don't have an `id` associated with them so that screen readers may not announce the columns of the table. Furthermore, a dozen bugs of severity "Blocker" — stating that `bpy is not defined` — are listed. This is however not a bug, but rather a side-effect of Python being set up in a custom way within Blender itself.

A lot of potential code improvements can be found, however, such as empty code blocks that can be removed, commented-out code, and functions that have a lower case and an upper case variant with the same name.

SonarQube also reports a section for technical debt. Even though over 3,000 instances of code smell are found within the Python codebase, this amount is still low considering the size of the project. The overall technical debt is estimated at 69 days of engineering work, which mostly pertains to the UI code. All in all, SonarQube considers the maintainability of the Python part of Blender's source code good, as the technical debt ratio is lower than 5%.

4.4.6 Conclusion

In conclusion, Blender adheres to a very strict and streamlined development process. They spent a significant time on code improvement and bug fixes. The small amount of SIG's Sigrid analysis results tell us that Blender violates a nontrivial number of good practices, while SonarQube tell us that Blender does not have as much technical debt as one would expect from a 25-year-old project. Especially given Blender's sheer complexity, we think it's code quality is surprisingly good and well-maintained, which also makes it easier for new developers to helpt contribute to Blender.

[51] SonarQube. https://www.sonarqube.org/
[52] SIG. https://www.softwareimprovementgroup.com/

4.5 Blender's Variability

This essay will outline how and to what extent Blender is variable. This "software variability is the ability of a software system or artifact to be efficiently extended, changed, customised or configured for use in a particular context"[53], and this means we will take a look at how many different "instantiations" of Blender are possible, and how these are obtained. We start with identifying these variable components, also referred to as "features", and modeling those in a feature model.

4.5.1 Variability analysis

Blender is available on Windows, macOS, Linux[54], and this choice constitutes the first feature we distinguish. This kind of platform variability is a form of compile-time binding, resulting in a separate binary for each platform. Functionally, these binaries are — as far as we know and have investigated — functionally identical, and choice of operating system entails no functional constraints. (The Windows version does seem to perform better.[55]) Blender even replaced GLUT with their own windowing system, called GHOST, in order to be multiplatform.[56]

Besides the stable builds, there are also unstable daily builds, containing newer features[57], and builds from experimental branches[58], which is a form of functional variability. Especially for developers, there is even more compile-time binding variability, as Blender offers several build targets. After a developer has cloned the Blender repository, he can build the application by simply running `make` on Linux and macOS or `./make.bat` on Windows. Additional flags can be passed on for customizing the builds. The available options can be retrieved by running `make help` or `./make.bat help`. The options that are relevant to the building of Blender are:

- `debug`: Debug binary, which simply disables certain compiler optimizations and includes debug information. We do not consider this build a feature.
- `release`: Identical to the official builds on blender.org builds, i.e. a complete build with all options enabled.
- `regular`: Similar to a release build, except CUDA and Optix.
- `lite`: Contains the minimum amount of features enabled for a smaller binary, faster building time, and less dependencies.
- `headless`: Omits the interface (and is meant for renderfarms or server automation).
- `cycles`: Build Cycles standalone only, without Blender.
- `bpy`: Build as a Python module which can be loaded from python directly.
- `deps`: Build library dependencies (intended only for platform maintainers).
- `developer`: Enable faster builds, error checking and tests, recommended for developers.
- `config`: Run CMake configuration tool to set build options.
- `ninja`: Use Ninja build tool for faster builds.

Besides these preset build targets, Blender's CMake configuration file offers hundreds of options[59], and almost any library and non-essential component of Blender can be enabled or disabled, leading to an

[53] Svahnberg, M., et al. (2005) 'A taxonomy of variability realization techniques', Software - Practice and Experience, 35(8), pp. 705–754.
[54] Blender, Download Blender. https://www.blender.org/download/.
[55] Blender Benchmark. https://opendata.blender.org/benchmarks/query/?group_by=os. Last accessed: 2020-04-06.
[56] Letwory Interactive. Blender: GHOST. http://www.letworyinteractive.com/blendercode/d5/d2e/GHOSTPage.html.
[57] Blender, Download Blender Daily Builds. https://builder.blender.org/download/.
[58] Blender, Download Blender Experimental Branches. https://builder.blender.org/download/branches/.
[59] Blender, Diffusion. CMakeLists.txt. https://developer.blender.org/diffusion/B/browse/master/CMakeLists.txt.

enormous amount of product variations.

Conditional compilation is used for specifying code that is to be used with or without certain targets. The `debug` build target does not result in a different codebase, as the options that are changed are mainly compiler options. The `headless` target, on the other hand, does have preprocessor macros (`#IFDEF`) that change what will be compiled. A `headless` build does not not contain any graphical interface, and as such, any GUI code does not have to be compiled. The build configuration for Blender seems to be somewhat complex considering the size of the codebase and the variability in build targets. When looking at the commit history of the build files however, it is apparent that they are not changed often. At present, it seems to be fully stable, though the build configuration files might prove to be an obstacle when more build targets are added.

4.5.1.1 Add-ons

An important feature of Blender, which we have thusfar only briefly touched upon, is Blender's Python scripting API and all the modding possibilities (a "mod" is a modification, hence a feature) that this brings. Blender comes bundles with a number of add-ons already, some are labeled as officially supported (by the Blender developers), some are labeled as community supported (by community developers), and the rest is labeled as testing, and these are not included in release builds. Besides these, anyone can script an add-on, distribute it, and have others import it. All add-ons can be enabled and disabled in Blender's Preferences window without restarting the program (because they are written in Python), indicating that these add-ons are bound at runtime. An add-on must have the following:

- `bl_info` Dictionary containing metadata bout the add-on.
- `register` Function that is run when enabling the add-on.
- `unregister` Function to unload anything that was setup. Called when the add-on is disabled.

Blender's documentation on writing add-ons[60] makes apparent that Blender uses the Observer pattern for their add-ons, as every add-on must have the `register()` and `unregister()` functions.

In order for an add-on to be accepted into the officially distributed add-ons catalog, it must meet several requirements [61]. These requirements are quite complete and strict:

- Contain add-on meta information in a `bl_info` dictionary.
- Define `register()` and `unregister()` functions.
- Be documented on an associated wiki page in the Blender wiki.
- Be evaluated and approved by another Blender developer.
- Pass the Flake8 / other PEP8 checker tool for style guide enforcement.
- Be compatible with the latest Blender release.
- Inclusion of binary data-files is to be avoided.

Furthermore, for an add-on to remain accepted into the official release, it must also receive continuous maintenance to keep the add-on working. Because of these strict guidelines, the number of default installed add-ons will not become too large to the point where it starts slowing down the application, and thus Blender should remain scalable. Furthermore, the add-ons do not modify Blender's internals and are mostly extensions to the UI.

[60] Blender, Add-on Tutorial. https://docs.blender.org/manual/en/latest/advanced/scripting/add-on_tutorial.html.
[61] Blender, Add-on guidelines. https://wiki.blender.org/wiki/Process/Addons/Guidelines.

4.5.1.2 Blender's Preferences options

Besides add-ons, the Blender Preferences window naturally contains many options for modifying the program, but many of these we either consider to be part of the program's functionality and thus not a (variable) feature, or they are too specific and mentioning them all is cumbersome and not informative. We do want to distinguish the option of rendering device for the Cycles rendering engine. Cycles is Blender's physically-based (photorealistic) ray tracing engine, and this can be run either on the CPU or on the GPU. Furthermore, when running on the GPU, there is the choice of graphics API: Nvidia's CUDA or Optix, or OpenCL. The Cycles engine furthermore has the option of enabling its experimental feature set.[62] However, as of writing, adaptive subdivision is the only experimental feature. All these Cycles-specific options are bound at runtime.

As for minor options, Blender offers a number of GUI translations. 8 languages are fully supported, 9 languages are works in progress, and the translation of 19 languages is just getting started. For audio device, there can be chosen between OpenAL or SDL.

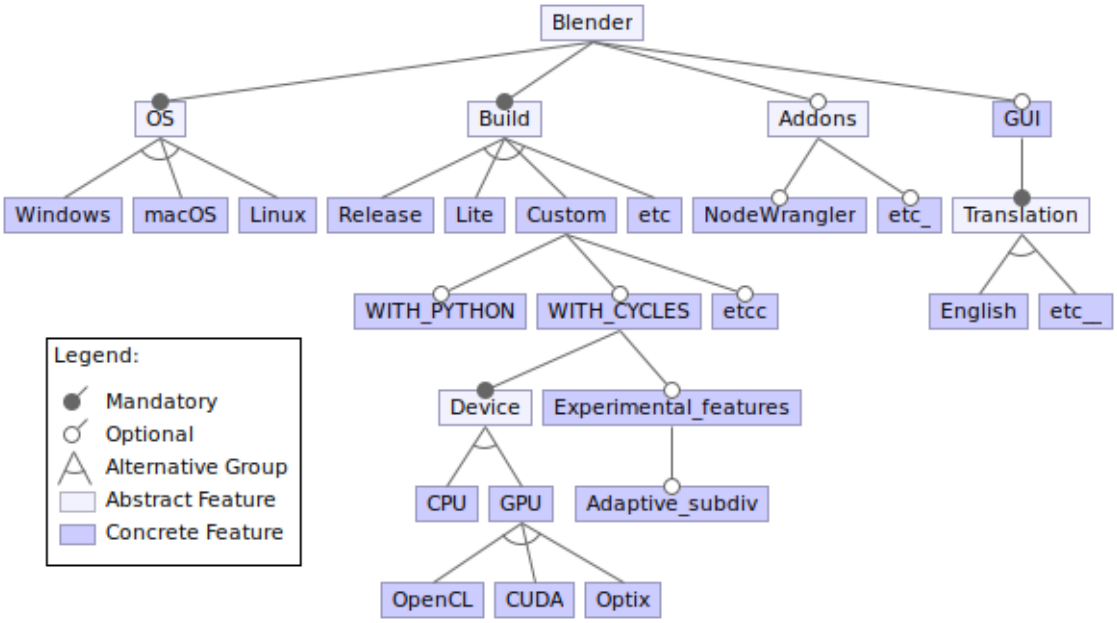


Figure 4.11: Feature model, created with FeatureIDE[63].

We created a feature model with the features we identified. All (additional) constraints arise between the CMake build options. For example, when `WITH_PYTHON` is disabled, we cannot use the Cycles rendering engine (`WITH_CYCLES`) and neither can we use the Draco mesh compression Python module (`WITH_DRACO`). As another example, when we build Blender as a Python module, the CMake configarion disabled the GUI. The CMake configuration (`CMakeLists.txt`) defines many, many such contraints, and some are rather complex, consisting if many nested if-else statements. Aside from the build options, however, we have not been able to identify any constraints that cannot be expressed in the feature model, and we think this is a

[62] Blender Developer Wiki, Cycles, Experimental Features. https://docs.blender.org/manual/en/latest/render/cycles/features.html.
[63] FeatureIDE. https://featureide.github.io/.

testament to Blender's versatilty.

4.5.2 Variability management

For stakeholders, the way all the variability is managed is quite straightforward. Most of the end-users interact with the three available operating system choices[64], and the translations that are available in the Blender settings. If there are new features they want to take advantage of, at the bottom of the download page they can also download the experimental builds. More invested end-users might also want to install the mentioned add-ons. The rest of the builds available from the CMake file in the source code is intended for the stakeholder category of developers, as there is no reason for end-users to have the source code.

As for the information that is available to the end-users and developers about variability management, most of it can be found on many of Blender's sites. For example, end-users can find information on add-ons in the Blender manual [65]. For developers, there is tons of information on different variability options on the Blender wiki. All the information starts in the Building Blender "hub" page [66]. From there, developers access information on resolving building issues[67], can get redirected to the building options[68] mentioned in the previous section, find links to what the best setup is for developing Blender[69], library dependencies that Blender needs[70], and lastly, information on other building options, such as Blender as a Python module[71] or building with CUDA and Optix[72].

The library dependencies that are needed are stored precompiled in an SVN repository and automatically downloaded when running `make update`. The precompiled libraries are built through the use of a CMake based system. On Linux, it is also possbile to use the System Package Manager to install many of the libraries, though losing the portability option this way. Besides the common build errors listed in the link above, it is also possible to report build problems on the Building Blender forum [73] or on the Blender IRC [74], so there are enough places for developers to help each other with building Blender. Inside the building options page mentioned in the previous paragraph, it is explained how to speed up the building, such as enabling address sanitizer in CMake, using the Ninja build system, or using ccache in Unix-based operating systems, which seems to be especially useful when switching between Git rivisions and branches. On the page intended for developers with links to different tools [75], it is also possible to find information on how to use distcc with Blender [76], which is a tool to distribute building C/C++ software across multiple Unix-like systems, and is useful for when developers need to rebuild the entire source tree many times.

4.5.3 Conclusion

Blender is highly variable, but most of this variability is aimed at developers or users building from source. For regular users, the most relevant form of variability manifests itself in the stable/nightly/experimental

[64] Blender, Download Blender. https://www.blender.org/download/.
[65] Blender, manual, Add-ons. https://docs.blender.org/manual/en/latest/editors/preferences/add-ons.html.
[66] Blender, Building Blender. https://wiki.blender.org/wiki/Building_Blender.
[67] Blender, Resolving Build Issues. https://wiki.blender.org/wiki/Building_Blender/Troubleshooting.
[68] Blender Developer Wiki, Build Options. https://wiki.blender.org/wiki/Building_Blender/Options.
[69] Blender, Building Environments. https://wiki.blender.org/wiki/Developer_Intro/Environment.
[70] Blender, Library Dependencies. https://wiki.blender.org/wiki/Building_Blender/Dependencies.
[71] Blender, Blender As Python Module. https://wiki.blender.org/wiki/Building_Blender/Other/BlenderAsPyModule.
[72] Blender, Building Blender with CUDA and Optix. https://wiki.blender.org/wiki/Building_Blender/CUDA.
[73] Blender, Building Blender forum. https://devtalk.blender.org/c/blender/building-blender.
[74] Blender, chat. https://blender.chat/channel/blender-coders.
[75] Blender, Development Tools. https://wiki.blender.org/wiki/Tools/distcc.
[76] Blender, distcc. https://wiki.blender.org/wiki/Tools/distcc.

builds and the platform variability. Across platforms, Blender is ultimately very consistent, at least in terms of functionality and from the perspective of the end-user. This makes Blender very predictible and compatible, and allows users on different platforms to collaborate.

Chapter 5

Bokeh

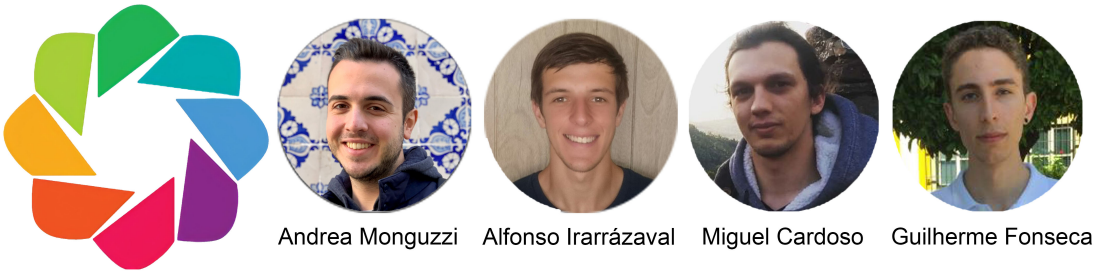


Figure 5.1: team

Publicly released in April 2013, Bokeh is an interactive visualization library for modern web browsers. It is suitable for the creation of rich interactive plots, dashboards and data applications. In fact, Bokeh does so in an elegant and concise way, without losing the ability to provide high-performance interactivity over large or streaming data sets. This library shows other remarkable qualities:

- **Flexibility** - with Bokeh you can create common plots or handle custom use-cases, it is up to you!
- **Interactivity** - Bokeh offers tools, widgets and UI events that allow you to drill-down into details of your data.
- **Power** - Bokeh is powerful! You can add custom JavaScript to help you with specialized cases.
- **Shareability** - with Bokeh you can easily publish your plots, dashboards and apps in web pages or even Jupyter notebooks.
- And the best part? Bokeh is an open source project!

Mainly written in Python and with a special focus on offering rich and interactive visualizations for web browsers through JavaScript, Bokeh differs from other data visualization libraries (such as Matplotlib). In addition, the project is in constant development, with hundreds of open issues.

In the following pages a structured analysis on Bokeh is carried out: the main focuses will be on the vision underlying the project and its possible future success, followed by a study on the architectural decisions made and on the code level perspective. Finally, we will conclude with a deep analysis guided by the main

characteristics of Bokeh.

5.1 Team

The team is constituted by the following members:

- Alfonso Irarrázaval
- Andrea Monguzzi
- Guilherme Fonseca
- Miguel Cardoso

which, although coming from different backgrounds, share a vivid interest for Data Visualization. Lured by the appealing characteristics mentioned above, the team decided to embrace Bokeh, deep dive into its design choices and code base and analyse it for the Software Architecture course, with the goal of learning from both the project and the course, and ultimately share their findings and the lessons learnt with the world!

5.2 Plotting Bokeh, an Analysis to its Present and Future

There is more than meets the eye. Data is everywhere. There are patterns in every aspect of life but sometimes we cannot see them, we cannot understand them and even worse, we cannot retrieve any value from them! Bokeh is one of the many tools that aims to solve this problem: it connects the dots and lessens the distance between what we see and what we understand. In this essay we will look into Bokeh and what motivated its creation, development and vision!

5.2.1 Behind what you can plot

Data is everywhere! Everyone knows that. Nowadays society thrives and lives on data. It's not surprising, then, that people try to use such data to reveal hidden patterns within. That is where data visualization comes into play: to lessen the distance between what we see and understand.

Python has been, as of late, the dominant programming language for data processing, as many libraries - such as Pandas, NumPy and ScyPy - have been optimized to work with large datasets, and users would like to have a visualisation tool on Python that can be easily connected to the processed data. Users usually have another, extra goal: to be able to share their findings in a way so that the visualization's users can easily access and interact with it. Bokeh brings all of the previous requirements together, by providing interactive plotting on web browsers through Python in an easy and efficient way.

5.2.2 The hardest plot

Every software has the same end goal: to be used and understood. However, every user is different, and with this difference emerges an important point: distinct users may think about the software in distinct ways. With that in mind, in order to develop successful software, it is important to explore users' perceptions around the system, anticipating their wants and needs.

When it comes to Bokeh, two types of users stand out: data enthusiasts and data specialists. While both types of end users expect Bokeh to act like a catalyst in plotting their data, we can draw the line between both groups when we talk about *how* they perceive and use the system. Data enthusiasts don't use Bokeh professionally. Sometimes, they just want to play a bit with the data and share their incredible plots. These

users value *ease* and *celerity* when using the system. On the other hand, data specialists may depend on it professionally. They require *efficiency*, *productivity*, *complexity* and *completeness* in order to retrieve the hidden value of the data and handle their custom use-cases.

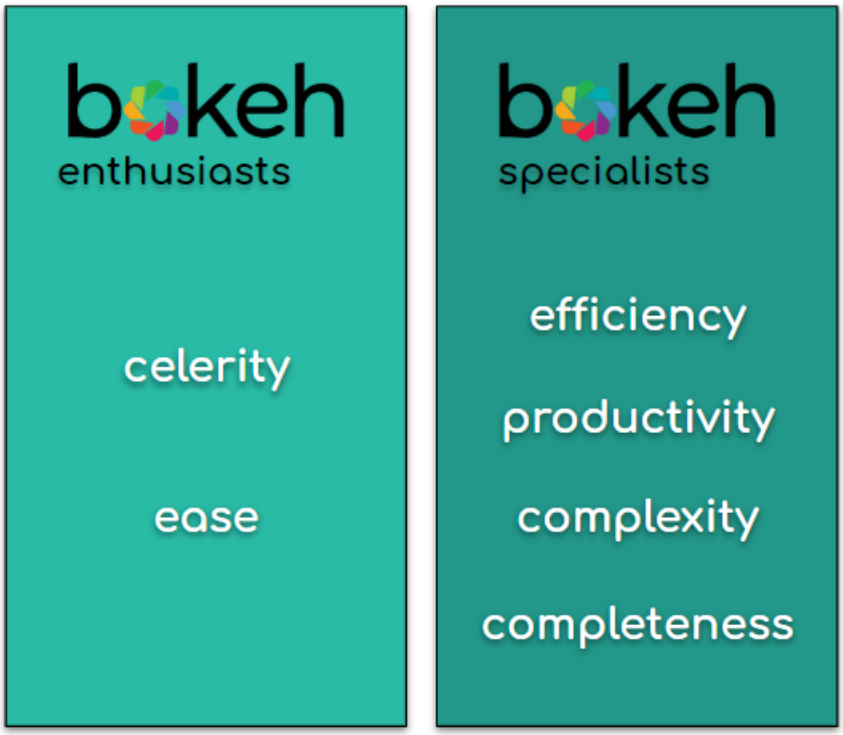


Figure 5.2: User Needs

This dissimilarity imposes a challenge on Bokeh: the system has to be viewed as easy to work with while having a powerful and complete interface. The real question comes up: Is Bokeh able to satisfy both mental models? In the next subsections we will look into that.

5.2.3 The reasons why you should plot with Bokeh

Does Bokeh generate value? Absolutely! Bokeh is an interactive visualisation library for modern web browsers, which can be described as: flexible, interactive, shareable, productive, powerful, and, last but not least, open source. These keywords define what Bokeh is and what it offers. Indeed, this library allows straightforward usage while giving the means to build cutting-edge specialised use-cases. This simple but important specification highlights the strong flexibility of the software: it is possible to start small, and eventually build up in order to use advanced tools and more complex and sophisticated features. It is clear that Bokeh aims to be for everyone: be a new user or an experienced one, the only requirement is knowing how to use Python. The fact that Bokeh is conceived as a Python framework means that the user can resort to PyData tools that he/she is familiar with, without the need of learning new tools and languages.

But Bokeh is not just Python! It allows to add custom JavaScript to support advanced or specialized cases,

providing the user a powerful customization possibility. Furthermore, all of the user plots, dashboards and apps generated with Bokeh can be easily published in web pages or Jupyter Notebooks for easy access.

However, Bokeh not only offers value on what it does - enhancing the understanding of the data - but on how it does so. For example, data scientists and developers can leverage Bokehs' capabilities to interact with the published results, to probe "what if" scenarios, to drill-down into the details of the data and also to visualize real-time data.

5.2.4 Plotting variables

Software is made for and by people and these people can be categorized into different groups of stakeholders.

> *'What is a "stakeholder"? Team members, system engineers, architects, and testers all have a stake in creating sound system form. There may be many more: use your imagination.'* [1]

In their book [2] Coplien and Bjørnvig define stakeholders taking into consideration two distinct parts of the system: what it *does* and what it *is*. The first part, related to the structure of the business over time, has as principal stakeholders *domain experts*, *system architects* and *business people.* On the other hand the second part, that relates to the user's view of the services provided by the system, has as principal stakeholders *end-users*, *user-experience people*, *interface designers* and *requirements people*. Furthermore, they divide the stakeholders in five major areas:

- the end users,
- the business,
- customers,
- domain experts,
- and developers.

To further our understanding, we looked at a wide range of different sources. In addition, we go into deeper detail and divide each major area defined in [3] in different sub-areas, in an attempt to refine the analysis of Bokeh's stakeholders. Such analysis can be found in the following tables.

End Users	**Description**
Specialists	Previous sections showed that Bokeh is undoubtedly a library of powerful characteristics. Consequently, Bokeh is used in the industry as a tool to help different projects. Even Elon Musk tweets about it! In fact, not only Data Scientists, but also Engineers, Researchers and other end users depend on Bokeh for their day-to-day endeavours.
Enthusiasts	The enthusiasts are the end users that do not use Bokeh professionally but rather in their free time activities mainly out of curiosity. Examples of this kind of end users are open source community members, students and overall enthusiasts.

[1] Coplien, James O., and Gertrud Bjørnvig. Lean architecture: for agile software development. John Wiley & Sons, 2011.
[2] Coplien, James O., and Gertrud Bjørnvig. Lean architecture: for agile software development. John Wiley & Sons, 2011.
[3] Coplien, James O., and Gertrud Bjørnvig. Lean architecture: for agile software development. John Wiley & Sons, 2011.

Business	Description
Business Sponsors	Bokeh is a sponsored project of NumFOCUS, a nonprofit charity in the United States. NumFOCUS provides Bokeh with fiscal, legal and administrative support to help to ensure the health and sustainability of the project (defined as an Assessor in [4]). Furthermore, Bokeh's donations are managed by NumFocus[5]. Bokeh is also sponsored by Anaconda, Nvidia, Quansight and REX Real Estate (defined as an Acquirers in [6]).

Domain Experts	Description
Core Team	Bokeh's core team offers the expertise needed to structure the project. The members of this team are responsible for the ongoing organizational maintenance and direction of Bokeh[7]. By March 2020 this team was compose by: Sarah Bird, Luke Canavan, Carolyn Hulsey, Mateusz Paprocki, Philipp Rudiger, Bryan Van de Ven.

Among the responsibilities of this team we have: - Reviewing and merging Pull Requests from other contributors - Making and implementing decisions about project infrastructure - Protecting and managing confidential project information such as service passwords - Handling all project financial matters - Addressing reports related to the Community Code

Developers	Description
Development Team	Developers are the prime oracles of technical feasibility[8]. In Bokeh, the development team comprises active contributors.
Maintainers	Maintainers are members of the Development team whose job is to maintain and enforce.
Other Contributors	Here we include testers, designers, etc.

Stakeholder engagement

Stakeholder engagement is essential when it comes to achieve a coherent, well designed and functional system. In regard to developed-related communication, it is done mostly through Github and Zulip. In

[4]Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[5]https://github.com/bokeh/bokeh#sponsors.

[6]Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[7]https://github.com/bokeh/bokeh/wiki/BEP-4:-Project-Roles#core-team.

[8]Coplien, James O., and Gertrud Bjørnvig. Lean architecture: for agile software development. John Wiley & Sons, 2011.

Github, developers interact through Issues and Pull Requests, whereas in Zulip developers interact in real time, discussing different topics related to the project.

End-user engagement is crucial as well. As Coplien and Bjørnvig say,

> *"[. . .] good end user engagement changes end user expectations."* [9]

This communication is possible with the help of a discussion website, Discourse. Here users and other community members can showcase their interesting projects and awesome works, receive general support or even discuss Bokeh's development. In addition, Twitter is also used as a platform for engagement, overall discussion and showcasing.

5.2.5 Variables and axis of Bokeh

As we know by now, data is taking over the world! And since progress does not stop, all data processing related tools will continue to grow even more. We talked about how Python, with the help of libraries such as NumPy and Pandas is one of the most used data processing tools, and we can only expect it to improve.

The following figure synthesis the context in which Bokeh lives.

Besides their strong participation in the data world, Bokeh still has room to grow. As many visualization libraries, it gets saturated for insanely large datasets, and that may be a potential area for improvement.

5.2.6 Bokeh and beyond

Progress should not be stopped. Bokeh is a software that is actually (March 2020) in continuous growth and development: just look at the pull requests!

This library allows you to see more and to see better. The only limits of what you can do with Bokeh are on the realm of human imagination: if you can imagine it, you can *probably* plot it.[10]

Moreover, Bokeh is open source, so you can help it grow even further and beyond. The future lies right ahead. The following image synthesizes Bokeh's roadmap.

But the future is not only about software: people are an important variable in the equation, since software is made by and for them. In this regard, we can mention the willingness of improving this project's documentation in an attempt to help the users finding the appropriate documentation and examples.

Additionally, some hints on the imminent future direction of the software can be found in the open issue list. Take, for example, this issue that jumps out, reporting a discussion about the possibility that the new version of Bokeh might stop supporting legacy web browsers in the future. For instance, this change can ultimately empower this library, while users stuck with obsolete web browsers will still be able to use previous versions of Bokeh.

Now that you can *plot* Bokeh, what will be your next graph?

5.3 Plotting Bokeh, an Analysis of its Architectural Variables

In order to architect, one must envision. In the last essay we discussed the *vision* underlying our project. Now we go into further detail in our architectural analysis of Bokeh. We shed some light on the different

[9] Coplien, James O., and Gertrud Bjørnvig. Lean architecture: for agile software development. John Wiley & Sons, 2011.
[10] https://demo.bokeh.org/

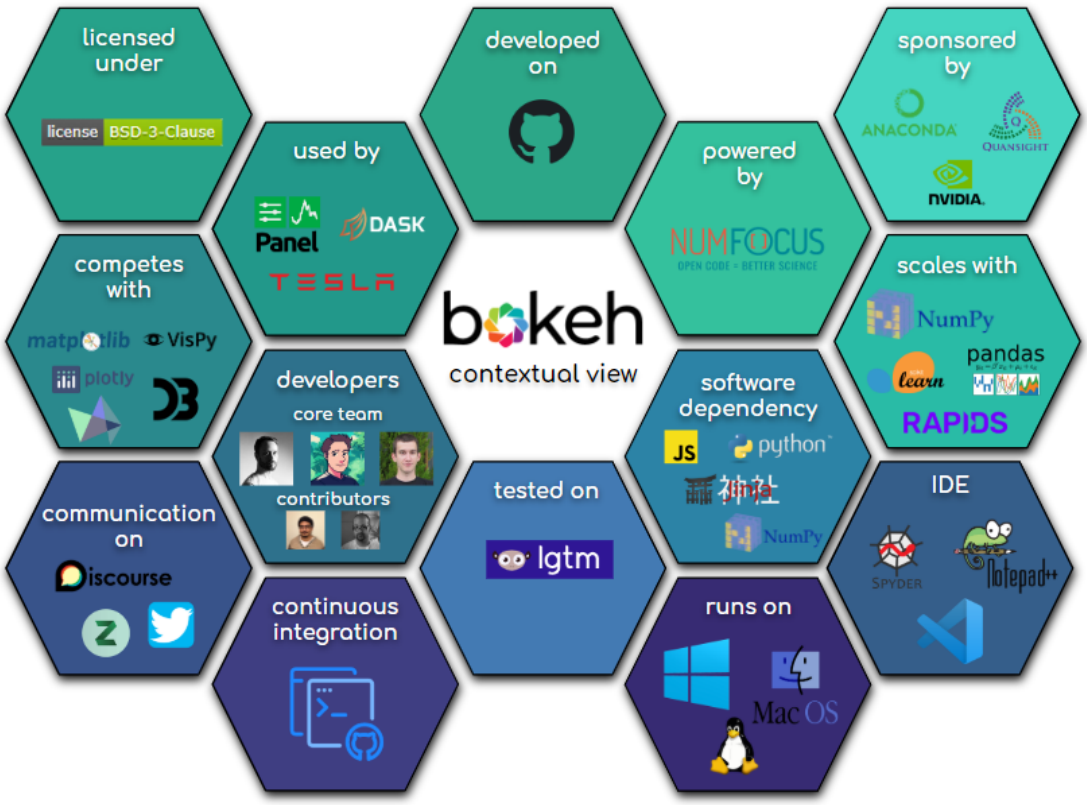


Figure 5.3: Variables and Axis

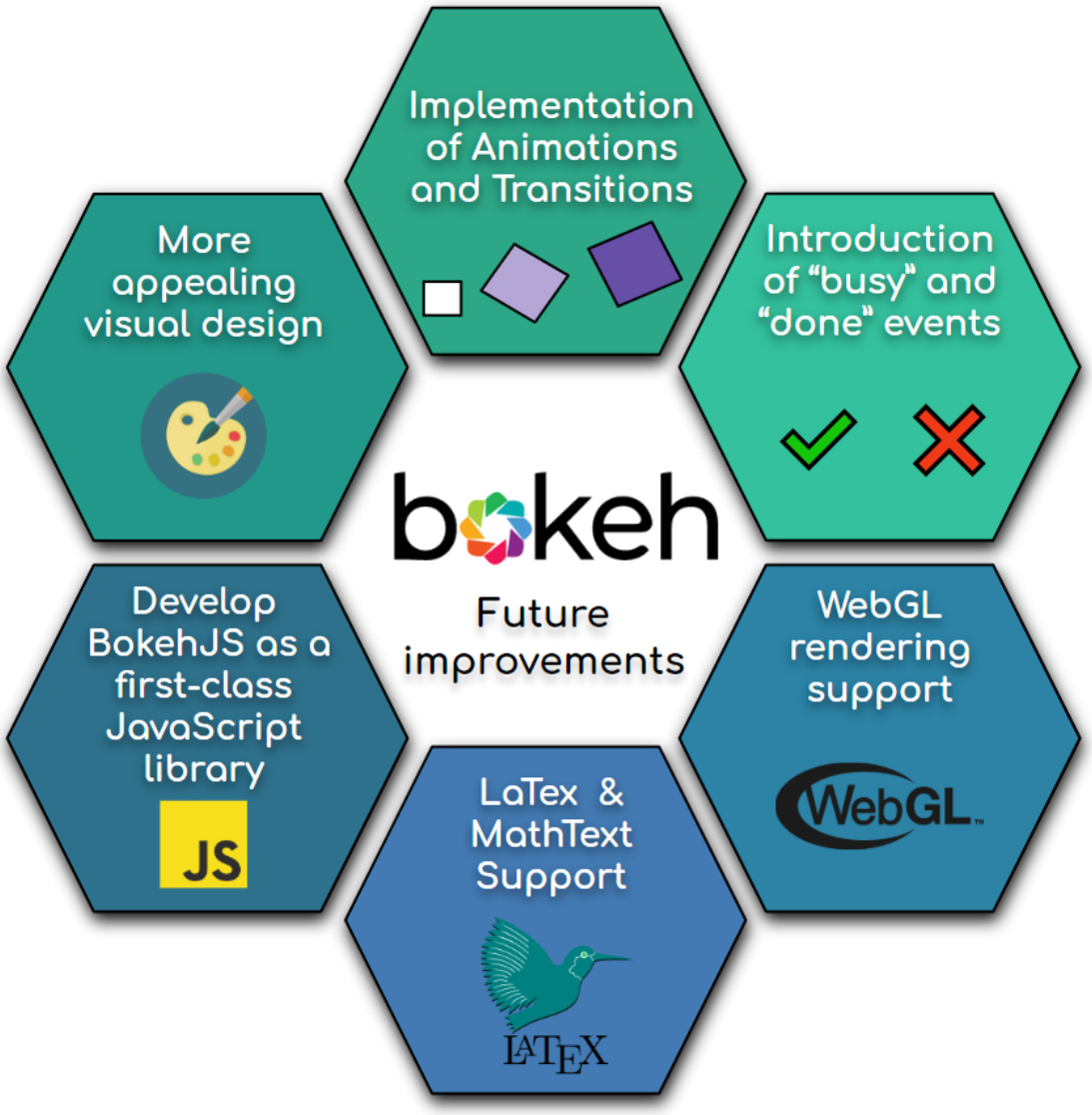


Figure 5.4: What lies ahead

architectural views that describe Bokeh; we address design choices and patterns applied to this project; and finally we look into its non-functional properties.

5.3.1 A bird's eye view of Bokeh's architecture

It is fair to assume that while using Bokeh one may think "wow, what an amazing graph I just plotted!". Indeed, Bokeh allows for the creative usage of data with the end goal of promoting insightful thoughts when one looks at it. However, there are also other perspectives from which you can look at Bokeh. Different stakeholders involved in this project share different viewpoints towards the system, which must be considered. To do that we resort to the 4+1 view model of architecture[11].

The **logical view** is one of the most relevant views of the system since it relates to the functionality provided to the end-users. As we discussed in the first essay, Bokeh is a visualization library that refers to a broad spectrum of users, offering different functionalities with the goal to be for everyone, from new users to experienced ones.

According with the definition of Kruchten, the **process view** takes into account some non-functional requirements. Moreover, it addresses concurrency and distribution, system integrity, and fault tolerance, focusing on the run time behavior of the system[12]. From Bokeh's perspective, the run-time view and non-functional requirements are essential... and do not worry! we will address these later, your curiosity shall be satisfied!

The **development** view describes the system from the viewpoint of software developers and testers. Thus, this view is concerned with the architecture that supports the development process, in order to

> ensure that there is order rather than chaos when it comes to the organization of the system's code.[13]

The fact that Bokeh is an open source project with more than 92.8k lines of both Python and Javascript code organized in a wide structure (check codeline organization figure) shows that the analysis of this view should not be underestimated, as you will see later in the essay.

The fourth viewpoint is the **physical view**. It concerns aspects of the system that are important after it has been tested and is ready to be deployed[14]. This view also describes the mapping of the software on the hardware, which in Bokeh's case is not so relevant, due to the fact Bokeh and BokehJS work on top of abstractions, like operative systems and/or browsers. Nevertheless, this view can be exploited to highlight some interesting aspects such as technology compatibility and the use of third-party software as we will explain in the dedicated section.

Now, what about the +1? This refers to a few selected use cases, or *scenarios*, which are used to check if the other 4 views work in harmony. This viewpoint is relevant for Bokeh as it is for any other software, since it helps with the validation and illustration of software design.[15]

Although not considered by Kruchten, there is an extension to the logical view that can be relevant: in a recent AMA given by Grady Booch, the growing interest in data as a central point in a system's design

[11] P. B. Kruchten, "The 4+1 View Model of architecture," in IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995.
[12] P. B. Kruchten, "The 4+1 View Model of architecture," in IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995.
[13] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.
[14] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.
[15] P. B. Kruchten, "The 4+1 View Model of architecture," in IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995.

was highlighted. As data becomes more and more important in our society, system's architectural elements should reflect the need for exchanging, understanding and representing it. In the context of our project, it makes even more sense since Bokeh lives on data.

5.3.2 Patterns in Bokeh's architecture

> "The purpose of a software pattern is to share a proven, widely applicable solution to a particular design problem in a standard form that allows it to be easily reused."[16]

Bokeh is no different from other widely used software systems. It also follows some architectural patterns. Bokeh's architecture facilitates an easy manipulation of the components and configuration of a plot from server-side code in Python or other languages. Furthermore, BokehJS can also be used directly as a standalone JavaScript library, with plot data embedded directly into the page, retrieved via AJAX calls, or supplied by a separate Bokeh Plot server[17].

We can easily identify the client/server design pattern:

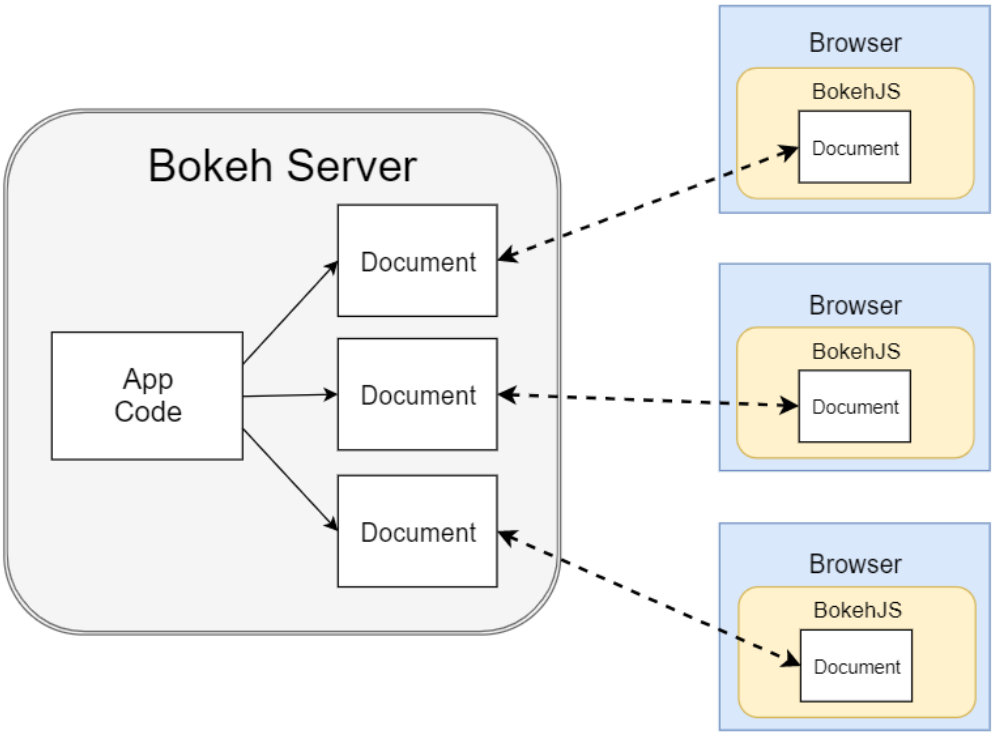


Figure 5.5: Bokeh Server-Browser interaction

A Bokeh server (left) uses Application code to create Bokeh Documents. Every new connection from

[16] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[17] https://github.com/bokeh/bokeh/blob/master/bokehjs/README.md

a browser (right) results in the server creating a new `document`, just for that session. The capability to synchronize between Python and the browser is the main purpose of this server.

The design pattern that allows this synchronicity is called **observer pattern**. We can think of Bokeh Server as the observer that is watching for all the sessions, here named the subjects. Bokeh can work as a simple and straightforward Python library that can display the plots or output them to an `.html` file.

But there is more. Bokeh uses delegation extensively for policies. The **delegation pattern** is a object-oriented design pattern that allows an object to handle a request by delegating it to another object.[18]

```
class Rectangle(val width: Int, val height: Int) {
    fun area() = width * height
}

class Window(val bounds: Rectangle) {
    // Delegation
    fun area() = bounds.area()
}
```

Figure 5.6: The Window class delegates the area computation to the rectangle.

In addition, Bokeh follows the **visitor design pattern** for processing object graphs[19]. In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. The algorithm is converted to a class of its own and visits the object structure where it is executed.

Bokeh follows other patterns as well, such as the **websocket protocol being implemented via explicit state transitions** and **proxies in various places**, yet explaining them goes beyond the scope of this blog post.

5.3.3 Developing Bokeh

Now, lets look at the files that define Bokeh.

Developers encapsulate source files with related code in modules, which are studied in detail along with their dependencies. This way developers know how to work on some parts of the code without affecting in an unintended way the remaining, safeguarding code maintainability and extensibility.

Note that, due to the sheer amount of dependencies and modules, not all of them could be visualized in readable manner, so we chose to only include the most important ones in the figure above.

Starting from the top, we have the **django** (purple) module (here we use django as a representation, since Bokeh allows for the usage of other web frameworks such as Flask). This module is highly dependent on a lot of different modules. This happens because it is possible to embed a Bokeh application into another web application, which, in its turn, highly depends on code that is partitioned in different modules.

[18] https://en.wikipedia.org/wiki/Delegation_pattern
[19] https://docs.bokeh.org/en/latest/docs/reference/document.html

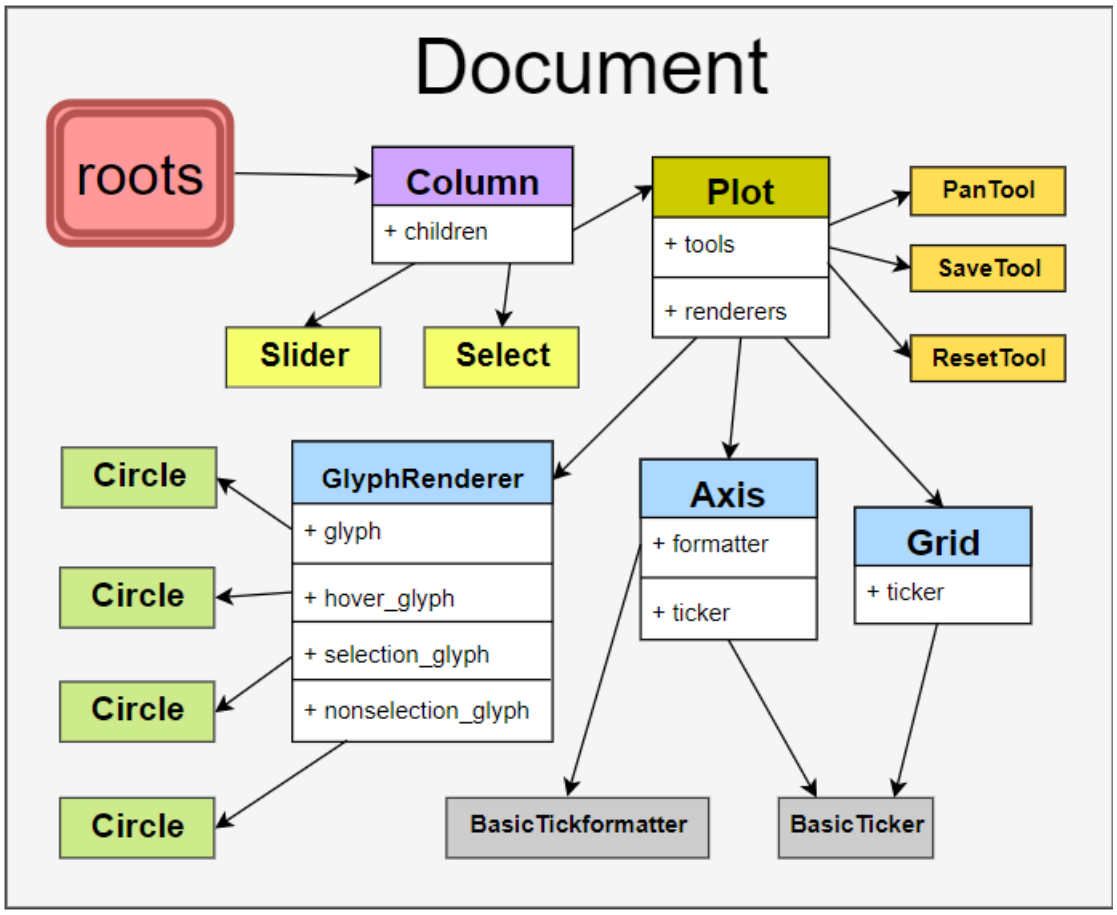


Figure 5.7: If you're wondering what is an object graph, there you go.

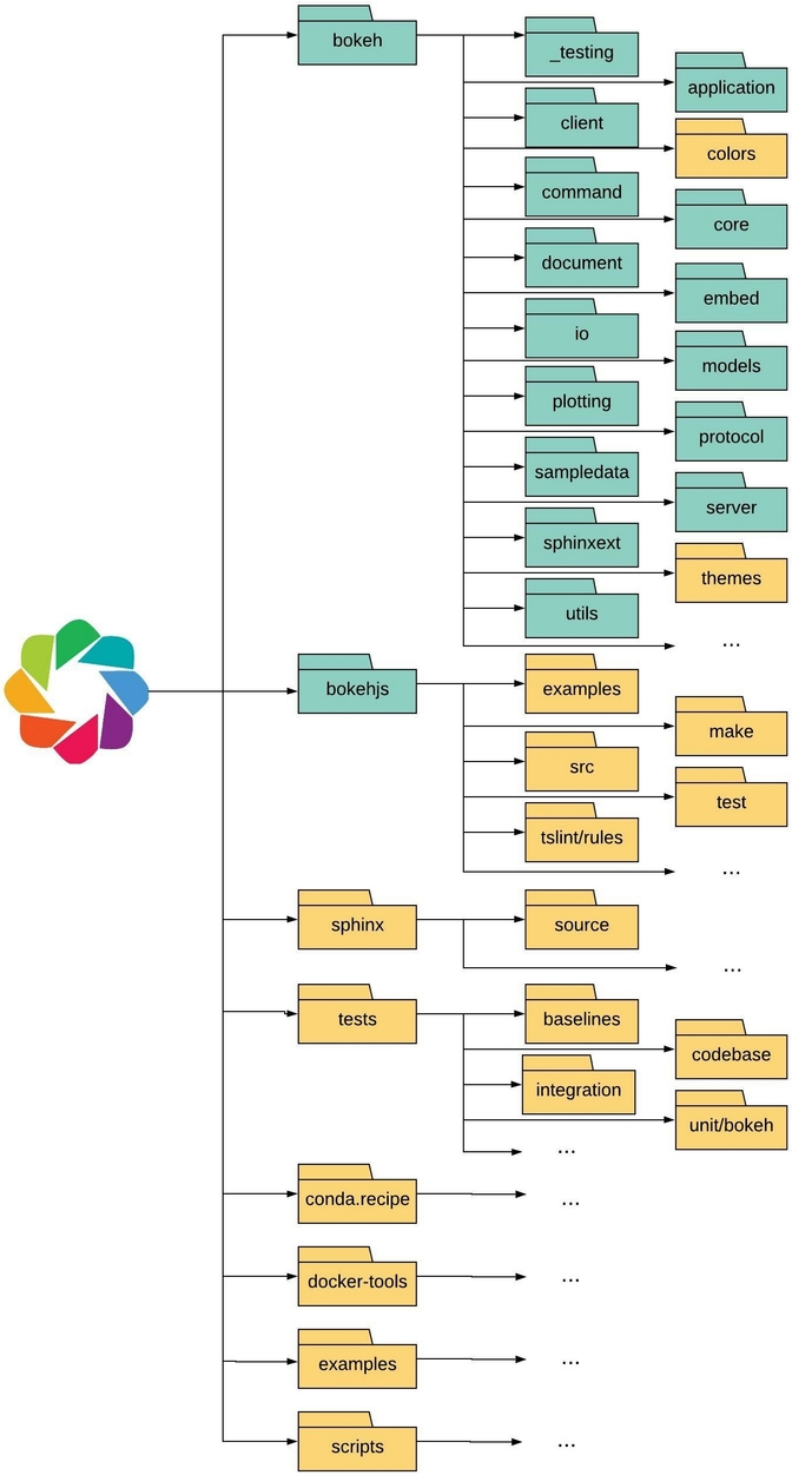


Figure 5.8: Codeline organization

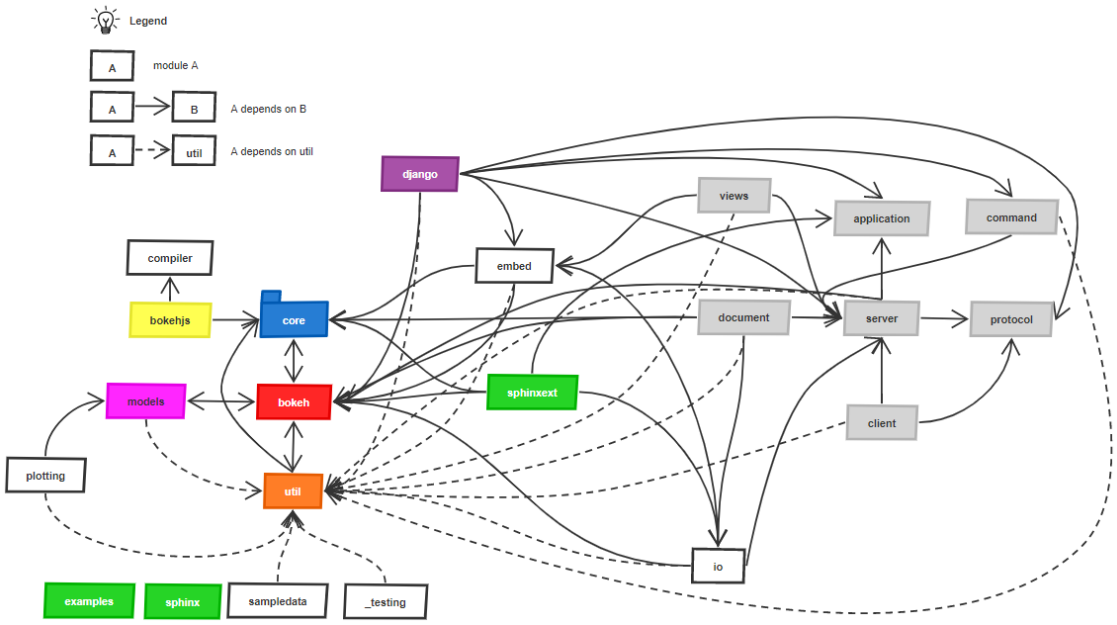


Figure 5.9: Bokeh's most important modules and their dependencies.

The **models** module (pink) provides the building block classes of Bokeh, the Models[20]: graphs, plots, axes, ranges, scales, widgets, etc. These low-level objects comprise a Bokeh object graph.

Next, we have **core** (blue), which is not truly a module but rather a package that represents an aggregation of modules used to implement Bokeh itself[21]. For example, the **validation** module can be used to perform integrity checks on an entire collection of Bokeh Models[22], while the **property.mixins** module provides the classes used to group properties of the Models.[23]

The **util** module (orange) provides a collection of utilities used to implement Bokeh's functionalities[24], be that functions to call browsers; JS compilation; dependency checking; etc. It is interesting to note that almost all of the most important modules depend on **util**.

The modules **sphinx**, **sphinxext** and **examples** (green) are relevant for documentation purposes.[25][26]

Bokeh (red) can be considered the top level module in the project. It contains useful functions and attributes, such as the *license* and current *version* of the project.[27] Note here that several modules have dependencies towards **bokeh** and it has dependencies towards them. This implies that the modules depend on each other.

The modules colored in gray are responsible for the client-server application. For example, the **document**

[20] https://docs.bokeh.org/en/latest/docs/dev_guide/models.html
[21] https://docs.bokeh.org/en/latest/docs/reference/core.html
[22] https://docs.bokeh.org/en/latest/docs/reference/core/validation.html#bokeh-core-validation
[23] https://docs.bokeh.org/en/2.0.0/docs/reference/core/property_mixins.html#module-bokeh.core.property_mixins
[24] https://docs.bokeh.org/en/latest/docs/reference/util.html
[25] https://docs.bokeh.org/en/latest/docs/dev_guide/documentation.html
[26] https://docs.bokeh.org/en/latest/docs/reference/sphinxext.html
[27] https://docs.bokeh.org/en/latest/docs/reference/0_bokeh.html

module is a container for Bokeh Models to be reflected on the client side BokehJS library (yellow)[28]; while the **protocol** module provides message protocols for communication between Bokeh Servers and clients.[29]

On a general note, this graph allow us to conclude that there is a tight coupling between Bokeh's components. There are also several cases of component entanglement and circular dependencies (**util** to **bokeh** to **models** to **util**).

When building the development view, it is also important to define system-wide standards to ensure technical integrity.[30] Bokeh does that in all sorts of ways. For example, the core team developed guidelines to manage issues and Pull Requests on Github.

5.3.4 Running Bokeh

One may still ask, how does everything really connect and work? Is Bokeh really a well-oiled machine?

Bokeh server can be served from a multitude of frameworks, be it `Django`, `Flask` or the default `Tornado` Framework. To serve Bokeh as a server all that you have to do is `bokeh serve {yourdocument}.py`, and a Tornado web server will start serving your document to the interwebs or just to your local network. Here, we have one for you.

This interaction between Bokeh and BokehJS is one of the things that make this open-source project so interesting.

First, the application code computes the `Document`, creating the object graph. Next, this graph is serialized and sent to BokehJS where it is deserialized and rendered. The application code is executed in the Bokeh server every time a new connection is made. The application code also sets up any callbacks that should be run whenever properties such as widget values are changed.

This allows scalability, concurrency and distribution. In a *perfect world*, one server can `serve` it all!

5.3.5 Deploying Bokeh

As Bokeh is a Python library, its execution only depends on the end-user running a compatible version of Python (3.6+). There is no cloud-based runtime dependencies but it does have system dependencies in order to correctly run locally[31]. For the latest version (2.0.0) we can identify the following Python libraries as dependencies for basic usage as well as optional dependencies needed in order to use all features available, and testing modules:

One may tend to ignore the importance of testing here. However, in a recent interview we did with the core team member Bryan Van de Ven, he actually identified testing as one of Bokeh's production main challenges:

> "For me personally one of the hardest areas is simply the enormous test surface for a project like Bokeh. If you look at the entire matrix of Python version x OS version x Browser version x Jupyter version x Tornado version x... It's just vastly more than we have resources to run real tests for. Things continue to improve as we are able to, but it's an ongoing challenge."

[28] https://docs.bokeh.org/en/latest/docs/reference/document.html
[29] https://docs.bokeh.org/en/latest/docs/reference/protocol.html#module-bokeh.protocol
[30] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.
[31] https://docs.bokeh.org/en/latest/docs/installation.html

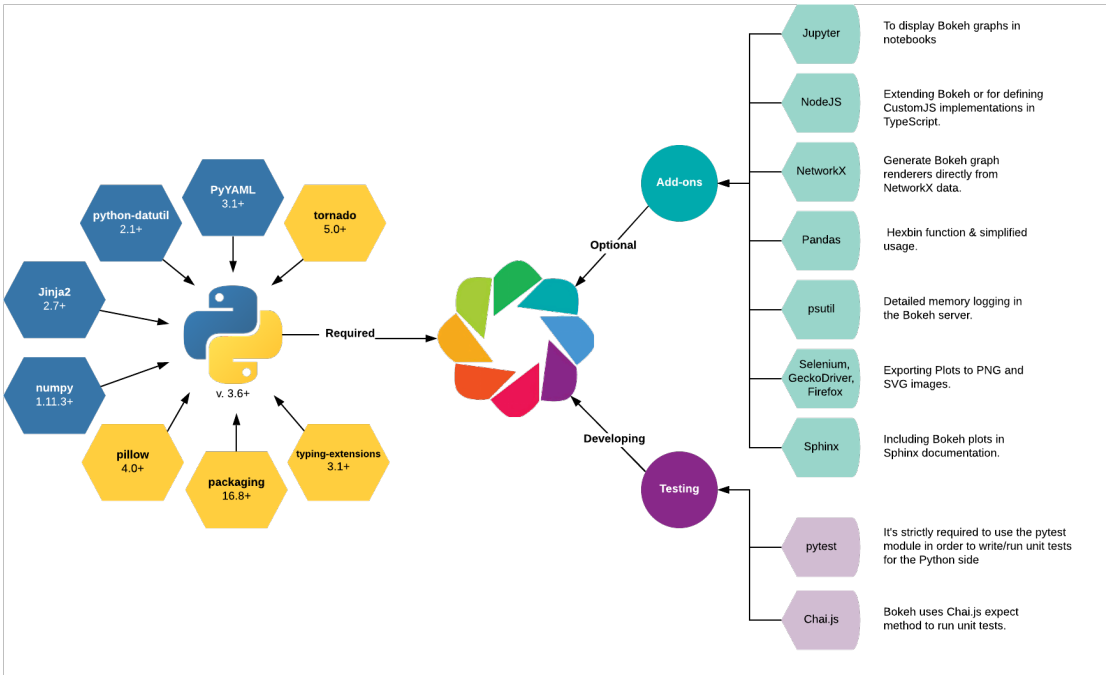


Figure 5.10: Bokeh's dependency graph

As of continuous integration, until the latest release Bokeh made use of GitHub-CI to run a full test build in every pull Request to master, but for version 2.0.0 a feature was added to include Subresource integrity, and that made developers build and upload part of deployment manually.

5.3.6 Not only about plotting

Bokeh is not just a data visualization library, but also a very good one. One way of measuring such claim is to look into its non-functional properties. We have identified the following areas in which Bokeh excels:

Feature	Description
Open Source & Transparency	As an open source system, Bokeh gets users involved in the development process, which allows the team to know where to center their efforts after listening to the end-user's demands directly. Also, it allows the amount of contributors to rise substantially. Open Source bring along great transparency and communication, and Bokeh really embraces the open source spirit by allowing anyone to enter their zulip channel, where there is constant interaction between contributors and the core team members.
Performance	Bokeh also has a very good performance in comparison to other standard Python data visualization libraries. In the first section of this notebook, we have made a simple, yet very conclusive test about how much better the performance is. What we did was plot 3 simple functions using both MatPlotLib and Bokeh, and time the processing and rendering time of each library. The results state that Bokeh is ~3x faster to process and show data, making it pretty safe to say that, at least in simple cases, it has a better performance.
Interactivity	In Part 2 of our notebook, you will realize that adding interactive tools to a plot is very simple. In this example we added basic mouse navigation events such as panning and zooming, and a save button to download your plot.
Readability	Plotting with Bokeh is very easy: variables and functions have very intuitive names, as well as methods and keyword arguments of functions. A good example of this is that in the previously mentioned notebook, to create a figure, you must call the `figure` function (Duh!), and if you want to add a label to, lets say, the X axis you only have to pass the keyword argument `x_axis_label="<label>"` (Duh!^2). And guess what keyword argument you must give to the figure in order to add tools to it! Yup, you guess it right! Tools. That is it. Bokeh can not get more straightforward than this!

Feature	Description
Documentation	The development team has gone the extra mile to document every aspect of Bokeh, even using Sphinx to generate HTML documentation for easy export. Interestingly, with the recent release of Bokeh 2.0, there has been an increasing difficulty in communicating all the changes brought to the library. Bryan Van de Ven addressed this issue on the interview, commenting on the fact that community support is an important complement to documentation: *"We have tried to document those changes [. . .] but there is no way to make sure users see or know about it, and I expect there will be an increased load of support questions for a long time."* It is clear then, that there is an interesting trade-off between the size and maturity of a project and the need for documentation, as Bryan, also underlined: *"once a project reaches a certainly level of size/maturity, all the important and hard problems are not technical ones, they are people/community ones."*
Security	Bokeh added on their latest release Subresource Integrity, which is "a security feature that enables browsers to verify that resources they fetch (for example, from a CDN) are delivered without unexpected manipulation"[32] making all of their resources more secure, however, there is a trade-off involved. As we interviewed Bryan Van de Ven, he brought up the fact that this integration made the release of Bokeh a little harder, or less automatic, as now developers would need to build and test locally before releasing.

That's it! Hopefully you have enjoyed this insight in how the architecture of Bokeh is *plotted* and how its various components are *drawn* together.

5.4 Plotting Bokeh, an Analysis of its Quality

Testing is the key to success. Quality is never an accident. All code is guilty, until proven innocent. Folk knowledge in Software Engineering is full of sayings like these ones. Nevertheless, we still live in a world where code quality is most of the times undervalued. The importance of testing, refactoring and evaluating technical debt should however not be neglected, since they are of major importance to safeguard code quality and architectural integrity.

The fact that open-source projects live from community contributions, as also reinforced by the core team member Bryan Van de Ven in a podcast, makes things harder.

> there's a lot of things to do and presently, not enough people to do them,

If, on one hand, it is evident that contributing for new features in open source projects is essential (check our roadmap in essay 1 for more information about those juicy features!), on the other hand, it is important to follow contribution guidelines and templates, and it is even more fundamental to adopt good Testing and Code Quality practices.

[32] https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

5.4.1 Bokeh and its quality processes

Quality is a must. Bokeh is no exception. In order to enforce this quality, only a hand-full (the core team) of developers are allowed to merge new code to the master branch, which means that every line of code is evaluated and checked whether it follows the core team guidelines. This helps the software to maintain a certain level and a coherent code style.

The core-team is also very active within the community, discussing all the issues and searching for the best solutions. These constant interactions allows trust to be built between the community and the core-team, and trust generates confidence and promotes better discussions, which in turn leads to the best possible outcome. Code cannot be separated from the people making it.

As Sara Mei's said in her keynote on "livable code" at RailsConf 2018[33]:

> "Build trust among team members and strengthen their connection. The code base will follow"

Bokeh, besides having the core-team as the *quality judges*, also uses LGTM, which is an awesome automated tool that evaluates code quality, has an unparalleled security analysis, automates code review and executes a deep semantic code search. With this tool, Bokeh prevents bugs from being merged but also grades its code quality comparatively to other projects.

Oh, did you think that was it? No. There is more!

Bokeh uses Github Continuous Integration, which means that they continuously integrate! What are they integrating you ask? Code, Tests, Deployment! Every time someone pushes something to the master branch at Github, the CI is activated and everything is tested again.[34]

This way, Bokeh can assure code quality.

5.4.2 Bokeh never stops: a look into the CI processes

Github CI, or Github Actions as they named it, automates Bokeh's workflow, easing the development process. This tool automatically builds to Windows, Ubuntu, macOS but also runs a plethora of tests making sure that *almost* nothing is broken!

Every push to the master branch or any Pull Request branch on GitHub automatically triggers a full test build on the Github Continuous Integration service.[35] You can check a list of the current and previous builds here.

Just a small glimpse of whats going on behind the curtains:

5.4.3 Testing Bokeh, the importance of tests and how to build a better future

It is fair to say that testing is not an easy job. In fact, Bryan stated in a recent interview conducted by us that testing is one of the hardest areas on the project.

> [...] one of the hardest areas is simply the enormous test surface for a project like Bokeh. If you look at the entire matrix of Python version x OS version x Browser version x Jupyter version x Tornado version x... It's just vastly more than we have resources to run real tests for. Things continue to improve as we are able to, but it's an ongoing challenge.

[33] https://www.youtube.com/watch?v=lI77oMKr5EY
[34] https://docs.bokeh.org/en/latest/docs/dev_guide/testing.html#continuous-integration
[35] https://docs.bokeh.org/en/latest/docs/dev_guide/testing.html#continuous-integration

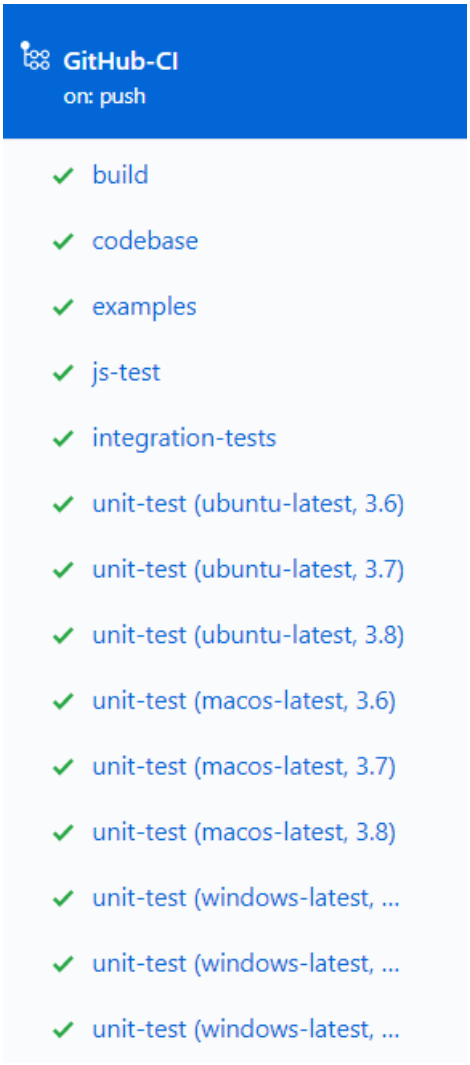


Figure 5.11: Github CI test checks

It is not surprising to say then that Bokeh relies heavily on user input, as shown in the figure below. Users contribute mainly through visual and manual testing, reporting the issues they find while running their applications.[36][37][38]

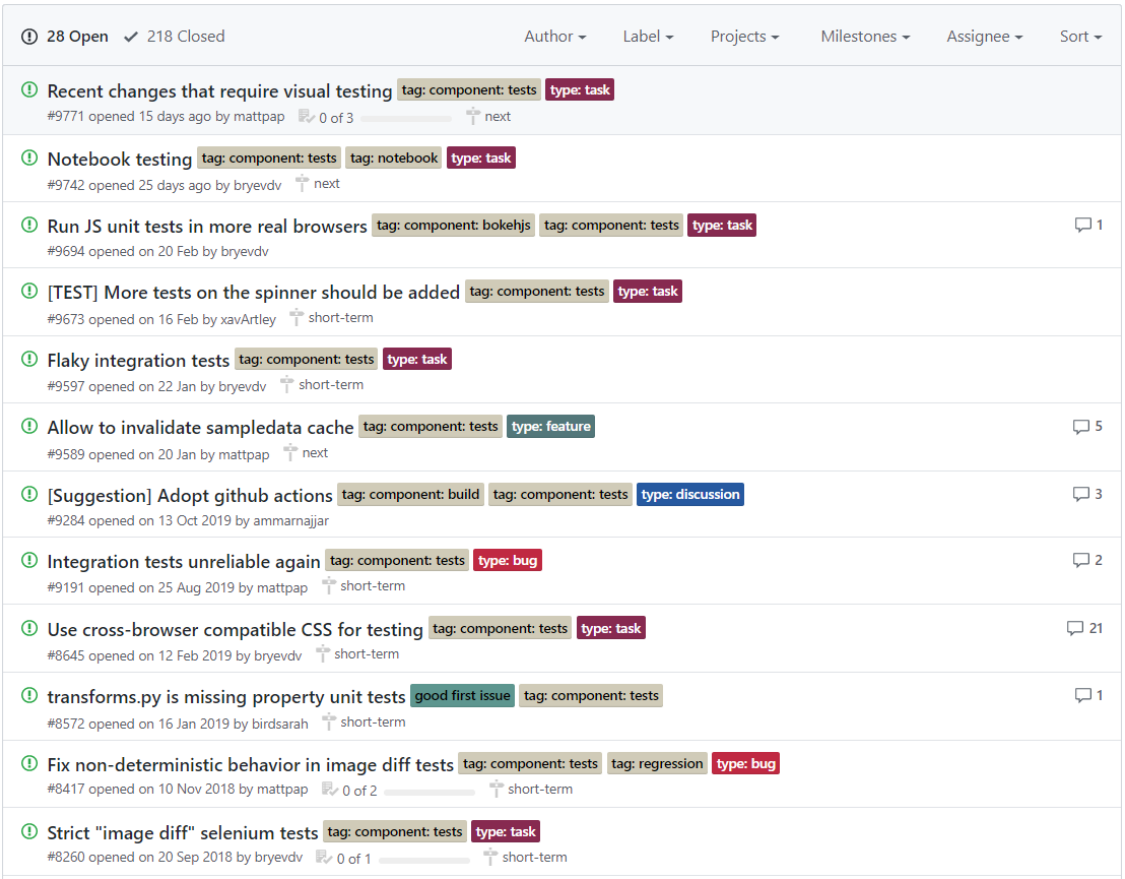


Figure 5.12: Label tag

However, a large and multi-language system like Bokeh cannot rely only on those kinds of tests to maintain capability and prevent regressions. For the past months the team has been asking for improvements in their automated testing process and encouraging contributions from the community, as shown in the figure below. The developer guide dedicates a whole page to testing. The core team mentality about the importance of tests is clearly stated there[39]:

> In order to help keep Bokeh maintainable, all Pull Requests that touch code should normally be accompanied by relevant tests [...] a Pull Request without tests may not be merged.

[36] https://github.com/bokeh/bokeh/issues/9724
[37] https://github.com/bokeh/bokeh/issues/9522
[38] https://github.com/bokeh/bokeh/issues/9703
[39] https://docs.bokeh.org/en/latest/docs/dev_guide/testing.html#writing-tests

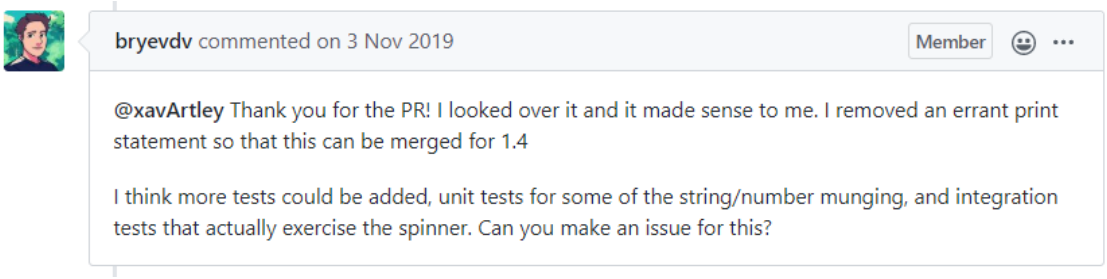


Figure 5.13: Bokeh encourages community participation

Besides the *"no test, no merge*" policy for new features, the team is also concerned with the quality of tests[40] even though it is not a priority.

Recently, all the test files were aggregated in a single `tests` directory.[41] The `tests` directory consists of Python unit tests, JavaScript unit tests, browser integration tests, codebase tests and example tests, grouping 41.7k lines of code, corresponding to 30.0% of the number of code lines in the project, 139.1k[42]. For instance, the example tests, among other things, check for image differences and can take 30 minutes to run.[43]

Unfortunately, Bokeh doesn't offer information regarding code coverage and the matter is not covered in recent issues nor Pull Requests.

5.4.4 Bokeh's hot activity

We used CodeScene to figure out the hotspots of Bokeh. Our guess was that the most important components were constantly changing, refactored or having its bugs fixed. And we were right! The hottest spot of Bokeh is `document.py` which is a core component of Bokeh's architecture: this Python file describes how a document works, which is "nothing" more than a collection of widgets and plots that can be served to users. `document.py` reigns on top but `figure.py`, `annotations.py`, `tools.py` and some other components also have a quite nice activity going on, and again, all of those are core components to Bokeh since combined they are what defines a Bokeh's plot.

This shows that Bokeh is active and looking forward to get better each day focusing on what it really matters!

We also noticed that there were several modules within BokehJS that have some activity going on, which makes perfect sense because, as you will see in the next section, BokehJS is the component that will suffer the most changes in the future.

5.4.5 From the architecture to the roadmap

There is always room for improvement and, as we mentioned in the roadmap of our first blog post, Bokeh has a great plan for the future.

[40] https://github.com/bokeh/bokeh/pull/9729#issuecomment-599622371
[41] https://github.com/bokeh/bokeh/issues/9533#issuecomment-567567036
[42] https://lgtm.com/projects/g/bokeh/bokeh/latest/files/?sort=name&dir=ASC&mode=heatmap&showExcluded=true
[43] https://discourse.bokeh.org/t/running-the-tests-locally/4996

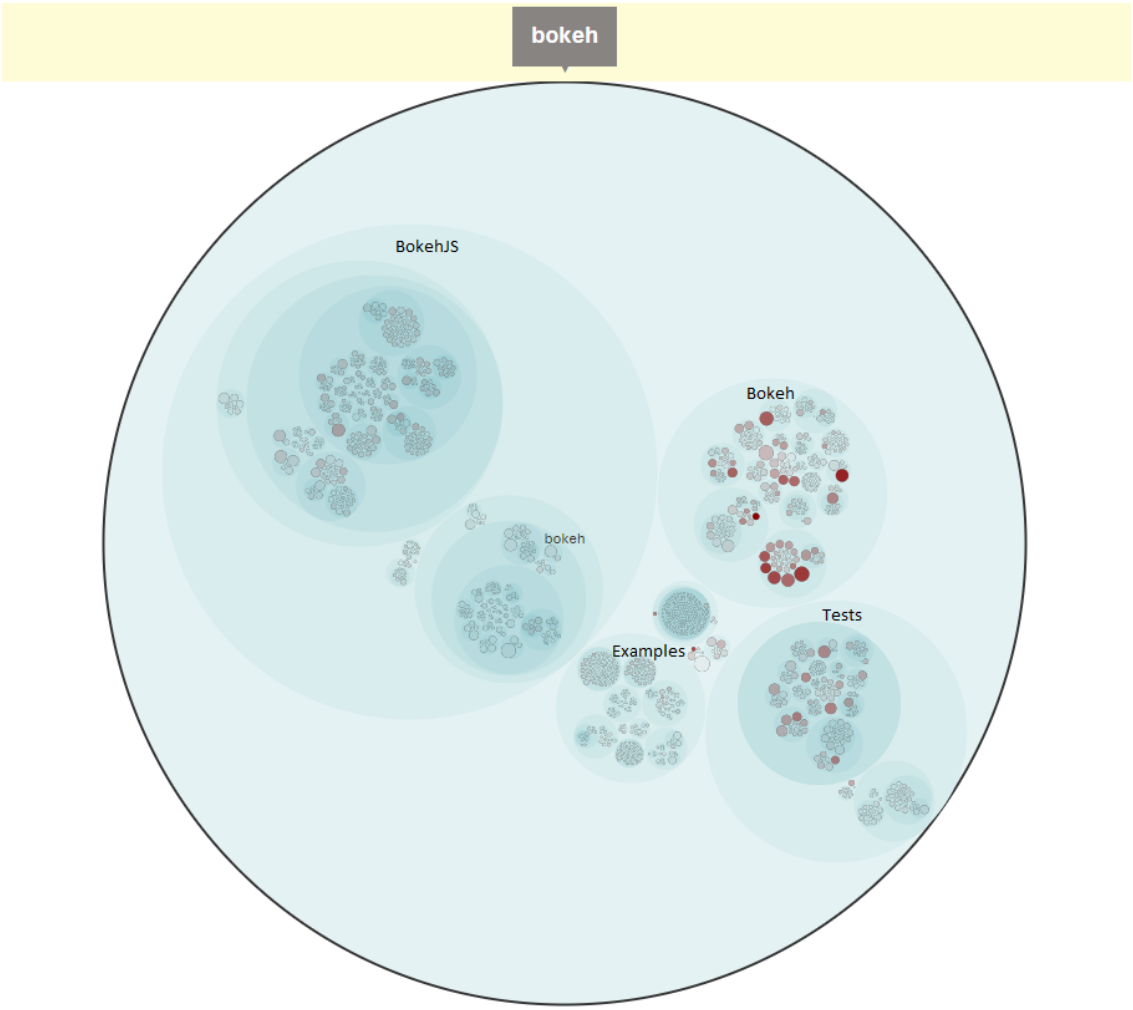


Figure 5.14: Hotspots: Red means hot!

The developers are highly concerned about improving BokehJS[44][45], as they want to develop BokehJS as a first-class JavaScript library. The core team expects this to attract a wider pool of contributors interested in maintaining the project[46] and helping with the development of new features, such as better animations, transitions and events.

We expect that in the next months BokehJS module gets a lot of activity, since, as shown from the references above, the code is not yet ready for all the changes.

5.4.6 A look into Bokeh's code quality

We already know that Bokeh has two independent parts that work together to create awesome stuff: Bokeh and BokehJS. In order to get started with this section, we must look into the structure and code distribution of both, along with how the code is maintained and finally describe the quality of the code regarding these concepts.

First off, we start with a dependency graph of the **main components** in Bokeh repository:

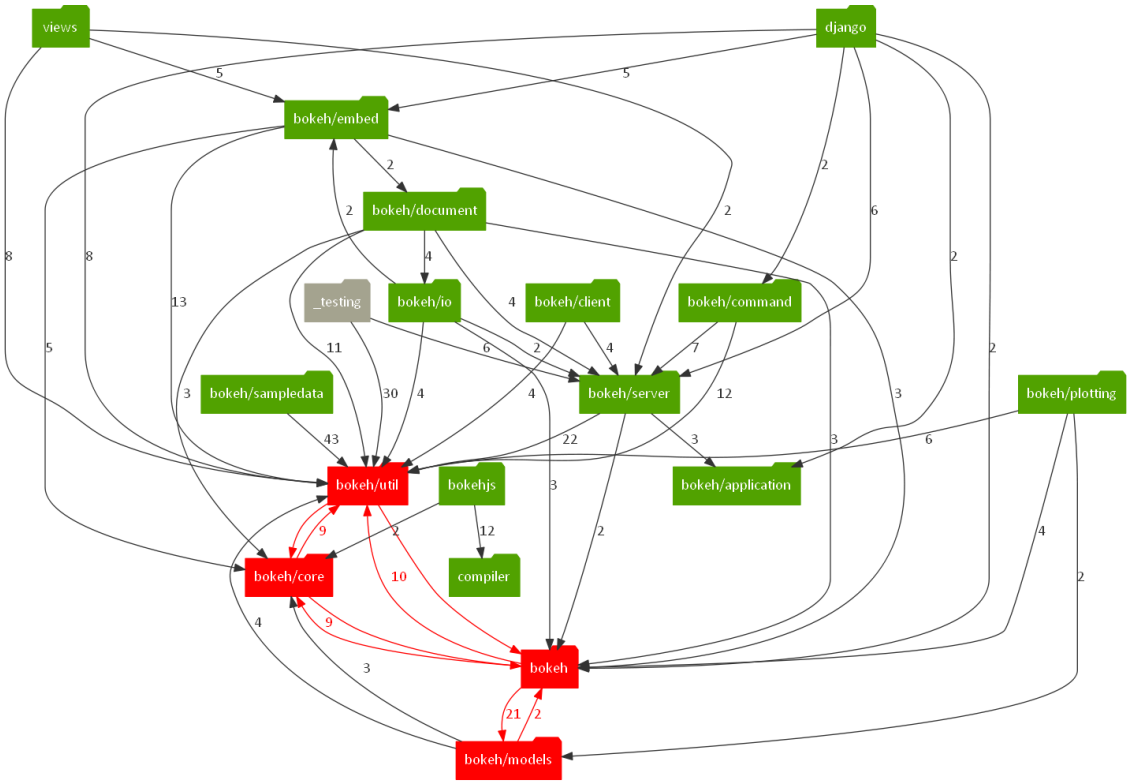


Figure 5.15: Bokeh's main modules dependency graph

With this we can discuss one of the concepts regarding code quality and maintainability: **cyclic dependen-**

[44] https://github.com/bokeh/bokeh/pull/9615
[45] https://github.com/bokeh/bokeh/pull/9808
[46] https://bokeh.org/roadmap/

cies. These occur when two modules/components depend on each other to work, that is, there are calls from *module 1* inside *module 2* and vice-versa. In the previous graph representation, we have identified 4 modules that suffer from this (colored in red). Perhaps the most serious case would be the `bokeh` module (Python side), that has cyclic dependencies with 3 different modules, so a change into this module may trigger errors in the modules `bokeh/util`, `bokeh/core`, `bokeh/models`, and also a change in any of these 3 may trigger an error in `bokeh` as well.

Another important issue to address is **code recycling**. This refers to when a chunk of code is present in many modules, and in order to give more consistency and avoid code repetition, a new module is created where that chunk is called from whenever needed. A very recent issue has been brought up regarding this in the BokehJS module, and Bokeh team is trying to improve this for future releases. Here one can appreciate that the team is concerned about their code quality, imposing code changes to improve it whenever possible. Other examples related to this are the introduction of global terminologies to help achieve an organizational standard, and further resources redistribution to make bokeh more managable, as can be seen in this issue.

Another interesting metric is how many Lines of Code (LOC) are destined for testing. The following statistics are taken from LGMT:

Code density from root (/):

Component	LOC (amount)	LOC (aprox %)
bokeh	40.8k	29.3%
bokehjs/test	15.6k	11.2%
bokehjs/< others >	39.7k	28.5%
conda.recipe	2	~0%
docker-tools	30	~0%
examples	11.7k	8.4%
scripts	1.4k	1%
sphinx	3.4k	2.4%
tests	26.1k	18.8%

total 139187 100%

disclaimer: the test module is Python related and bokehjs/test is JS related

This gives us a total of 41.7k LOC (30%) of code destined only for testing, which reflects the efforts of Bokeh's team to have a well tested developing environment and stable releases.

LGMT also provides a comparison tool to check how Bokeh performs in terms of code quality to similar Python and JavaScript projects. In both cases Bokeh was graded with an A+, being on top of similar projects, such as Ansible (this project is also being studyied in this course: you can learn more about Ansible here).

5.4.7 Refactoring Bokeh

In order to get more insight about where should refactoring be made, we used SonarQube, and the results were rather surprising.

SonarQube identifies 3 types of refactoring candidates (Bugs, Vulnerability and Code Smell), and separates them in 4 different severity levels (Blocker, Critical, Major and Minor).

5.4.7.1 Bugs

Here we have detected 48 different bugs that could use refactoring in order to improve code quality. Most of them are suggestions, and not many of them apply, but some are really amazing. The most severe (blocker) case found (twice) is in the following code piece:

```
def __init__(self, *args, **kwargs):
    return super().__init__(*args, **kwargs)
```

Remove this return value. Why is this an issue? 5 months ago L198
Bug Blocker Open Not assigned 5min effort Comment No tags

Figure 5.16: Remove return Bug

> "By contract, every Python function returns something, even if it's the None value, which can be returned implicitly by omitting the return statement, or explicitly. The **init** method is required to return None. A TypeError will be raised if the **init** method either yields or returns any expression other than None. Returning some expression that evaluates to None will not raise an error, but is considered bad practice."

The solution is as simple as to remove the `return` word and code should work.

5.4.7.2 Vulnerability

This refers to any security vulnerability that the code may have. Here we only have 5 issues/warnings, of which 4 are the same.

1. Phishing (Blocker)

```
<div class="bk-root">
    <a href="https://bokeh.org" target="_blank" class="bk-logo bk-logo-small bk-logo-notebook"></a>
```

Add rel="noopener noreferrer" to this link to prevent the original page from being modified by the opened link. Why is this an issue? 7 months ago L25
Vulnerability Blocker Open Not assigned 1min effort Comment phishing

```
    <span id="{{ element_id }}">Loading BokehJS ...</span>
```

Figure 5.17: Noreferrer Bug

> "When a link opens a URL in a new tab with target="_blank", it is very simple for the opened page to change the location of the original page because the JavaScript variable window.opener is not null and thus"window.opener.location can be set by the opened page. This exposes the user to very simple phishing attacks"

   The solution, again is very simple, we only need to add `rel="noopener noreferrer"` to the *a* tag. (`noopener` for newer browsers and `noreferrer` for older)

2. Jinja Autoescaping

   > Escaping HTML from template variables prevents switching into any execution context, like < script >. Disabling auto escaping forces developers to manually escape each template



Figure 5.18: Autoescaping Bug

variable for the application to be safe. A successful exploitation of a cross-site-scripting vulnerability by an attacker allow him to execute malicious JavaScript code in a user's web browser. The most severe XSS attacks involve:

- Forced redirection
- Modify presentation of content
- User accounts takeover after disclosure of sensitive information like session cookies or passwords

The solution? just add the `autoescape=True` *kwarg* to the `Enviroment` instantiation. Alternatively, one could specify where the html would be able to auto escape.

5.4.7.3 Code smells

These are mostly not-so-good coding practices that could be easily improved. Most of them correspond to cognitive complexity, and aim to reduce it by further modularization or code simplification. One example of this would be:

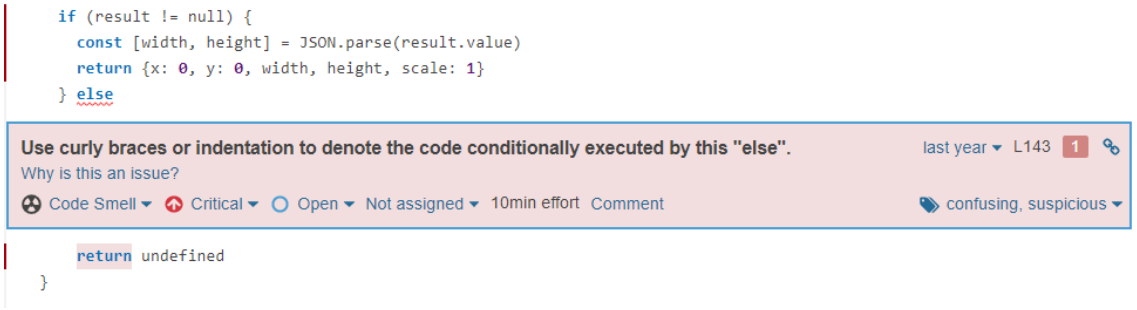


Figure 5.19: Curly Braces Bug

"In the absence of enclosing curly braces, the line immediately after a conditional is the one that is conditionally executed. By both convention and good practice, such lines are indented. In the absence of both curly braces and indentation the intent of the original programmer is entirely unclear and perhaps not actually what is executed. Additionally, such code is highly likely to be confusing to maintainers."

The solution is very straightforward: add curly braces (`{}`)

The most severe (Blocker) issues, however, are:

1. Same Value return

Functions are meant to return different values according to the information they process (There's no point on having a function that always returns the same value). In `bokeh/core/property/visual.py` we see the following code:

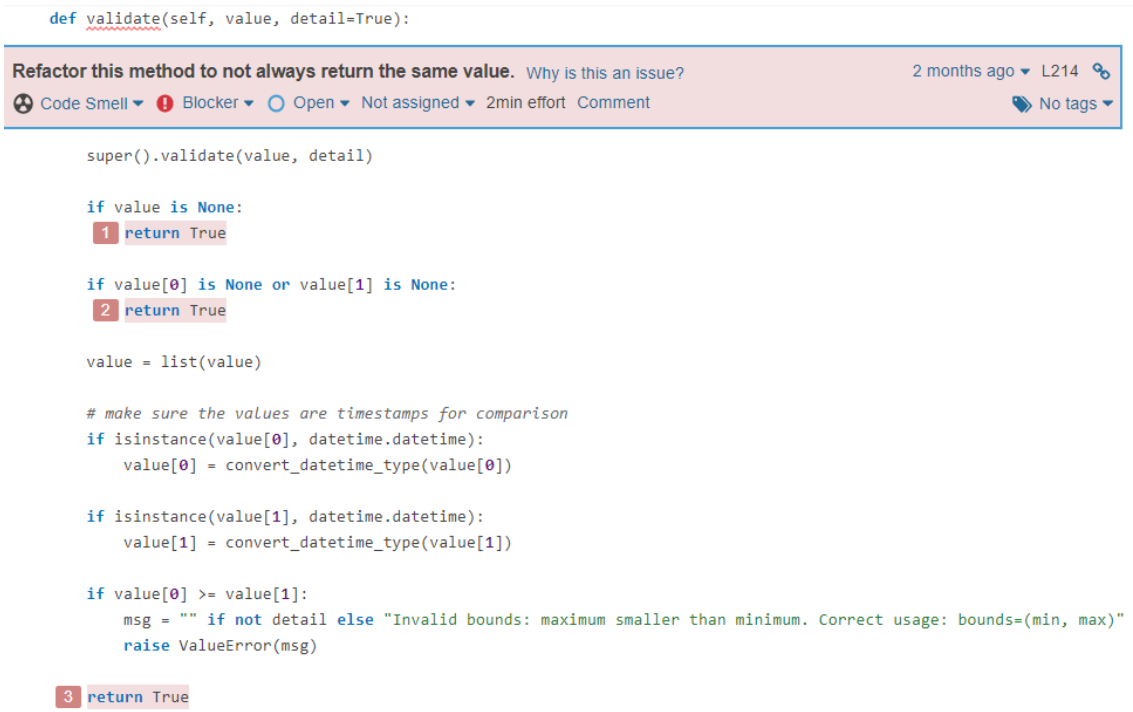


Figure 5.20: Same Return Bug

As we see into this issue, we can realize that the return value doesn't really matter, but the goal here is to whether the function raises an exception and the return value is unnecessary. So we will not take into account SonarQube suggestion, but rather propose our own solution: the most logic option would be to replace all of this with a `try/except` expression, and change the 2nd and 3rd `if` with `elif`, that way no explicit return value is required and if the conditions are satisfied, the Error will rise anyways.

2. Switch case break

   > When the execution is not explicitly terminated at the end of a switch case, it continues to execute the statements of the following case. While this is sometimes intentional, it often is a mistake which leads to unexpected behavior.

   In this case it clearly is not intentional, as the two cases are completely different, not related strings. So the correct thing would be to add a `break` statement right after `this.webgl = global_webgl`

With all of this said, we can see that there is still room for refactoring. Some issues may not have any substantial effect on the execution itself, but that doesn't mean they don't improve the code quality. It's good to see, however, that the issues we have detected are not that serious, and in most cases really easy to solve!

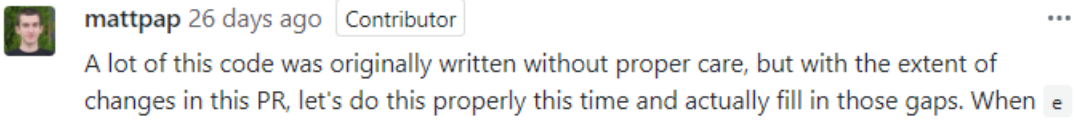


Figure 5.21: Switch Case Bug

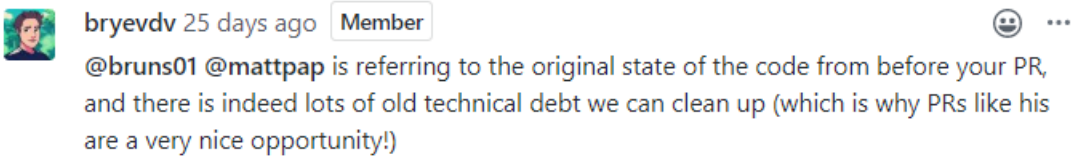
5.4.8 Is it time to pay… the technical debt?

Technical debt is a metaphor coined by Ward Cunningham that reflects the cost of additional rework by choosing a faster solution now, neglecting internal quality, instead of a better one that will imply more work.[47]

Pull Request #9732 gives an interesting overview on how technical debt works. In this PR a contributor suggested a new feature. One member of the core team commented that the code being changed was originally written without much care, as can be seen in the figure below.



The file was not refactored since it was originally written, as the extra effort was not worth the gain. Other member of the team realizes that they are paying this technical debt gradually, as more pull requests are opened.



[47] https://www.martinfowler.com/bliki/TechnicalDebt.html

Furthermore, recently Bokeh released version 2.0.0. This allowed the developers to shed some technical debt. Bryan commented on this issue in the interview:

> [...] 2.0 was the opposite of difficult: it was a great change to shed a lot of technical debt, dropping python 2 and python 3.5 support, e.g. allowed us to make many things much simpler, discard bothersome or outdated things, etc.

We hope to have convinced you that is it not only the foundations that matter, without maintenance and proper care, even the strongest pillar rots. One needs to safeguard the quality and architectural integrity of any software.

5.5 Plotting Bokeh, an Analysis of its Collaboration

Product development involves two, fundamental, elements: a technical one and a social one.[48] One can only dream of a successful project when both these components are harmoniously aligned. In previous essays we analysed both elements. We focused specially on Bokeh's technical side, studying its properties, processes, development, deployment, architecture, code quality and testing.[49][50][51] Nevertheless, we also shed some light on the social component, describing the importance of the different groups of individuals involved in the project.

In this last essay we will dive deeper into its social element, that is, how Bokeh as an organization handles collaboration and communication. Ultimately, this study will reflect on how and when the different individuals involved in the development process interact, and how can those interactions be used to enhance development productivity.

5.5.1 Why should we trust in communication?

The relationships between the software developers are as important as the relationships between the structure and the components of the software, since coordinating product design decisions requires good communication among the engineers who make and implement them.[52]

The core team is well aware of its importance, since the discussion on how communication should be handled has been a recurring matter. Core team member Mateusz Paprocki raises an important point in an issue comment.

[48] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf
[49] https://desosa2020.netlify.com/projects/bokeh/2020/03/09/product-vision.html
[50] https://desosa2020.netlify.com/projects/bokeh/2020/03/19/vision-to-architecture.html
[51] https://desosa2020.netlify.com/projects/bokeh/
[52] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf



mattpap commented 10 days ago Contributor

> As an aside, I'd really prefer to have this sort of conversation off GitHub, so I'd encourage you join the Zulip chat.

GitHub is exactly the right place for this kind of discussion, because it's an important and open one, and actually a discussion and not a chat. It also needs to be preserved, so that in some time, possibly years, people will be able to figure out why we ended up with bokeh that we ended up. The only historically reliable place to preserve such discussions is the repository, either in the issue tracker, PRs or the version controlled wiki. Anything else is a liability and good path towards either eventually or immediately (e.g. private E-mails) erasing valuable information. You can use a chat for issues that intrinsically require immediate back and forth, but any conclusions of value must be preserved in the repository. Additionally I'm not going to search multiple places to get answers I need. The only exception are private discussions, which unfortunately can't happen on GitHub. Though in that case discourse would be a preferred choice over a chat.

Mateusz argues that discussions on GitHub bring transparency for the whole community, and their role should not be undervalued. Software Development literature is also not indifferent to this issue, as stated by *Cataldo et al*.[53]

> The product development literature argues that information hiding, which leads to minimal communication between teams, causes variability in the evolution of projects, frequently resulting in integration problems.

Moreover, Mateusz expresses the importance of preserving valuable information such as discussions on design and architectural decisions in the fewest possible number of channels and tools. In fact, he acknowledges the importance of using a collaborative tool such as GitHub as a centralized communication device in order to preserve vital information, so nothing gets lost and everything can be widely and easily accessed.

Nevertheless, Bryan van de Ven, other member of the core team, voiced on several occasions[54][55] the importance of interpersonal interactions and overall communication as a way to inform and learn from each others.

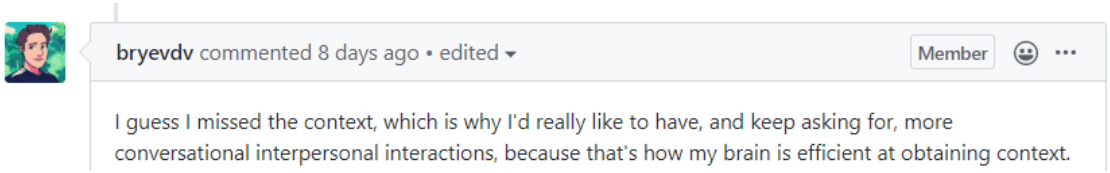


bryevdv commented 10 days ago • edited Member Author

> GitHub is exactly the right place for this kind of discussion, because it's an important and open one, and actually a discussion and not a chat

This ignores human concerns. Namely that I would sometimes just like to have a casual, exploratory, interpersonal conversation to hopefully learn from each other and inform our opinions, before getting to the point of long drawn out GH discussions, which I do not always have the stamina or desire for. Case in point.

[53] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf
[54] https://github.com/bokeh/bokeh/issues/9842#issuecomment-605669623
[55] https://github.com/bokeh/bokeh/issues/9858#issuecomment-606735672



Additionally, on a recent interview we conducted, Bryan stated:

> [...] once a project reaches a certainly level of size/maturity, all the important and hard problems are not technical ones, they are people/community ones.

We are aware of the fact that implementing good collaboration and communication practices is not an easy task for any organization. For an open source software it is even more challenging, as usually a main portion of the contributors is geographically distributed.

Ultimately, we argue that both types of communication discussed previously should not be underplayed and are equally important. However, we raise our concerns regarding the nature of that communication, and encourage transparency as a way to get users involved in the development process.

5.5.2 Collaboration analysis: the method

In this essay we will explore the communication and coordination patterns showed by Bokeh's developers, in order to assess development productivity and thus predict how resolution time of modification requests can be reduced. Moreover, as Conway's law recalls us, by studying the communication structure of the organization, one can make relevant inferences about the design of the structure of the final product.[56]

We will base our analysis on Pull Requests (PR) and Issues on GitHub, as it is the central tool in Bokeh's development. We know, however, that development-related communication is also done in Zulip, Discourse and private channels, as explained in our previous essay.

To carry out our analysis we resort to the Congruence Framework introduced by *Cataldo et al* based on the idea that in software development there are two components, the technical and the social one, which need to be aligned to have a successful project.[57]

Socio-technical congruence can then be defined as the match between the coordination requirements established by the dependencies among tasks and the actual coordination activities carried out by engineers.[58]

This highlights two important elements that combined originate congruence: the set given by the description of which individuals are working on which tasks and the set regarding the dependencies among tasks. Both of them can be modelled using matrices that, once combined, output the coordination requirements matrix, representing the extent to which each pair of developers needs to coordinate their work. Having this matrix, it is then possible to compute *congruence*, defined as the proportion of coordination activities that actually occurred relative to the total number of coordination activities that should have taken place.

Nonetheless, we are aware that the level of dynamism in an open source project, such as Bokeh, is different from the one studied in the paper. This inevitably forces adjustments to the framework explained above.

First of all, instead of analyzing which individuals are working on which tasks, we investigate which modules were changed by which developers. This is a necessary workaround, as Bokeh doesn't resort to

[56] https://en.wikipedia.org/wiki/Conway%27s_law
[57] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf
[58] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf

task assignment policies. Furthermore, analyzing module changes instead of file changes allow us to explore previous work. After all, we have commented on Bokeh's module structure before.

Secondly, when it comes to the study of relationships or dependencies among tasks, we apply 2 different methods:

1. We use the module coupling analysis provided by Sigrid and introduced on our previous essays;
2. We examine the set of modules that were changed together in a specific Pull Request, which we call Modules Changed Together (MCT) method, inspired by the Cataldo's Files Changed Together (FCT) method.[59]

The reason to complement the analysis of syntactic dependencies (in this case functional calls between different modules) with the MCT approach is outlined in the paper. *Cataldo et al* argue that functional dependencies analysis captures a narrow view of relationships among tasks. The motivation for MCT is also rather straightforward:

> when a modification request requires changes to more than one file, it can be assumed that decisions about the change to one file in a modification request depend in some way on the decisions made about changes to the other files involved in implementing the modification request.

Finally, to assess the number of coordination activities carried out by developers we analyse comments and reviews on Pull Requests and subsequent Issues referenced on the Pull Request. From now on we will refer to this combination as a *Task*. We define that a developer communicated with another developer if both commented on the same Task, regardless of whether they actually interacted directly, since we assume that developers read all comments on Tasks they also comment on.

All the data used in this study was collected using the GitHub API. We analysed every PR opened between the release of v1.0.0 (October 2018) until the release of v2.0.0 (March 2020) and all the issues related to each of those PRs. The analysed timespan is roughly 18 months of development. We supplement this study with plots, graphs and other visuals done in Bokeh - it is appropriate to use Bokeh to analyse Bokeh, right?

To be able to retrieve all of the necessary mentioned data we took the initiative, and engineered a tool. We built a pipeline that scraps the necessary data using the Github API, processes it, discarding what is not needed and computing what is relevant. The output of these processes is then directed to the front-end and displayed using figures created with Bokeh.

We (will) also make available the source code to reproduce these experiments. The interested reader can apply them to his/her favorite GitHub projects. Note that it is open-source, so we are more than happy to discuss improvements.

5.5.3 Collaboration analysis: the results

We start by doing an overall assessment of the data collected throughout the 18 months of development. During this period, we identified 700 PRs opened by 154 different collaborators. Figure 1 combines this numbers, showing how many PRs developers with more than 2 PRs opened.

In addition, we identified a total number of 4876 comments on Tasks, and divided this number by developer.

We also collected data on how many reviews each developer did (Figure 2), on a total of 549 reviews.

[59] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf

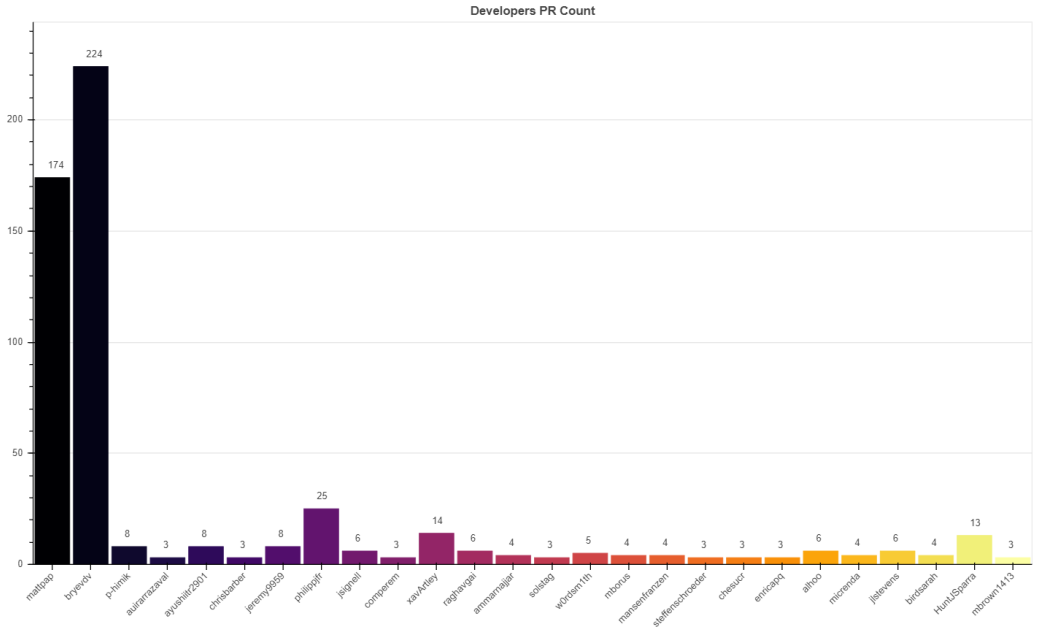


Figure 5.22: 1. PRs by developer

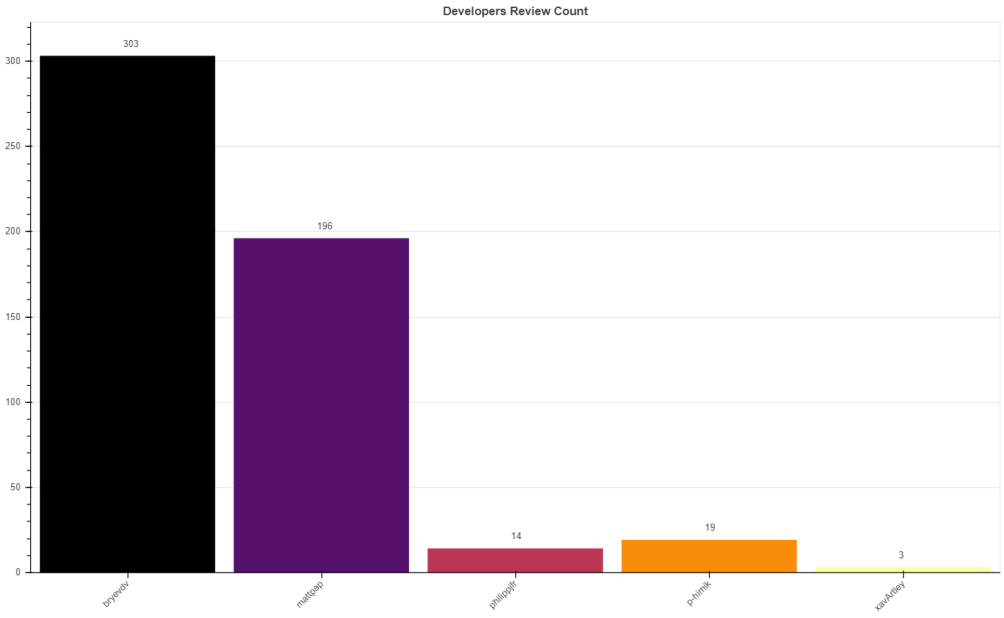


Figure 5.23: 2. Reviews by Developer

Finally we investigated how many times every each of the 54 considered modules was changed.

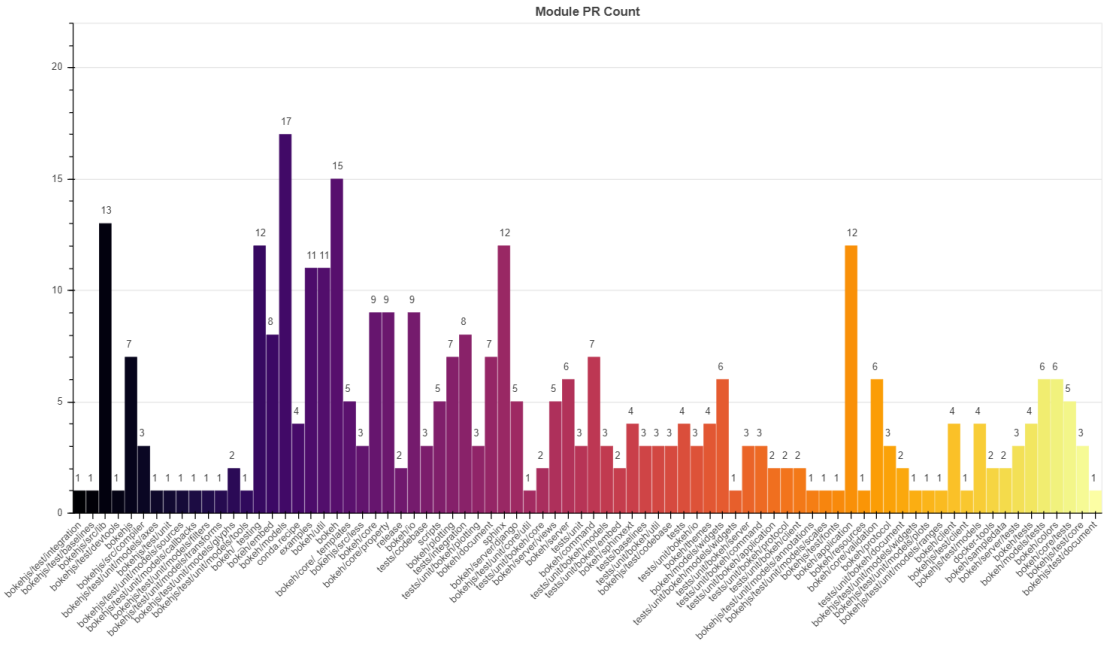


Figure 5.24: 3. PRs by Module

Thanks to this dataset, we were able to build the matrices according to the procedure explained in the previous section as well as the coordination requirements matrix. This matrix compared against the actual coordination activities matrix, provides a measure of congruence.

We start by analyzing a developer by modules matrix (Figures 4 and 5). Each cell *ij* indicates the number of changes that the individual *i* did to module *j*. We refer to this matrix as *Ta*.

At a first look we can see how a small set of developers is in charge of a large portion of the changes that took place during the 18 months.

The members of the core team, Mateusz and Bryan together touched on every module. Mateusz gave more contributions to BokehJS (Javascript) while Bryan to Bokeh (Python). They are the main contributors to the project and its principals pillars.

Besides **Bokeh** module, the module **Bokeh/models** was changed by a lot of different contributors, which makes sense since Models represent the building block classes.[60]

Next, we compute the matrix that describes which modules are changed together according to the inspected PRs (Figure 6). We call it *Td*. Each cell *ij* (or cell *ji*) in *Td* indicates the number of times that module *i* and *j* where changed together.

It is possible to see that, for instance, the pairs of modules **Bokeh/util** and **Bokeh/embed**, **Bokeh** and **Bokeh/core** are usually changed together. In fact, each pair is tightly coupled together if we resort to the

[60] https://desosa2020.netlify.com/projects/bokeh/2020/03/19/vision-to-architecture.html

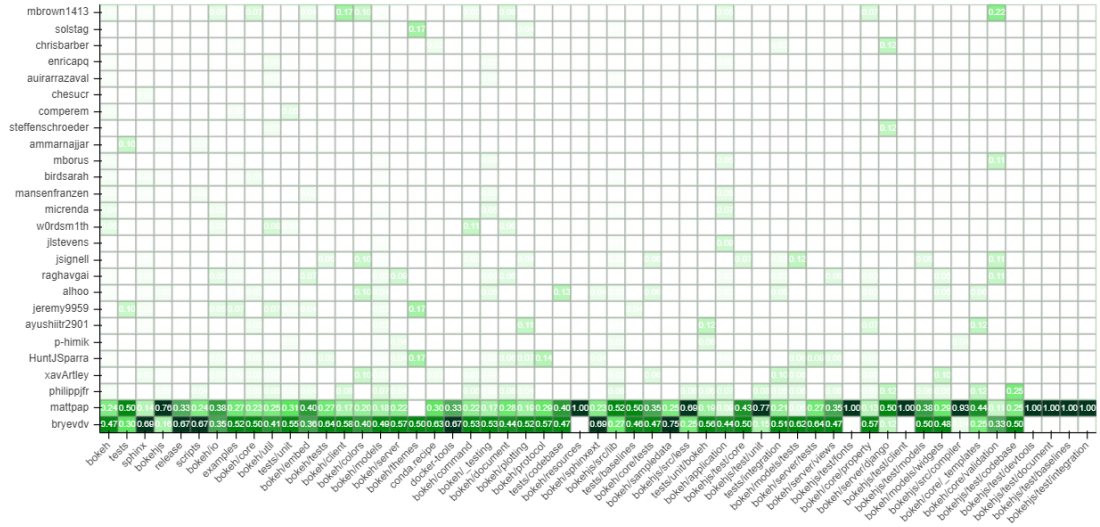


Figure 5.25: 4. Developer by Modules Matrix percentage

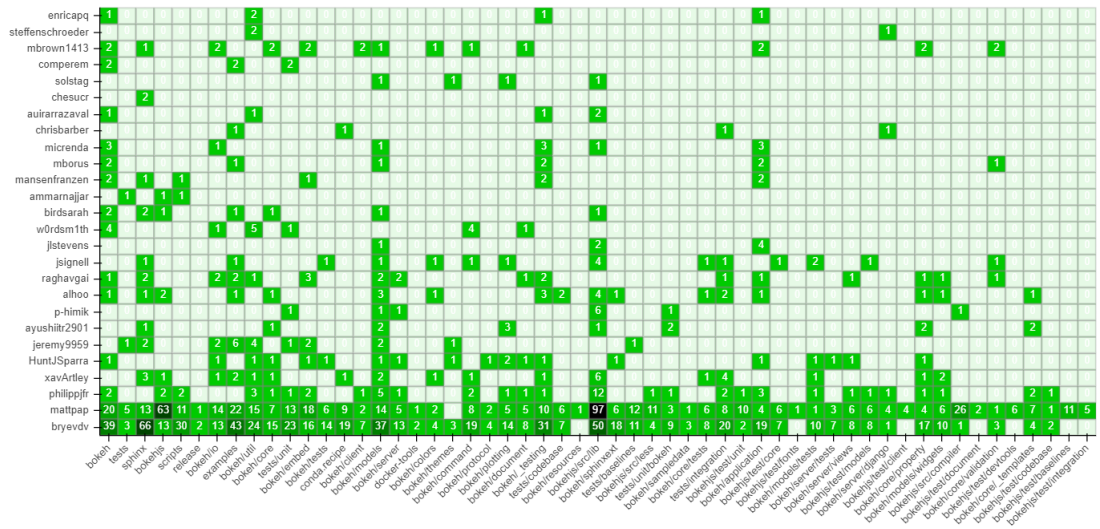


Figure 5.26: 5. Developer by Modules Matrix absolute



Figure 5.27: 6. Modules by Modules Matrix

syntactic analysis provided by Sigrid. It is also interesting that the last pair of modules, besides being usually changed together, share cyclic dependencies.

On the other hand, the modules **Bokeh** and **Bokeh/_testing** are not tightly coupled when it comes to Sigrid's analysis, but are usually modified in the same Pull Request.

Now that we have matrices *Ta* and *Td* we can compute the coordination requirements matrix, *Cr* (Figure 7). We have that *Cr* = *Ta* x *Td* x *TaT*, with *TaT* being the transpose of Ta.[61]

Finally, we show the matrix that specifies the coordination activities that actually occurred, *Ca* (Figure 8).

Cr gives us information on how much each pair of people need to collaborate in order to coordinate their work. The bigger the value in each cell, the more we expect both developers to interact with each other. We compare it against the actual number of coordination activities carried out by the developers, *Ca*. As previously discussed, the identification of which set of individuals should be coordinating activities is an important step in enhancing productivity and quality in the development process. When collaboration is not properly evaluated, several problems can arise:

> Information hiding led development teams to be unaware of others teams' work resulting in coordination problems.[62]

Analyzing *Cr*, one can see that it is expected that Mateusz and Bryan coordinate a lot their activities, as they are the developers which contribute the most. They are also expected to be the bridge between Bokeh's development and new contributors. Unsurprisingly, this is actually the case, as can be seen by *Ca* and Figure 9. Bryan and Mateusz are the central pieces of this network.

Congruence could then be computed following the formula present on the paper[63], which would give us a value between 0 and 1 representing the proportion of requirements that were satisfied.

5.5.4 Bokeh and BokehJS, an analysis

In the roadmap for the future[64] we stated that a main goal for Bokeh's core team is to develop BokehJS as a first-class Javascript library. This was also referred by Bryan in the interview:

> I'd also like to continue to encourage BokehJS as a pure JS tool that can be used on its own. My hope is this might attract new JS contributors to the project.

The question that arises is: "Are Bokeh and BokehJS able to evolve separately?". Our analysis showed that changes in BokehJS are usually followed by changes in modules on the Python side. Comments on some issues are also prof of that. This inevitably shows that in order to BokehJS evolve as a separate tool, coordination in communication and collaboration needs to fill an extremely important role on the future of Bokeh.

5.5.5 Shortcomings and conclusion

We identify some shortcomings with out method.

[61] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf
[62] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf
[63] https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf
[64] https://desosa2020.netlify.com/projects/bokeh/2020/03/09/product-vision.html

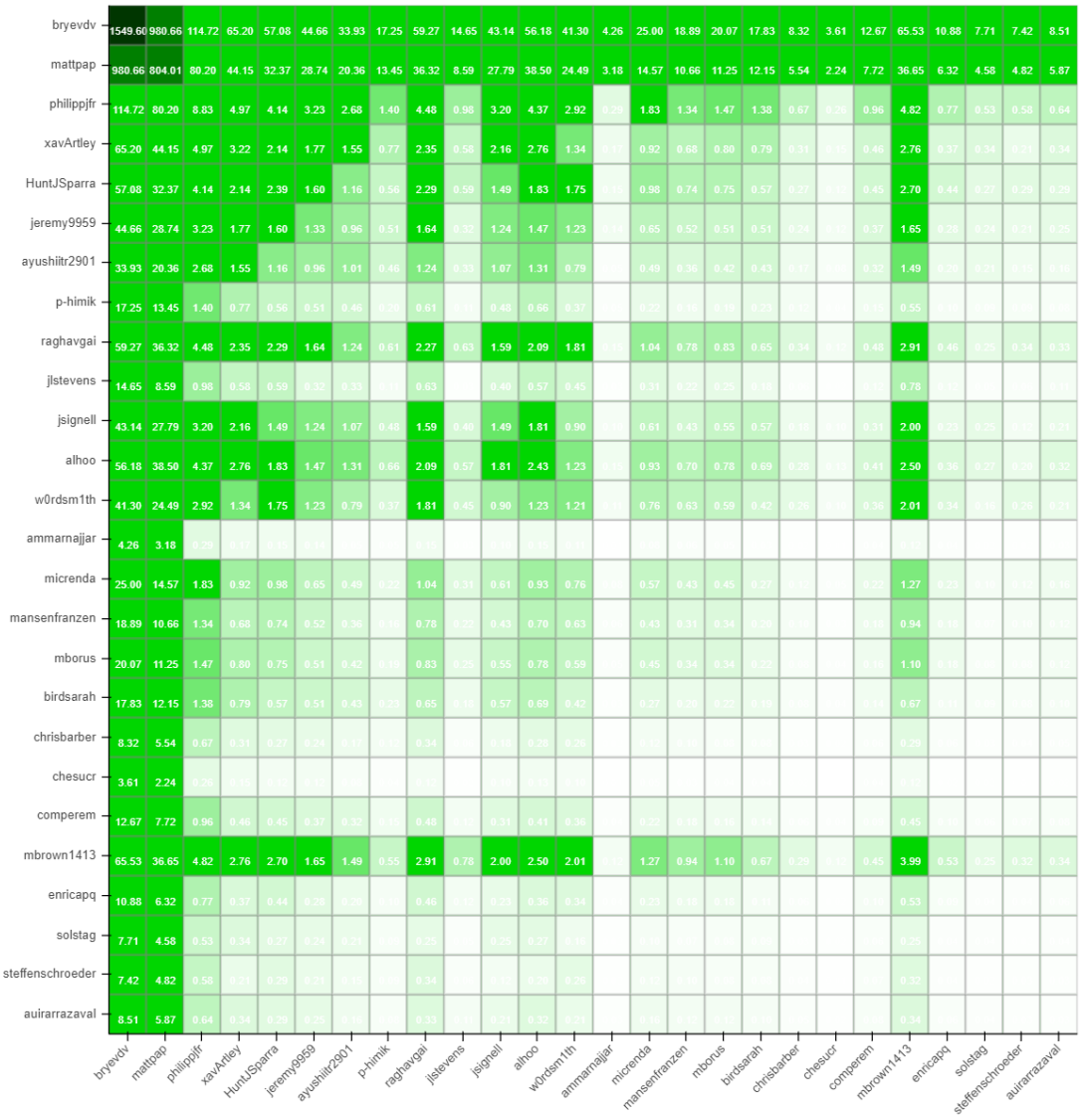


Figure 5.28: 7. Matrix Cr

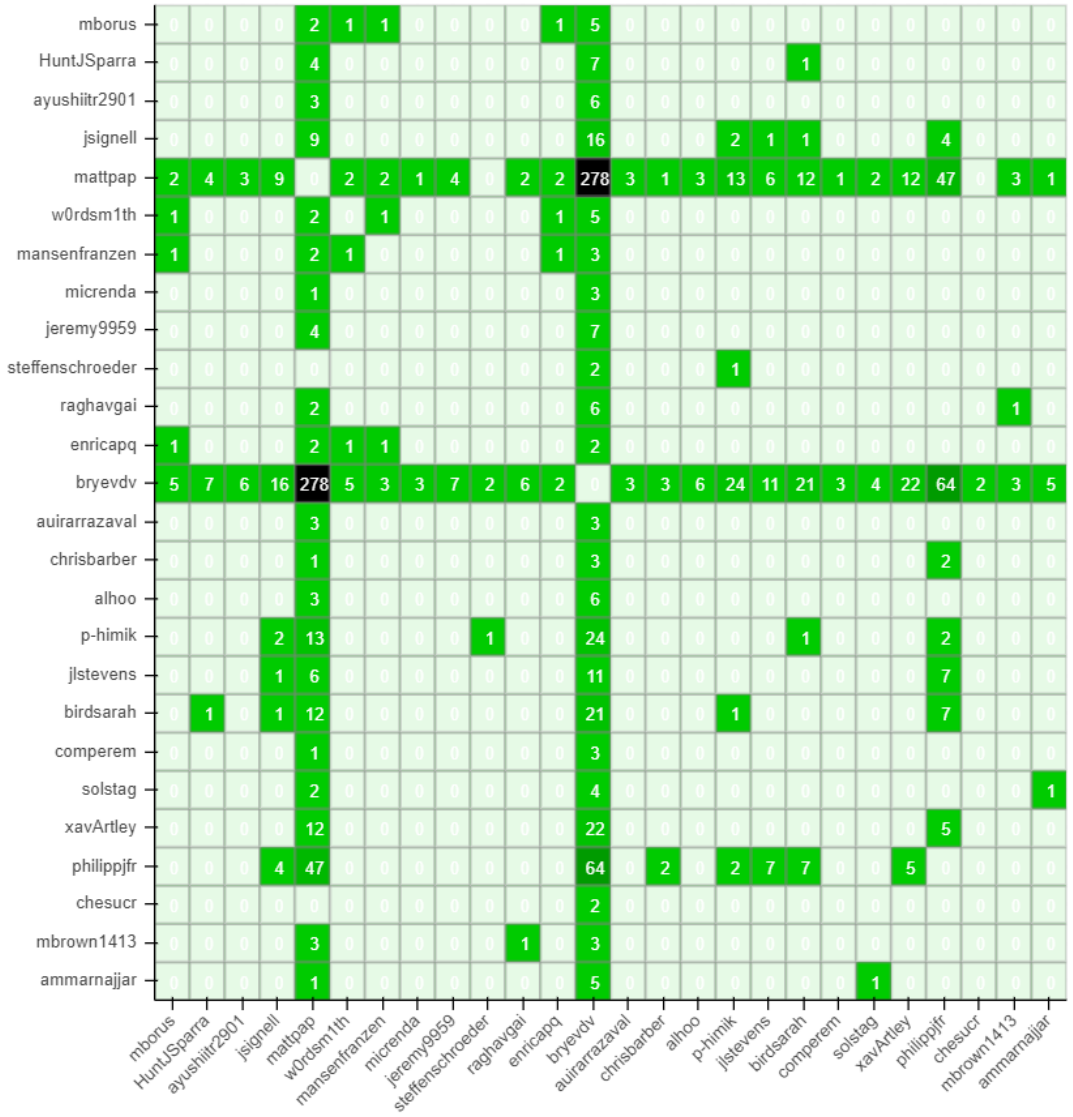


Figure 5.29: 8. Matrix Ca

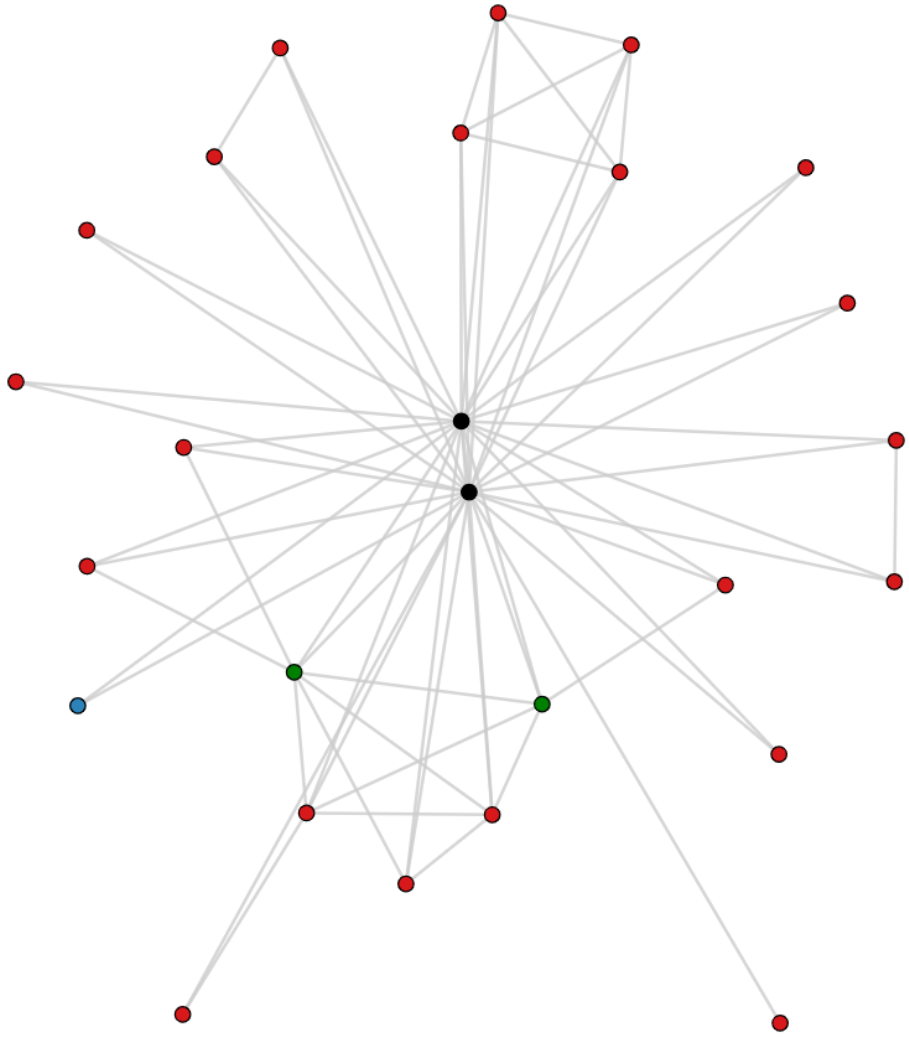


Figure 5.30: 9. Communication network in Bokeh. Black nodes represent Mateusz and Bryan. Green nodes other members of the core team. The blue node represents one of the authors of this essay.

First, we deliberately decided not to analyse all means of communication. However, doing that could have given us different and relevant new insights.

Secondly, we didn't use label information to filter Pull Requests. This could have given us the ability to only focus on feature additions, which would be in itself interesting.

Finally, we did not take into consideration that usually the work in an open source project is not evenly distributed among all contributors.

Nevertheless, this essay allow us to conclude that communication represents a fundamental element in the development of a software and should not be undervalued. Good coordination and collaboration practices are also mandatory, since communication overhead can be detrimental for projects of considerable size.

Chapter 6

Docker Compose

Docker Compose is a tool to manage multi-container docker applications running on a single host just by configuring a YAML file. it allows the management of multiple docker isolated environments, networks, and information volumes.

Docker Compose is a good fit for single-host deployment, development, and automated testing environments. PaaS, SaaS, and IaaS providers like; Heroku, Amazon Web Services, and Digital Ocean, among others, encourage its usage for local development.

6.1 The Architectural Journey

This chapter analyzes Docker Compose from an architectural perspective; Exploring source code, documentation, and people interaction.

Our journey will start at sea level, looking into the product vision. From there, we do a full descent, traveling to architecture, going deeper into code, and at 10.000 meters depth analyze configuration, collaboration, sustainability, and any insightful discovery during this journey.

6.2 Team

- Andrea Nardi
- Kanya Paramita Koesoemo
- L.C Guerchi
- Manisha Sethia

6.3 What is Docker Compose, and why does it matter?

Docker Compose is an **orchestration tool** that manages multiple docker containers in a single application, allowing the running of application components in isolated environments without unnecessary complexities. Compose automates the **building, deploying and running of the docker containers** allowing engineers to develop and deploy without building, dependencies, and compatibility issues. Compose achieves such

isolation using container-based technology which allows a lower CPU and memory overhead compared to hypervisors and virtual machines.

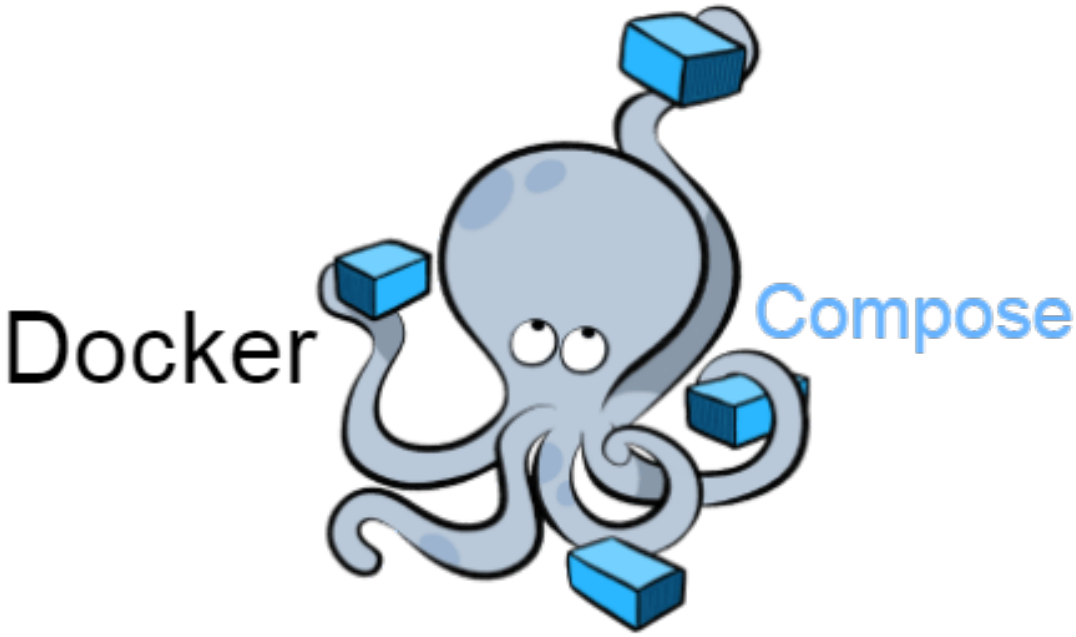


Figure 6.1: Docker Compose Logo

What will I learn from this blog-post?

This blog-post will discuss a high-level overview of Docker Compose and its motivations. The **end-user mental model** explains how the user perceives and uses Compose. Then, the **capabilities and requirements** of Compose are addressed. Later, the **stakeholders** influencing these requirements and their goals are discussed. Next, the **context** of Compose in terms of the People, Technical External Systems, Business, and Life-cycle is addressed. Finally, the future **roadmap** is presented.

- **End-User Mental Model**
- **Key Capabilties and Requirements**
- **Stakeholders**
- **Context**

6.3.1 End-User Mental Model

How are Docker Compose users actually using Compose?

The End-User mental model of Compose mainly consists of three parts, namely : 1. Service 2. Network 3. Volume

Whenever the user defines a `docker-compose.yml` file, the end-user has to define those three components.

6.3.1.1 Service

A service is a docker container defined by the end-user to carry a specific task. Services in Compose can be subdivided into more parts, listed below:

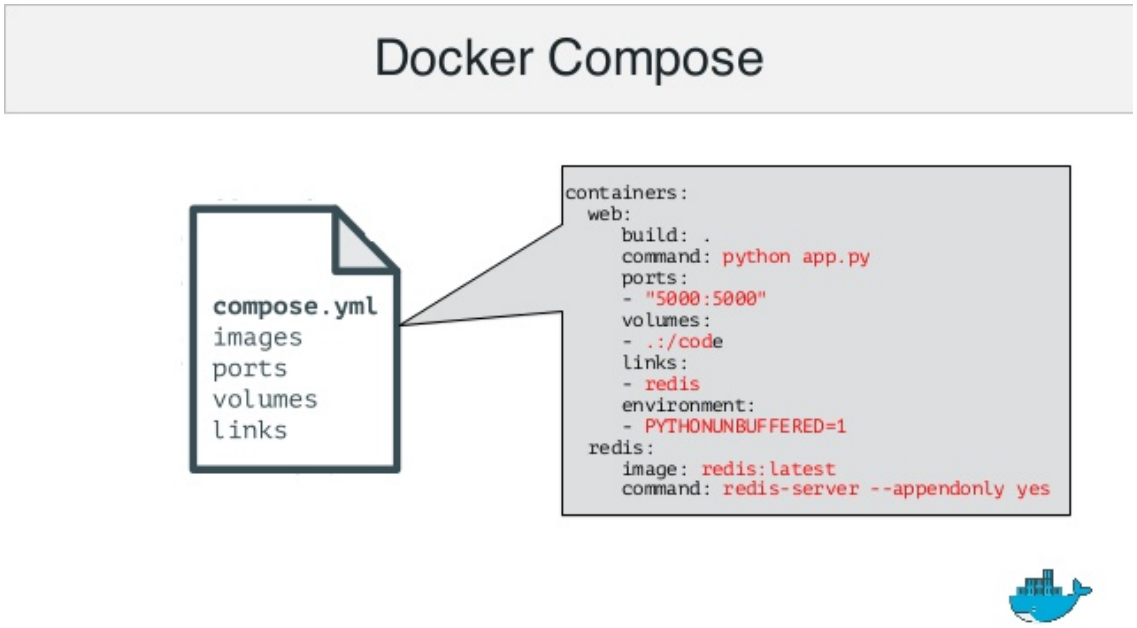


Figure 6.2: Service in Compose

- **Image**: From which Docker image is the service based on.
- **Build**: Which `Dockerfile` specifies the service building process.
- **Ports**: Which port of the Docker container maps to a port of the host machine.
- **Volume**: What file system should be mounted to the Docker container file system. The file system could be from the host machine or user defined. The user defined file systems are discussed in more detail here.
- **Link**: Makes the interdependence of services explicit.

6.3.1.2 Volume

Volume is configurable; the user can decide to mount on the services file systems. The data in volumes persists between Docker containers restarts.

6.3.1.3 Network

Network is a user-defined local area network with options to configure which services are part of it. Networks are usually defined for the services to communicate between each other with a network protocol (e.g. UDP, TCP, HTTP).

6.3.2 Key Capabilities and Requirements

Technically speaking, what are the requirements and capabilities of Compose?

The driving idea behind Compose is to enable IaC (infrastructure as code). IaC eliminates manual provisioning of systems. Compose allows local development and facilitates zero time deployment from anywhere. It allows for complex applications to be shared with its customer's stakeholders and enables personal infrastructure without requiring external IT resources. Compose helps to configure resource allocation, connections with containers, environment variables, state-specific data along with persistent data. [1]

The top-level functional requirements of Compose are: [2][3]

6.3.2.1 Application

- Define complex computational systems with few lines of code
- Define an application with multiple services
- Enable building, starting, launching an application using the command-line tool in detached mode.
- Communicate with the Docker engine to enable running applications such that it behaves like a single container.

6.3.2.2 Containers

- Create, stop and remove containers
- Display process information of the containers
- Connect a container to other environments
- Operate on the container as a normal Docker container

6.3.2.3 Networks

- Instructs Docker engine to create a network over which the applications' containers can communicate
- Remove defined customer network

6.3.2.4 Volumes

- Configure data volumes
- Define environment variables

Can you give a use-case for more perspective?

Suppose you have a two container system running on two separate hosts. Compose will build systems such that you can run both services from the same host system without putting much effort into network configurations. This enables continuous simplified delivery [4].

[1] Joshua Cook. 2017. From Docker for data science : building scalable and extensible data infrastructure around the Jupyter Notebook server.

[2] Joshua Cook. 2017. From Docker for data science : building scalable and extensible data infrastructure around the Jupyter Notebook server.

[3] Faizan Bashir. 2018. A Practical Introduction to Docker Compose. *hackernoon.com*. Retrieved March 8, 2020 from https://hackernoon.com/practical-introduction-to-docker-compose-d34e79c4c2b6

[4] Joshua Cook. 2017. From Docker for data science : building scalable and extensible data infrastructure around the Jupyter Notebook server.

6.3.3 Stakeholders

Who decides these capabilities and requirements?

Major stakeholders influence the decisions made regarding Compose. These are discussed below. [5][6]

6.3.3.1 Businesses

Docker Inc. is the acquirer of Compose. It is responsible for generating revenue via subscriptions to Compose Support. This means making Compose scalable, ensuring fast software delivery, and acquiring investments to build the open-source platform.

Companies like *Circle Cl, FLocker and Codefresh* are in partnership with Compose. These partners wish to provide solutions leveraging the Compose API, benefit from co-marketing with *Docker Inc.* and provide confidence to customers to use and install Compose via their solution [7].

6.3.3.2 End-users

The end-users are using Compose to build/view applications. Their goal is to use Compose for fast performance, UI responsiveness, and easily transform the template into application in minutes[8].

6.3.3.3 Customers

Customers are regarded as businesses using Docker Compose within their organization to deploy applications (e.g. *Harvest* and *Key location*). Their goals include scalability, easy configuration, and fast development setup.

Some technical goals could be the usage of standard Docker API, multi-container descriptor simple configuration, and Kubernetes integration.[9] [10]

6.3.3.4 Domain experts

The main developers along with *Docker Inc.* are most likely the domain experts. Their role is to guide Compose to follow its roadmap and intercept possible dangers within the platform (using their experience).

6.3.3.5 Developers/Testers

The developers and testers are building and testing Compose. Their goals are to have access to well-written documentation, reliable networking, and versions with working features. They expect less volatility in the

[5] James O Coplien and Gertrud Bjørnvig. 2011. Lean architecture : for agile software development. Wiley, Chichester. Retrieved from http://www.leansoftwarearchitecture.com

[6] Introduction and Goals | arc42 Documentation. *docs.arc42.org*. Retrieved from https://docs.arc42.org/section-1/

[7] Stake Share. Docker vs Docker Compose | What are the differences? *StackShare*. Retrieved from https://stackshare.io/stackups/docker-vs-docker-compose

[8] Stake Share. Docker vs Docker Compose | What are the differences? *StackShare*. Retrieved from https://stackshare.io/stackups/docker-vs-docker-compose

[9] Stake Share. Docker vs Docker Compose | What are the differences? *StackShare*. Retrieved from https://stackshare.io/stackups/docker-vs-docker-compose

[10] vsupalov. 2020. Is Docker-Compose Suited For Production? vsupalov.com. Retrieved from [https://vsupalov.com/docker-compose-production/]

system, and not to be dependent on a single machine; both currently unsatisfied by Compose [11].

6.3.4 Docker Compose Context

What are the aspects involved in building such platform?

Context is a multivalent world which we define as a combination of Bass's architectural context categories: technical, business, professional, and life-cycle [12] with the intuition provided by Rozanski & Woods [13] in their definition:

> "The Context view of a system defines the relationships, dependencies, and interactions between the system and its environment—the people, systems, and external entities with which it interacts."

We analyze four subsets covering the big picture. All subsets consist of interactions between them.

6.3.4.1 The Present (2020)

6.3.4.1.1 People Since Compose is an open-source system, the word *people* make more sense than *professionals*. Compose community is part of the Docker Community [14] and follows the same guidelines. Chatting is the main communication channel, mostly centered around Q&A, at the user level. To find insights about the Compose organization, we took a look at the releases and commit history since 2013. The commit history presented in Table 1, shows that a small group of participants did most of the commits. The absence of recent pushes from top developers suggests that new developers started to take ownership over time. According to Sebastiaan van Stijn (Open Source Manager at Docker), the changes of ownership is because the Compose project was initiated by Aanand Prasad and Ben Firsh and later acquired by Docker. Prasad and Firsh moved to different roles inside Docker, and are no longer working on the project.

Nowadays, the maintenance of the project is done within the *developer solutions*, a team inside Docker where people work on projects *in rotation*. This helps spreading knowledge and prevents a project becoming dependent on a limited number of people. Releases in 2020 are authored by an automated pipeline. Before, it was done by some of the active top contributors. According to Van Stijn, the developers' influence in their release cycle depends on the nature of the features/changes. As the main "Docker" CLI (the docker stack deploy), the Docker app plugin, and Compose itself use the same file format, the Docker team tries to keep those projects *in sync* by functionality added to the Compose file schema. Changes specific to the Compose command-line can be contributed by the community; those will be evaluated and included if they address a common use case.

6.3.4.1.2 Technical External systems External technologies that influence Compose are Docker, Linux distributions, Linux shells, Secure Shells (SSH), Python implementation for YAML pyyaml, orchestrators like Kubernetes and Swarkit, and docker registry servers.

From all these external systems, Docker is the biggest influence for Compose as external technology and on the *people* side.

---

[11] Stake Share. Docker vs Docker Compose | What are the differences? *StackShare*. Retrieved from https://stackshare.io/stackups/docker-vs-docker-compose

[12] Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice. Addison-Wesley Professional.

[13] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

[14] Docker Community - Connect with Docker Enthusiasts | Docker. *www.docker.com*. Retrieved March 8, 2020 from https://www.docker.com/docker-community

6.3.4.1.3 Businesses Online, we found suggestions to use Compose for local dev environment in many IaaS, PaaS and SaaS providers like Digital Ocean, Amazon or Heroku. This indicates that most usage occurs within software companies.

6.3.4.1.4 Life Cycle We analyzed the life of Compose by looking at the bug/features frequency over releases during the last two years. We noticed that Compose is in continuous evolution. Table 2 shows features/bug fixes that occur at a constant rate. Features added in the last two years indicate that Compose orbits around security, optimization, compatibility, and more configuration options.

6.3.4.2 The Future (+2020)

According to Van Stijn, the **roadmap** will keep Compose as an easy-to-use solution for developers, focusing on *commonly used features* and avoiding complexity.

All information below exceeds the 1500 word limit

6.3.5 Thanks

Special thanks to Sebastiaan van Stijn for answering all our questions and guide us in this Compose adventure. Details of the interview can be found below.

6.3.6 Release table Features and Bug fixes:

Author	Number of commits
Joffrey F	1510
Aanand Prasad	1181
Daniel Nephin	780
Ben Firshman	522
Ulysses Souza	164
Mazz Mosley	159
Harald Albers	94
mnowster	70
Steve Durrheimer	38
Nicolas De Loof	32

Table 1: Top 10 Authors, for 2013-2020 in Docker Compose.

Date	Release	Features	Bugfix	Author
2020-02-24	1.26.0-rc2	yes	yes	docker-dsg-cibot
2020-02-03	1.25.4	no	yes	docker-dsg-cibot
2020-01-31	1.25.4-rc2	no	yes	docker-dsg-cibot
2020-01-23	1.25.3	no	yes	docker-dsg-cibot
2020-01-20	v1.25.2	yes	yes	docker-dsg-cibot
2020-01-20	v1.25.2-rc2	yes	yes	docker-dsg-cibot
2020-01-06	1.25.1	yes	yes	ulyssessouza

Date	Release	Features	Bugfix	Author
2019-11-29	1.25.1-rc1	no	yes	ulyssessouza
2019-11-18	1.25.0	yes	yes	ulyssessouza
2019-10-28	1.25.0-rc4	yes	yes	rumpl
2019-10-28	1.25.0-rc3	yes	yes	rumpl
2019-08-07	1.25.0-rc2	yes	yes	ulyssessouza
2019-06-24	1.24.1	no	yes	rumpl
2019-05-23	1.25.0-rc1	yes	yes	ulyssessouza
2019-03-28	1.24.0	yes	yes	ulyssessouza
2019-03-22	1.24.0-rc3	yes	yes	ulyssessouza
2019-01-14	1.24.0-rc1	yes	yes	rumpl
2018-11-28	1.23.2	no	yes	shin-
2018-11-01	1.23.1	no	yes	shin-
2018-10-30	1.23.0	yes	yes	shin-
2018-10-17	1.23.0-rc3	yes	yes	shin-
2018-10-08	1.23.0-rc2	yes	yes	silvin-lubecki
2018-09-26	1.23.0-rc1	yes	yes	shin-
2018-07-17	1.22.0	yes	yes	shin-
2018-07-05	1.22.0-rc2	yes	yes	mnottale
2018-06-21	1.22.0-rc1	yes	yes	shin-
2018-05-02	1.21.2	no	yes	shin-
2018-04-27	1.21.1	no	yes	shin-
2018-04-09	1.21.0	yes	yes	shin-
2018-03-31	1.21.0-rc1	yes	yes	shin-

Table 2: Release relation Bug/Feature in docker-compose for 2018-2020.

6.3.7 Interview Quotes of Sebastiaan van Stijn

“Compose (formerly known as”Fig“) project was started by Aanand Prasad and Ben Firsh. Their company (and the project) was later acquired by Docker, where development was continued. Ben and Aanand moved to different roles inside Docker, and later went on new journeys, so they’re no longer working on the project. Nowadays, the maintenance of the compose project was then moved to the”developer solutions" team inside Docker, which is responsible for various projects and products. Within that team, people work on projects “in rotation”, which helps spreading knowledge and prevents a project depending on a limited number of persons."

“. . . it depends on the features or changes nature; given that both the main”docker" CLI (the docker stack deploy), the docker app plugin, and Docker Compose itself use the same file format, we try to keep those projects “in sync” on functionality added to the compose file schema. This means that a change added in the docker CLI should also find its way into docker-compose “standalone” eventually. We may move the compose specification to a separate repository to make it more clear where the “canonical” location is to introduce changes in the schema.

“I’ll be honest; there’s been some feature-creep in the past where features have been accepted for corner-cases. Adding features is a lot easier than removing/deprecating them, so we’ll

> have to live with some of those, or find a migration path to improve. The goal of Docker projects/products is to make things easy to use and to provide sensible defaults when possible. Some features added in the past don't"fit" that principle, so it may have to be looked at some point. Finally changes based on user and customer feedback, research; if we identified workflows that can be improved by adding a feature, such features can be added to the roadmap. Regarding control of releases, there is not a build master,that's a rotating role, not a fixed person."
>
> "All through GitHub; open a ticket to propose enhancement or feature and/or open a pull request to propose a change. Providing a good description of the use-case that the change addresses is important there, as it helps to decide if the proposed solution makes sense, or if other solutions should be considered/designed."
>
> "While some things"settle down", the container ecosystem is still a fast moving"landscape". We want docker-compose and the compose-file format to be a great, easy-to-use solution for developers. The compose-file format is very popular as it (despite some"odd" choices from the past) has a nice balance between "commonly used features" and not being overly complex. We see adoption of the format for many uses; for some as an "end format" (running containers locally), for others as an intermediate format (deploying to kubernetes, or generating kubernetes YAML's from the compose file)." - Sebastiaan van Stijn

Written by Manisha Sethia, Kanya Paramita, Andrea Nardi, Lucio Guerchi

6.4 Relevant Architectural Views

Docker Compose is a lightweight orchestration tool for deploying multi-container applications on a single host machine; it easy to use for Docker users, and does not require extensive configuration.

Docker Compose is part of the toolkit in Docker. The previous essay discusses the vision of Compose. In this essay, we bridge the gap between the vision and the architecture.

> *What is the framework used to explain the architecture of Docker Compose?*

The framework leveraged in this essay aims to facilitate three main learning goals for the reader:

1. Compose within Docker Architecture

We take inspiration from the "4+1" model to provide an **intuitive understanding of Compose's role within Docker** and its end-user. The "4+1" model [15] states a unique viewpoint named "Scenarios" which illustrates use-cases that reflect the sequence of interactions between objects and processes. This essay provides a use-case of how the end-user can use Compose to develop/test their application.

2. Relevant architectural pattern

Many standard architectural patterns exist within Software Architecture. This section discusses what is the relevant architectural pattern for Compose and the motivation behind it.

15

2019. 4+1 Architectural View Model - CodeOpinion. CodeOpinion. Retrieved from https://codeopinion.com/41-architectural-view-model/

3. Formal representation of the architecture of Compose

For a formal understanding, the architectural viewpoints discussed by Rozanski & Woods [16] will be used. This is followed by a discussion on what are the trade-offs for the function/non-functional properties of the system.

Since Compose is a simple orchestration tool, many viewpoints are not relevant here. The main criteria for determining the relevancy of the viewpoints are applicability and addressal to significant stakeholders. With Rozanski & Woods [17] the chosen viewpoints are:

What aspects are considered from Rozanski & Woods [18] ?

View-Point	Model	Description	Stakeholders
Development	Module Structure Models	Describes the architecture that supports the software development process [19]	Developers/Testers
Functional	Functional Structure Model	Describes the system's functional elements, their responsibilities, interfaces, and primary interactions. [20]	All
Deployment	UML Deployment Model	Describes the environment into which the system will be deployed [21]	System Administrators, Developers/Testers

[16] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.
[17] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.
[18] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.
[19] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.
[20] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.
[21] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

6.4.1 Compose within Docker Architecture

How does Compose interact with Docker components?

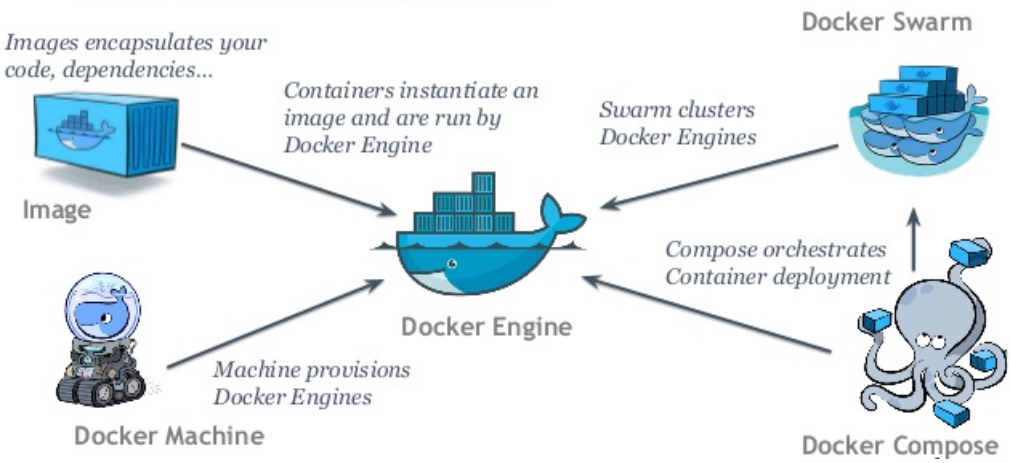


Figure 6.3: Compose with Docker Components

Figure 1 [22] gives an overview of Compose's interaction with other components of Docker. Swarm is also an orchestration tool for scheduling and managing across multiple host machines, whereas Compose deals with a single-host machine.

Where is Compose in the lifecycle of a Docker Application?

Compose can play a role in many stages within an application's life cycle. Developers can use Compose to run their application locally (enables **end-to-end testing**). The continuous integration process can use Compose to run and build the app (enables **automated testing**).

During the testing phase, **single-server environments** can be deployed for user testing (promotes **minimality**). During production, Compose can be used for defining the format of the application however, due to several limitations of Compose, other orchestration tools such as Swarm are popular to use. Thus Compose is mainly useful within the **developing, testing and staging** environments of an application.

Figure 2 shows a use-case of the end-user (developer of an application) using Compose to enable the testing/release of a feature that they are developing. [23]

The end-user has three main tasks, namely : 1. Define user's application using **Dockerfiles** 2. Define the services that are needed to run the application within the **docker-compose.yml** 3. Run the application using **docker compose up** command.

Figure 2 also shows how Compose interacts with the Docker registry (i.e. Docker Hub) to facilitate these tasks. It pulls the images created from the DockerFiles from the Docker registry and uses this to run a multi-container application.

[22] MariaDB Corporation. 2017. Getting Started with MariaDB with Docker. Retrieved March 15, 2020 from https://www.slideshare.net/MariaDB/getting-started-with-mariadb-with-docker-84370473

[23] Brodie Mitchel. 2019. Dev Environment Setup Docs For Mac. limibuyer. Retrieved March 15, 2020 from http://limibuyer.dx.am/dev-environment-setup-docs-for-mac.html

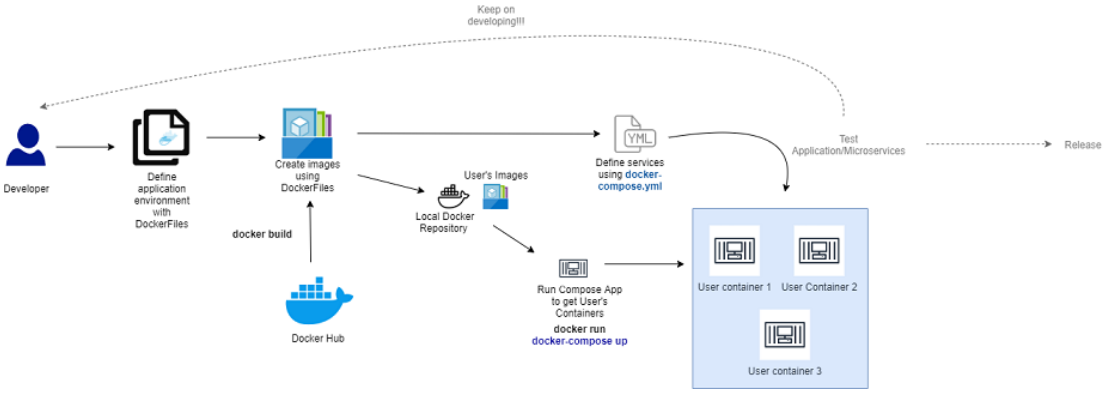


Figure 6.4: Docker Compose - End-User View

With an understanding of how Compose fits into the Docker architecture, the next sections will delve deeper into the architecture of Compose itself. This means that the focus shifts from the end-user of Compose to other stakeholders of Compose, including business owners and developers/testers.

6.4.2 Relevant Architectural Pattern

What is the architectural pattern applicable for Compose?

While Docker implements a client-server architecture, Compose implements a **monolithic** architectural pattern.

> "A **Monolithic application** has a single code base with multiple modules. Modules are divided as either for business features or technical features. It has a single build system that builds the entire application and/or dependency. It also has a single executable or deployable binary" [24]

A monolithic architecture is self-contained, with components of the software being interdependent and interconnected rather than loosely coupled. [25]

What are the advantages and disadvantages of a Monolithic architectural pattern?

The main advantage of a monolithic architecture is **simplicity**. There is simplicity in deployment due to all actions being able to be performed using a few commands. The pace of development is naturally increased as developers do not have to take into consideration the other tools working with Docker.

The disadvantage of a monolithic architecture is that as new features get introduced, the **codebase and dependencies increase**. This complicates the workflow, resulting in either building a new toolbox all together (for the new features) or decomposing the toolbox into a new architectural pattern (e.g. micro-services architecture).

Another disadvantage is that monolithic applications are **not reusable** meaning, one feature implemented in Compose cannot be leveraged into another toolbox of Docker.

[24] Monolithic vs Microservice Architecture - DZone Integration. dzone.com. Retrieved March 19, 2020 from https://dzone.com/articles/monolithic-vs-microservice-architecture

[25] Coding the Architecture. 2014. What is a Monolith? - Coding the Architecture. Codingthearchitecture.com. Retrieved from http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html

The entire Compose toolbox needs to be integrated within the other toolbox for the feature to be used. Lastly, any issue within the monolithic toolbox affects the entire toolbox. This results in **harder maintainability and more expensive to test**.

What is the motivation behind Docker Compose employing a monolithic architectural pattern?

From the discussion above, it seems that the disadvantages of a monolithic architecture outweigh the advantages. However, these disadvantages become prominent when dealing with complex systems. Compose has limited components and functionalities. Therefore, perhaps the disadvantages of maintainability and feature re-use are not as relevant anymore. Docker has many toolboxes such as Swarm that are achieving Compose's intentions with more complex functionalities.

This eradicates the need for introducing complex features within Compose itself. It is important to keep in mind the roadmap of Compose (as discussed in Essay 1). It indicates that Compose will be an **easy-to-use solution, with commonly-used features**. This is the reason why Compose implements a monolithic architecture, and "can get away with it".

Now that we have an idea of the architectural pattern and its motivation for Compose, let us delve into a formal representation of the architecture of Compose.

6.4.3 Development view

As discussed above, the Compose system is organized in a way that it can run multiple Docker containers as services using only a single command, instead of running each container separately (Figure 3).

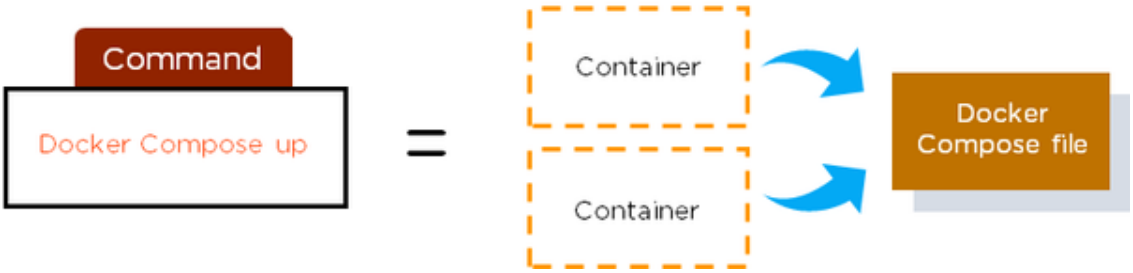


Figure 6.5: Docker Compose Command

6.4.3.1 Module Structure

Figure 4 shows the development view of Compose. The source code of Compose is conceptually organized into 3 main modules:

- **Core**, source code for core functionalities of Compose
- **Script**, responsible to build and run Compose core source code
- **Tests**, define and run functionality, integration, and unit tests for the core source code

The Core module is divided further into submodules (layers) that represent Compose's main functionalities.

CLI layer is in charge of creating a command-line interface for end-users to run Compose's feature command. It contains components that define all the CLI commands, configure Docker client and some other CLI

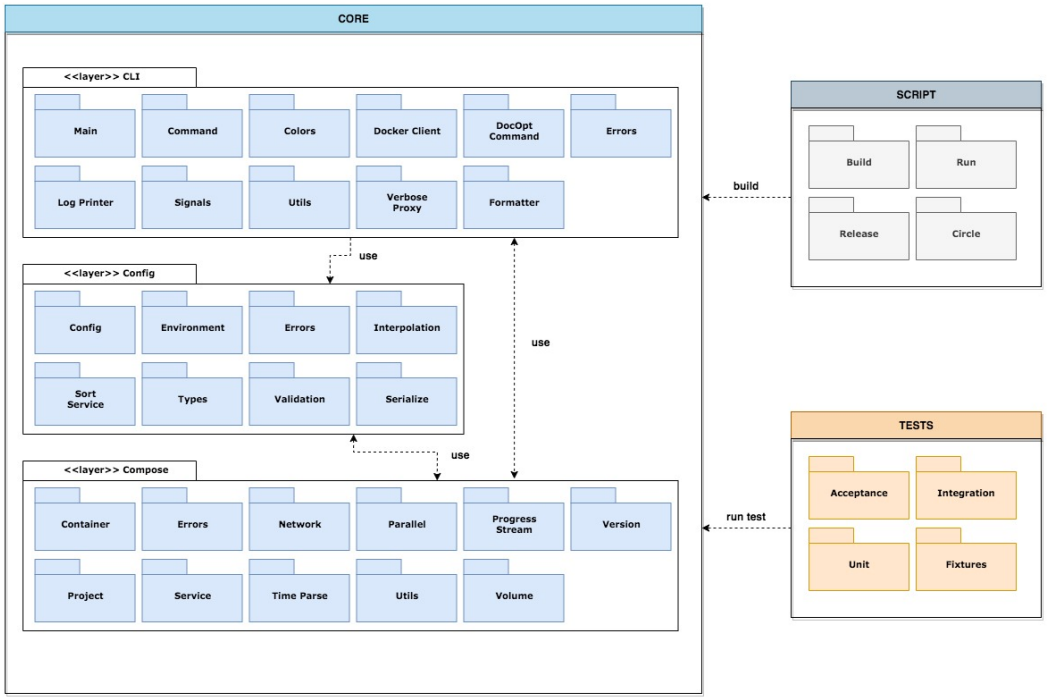


Figure 6.6: Docker Compose Development View

utilities. CLI layer calls Config layer to get configurations information and Compose layer to pass on command for them to execute and get information about services and containers.

Config layer's job is to load all configurations from Docker Compose YAML file and environment variables. Its components read and parse `docker-compose.yml` file, detect and parse environment variables or file, and do validation and interpolation for these configuration values. It interacts with Compose layer and uses some utilities from Compose's Utils component.

Compose layer constructs all the main objects and processes of Docker Compose. The components here are responsible for building projects, services, containers in a service, volumes, and networks by communicating with Docker. It uses Config layers to get configuration information and CLI layers to stream on information of Compose processes.

6.4.4 Runtime view

The runtime view of Docker Compose (see Figure 5) is done with the UML notation of the functional viewpoint provided by Rozanski & Woods [26]. The runtime view is meant to display the component's function, dependencies, and interactions. The methods of interaction from the end-user to the system are mainly through the docker-compose client, the `docker-compose.yml` configuration file, and the `.env` configuration file. Once Docker Compose deploys the containers in the Docker Daemon, the end-user can interact with the containers through a docker client.

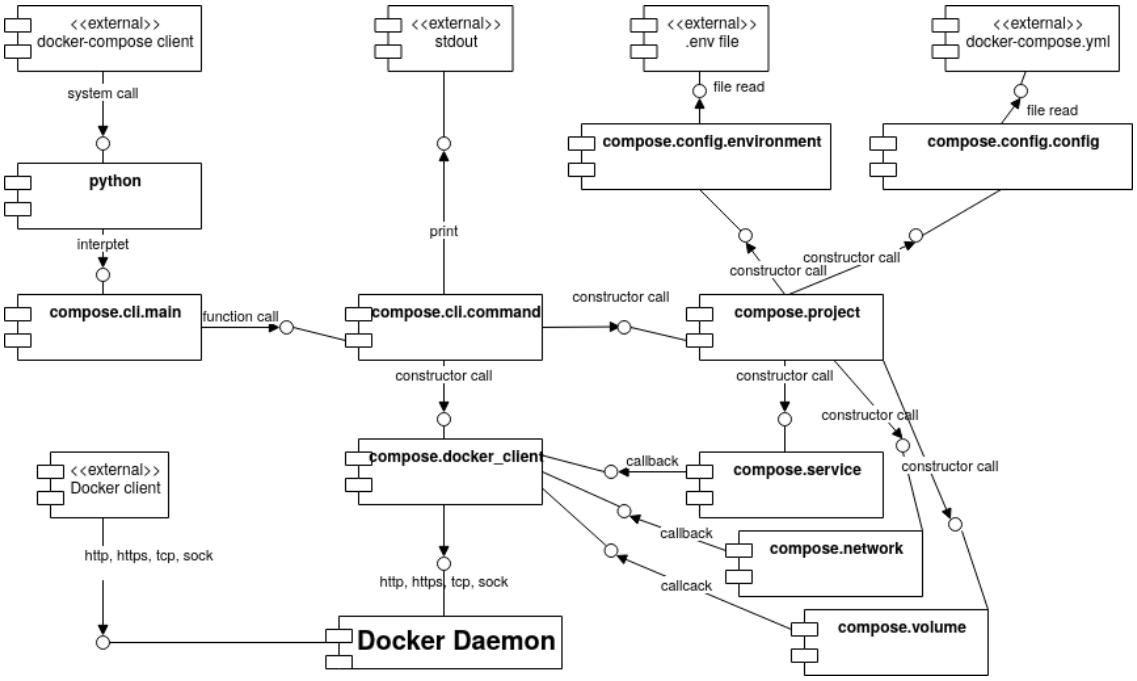


Figure 6.7: Runtime View

[26] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

6.4.5 Deployment view

Docker Compose is an orchestrator; it runs as a binary file at OS level in MAC, Windows, and Linux machines. At run time, it manages the startup of Docker images, networks, and volumes, inside a single host machine.

Figure 6 shows a deployment view of a containerized application managed and configured by Docker Compose.

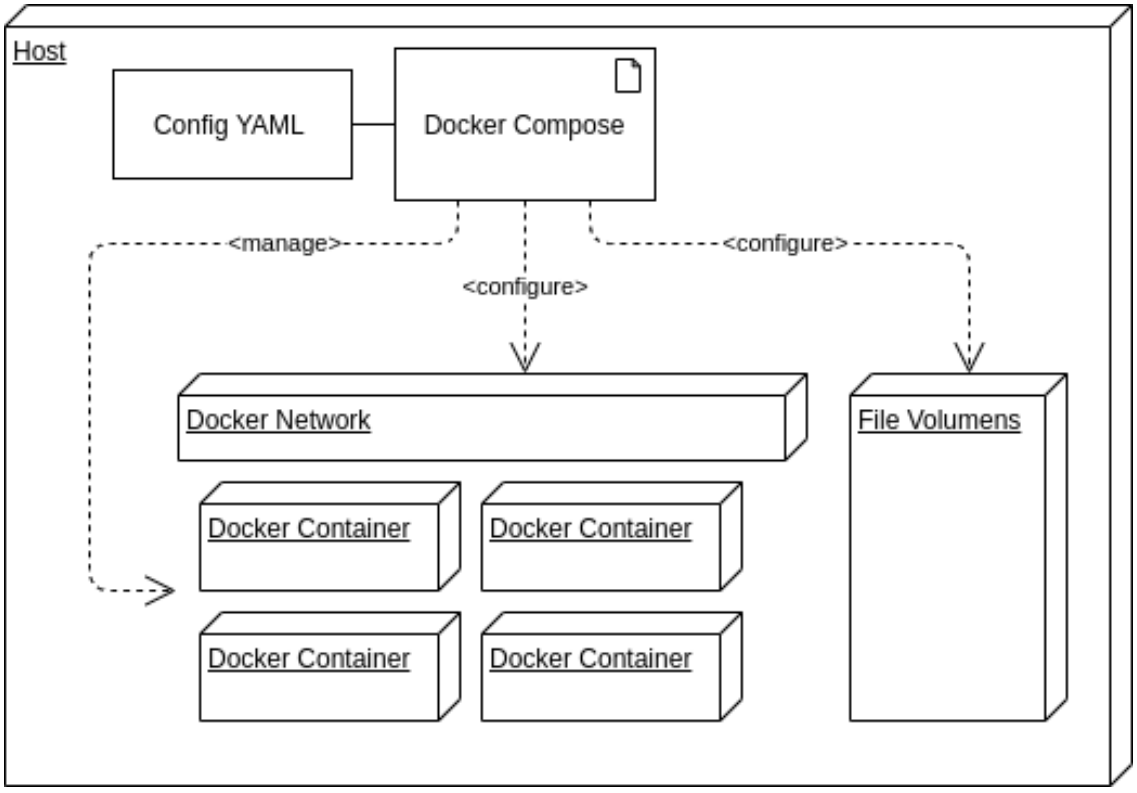


Figure 6.8: Deployment View

6.4.6 Docker Compose Functional / Non-Functional properties tradeoffs

In software architecture, it is common to divide properties into functional and non-functional; those that are functional are about what the system should do, meanwhile non-functional are about how the system should behave.

For Docker Compose, there are no significant trade-offs between non-functional and functional properties. For example, support for BuildKit, coined as a feature, is meant to improve performance, storage management, security, besides overall functionality. The main reason behind this is that end-users are concerned with how time-consuming, secure, and memory friendly Docker Compose is. For this reason, performance, efficiency, and security become core features. As a result, non-functional properties like

integrity, availability, and fault-tolerance could be considered out of scope; that means, it works or does not. Meanwhile, performance, efficiency, and security are considered key features.

Written by Manisha Sethia, Kanya Paramita, Andrea Nardi, Lucio Guerchi

6.5 Quality and Technical Debt

We analyze the integration process, code quality, test coverage, and how this is facilitating the roadmap of Compose. Then, we delve a bit deeper into technical debt and refactoring solutions of newer features and the entire system as a whole. Overall, Compose system has an organized and aligned development process with high test coverage. However, there's much improvement and refactoring needed on the code quality and entanglement itself.

6.5.1 Coding the architecture

Commits are individual file changes used in version control tools like Github. Counting the number of times that a commit modifies a file indicates code activity (hotspots).

We analyzed file history from 2013 to 2020 to identify code hotspots excluding testing and CI/CD related files. From our analysis (Fig.1), we found that compose/service.py (513), compose/config/config.py (402), compose/cli/main.py (399), and compose/project.py (276), were the most committed files of the Docker Compose project; these files belong to key components, Compose, CLI and config. As a side, note "fig" was the previous name of "compose" when the project was not yet part of Docker.

From 2019 to March 2020 (Fig.2), we found that compose/cli/main.py (27), compose/service.py(24), compose/project.py(22), and compose/config/config.py(11) were the most committed files, following the same trend in hotspots that we encountered in previous years; that means that there were not architectural changes during the last months of development.

6.5.2 From Roadmap to Code

Starting 2020, Docker Compose 1.25.1 was released; this release added support to Buildkit which enables faster builds. After that, the repository shows small stories that deal with additional functionality and bugfixes. Stories like those will be developed in the hotspots mentioned in the previous section.

Epics are bodies of work used in agile methodologies that could be subdivided into stories. We based our findings of expected features on the analysis of Epics in combination with recent repository activity and the latest strategies from the Docker project.

Looking at active Epics, we expect efforts in:

- Removal Python 2
- Proposals to rely on Docker CLI
- Reconcile docker-compose schema on docker/cli vs docker/compose

For March 2020, the roadmap of Docker Compose is about code cleaning and alignment. The reason behind this is the required removal of Python 2 due to its end of life (January 2020); this issue is now under scope discussion. Another goal found in Epics is to make Docker Compose work in harmony with Docker. To make this possible, architectural level decisions and stakeholder coordination might be needed to avoid duplication of schemas between the two projects.

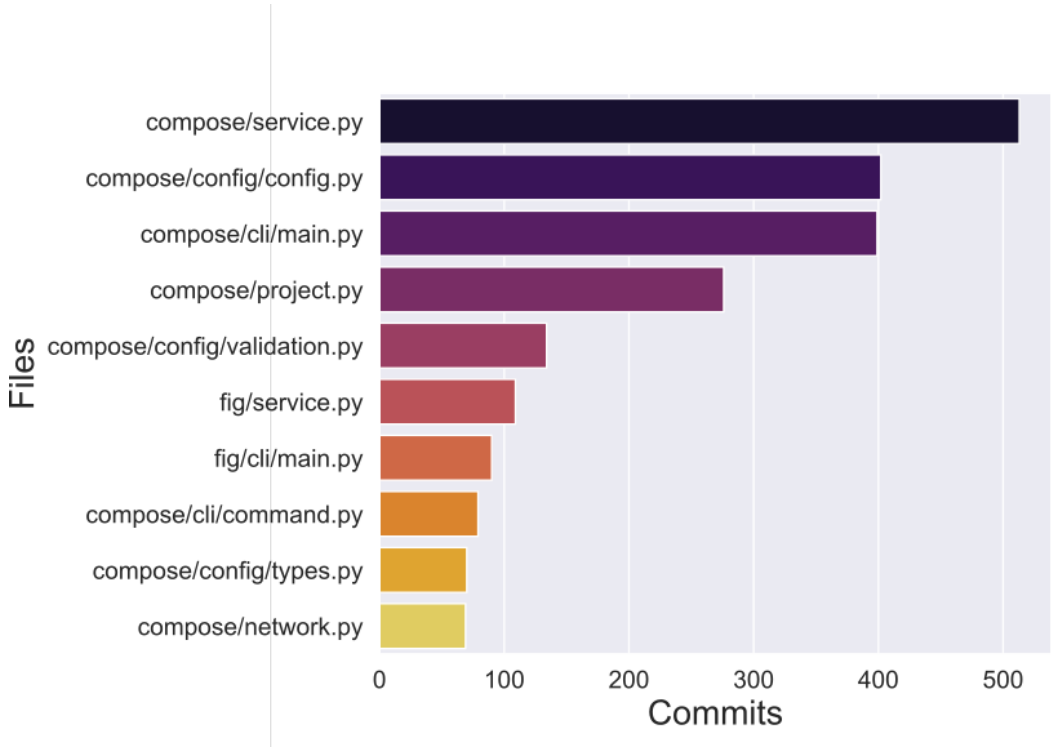


Figure 6.9: Top 10 most commited files (2013-2020)

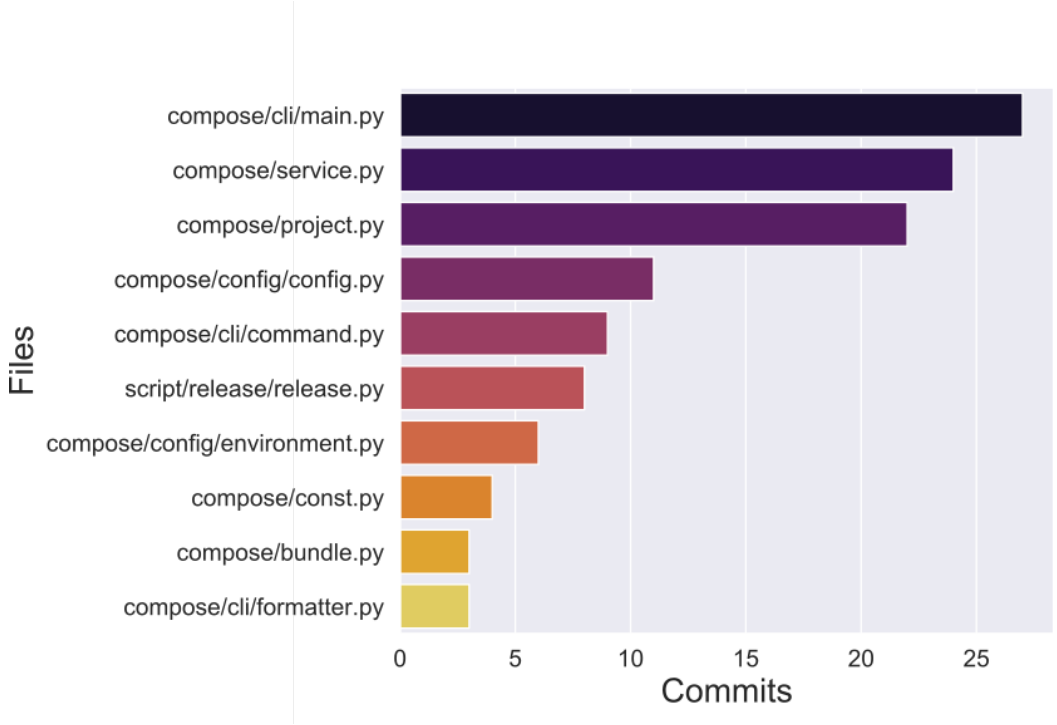


Figure 6.10: Top 10 most commited files (2019-2020)

For the rest of the year, we can predict that as Docker refocuses on developers, Docker Compose will remain as a lightweight orchestrator for developers in constant syncing with the recently announced public Docker roadmap; an interesting approach that leaves space for decision making about the direction to the community. As far as we don't see any architectural changes, a keep things simple mindset, we predict that changes might remain in the same top 10 most modified files on the Docker Compose project presented in the previous section.

6.5.3 Code Quality

How good or bad the code quality is rather subjective and depends on what the organization considers important. From the roadmap explained, Compose main concern is around avoiding complexity and support maintainability and that `cli` and `config` will be the most affected components.

We use Sigrid, SonarCube and CodeFactor as static analysis tools to give an overview of Compose code quality. Overall, Compose has an average code quality as reported from each tool:

- Sigrid reports great score in code volume, duplication, and component balance but low in components independence and unit complexity
- SonarCube reports great score in technical debt and code smells measurement
- CodeFactor reports an overall code quality score of C- based on complexity, maintainability, and security

6.5.3.1 Code Complexity

We use McCabe Cyclomatic Complexity as it is one of the well-known and widely used metrics [27]. Cyclomatic complexity simply measures the number of paths run through code[28].

CodeFactor identifies Compose code cyclomatic complexity as low-risk problems. There are 8 issues low-risk complexity found in some code files as shown in Fig.3.

Only 8 methods identified having complexity issues (Fig.4), all also low risk. Some methods from `cli` and `compose` are identified as having complexity issues. But if we take a look at the low-risk complexity score which ranges around 16-23, it's still considered manageable.

Other metrics are code duplication levels and unit sizes as reported by Sigrid (Fig.5). Compose has low duplication percentage which is good. While in terms of unit size, it can be seen that only 50% of them are in good condition.

6.5.3.2 Code Maintainability

Code maintainability can be seen as the effort needed to make specified modifications to a component implementation [29]. Fig.6 shows Compose component entanglement graph as reported by Sigrid. It can be seen that there are many dependencies between `cli` and `compose` component which ideally shouldn't exist based on the layer structurization.

[27] A. F. Nogueira, "Predicting software complexity by means of evolutionary testing," 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, 2012, pp. 402-405.

[28] Bhatti, H. R. (2011). Automatic Measurement of Source Code Complexity (Dissertation). Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:ltu:diva-46648

[29] SEI Open System Glosssary. https://www.sebokwiki.org/wiki/Open_System_(glossary)

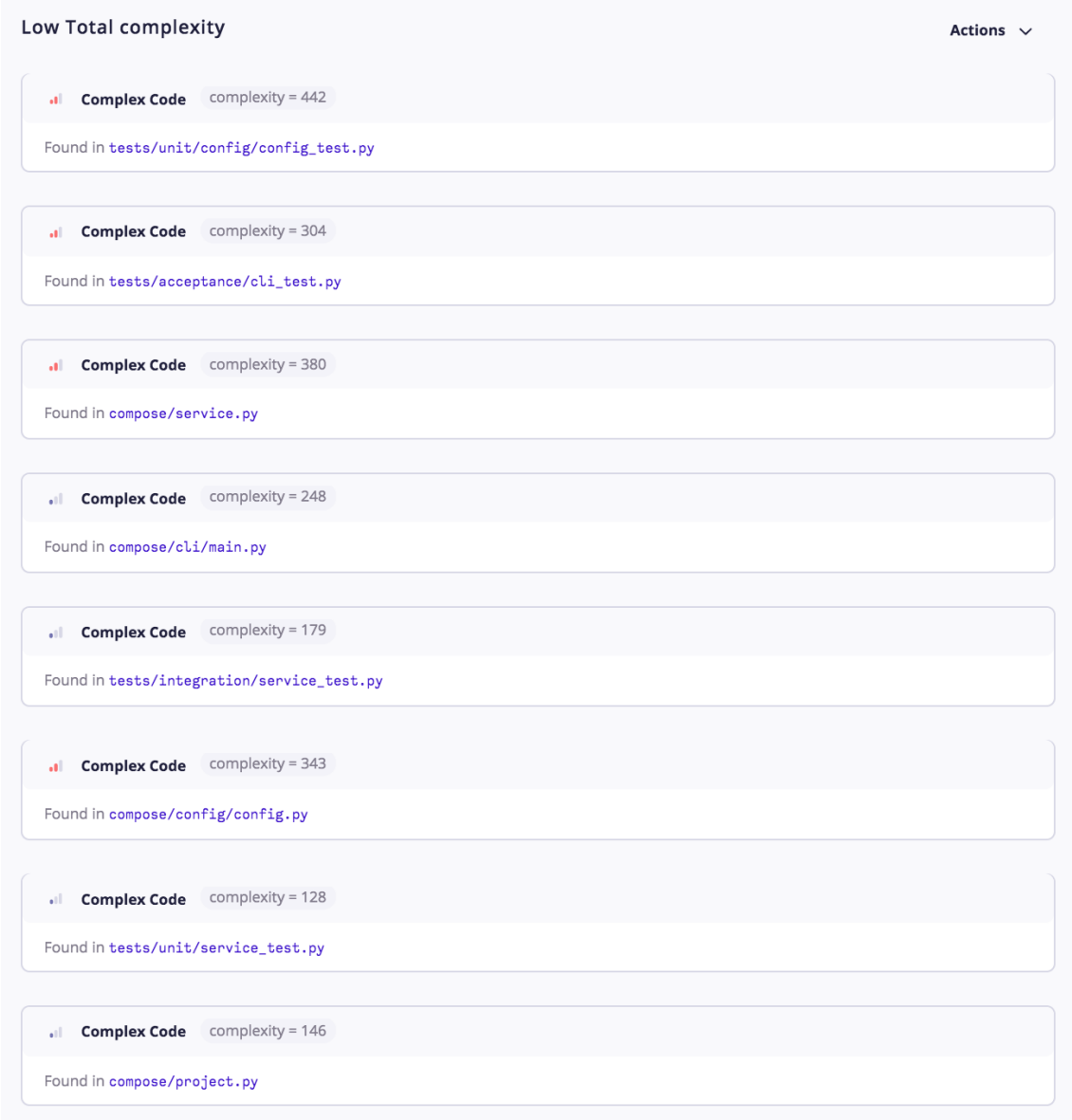


Figure 6.11: Compose source code files with complexity issues

Complex Method complexity = 18

Found in compose\cli\main.py:989-1124

```
989     def up(self, options):
990         """
991         Builds, (re)creates, starts, and attaches to containers for a service.
992 
993         Unless they are already running, this command also starts any linked services.
```

View more

Complex Method complexity = 16

Found in compose\config\config.py:856-909

```
856 def finalize_service(service_config, service_names, version, environment, compatibility):
857     service_dict = dict(service_config.config)
858 
859     if 'environment' in service_dict or 'env_file' in service_dict:
860         service_dict['environment'] = resolve_environment(service_dict, environment)
```

View more

Complex Method complexity = 17

Found in compose\network.py:209-236

```
209 def check_remote_network_config(remote, local):
210     if local.driver and remote.get('Driver') != local.driver:
211         raise NetworkConfigChangedError(local.true_name, 'driver')
212     local_opts = local.driver_opts or {}
213     remote_opts = remote.get('Options') or {}
```

View more

Complex Method complexity = 16

Found in compose\config\serialize.py:53-87

```
53 def denormalize_config(config, image_digests=None):
54     result = {'version': str(V2_1) if config.version == V1 else str(config.version)}
55     denormalized_services = [
56         denormalize_service_dict(
57             service_dict,
```

View more

Complex Method complexity = 23

Found in compose\config\serialize.py:125-169

```
125 def denormalize_service_dict(service_dict, version, image_digest=None):
126     service_dict = service_dict.copy()
127 
128     if image_digest:
129         service_dict['image'] = image_digest
```

View more

Figure 6.12: Compose source code methods with complexity issues

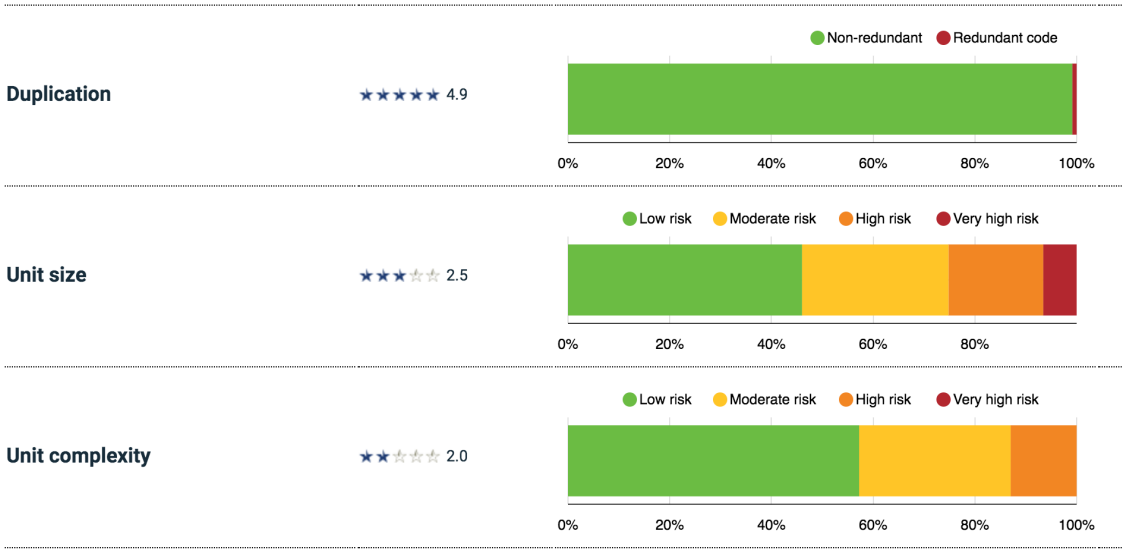


Figure 6.13: Compose source code methods with complexity issues

Unit interfacing score indicates the size of the interfaces of the units of the source code in terms of the number of interface parameter declarations. Approximately 30% of unit in `config` and `cli` components have a medium to high-risk score which can be a problem to the code readability (Fig.7). Module coupling score indicates the number of incoming dependencies for the modules of the source code. It might create a problem in making changes to component `config` as it has quite a huge number of module coupling (Fig.7).

We look into more detail of the code using SonarCube where we identify some of the code smells within Compose main components. Overall, Compose code smells dominate in the `test` component while there are only a few detected in the `config`, `cli` or `compose` component. `config` component itself only contains 14 code smells. Fig.8 shows an example of it where a method Cognitive Complexity score higher than allowed. Cognitive Complexity is a measure of how hard the control flow of a function is to understand. Functions with high Cognitive Complexity will be difficult to maintain [30].

6.5.4 Testing Assessment

The testing environment used by the Docker Compose project is pytest[31]. The tests are:

- End-To-End tests
- Integration tests
- Unit tests

These tests are performed by the continuous integration system of the Docker Compose project. All the tests are done using Python 2.7 and Python 3.7, for Alpine and Debian Docker containers.

Each End to End tests a `docker-compose.yaml` configuration file. The configuration files can be found in the folder `/tests/fixtures`.

[30] Cognitive Complexity: A new way of measuring understandability. https://www.sonarsource.com/docs/CognitiveComplexity.pdf
[31] Just Say No to More End-to-End Tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html?

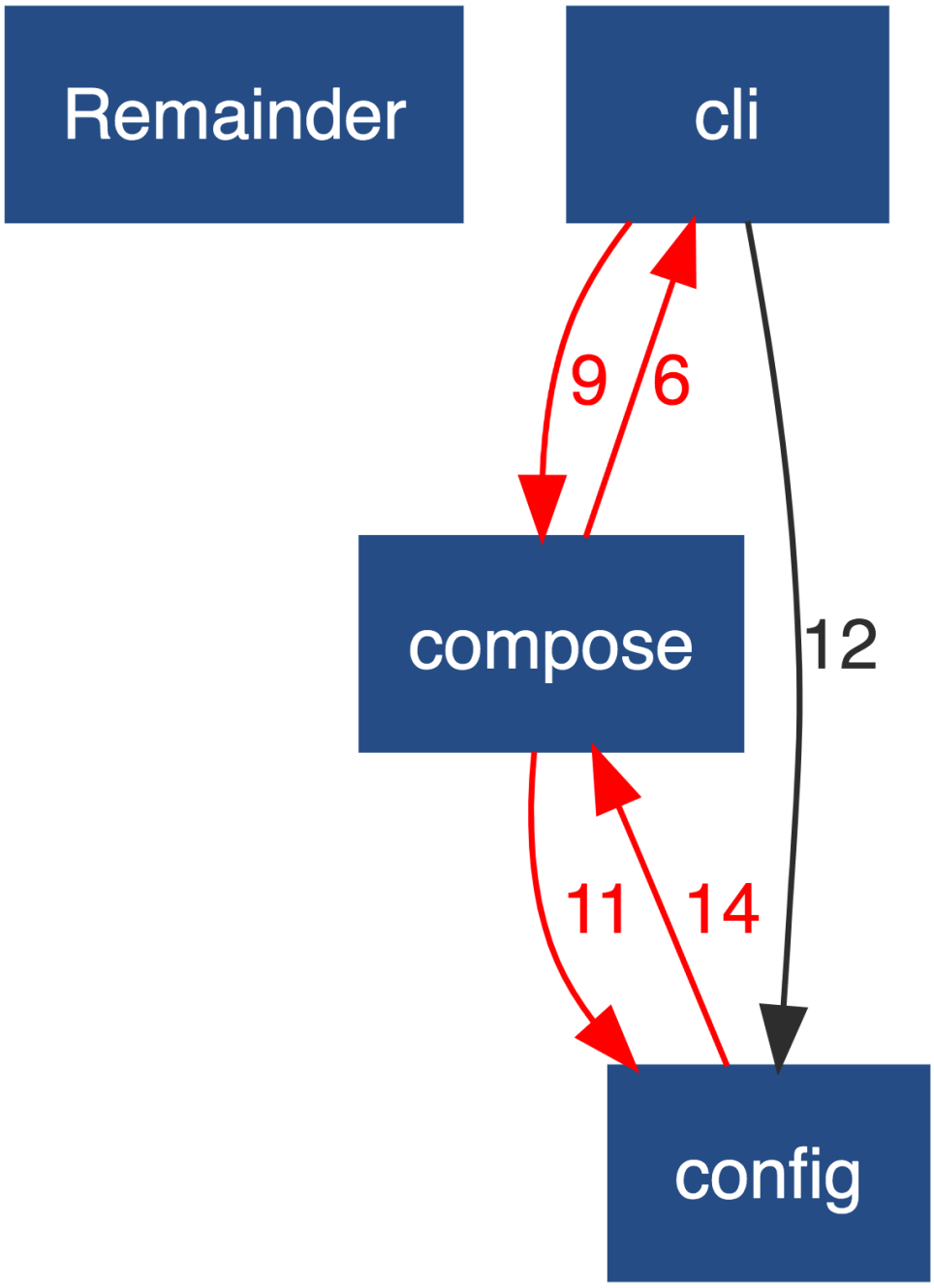


Figure 6.14: Compose component entanglement

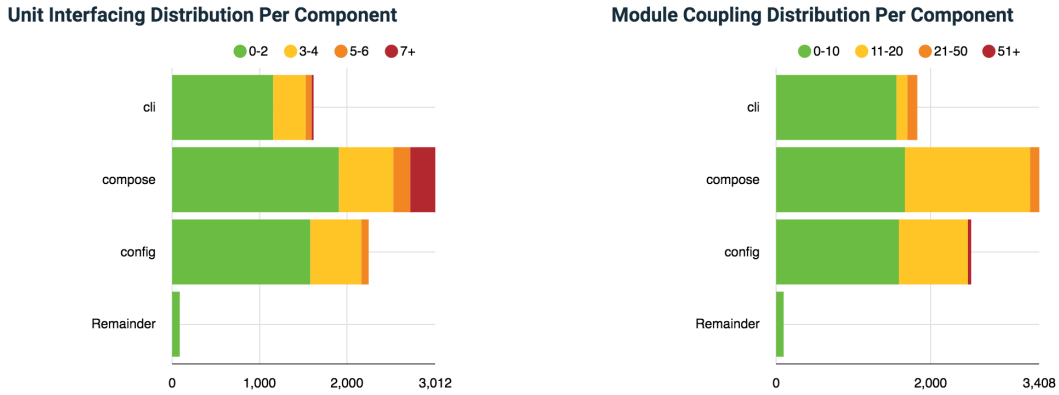


Figure 6.15: Compose module and unit interfacing



Figure 6.16: Compose code smell example

Integration tests don't test specific `docker-compose.yaml` config file, but rather test higher-level components like `volume`, `network` and `service`.

Unit tests test smaller components. In this project, there are three subcategories of tests:

- **cli**: tests regarding parsing of the cli command call
- **config**: tests regarding parsing of the `docker-compose.yaml` or `.env` configuration file
- **generic**: tests that don't fall in either of those categories.

6.5.5 Test Assessment

SIG system reported the test to code ratio around 200%. This is one of the projects with the highest test to code ratio analyzed in the SIG system for DESOSA 2020.

The lines of code are distributed between test types as represented in the following table. As you can see, the percentages of lines of code are somewhat compliant with the testing pyramid[32]

Test type	lines of code	percentage lines of code
End-To-End	2951	15%
Integration	4765	25%
Unit	11207	60%

If we take a look at the coverage percentage (Fig.9), we can see in the following figure that every file in the project receives on average a roughly 90% coverage. The coverage table and the coverage percentage was computed Coverage.py.

6.5.6 Community Awareness

On the discussion in Docker Compose GitHub page, we analyze each of the labels of both open and closed issues and pull requests to indicate the concern of the discussion itself (Fig.10). As seen on figure, label `kind/feature` and `kind/bug` sit on top of most discussion indicating most efforts done on improving features and resolving bugs as technical debt, respectively. There also has been some works done on testing improvement (label `area/tests`) although not as much while none indicate high interest or concern on code quality.

6.5.7 Refactoring Suggestions

We use Sigrid to analyze refactoring suggestions.

System Properties - Units (function/methods), Components	Description	Performance
Code Duplication	Harder to maintain when fixing bugs and adding changes	4.9/ 5.0
Unit Size	Desirable to build single responsibility handling units	2.5/5.0

[32] Just Say No to More End-to-End Tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html?

System Properties - Units (function/methods), Components	Description	Performance
Unit Complexity	Low complexity means fewer test cases required, easier to understand and fewer execution paths needed	2.0/5.0
Unit Interfacing	Units with more parameters have larger interfaces thus are harder to modify and error-prone.	1.6/5.0
Module coupling	Loosely coupled modules are easier to understand, test and change.	1.8/5.0
Component Independence	Loosely coupling components inhibits changes in one component affecting the other.	0.5/ 5.0
Component entanglement	Limiting communication lines between components makes it easier to change components in isolation to further extend the architecture	1.0/5.0

6.5.7.1 Unit size

Since Docker Compose only has a few units, it is expected that the size of these units will be large (Fig.11).

6.5.7.2 Unit Complexity

The complexity occurring in cli.py is due to the **docker compose up** command being able to build, (re)create, start, and attach to containers for a service (Fig.12). These responsibilities can be enabled through **multiple command lines instead of one**. The next complexity is caused in service.py. A common theme within this service.py file is the lack of abstraction.

6.5.7.3 Unit Interfacing

The extensive parameters show up in **build** functions and functions/methods related to **docker compose up** (Fig.13).

6.5.7.4 Module Coupling

Errors.py and utils.py have most coupling. This is expected as errors.py and utils.py have most dependencies on other modules. This seems inevitable (Fig.14).

6.5.7.5 Component Independence

Many functions are dependent on the services (Fig.15). A manifestation of this for the end-user is that services are built once, and any change to the **service requires rebuilding** again. To facilitate component

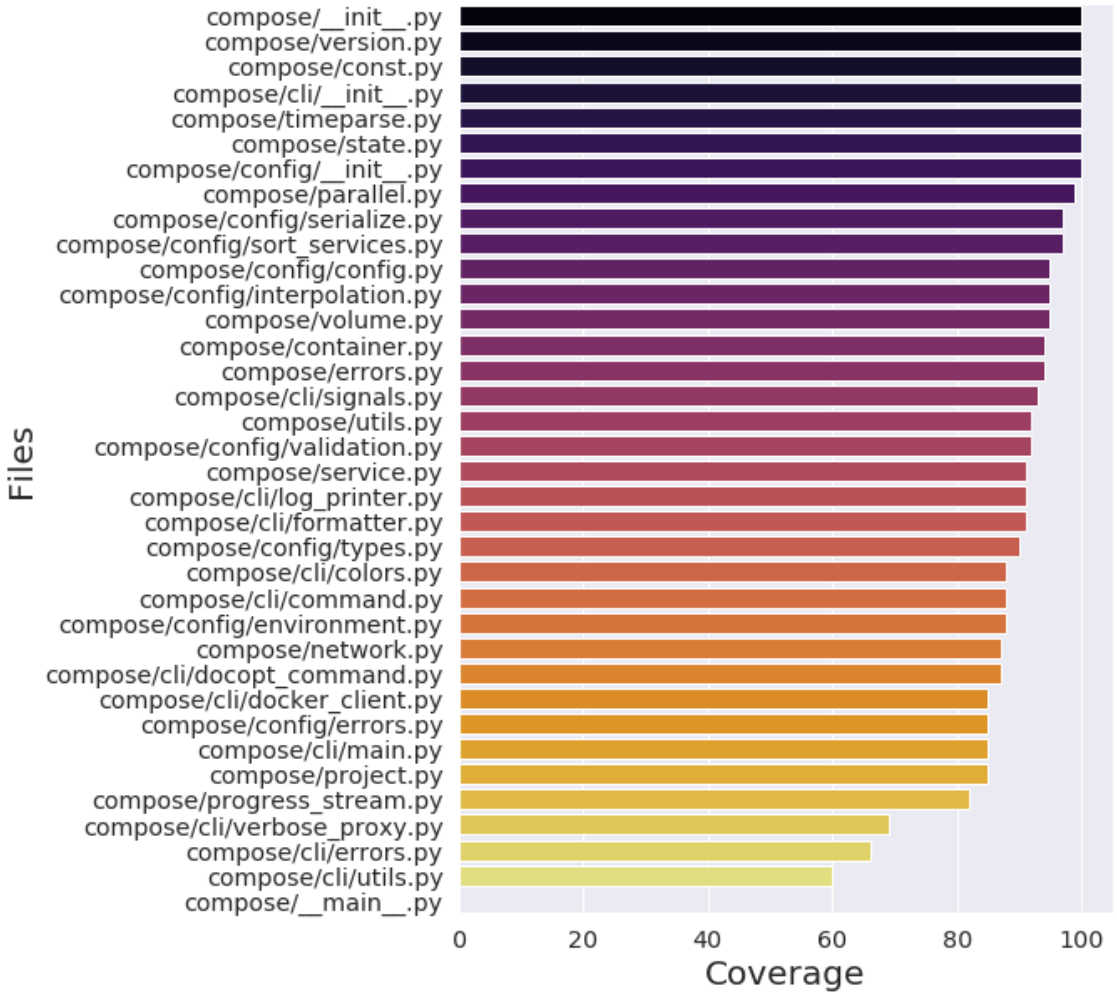


Figure 6.17: coverage

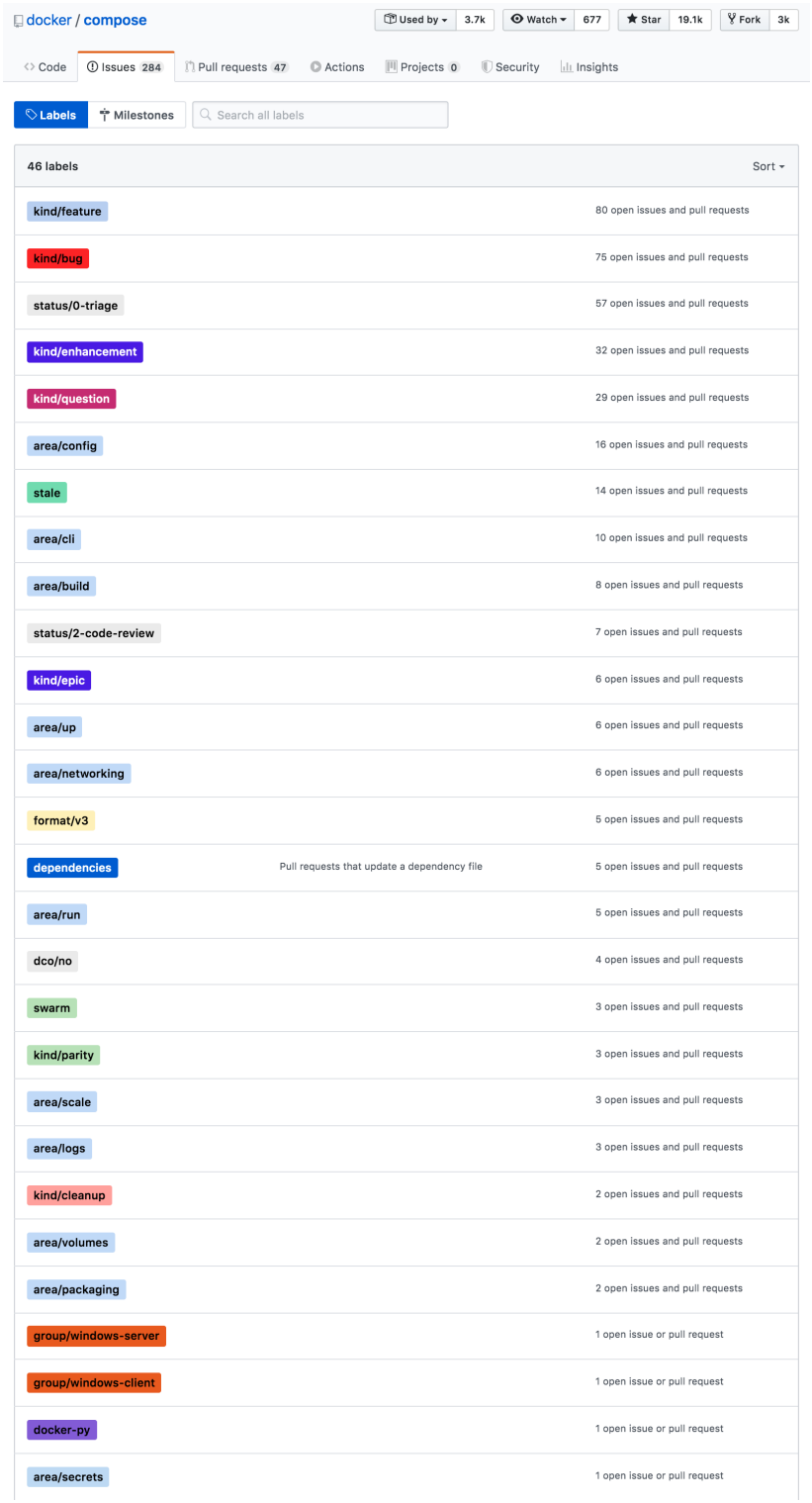


Figure 6.18: Compose issues and PRs

Name	Lines of code	McCabe complexity	Number of parameters	Component	Technology
config.py	99	1	0	compose/config	python
TopLevelCommand.up(options)	78	18	1	compose/cli	python
setup.py	76	9	0	Remainder	python
Service._get_container_host_config(override_options,one_	75	5	2	compose	python

Figure 6.19: Unit Size

Name	Lines of code	McCabe complexity	Number of parameters	Component	Technology
TopLevelCommand.up(options)	78	18	1	compose/cli	python
Service._get_container_create_options(override_options,n	61	19	4	compose	python
Service.build(no_cache,pull,force_rm,memory,build_args_(	53	15	10	compose	python

Figure 6.20: Unit Complexity

Name	Lines of code	McCabe complexity	Number of parameters	Component	Technology
Service.build(no_cache,pull,force_rm,memory,build_args_(	53	15	10	compose	python
Project.up(service_names,start_deps,strategy,do_build,tim	48	10	16	compose	python
_CLIBuilder.build(path,tag,quiet,fileobj,nocache,rm,timeou	35	7	26	compose	python

Figure 6.21: Component Interfacing

Name	Lines of code	Fan-in	Component	Technology
errors.py	43	69	compose/config	python
errors.py	129	33	compose/cli	python
utils.py	121	29	compose	python
service.py	1373	17	compose	python

Figure 6.22: Module Coupling

independence, an **architectural refactoring** seems the best way to go.

Name	Lines of code	Component	Technology
service.py	1373	compose	python
config.py	1153	compose/config	python
main.py	960	compose/cli	python

Figure 6.23: Component Independence

6.5.7.6 Component Entanglement

Compose, compose/cli, and compose/config are cyclically dependent on each other. This entanglement is expected due to the monolithic nature of Compose (Fig.16). (See Architectural Refactoring)

Description	Weight
Cyclic dependency between compose and compose/config.	11
Cyclic dependency between compose and compose/cli.	6

Figure 6.24: Component Entanglement

6.5.7.7 Architectural Refactoring

As discussed in the previous essay, Compose is based on a monolithic architecture pattern. To refresh, a monolithic architecture allows simplicity in development and testing at the cost of size and complexity. At some point, the system becomes too large and complex to understand/change.

We suspect that refactoring in this system is not limiting to code refactoring, however, it requires a complete architectural refactoring.

6.5.8 Technical debt present in the system

We refer to an academic definition of technical debt defined as :

> "Technical debt describes the consequences of software development actions that intentionally or unintentionally prioritize client value and/or project constraints"

We divide the list of identified technical debts [33] into three categories:

6.5.8.1 Planned

- Cyclic dependencies between components (i.e. compose, compose/cli and compose/config)

[33] arc42 Documentation. https://docs.arc42.org/section-11/

- **docker compose up** and build functions having large interfaces

6.5.8.2 Unintentional

- Lack of coding guidelines relating to feature developments
- Lack of testing standards and guidelines
- No testing coverage functionality
- Single Responsibility Principle violated (e.g. in Services.py)
- Lack of abstraction (e.g. increased complexity in units)

6.5.8.3 Inevitable

- High coupling (i.e. between projects.py, services.py, and config.py)
- Certain components having high complexity
- Components having overall higher dependencies

6.6 System and People Collaboration in Docker Compose

Until now, we analyzed Docker Compose from various traditional software engineering dependencies stand points (development view, runtime view, operations view). This does not give us a full perspective of the architectural structure of the project.

In this essay, we will tackle the project from a social perspective.

In **Module Coupling**, we will tackle Docker Compose modularization, as it is a reflection of the organizational structure of Docker Compose, as stated by Conway's law [34].

As demonstrated by [35] logical dependencies[36] capture a more realistic component dependencies. We will also investigate the methods that Docker Compose uses to tackle coordination. As shown by [37], congruency between dependency and coordination is a major factor for the efficiency of delivery time from a change request. Of course, because we are working in a open-source framework, change requests can be seen as issues. This will be discussed in **Developers Communication in Docker Compose**.

Finally, we analyze whether Docker Compose code structure is a reflection of the social structure in Docker compose in **Effects Between Code and People**.

6.6.1 Module Coupling

Why does module coupling matter in this case study?

[34] Conway, M. E. (1968). How do committees invent. Datamation, 14(4), 28-31.

[35] Cataldo, M., Herbsleb, J. D., & Carley, K. M. (2008, October). Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement (pp. 2-11).

[36] Gall, H., Hajek, K., & Jazayeri, M. (1998, November). Detection of logical coupling based on product release history. In Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272) (pp. 190-198). IEEE.

[37] Cataldo, M., Herbsleb, J. D., & Carley, K. M. (2008, October). Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement (pp. 2-11).

According to Conway's law, software structure is a reflection of the organizational structure of its developers. Since Compose only has three high-level components (see previous essays), thus we visualize the structure of lower-level modules within these components.

We use Sigrid to obtain dependencies within modules of Docker Compose. Using this information, we visualize significant components and their dependencies to obtain an idea on the structuring. This can be seen in Figure 1. The colors, green, blue, and yellow, indicate the level of coupling (blue = high, green = medium and yellow = low coupling). The coupling level is determined by Sigrid based on the number and strength of incoming dependencies. A **link** between two modules indicates a dependency between them, whilst the **arrow** indicates the direction of dependency. The **thickness** of a link determines the relative strength of the dependency within the graph so the thicker the link is, the stronger the dependency. It is important to note that segregation of levels and strength are relative to one and other, rather than based on a pre-defined value.

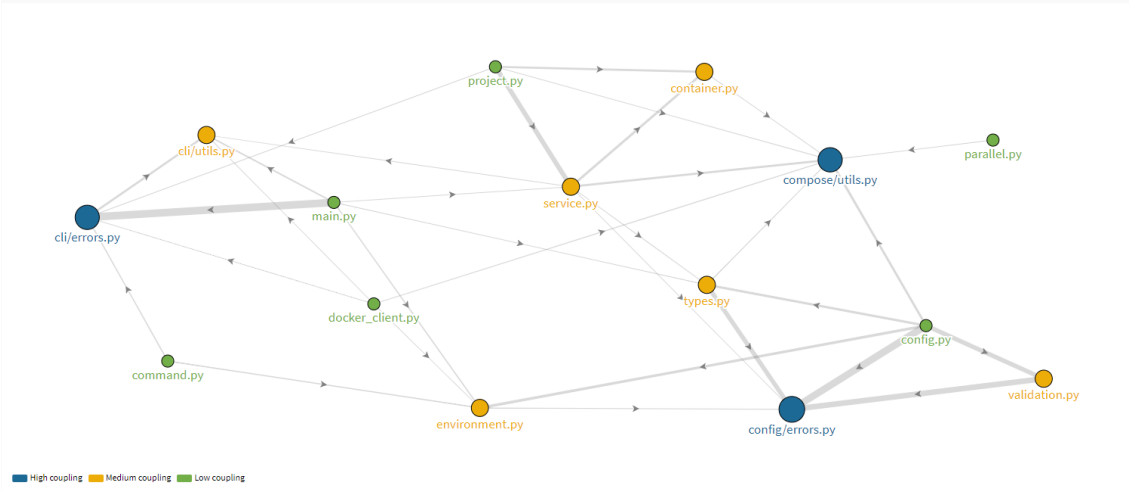


Figure 6.25: Overall module coupling structure

Below, we discuss some interesting module couplings we see in Figure 1.

Highly coupled modules - **config/errors.py**: Handles errors such as circular references, dependency errors and so on. It is highly coupled with many modules as it defines errors for multiple modules within *config*. - **cli/errors.py**: Handles errors regarding *cli* module such as timeout, api and connection errors. Most of its coupling is with modules of *cli*. - **compose/utils.py**]: Handles parsing jobs. It is coupled with components from *config*. The strength of depedencies is lower than *config/errors.py* and *cli/errors.py*.

Medium coupled modules - **environment.py**: Handles representation of the environment, and parsing the .env file. There is more variation in the number of incoming and outgoing dependencies compared to highly coupled modules. It is directly/indirectly connected to most highly coupled modules. - **container.py**: Handles representation of a container. We see a similar pattern of coupling as in *environment.py*. - **service.py**: Handles representation of a service, and building containers related to the service. We see a similar pattern of coupling as in *environment.py*.

Low coupled modules - **main.py**: Responsible for actually defining and running multi-container applications with Docker. It is loosely coupled, which is expected as *main.py* is only run when absolutely needed. -

config.py: Aggregates all relevant information from *config* components and processes them.

What can we infer from the structure of the coupled modules?

If we were to view the module coupling as a network, we could easily point out communities and how they are connected. For example, highly coupled modules can be seen as the *core* of a community (see Figures 2, 3, 4). Medium coupled modules can be seen as *community bridges*. Community bridges can be interpreted as nodes that efficiently connect different communities. Low coupled modules can be seen as either weak community bridges, or simply facilitating a role within a community.

Why are we suddenly comparing coupled modules with communities/networks?

The organization of collaborators is intuitively characterizable by ideologies such as communities and networks. Therefore, by trying to understand module coupling within Docker Compose through the lens of a network, we aim to guide the reader to the next sections.

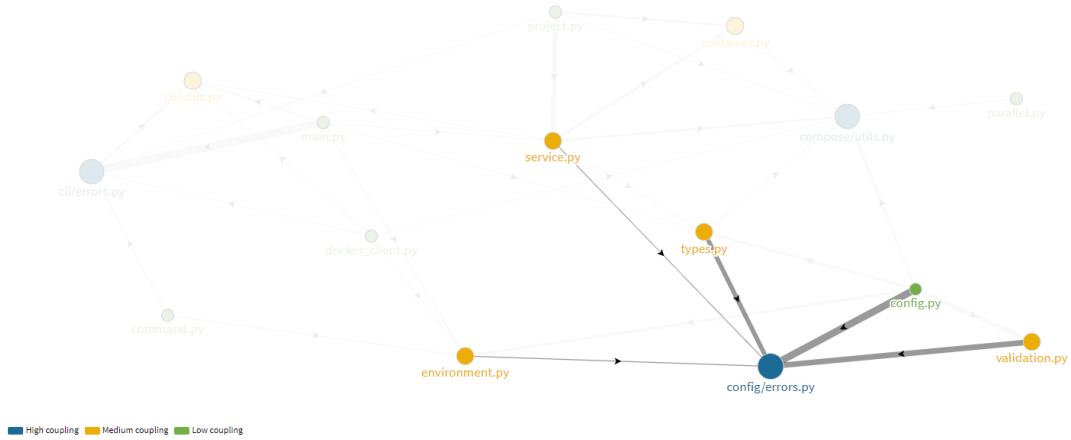


Figure 6.26: Module coupling for config/errors.py

6.6.2 Developers Communication in Docker Compose

How do developers involved in Docker Compose organized? How do they communicate and coordinate with each other?

Docker Compose communication consists of two public channels; Github and Slack. Slack is more about Q&A on the user side of Docker Compose, and Github is where the developer's communication happens. Inside *Docker Inc*, developers rotate teams in the Docker Ecosystem (check our interview with Sebastiaan van Stijn for more details), but they follow the communication methodology described in Figure 5.

GitHub is the main channel, where communication spins around issues supported by a short description, with a blog-style discussion system provided by Github that allows sequential order of comments, and a well-defined tagging system customized by the Docker team. To keep this channel active, response times, and tracking of activity done by the contributor manager and response times by Docker developers play a significant role.

Issues could be of different kind:

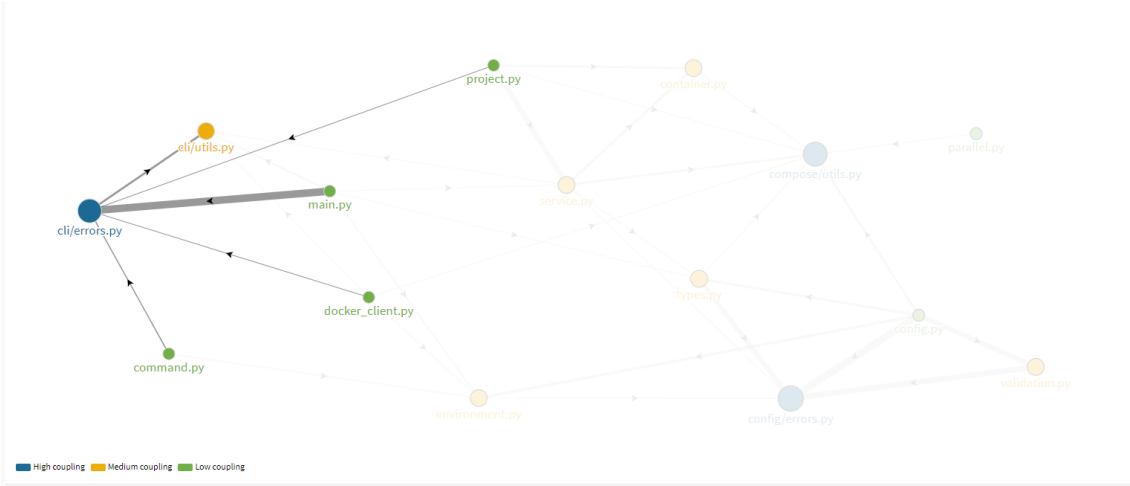


Figure 6.27: Module coupling for cli/errors.py

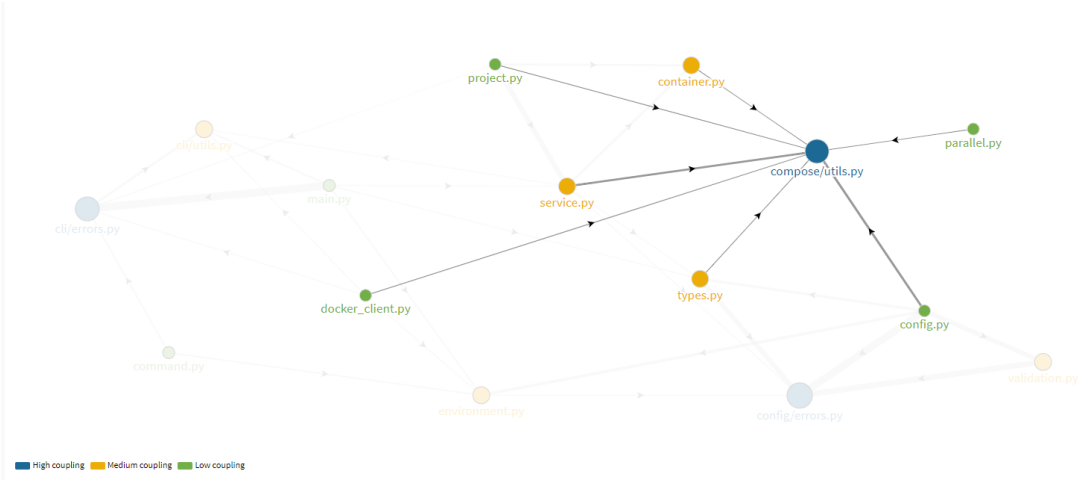


Figure 6.28: Module coupling for compose/utils.py

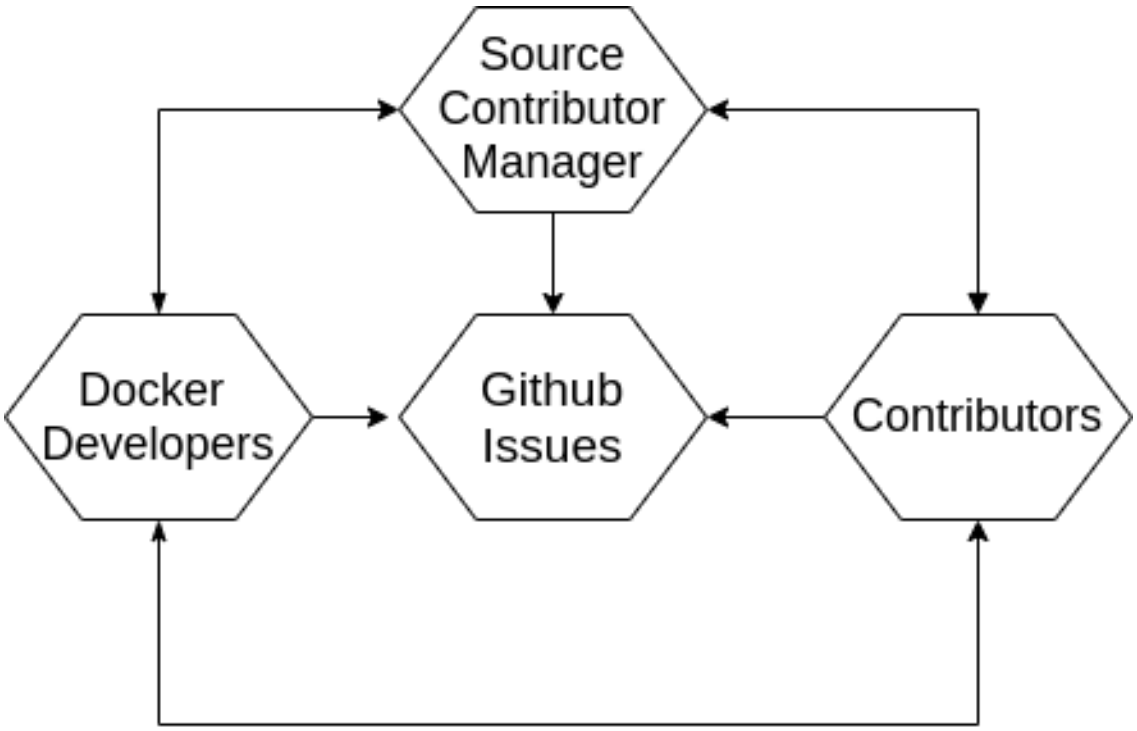


Figure 6.29: Communication spins around issues

- Epics
- Stories
- Custom Tagged

Docker Compose developers also open up their discussions to the public where we can read about their ideas and challenges regarding an issue and follow an agile process. For example, take a look at this epic and this issue regarding a feature.

The rules become more strict and follow a well-defined process described in the contributing guideline. One of the most significant moments is the merge request, which contains a code review. Merge requests might open a conversation where contributors and developers together discuss the solution.

Click me to see how a merge Request looks like!

To summarize, the most interesting aspect of the communication in Docker Compose is that with clear guidelines, test coverage, and constant community management is possible to have over the life of a system different team members and external contributors working together. The positive effect of this methodology is that it allows collective code ownership or shared code and an increase in Busfactor.

Finally, we think that the engineering of this system, mainly the continuous integration pipeline, and test coverage are the elements that allow a clear communication process with the rotation of developers as they ease coordination, particularly with external contributors.

6.6.3 Effects Between Code and People

> Is there a relationship between Compose code system design and the organization of the people itself?

Inevitably, a system organization is highly affected or affecting the organization of the people involved. Conway's Law [38] explained that the choices made by people from the organization before and when designing any system often fundamentally shapes the final output of the design. Conway view a system of both code and people can be illustrated by this linear graph shown in Figure 6. As stated from Conway's paper [39], this linear graphs provides an abstraction which has the same form for the two entities we are considering: the design organization and the system it designs. A system is a commitee in the people organization, while subsystem represent subcommitte and interface is a coordinator of the subcommittee.

If we compare the two network graphs of the code modules organization coupling and developers communcation from previous section above, we do not see any similarity in these two graphs. As the main code components of Compose are `cli`, `config` and `compose`, these importance should somehow reflected in the communication system of Compose's developers that includes the contributors. From the code system perspective, the importances are also not represented because on each main component the high coupling centered around part of code that functioning on error configuration. Analyzing this using Conway's principle, this fact about code coupling are delivering a message that errors are the main concerns of the organization while that is not true.

However, we don't see the way that the developers are organized is following Conway's principle as it is not a representation of the Compose's system design. This might also be the reason of why there's a faulty in the coupling of the modules because the developers communication network is not aligned with the Compose system design itself. The subcommittees, namely `Source Contributor Manager`, `Contributors`, and

[38] Conway, M. E. (1968). How do committees invent. Datamation, 14(4), 28-31.
[39] Conway, M. E. (1968). How do committees invent. Datamation, 14(4), 28-31.

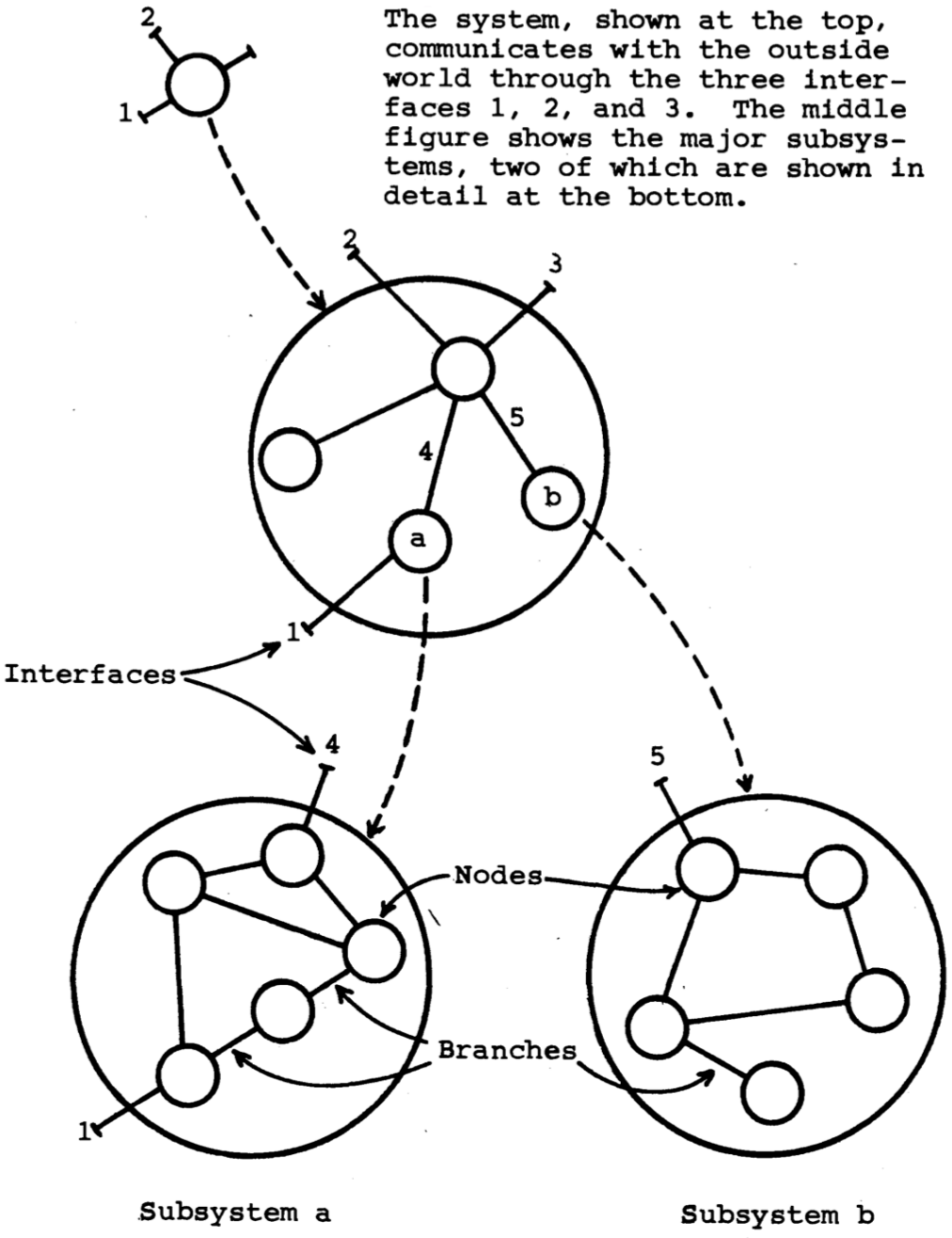


Figure 6.30: Conway's Linear Graphs

`Docker Developers`, doesn't follow the organization in the Compose system code. We don't see any coordinator specifically assigned for each subcommittess as well. The main coordination is held by the role of Source Contributor Manager (that now consist of 3 people) through GitHub issues discussion and pull request reviews.

Although, we see a small representation of Compose components and modules in the way the Github Issues are organized through labels as shown in Figure 7. Still, these labeling systems does not entirely sync up with the Compose modules. There's a label `area/packaging` that covers a different type of concept around the system that is not code modulation because there are no particular code module designed to handle packaging.



Figure 6.31: Compose Github Issues Labels

The code contributing guideline only communicates the general coding guidelines and not particularly about how code should be organized between modules and components. So we see that lack of communication around how source code should be correctly organized leads to things such as false module coupling.

Chapter 7

ESLint

ESLint is a well-known open source project. Initially, it was started by Nicholas C. Zakas in June 2013. It is a tool for identifying and reporting on problematic patterns found in ECMAScript/JavaScript code. Traditionally, code execution was needed for finding errors or syntax errors. ESLint prevents this by supporting static analysis. It supports both style checking and traditional linting of code. The code analysis is performed based on rules, which are configurable and customizable. This enables developers to write their own analysis rules which work alongside the default ESLint rules. Using ESLint, it is possible to automatically fix problematic code.

ESLint is supported by several companies, individuals and organizations. Donations enable the maintenance and development of ESLint. The team consists of a Technical Steering Committee, Reviewers and Committers. The open source community is welcome to suggest changes, bug fixes and contributions.

The entire project is written in JavaScript, it requires Node.JS and it works on Windows, Mac and Linux. ESLint is used by several companies and organizations, some which can be found at https://eslint.org/users.

The next posts will contain an in-depth analysis of the ESLint architecture.

7.1 The vision behind ESLint's success

Understanding the product vision of ESLint is a good starting point for our architectural analysis. The vision of a product has an impact on design decisions. We cover the vision in six sections. First, the intended achievements of ESLint (section 1) and a description of the end-user mental model (section 2). Other key points of ESLint's vision are the key capabilities and properties of the system (section 3), the stakeholders (section 4) and the current and future context (section 5). We end the product vision with a detailed roadmap, containing the future focus of ESLint, in section 6.

7.1.1 Goal of ESLint

ESLint is focused on helping developers code. It is a tool for identifying and reporting on problematic patterns found in ECMAScript/JavaScript code. JavaScript is a weakly typed programming language and errors are hard to avoid. Traditionally, code execution was needed to find the errors. This is the time-

consuming problem that ESLint solves, it discovers problems without the need for code execution. ESLint has several ideas to achieve this goal in the most convenient way for users.

These ideas can be found on the website of ESLint, which contains detailed information about the project[1]. The website states the primary reason why ESLint was created:

> "The primary reason ESLint was created was to allow developers to create their own linting rules." - JS Foundation and other contributors, source

Since every developer has different needs and requirements, it is convenient that ESLint allows custom rules. Additionally, it is developed with JavaScript and Node.js, which results in a convenient installation for developers (with npm) and fast runtime. Based on the quote and this statement, we conclude that ESLint is very user-oriented. The front page of the website displays three main functionalities[2]:

- **Find problems:** using static analysis.
- **Fix automatically:** solve the found problems automatically.
- **Customize:** write your own customized rules, that can work alongside the default rules of ESLint.

All these functionalities help ESLint to achieve their goal of helping developers code with JavaScript in a time-efficient way.

7.1.2 End-User Mental Model

The target audience of ESLint are developers that use either ECMAScript or JavaScript as language. ESLint supports these end-users by finding mistakes or vulnerabilities without the need for code execution. This aids them in writing mistake and vulnerability free code. ESLint needs to support developers in the least time-consuming way possible.

The end-user mental model for the developer is straightforward. The first aspect is that the developers want to install ESLint with a single command, after which they can immediately start using the tool with sane defaults. Secondly, when the need arises they want to be able to have their own linting rules. ESLint needs to support these custom rules of developers. The last aspect is that customizing the rules should be easy. Rules should be configurable with a simple config file and a collection of battle-tested rules should be available.

7.1.3 Capabilities and Properties

In order to accelerate developement and keep the project on track, it is important to keep an overview of the capabilities and properties.

7.1.3.1 Capabilites

ESLint is very straight forward in what it wants to achieve, but does it well. We can identify two main features that make up the ESLint project. With these two capabilities, ESLint is able to achieve its goal of helping developers with their projects.

- **Find problems in projects**: The ability to identify problems in a projects allows developers to create higher quality code and identify bad practices quicker.

[1] JS Foundation and other contributors, website
[2] JS Foundation and other contributors, website

- **Fix issues automatically**: ESLint can automatically fix issues based on the problems found. This increases the productivity of developers drastically. Developers will spend much less time fixing problems since ESLint will fix a lot of them automatically.

7.1.3.2 Properties

In this part, we will focus more on the properties of ESLint outside of analyzing and fixing source code. This is more focussed on how developers are able to incorporate ESLint into their workflow and what properties of ESLint allow developers to interact with ESLint easily.

- **Integration with text editors**: ESLint has integration with many text editors which makes ESLint easily accessible.
- **Continuous integration**: ESLint supports the use continuous integration to allow projects to maintain a certain code quality by setting a limit to the amount of errors/warnings for example.
- **Customizable rules**: Since ESLint is open-source, any developer can add its own rules. In the long term this will make ESLint much better, since new rules will only increase code quality even further.

7.1.4 Stakeholder Analysis

In *Software Systems Architecture*[3], Rozanski and Woods, stakeholders are defined as "groups of people with their own interests, requirements and needs they need from the system." Using public sources, ranging from GitHub issues to websites, we performed an analysis of these stakeholders. As a result of the project being open source and non-profit, some stakeholders are not as relevant. Identifying the stakeholders is fundamental to understand the role of architecture in the development of ESLint.

In the following table the major stakeholder types, their identified entities, and context is given.

Type	Persons or Groups	Description
Acquirers	Founding developer (Nicholas C. Zakas)	The procurement of the project was originally done by the founder Nicholas C. Zakas in June 2013 who saw a need in a customizable JavaScript linter.
Assessors	Technical steering committee* & maintainers	There is no clear assessor group, due to the open-source and low-risk nature of the project. Legal wise, the maintainers make sure contributors agree with the license agreement.
Communicators	Technical steering committee*	The technical committee communicates the architecture and contribution direction to aid developers in contributing and communicate with (potential) sponsors to gain funding and continue development.
Developers	Technical steering committee*,ESLint Team Contributors	Includes all the people who contribute to the repository: more specifically the ESLint team who are the largest contributors, but also the external small one-off contributors. Improving the system also makes their developing experience better.

[3]Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

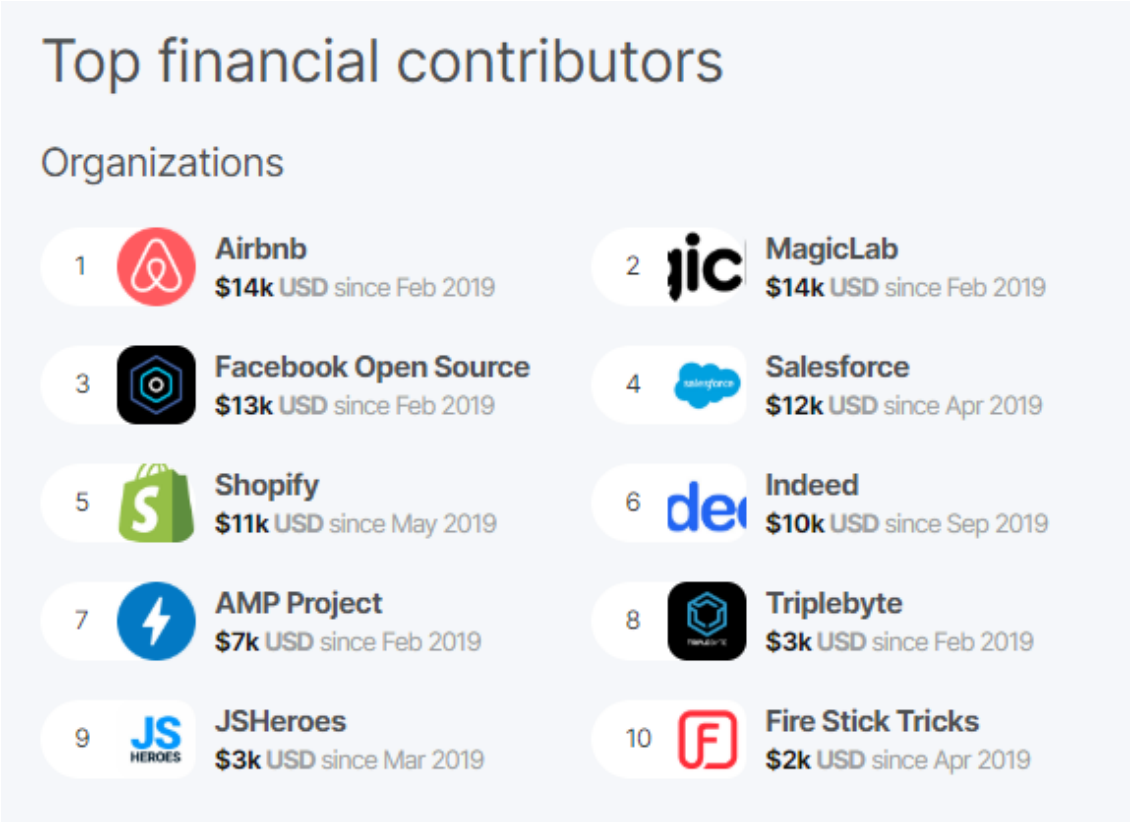
Type	Persons or Groups	Description
Maintainers	ESLint Team	The team ensures that pull requests made by contributors adhere to all standards (quality, testing, documentation) and decides whether a feature should be merged in or not.
Suppliers	NPM, GitHub	Being commercial companies, other than providing a service their interest also lies in attracting large software projects.
Support Staff	ESLint Team, ESLint Community	While there is no dedicated support staff, the main support is given by the maintainers on GitHub issues and the community in Gitter and Google Groups.
System Administrators	Users	Users themselves are system administrators, they run the ESLint system on their projects once it is deployed.
Testers	Developers	We have not identified a separate testing group from developers, new features such as rules instead require developers themselves to supply a test suite, as can be seen here for example.
Users	Developers and companies (e.g. Facebook, Netflix, Airbnb) which employees use ESLint	ESLint is used and installed by millions of developers every week. These users want a system that implements linting rules and is as stable as possible. The interest of the users is that their code quality will increase.

Technical steering committee: (Nicholas C. Zakas, Brandon Mills, Toru Nagashima, Kai Cataldo)

7.1.5 Other Stakeholders

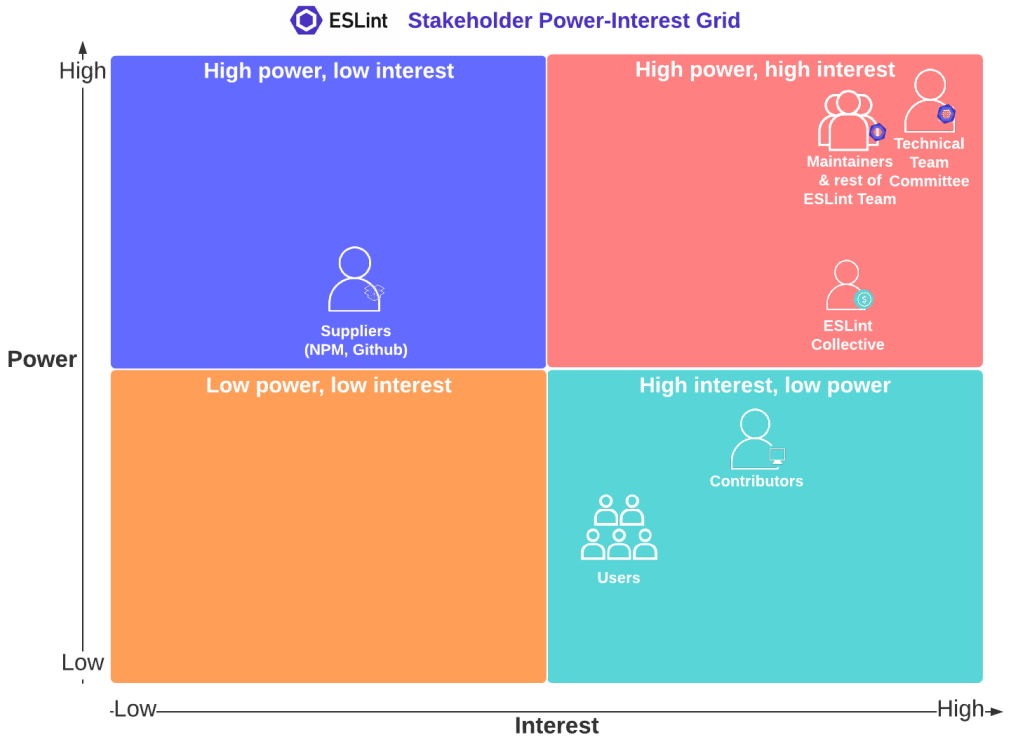
7.1.5.1 Sponsors

A special type of stakeholder we identified are the sponsors, namely the ESLint Collective, consisting of large companies such as Facebook, Airbnb, and Shopify (in addition to individual backers). The purpose of the money mainly is to pay ESLint team members for development. Sponsors, however, do not get any extra influence in the development decisions of the project. Other than some small advertisements on the GitHub page, their interest solely lies in the continued development of the project, as the tool is valuable for their companies.



7.1.6 Power-Interest Grid

Another perspective we can use to analyze stakeholders is a power-interest grid. In the following figure, we lay out the previously identified stakeholders in that grid.



7.1.7 Current and Future Context

As of right now, ESLint works only for Javascript and ECMAScript. So far there are no intentions of extending ESLint to other languages. The main reason ESLint was created is, as stated before, to give developers the liberty to create their own rules. Unlike other tools that focus on coding style, ESLint doesn't force one particular style but allows the developers to choose their own. As for the future, the creators intend to add different types of rules and formatters for the developers to use.

7.1.8 Roadmap

ESLint does not have a clear roadmap, but it creates regular releases. In each release, it is stated which new features have been implemented and which bugs have been fixed.

Aside from that ESLint seems to work with issues on their GitHub repository. The new rules that have to be implemented are listed in a document under different categories.[4] The categories are TypeScript-specific, Functionality, Maintainability, Style, Testing, Miscellaneous, Security and Browser. In this list, it is clear which rules have to be implemented first, but there is no deadline for any.

[4]Github roadmap file for Eslint, website

7.2 Behind the ESLint Architecture

Following our last post, we will discuss the architecture of ESLint. Architectural views, styles and design patterns will be discussed. Also, deployment and non-functional properties will be covered. This post gives an overview of the architecture of ESLint.

7.2.1 Relevant architectural views

In the book, *Software Systems Architecture* [5], Rozanski and Woods describe the concept of using viewpoints and views to aid in the architectural design process. For basic systems, it is possible to create a single architectural design for the complete system. However, for larger complex systems, this is not feasible. You can, however, model smaller parts of the system which together describe the architecture of the complete system. This is what Rozanski and Woods called an architecture view on your system.

> "An architectural view is a way to portray those aspects or elements of the architecture that are relevant to the concerns the view intends to address—and, by implication, the stakeholders for whom those concerns are important." - Software Systems Architecture by N. Rozanski and E. Woods [6]

In the book of Rozanski and Woods, a couple of views for a system are presented, we will use these views for our analysis. However, every architect is, of course, free to choose what views to use for their system, First, we will describe these views shortly, based on the description given in the book [7]:

- **Functional**: Describes the system's functional elements, their responsibilities, interfaces, and primary interactions.
- **Information**: Describes the way that the architecture stores, manipulates, manages, and distributes information.
- **Concurrency**: Describes the concurrency structure of the system that identifies the parts of the system that can be executed concurrently and how this is coordinated and controlled.
- **Development**: Describes the architecture that supports the software development process.
- **Deployment**: Describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment.
- **Operational**: Describes how the system will be operated, administered, and supported when it is running in its production environment.

For each of these views we will analyze how relevant these views are for ESLint:

- **Functional**: The functional view is an essential part of the architectural model. The functional view will always be important in systems, since the goal and use of the system should be clear to all.
- **Information**: ESLint is focussed on processing source code. It does not need expansive databases or distributed storage systems to function correctly. Therefore this view is a less important one for ESLint.
- **Concurrency**: This is an important view since a vast amount of files are to be scanned. If it was possible to be done in parallel, this could be a nice improvement. However, it is not detrimental if this

[5] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

[6] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

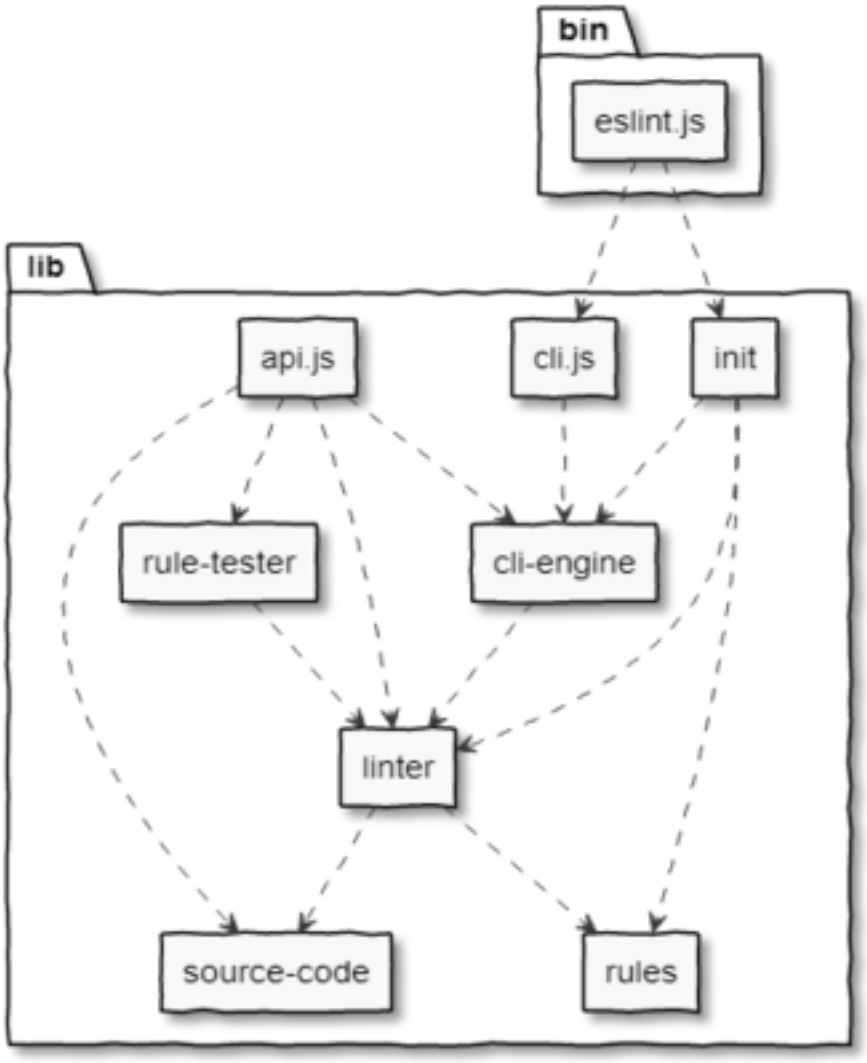
[7] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

is not done since it is not weird to expect a system that scans all source files to take a bit of time to complete. Therefore this view is relevant, up to a point.
- **Development**: Since the ESLint project is open-source, the development view is definitely one to be focussed on. ESLint provides excellent documentation for developers on how to contribute to the ESLint project.[8]
- **Deployment**: The deployment view is reasonably important since ESLint has to be easily usable. Without a good view on how and why users use ESLint, making decisions about the deployment of ESLint will be much harder.
- **Operational**: The operational view is less relevant for ESLint since it is deployed as a package. Once it is installed, users can update the package to a newer version, apart from that, it 'just works'.

7.2.2 Architectural Style and Design Patterns

On the first inspection of the organization of the system, it is clear that ESLint is rigorously based on the architectural style of componentization. This is also documented by themselves[9] and will be elaborated upon in the next section(s). The figure below shows the separation of components. Components such as `Rules` and `Source-Code` do not have dependencies, because other components, like `Linter`, tie these together. The isolated modules allow for better testing, higher code quality and lower the threshold for new contributors since they can get up to speed rapidly. These are important properties for a widely used system that is open-source.

[8] https://eslint.org/docs/developer-guide/
[9] https://eslint.org/docs/developer-guide/architecture



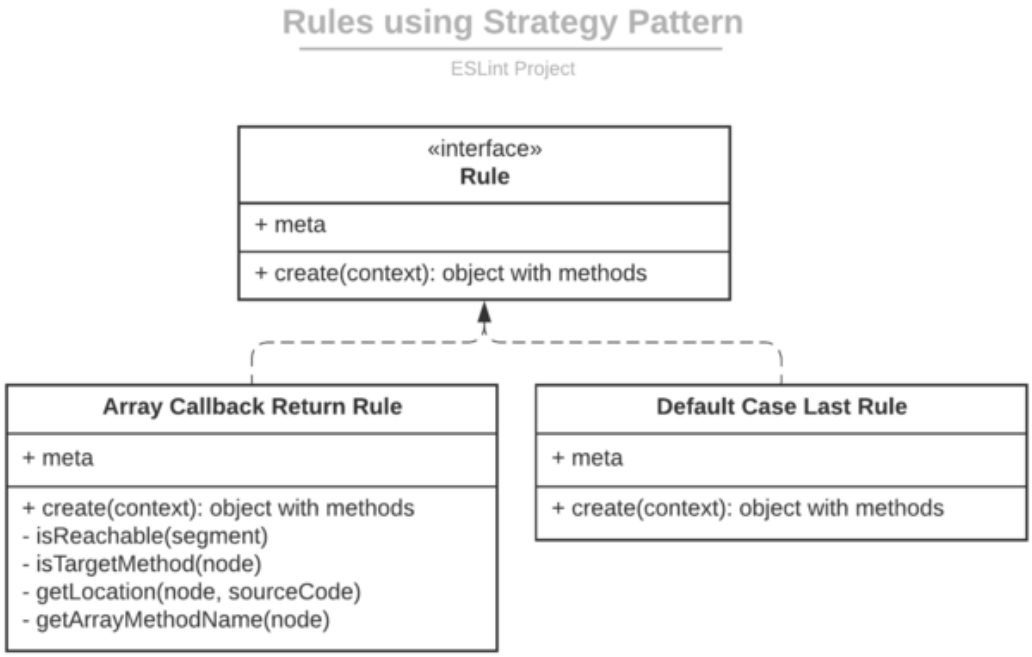
The architecture of ESLint, as supplied by the team themselves[10].

If we venture a step lower, we can't ignore the strategy design pattern[11]. This pattern solves the problem of adding and using new rules with custom linting behavior, without changing the other components such as the actual `Linter` engine or source code parser `Source-Code`. This is closely related to the open-closed principle, that states:

> "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"[12]

[10] https://eslint.org/docs/developer-guide/architecture
[11] https://en.wikipedia.org/wiki/Strategy_pattern
[12] Meyer, B. (1988). "Object Oriented Software Construction", Printice Hall. Inc., Upper Saddle River, NJ, USA.

Implementing a rule requires the implementation of the (very specific) interface that ESLint has provided. Since Javascript does not have interfaces, it is rather enforced through documentation, test-suites and manual checking by maintainers.



7.2.3 Overview of the ESLint system

In this section the development view will be discussed of ESLint. This describes the architecture that supports the software development process. The stakeholders who have the most use for this point of view are the software developers and testers.

First, we start by having a look at the required tools to develop in ESLint and the general structure of the modules in the repository. To develop in ESLint, 2 external tools are required:

- **Node.JS**
- **npm**

The directory structure of the repository looks as follows[13]:

- **bin**: executable files that are available when ESLint is installed
- **conf**: default configuration information
- **docs**: documentation for the project
- **lib**: contains the source code
  - **formatters**: all source files defining formatters
  - **rules**: all source files defining rules

[13] https://eslint.org/docs/developer-guide/source-code

- **tests**: the main unit test folder
  - **lib**: tests for the source code
    - **formatters**: tests for the formatters
    - **rules**: tests for the rules

Looking at the directory structure, it is obvious that the setup of ESLint is pretty basic. The files in which the developers and testers will primarily work in is the libs and test folders. Here they can develop the rules[14], formatters[15] and write unit tests in the corresponding test folder. Additionally if the developers want to configure the parsers[16] they need to work in the conf folder. External parsers can also be imported into plugins. Plugins[17] serve as the export module for npm which contains the rules and formats that are used in JavaScript.

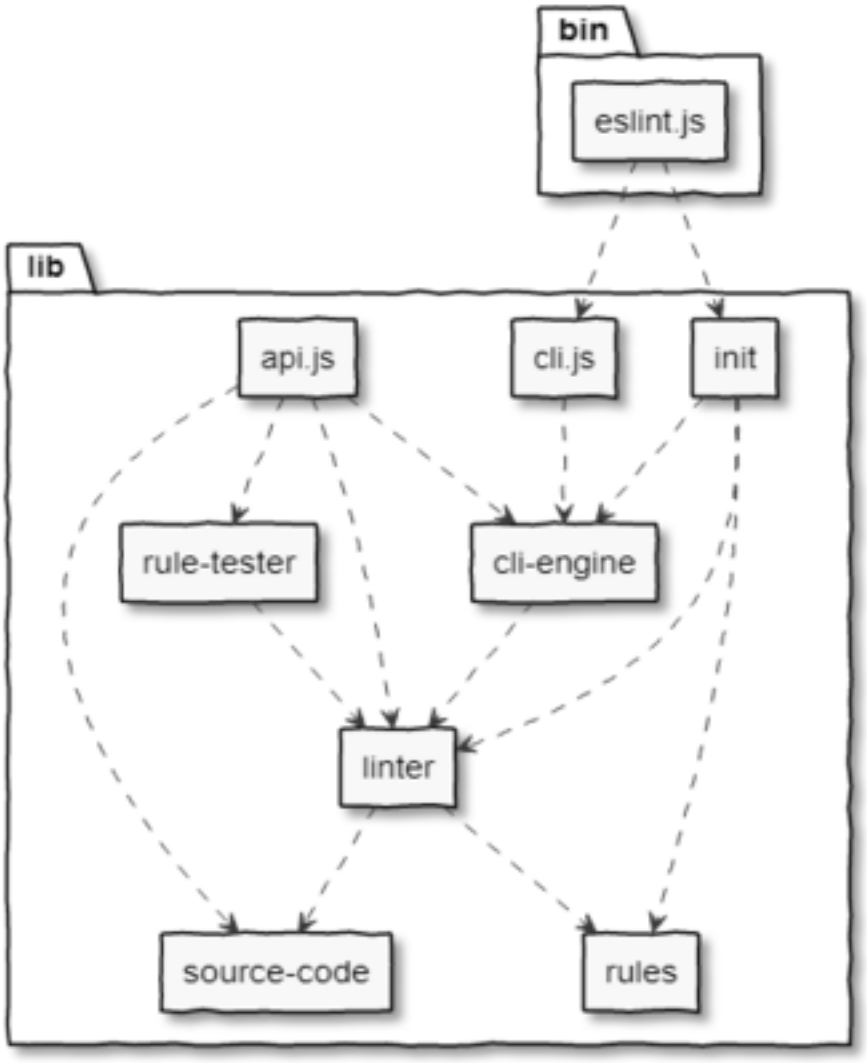
ESLint consists primarily of unit tests. For most files in the source folder there is a corresponding file in the test folder. To develop unit tests the npm module Mocha is required. Mocha provides a skeleton code for unit tests and thus are needed if developers intend to make contributions in ESLint. Finally the tests are run through npm.

7.2.4 ESLint architecture during run-time

ESLint provides a CLI interface that is used to execute ESLint [ˆeslintcli]. Once installed, this binary file, called `eslint` can be found in the `./node_modules/bin/`. The user uses this binary `eslint` file to execute ESLint. Integrations with ESLint, like with text editors, also use this binary file to execute ESLint.

The flow of the ESLint project during run-time is as shown in the following diagram from the ESLint architecture documentation[18]:

[14] https://eslint.org/docs/developer-guide/working-with-rules
[15] https://eslint.org/docs/developer-guide/working-with-custom-formatters
[16] https://eslint.org/docs/developer-guide/working-with-custom-parsers
[17] https://eslint.org/docs/developer-guide/working-with-plugins
[18] https://eslint.org/docs/developer-guide/architecture



We can see that this flow starts from the `eslint` component. We will now explain the flow of ESLint and each of its components during run-time[19], starting from the `eslint` binary:

- Execute `eslint` binary with options and the patterns (a format that allows the user to specify certain files or folders) for all files. This file then calls `cli.js` while passing on all arguments.
- `cli.js` parses all options and the patterns. Next, performs some logging when specified by the user and validates all options specified by the user. Next a "cli-engine"' instance is created using the options and file patterns.
- `cli-engine` iterates all files and uses a `linter` instance to get all the results. This `linter` instance is created using the options passed along to the `cli-engine` instance.
- `linter` is the final 'major' component. The linter inspects the source code, execute rules on the

[19] https://eslint.org/docs/developer-guide/architecture

source code and reports the results. If the user specified the code to be fixed, then the linter also fixes the problems in the source code.

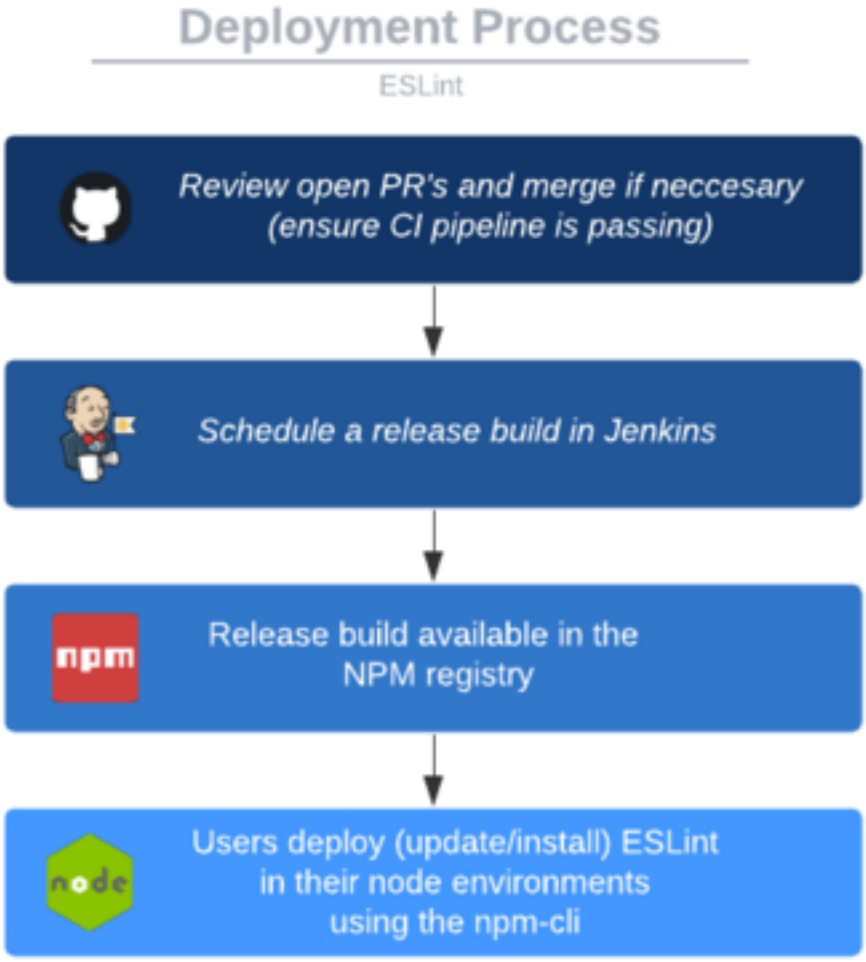
7.2.5 Deployment of ESLint

As explained in the section "Relevant architectural views", the deployment view describes the environment into which ESLint will be deployed including the dependencies it has on its runtime environment, according to Rozanski and Woods[20]. In this section, we will discuss how ESLint is deployed. Before deployment, the system should be tested and should be ready to go live. Otherwise, a potentially broken product will be delivered to the users.

First, we discuss the deployment requirements. The hardware requirements of ESLint are straightforward. A user needs a computer with internet connection that can run NodeJS (version `^8.10.0`, `^10.13.0`, or `>=11.10.1`, retrieved March 14th[21]). Apart from hardware, there are also third-party software requirements. The key player is the NPM packaging system, which is a software registry system that end-users can use to deploy packages into their projects. The ESLint team deploys a new package every week[22], which is done by making the new changes available in their NPM package. NPM takes care of a large number of deployment concerns, such as the installation of the tool, storage and resolving potential dependency conflicts between ESLint and other packages[23].

The complete deployment process is shown in the image below. After changes are made, Jenkins is used to scheduling a release build. Jenkins is an open-source automation server that helps to build, deploying and automating projects[24]. At some point, a six-digit 2FA code has to be entered from an authenticator app[25]. After deployment, the ESLint maintainers keep watching the release to verify that it functions as intended. The deployment is cross-platform, ESLint can be deployed to any operating system[26].

[20] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[21] https://eslint.org/docs/user-guide/getting-started
[22] https://eslint.org/docs/maintainer-guide/releases
[23] guide/releaseshttps://docs.npmjs.com/cli/npm
[24] https://jenkins.io
[25] https://eslint.org/docs/maintainer-guide/releases
[26] https://eslint.org/docs/maintainer-guide/releases



ESLint itself is lightweight, runs on its own and does not have many requirements. For example, it does not need an internet connection to function properly. All it needs is a bit of storage space and processing power. Users can customize the configuration of ESLint to adapt the system to their needs[27].

7.2.6 Non-functional properties

ESLint is designed to run for JavaScript. So on any platform JavaScript is running ESLint can also run. ESLint is easily installed through npm and has a fast runtime environment through Node.js. There is a lot of freedom for the user to choose their coding style. Everything is pluggable, the Formatter API and Rule API are deployed through command lines. Finally, ESLint makes sure that the project is as transparent as possible, so that everyone knows how to contribute to ESLint and what the requirements are to merge it in

[27] https://eslint.org/docs/user-guide/configuring

the project [28].

7.3 ESLint's variability management

In this last post, we will assess and analyze one of the features that ESLint takes pride in and differentiates itself with compared to competitors, namely the large variability in the system realized through configurations. We will go into what is variable, what benefits this has and how this complex configuration can be managed by users in a sane manner. Lastly, we will look at how this is implemented and finish off with an assessment done by ourselves.

7.3.1 Variability Modeling

In this section we will delve into the variabilities of the system, that is, features and settings that can be changed in the ESLint package. We will additionally look at incompatibilities and provide a feature model.

7.3.1.1 Variabilities

While ESLint is the perfect example of a tool that has one goal and does it well, it is also designed to be fully configurable from the start so it can be used in any project.

7.3.1.1.1 Project Configuration

- *Specify code parser*
- *Specify project environment*

The code parser can be configured and swapped out completely. By default it accepts ECMAScript 5 code, so developers who use different dialects e.g. Typescript will want to use `@typescript-eslint/parser` instead. Depending on the code in the project, users will also want to specify a project environment such as `browser` or `node` to load the correct initial global variables.

7.3.1.1.2 Linting Rule Configuration

- *Specify rules*
- *Disable rules using comments in code*
- *Specify ignored files and directories*
- *Specify plugins*

Before running, the user can specify which rules are enabled and which are not. This can be done by defining them in configuration files or using inline comments to disable specific lines or code sections. ESLint also allows the user to ignore files or complete directories and exclude them from the results and linting.

ESLint provides the user with a large set of rules which are very extensive and useful for developers. However, if a user decides that he wants to add more rules, plugins can be installed by specifying plugin repositories in the configuration file. This allows for easy extension of the base ruleset.

[28] https://eslint.org/docs/about/

7.3.1.1.3 Auto Fixing Features

- *Specify the use of auto-fixing code*

The auto fixing feature can be turned on to automatically fix types of linting problems, which can be errors, suggestion or layout problems. Developers benefit from enabling this because they will spend less time on manually fixing problems.

7.3.1.1.4 Output Variability

- *Specify the output method of the result of linting (& warnings etc.)*
- *Specify caching processed files*
- *Debug information*

How ESLint outputs the results can be specified before executing. The output file and format of the results can be adjusted. Debug information can also be included. This benefits users since the user can specify the output to their needs. Debug logs are also helpful for maintainers to solve bugs in reported in issues. On top of that, integrating ESLint can be made easier by using a more 'machine-readable' format like JSON.

7.3.1.2 Variabilities Incompatibilities

- *Linting-rule configuration and auto fix*

ESLint has the feature of automatically fixing your code, however the auto-fix feature might introduce new linting errors. Some rule configurations might thus end up not working well with the auto-fix feature. ESLint tries to solve this problem by repeatedly executing the fixes up to a maximum of 10 times until or no more linting errors are found. [29]

- *Plugin incompatibilities*

A user configuring external plugins should make sure that the rules are compatible. For example, having a rule that marks semi-columns as errors and another rule that marks them as required, will always lead to linting errors being reported.

- *Good configuration*

An example of an allowed configuration of variabilities is laid out in the getting started guide, which consists of the default ESLint rules with the Node platform with a version of `8.10.0` or higher. The linter is then run without any special options, so no use of auto-fix.

7.3.1.3 Feature model

Additionally, we have created a feature model that describes the features laid out in the previous sections. It is used to visualize features and the relationships between them:

> In software development, a feature model is a compact representation of all the products of the Software Product Line (SPL) in terms of "features". [30]

[29] Applying rule fixes, https://eslint.org/docs/developer-guide/working-with-rules#applying-fixes
[30] Feature model, retrieved 7 April, https://en.wikipedia.org/wiki/Feature_model

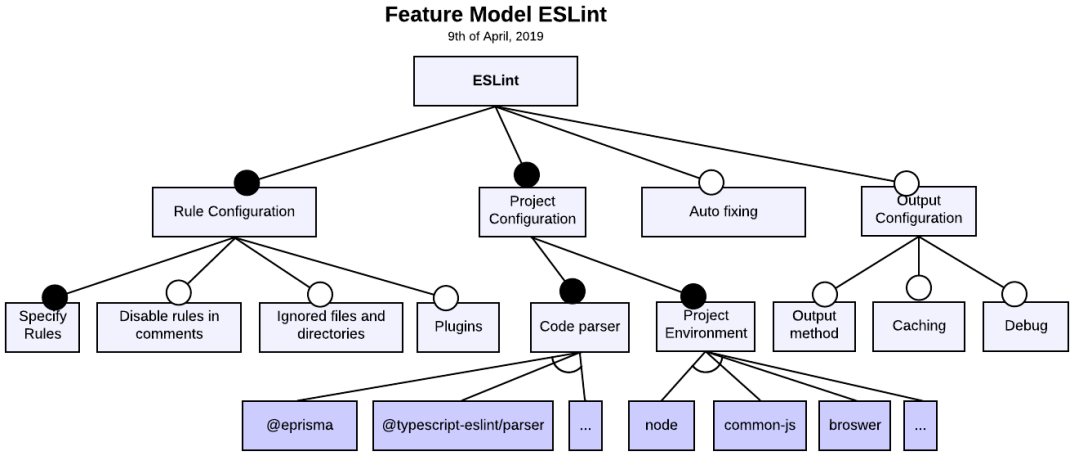


Figure 7.1: Feature model of ESLint

7.3.2 Variability management

ESLint is a system that is used and built by developers. In our blogpost 'The vision behind ESLint's success', we identified 10 stakeholders. All stakeholders that interact with ESLint are developers, either for their project or for ESLint. This is an important fact for our analysis of the variability management of ESLint. The main division that we assume between the stakeholders is that the developer can be a client or ESLint developer. First, we discuss the software variability.

7.3.2.1 Software variability

The variability management for software is restricted to JavaScript or similar languages like TypeScript or ECMAScript since those are the languages for which ESLint provides linting. ESLint provides configuration with clear documentation for software variability management[31]. Additionally, different versions of ESLint can be downloaded. We will explore this and platform variability in the next section: platform variability.

7.3.2.2 Platform variability

ESLint uses two distribution platforms: NodeJS[32] and Yarn[33]. First, we focus on clients of ESLint. Remember that those clients are developers themselves. Developers need NodeJS or Yarn to download ESLint. The download process is very familiar to developers since NodeJS is a frequently used tool for JavaScript and TypeScript development environments. ESLint is a tool that provides linting for those environments, so it is reasonable to assume that developers are already familiar with NodeJS. Apart from NodeJS, Yarn is also a well-known package manager. If a developer has NodeJS, ESLint can be downloaded on any operating system using the command:

[31] ESLint configuration documentation, retrieved 7 April, https://eslint.org/docs/user-guide/configuring
[32] Node.JS, retrieved 7 April, https://eslint.org/
[33] Yarn, retrieved 7 April, https://yarnpkg.com

```
npm install ESLint
```

or

```
yarn add ESLint
```

A specific ESLint version `x.xx` can be downloaded using the following command:

```
npm install ESLint@x.xx
```

As a side note, ESLint can be updated using the same package managers.

We can conclude that developers need NodeJS or Yarn to download ESLint, which shifts the variability aspect to those systems. If NodeJS or Yarn can be installed, ESLint can be installed as well. After some research, we found that both Yarn and NodeJS support a wide range of operating systems[34][35]. NodeJS supports 7 platforms with pre-built installers[36]. If the developer uses another platform or operating system, it is still possible to install NodeJS manually[37]. On the other hand, Yarn offers pre-built installers for 8 operating systems. Other operating systems can be used if NodeJS is installed, which is interesting.

To conclude this section, the installation procedure of ESLint developers is different than the client or user stakeholder group. The source code needs to be downloaded from the ESLint GitHub[38]. The developer can download all required packages by running the following commands in the (downloaded) ESLint folder:

```
cd eslint
npm install
```

Some additional steps can be read in the developer setup documentation of ESLint[39]. The point is that `npm` is used in this installation procedure as well. From this, we can derive that the software and platform variability is very similar for the two important stakeholder groups, client developers and ESLint developers!

7.3.3 Variability implementation mechanism and binding time

In order to understand the design choices made by the developers, we first need to have a look at the implementation mechanisms and binding times of the features. To do this we will have a look at two variable features: specifying the code parser and specifying plugins. With this, we will look at the mechanisms and binding time from 2 different variabilities. From this, we will derive a conclusion to understand the design choices of the developers.

7.3.3.1 Specifying code parser

Eslint uses by default Espree[40] as its parser. However, it is possible to set up a different parser as long as it meets the following requirements:

- It must be a Node module loadable from the config file where it appears.
- It must conform to the parser interface[41]

[34] Node.JS download page, retrieved 7 April, https://nodejs.org/en/download/
[35] Yarn download page, https://classic.yarnpkg.com/en/docs/install#mac-stable
[36] Node.JS download page, retrieved 7 April, https://nodejs.org/en/download/
[37] Node.JS download page, retrieved 7 April, https://nodejs.org/en/download/
[38] ESLint GitHub, retrieved 7 April, https://github.com/eslint/eslint
[39] ESLint developer setup, retrieved 7 April, https://eslint.org/docs/developer-guide/development-environment
[40] Espree repository, https://github.com/eslint/espree
[41] ESLint plugins, https://eslint.org/docs/developer-guide/working-with-plugins#working-with-custom-parsers

The following parsers are compatible with ESLint:

- Esprima[42]
- Babel-ESLint[43] - A wrapper around the Babel parser that makes it compatible with ESLint.
- @typescript-eslint/parser[44] - A parser that converts TypeScript into an ESTree-compatible form so it can be used in ESLint.

In order to add the custom parser in the configuration, the user has to add the `parser` option in his `.eslintrc` file. This rule has to be added with the other rules in the `parserOptions` property.

In the parser options the user is allowed to specify the JavaSript language options that need to be supported. By default, ESLint expects ECMAScript 5 syntax but this can be changed to other versions as well JSX options. The parser options are set in the `.eslintrc.*` file by using the `parserOptions` property. The available options are:

- ecmaVersion - to specify the version of ECMAScript syntax that the user wants to use.
- sourceType - set to "script", default, or "module" if the code is in ECMAScript modules.
- ecmaFeatures - an object indicating with additional language features that can be used:
  - globalReturn - allow return statements in the global scope
  - impliedStrict - enable global strict mode[45]
  - jsx - enable JSX[46]

An example of `.eslintrc.json` file:

```
{
    "parserOptions": {
        "ecmaVersion": 6,
        "sourceType": "module",
        "ecmaFeatures": {
            "jsx": true
        }
    },
    "rules": {
        "semi": "error"
    }
}
```

7.3.3.2 Specifying plugins

ESLint supports the use of third-party plugins. Before using the plugin, you have to install it using npm. To configure the plugins a list of plugins have to be stored in the property `plugins`. In this list the `eslint-plugin-` can also be stored. An example follows:

```
{
    "plugins": [
        "plugin1",
```

[42] Esprima, https://www.npmjs.com/package/esprima
[43] Babel, https://www.npmjs.com/package/babel-eslint
[44] typescript, https://www.npmjs.com/package/@typescript-eslint/parser
[45] strict mode, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
[46] JSX, https://facebook.github.io/jsx/

```
        "eslint-plugin-plugin2"
    ]
}
```

The naming differs based on the scope. For example, giving as a name of a plugin `jquery` would look for a non-scoped package with the name `eslint-plugin-jquery`. Whereas if you want to look for a scoped plugin you will have to type `@jquery`, which will look for the plugin `@jquery/eslint-plugin`. When using rules, environments or configs defined by plugins, they must be referenced following the necessary conventions[47].

By looking at the implementations, we can see that the underlying mechanism is Parameters and Configuration Files. The parameters are fixed and the user can select out of an array of it, by which ESlint will compile. If a change occurs in the configuration file then a reboot is required. As such the mechanism uses a load-time binding. This leads to flexibility in reconfiguration, which is great since ESLint focuses on the freedom of the user and a wide variety of coding styles. The conditions are checked at load time, which is also important as ESLint is run thus making sure that the conditions work at load time is sufficient. This allows for a fast and flexible system, which is exactly what meets ESLints requirements. The system also becomes scalable, as adding new rules/plugins/parsers can be done easily with new constraints and properties in the configuration files. From this, we can conclude that the developers chose the correct implementation mechanisms needed for ESLint.

[47] ESLint configuration documentation, retrieved 7 April, https://eslint.org/docs/user-guide/configuring

Chapter 8

Gatsby



Figure 8.1: The project team working on Gatsby

Gatsby allows developers to create react apps with full functionality, not just static websites. The uniform design language for websites built with Gatsby allows the team to work in a codebase with a single design methodology, while not having to think much about the different sources of data, performance metrics and scalability. The structure for the code is predetermined, and only the website itself has to be built.

8.1 Usage

The Gatsby framework is used to build a variety of applications, from simple single page apps and blogs to fully fledged webshops and complex applications. The project offers the developer a great source of information about building with it in the form of documentation, examples on how to use a feature, starter kits (a functioning starting point which is ready to expand upon) and blog posts doing deep-dives into a topic. If some feature is not available, you're free to implement a plugin which slots into the Gatsby build pipeline and adapt Gatsby to your personal needs.

8.2 The great Gatsby and its vision

This post is the first of a four-part series in which we will explore Gatsby from a software architectural perspective. As the core team knows Gatsby best, let's start with their take on it:

"Gatsby is a free and open source framework based on React [1] *that helps developers build blazing fast* ***websites*** *and* ***apps****"*

Let's dive right in. The Gatsby developers value open source and hence strive to create the largest and most inclusive open-source community. In 2018, the team behind Gatsby defined its company core values. One of these values state: "You belong here". They support this claim by explaining how anyone can contribute [2] to Gatsby, hosting peer-programming sessions and rewarding contributions by sending free swag all across the world. Also, they think that the future of the web will be radically different than it currently is. Instead of websites and data sources being connected in a one-to-one fashion, they believe web applications will internally become a "content-mesh" [3], consisting of several data sources connecting to several user-facing platforms.

8.2.1 Gatsby's specialty

One of the great features is that websites compiled with Gatsby are fast by default. In web development terms, this means that it is actually possible to get a lighthouse [4] score of 100%! This performance is achieved in three steps:

1. **Data sourcing:** When compiling a Gatsby website, it pre-fetches all data necessary to run from various data sources. These include your file system, any REST API data and data from your favorite CMS. This data is then exposed over one single well-formatted GraphQL API. Want to add another CMS? Just add a plugin in `gatsby_config.js` and you are done!
2. **Page building:** Some pages in a Gatsby website use a backend call which is executed when viewing the page to retrieve required data. These page-queries use the GraphQL API to combine all data from different sources required to view a page correctly, such as: texts, images and translation strings. This step eliminates the need for an abundance of calls and combines these in one single call.
3. **Page loading:** To further enhance the page load time, Gatsby uses techniques such as HTTP/2 push, pre-caching and lazy-loading as part of the PRPL-pattern [5].

Using this pipeline for building, websites built with Gatsby require no communication with a data server to be browsed. This also adds a security benefit: If someone manages to get access to your code, your database is safe.

The second reason to choose for Gatsby, is that it focuses a lot on developer experience (DX). Gatsby has a great developer community, extensive documentation, simple tooling, lots of plugins and uses a widely used programming language. On top of that, Gatsby has a team of paid professionals that are there to help the project grow and flourish.

8.2.2 The world around Gatsby

While having only started in 2015, Gatsby has already grown into a huge open source community with more than 3000 people contributing to their main repository alone. Because of the community's size, contributing has a side effect of building up a referral network, which can be used to segue into paid work. Besides

[1] https://reactjs.org/
[2] https://gatsbyjs.org/contributing/where-to-participate
[3] https://www.gatsbyjs.org/blog/2018-10-04-journey-to-the-content-mesh/
[4] https://developers.google.com/web/tools/lighthouse/
[5] https://www.gatsbyjs.org/docs/prpl-pattern/

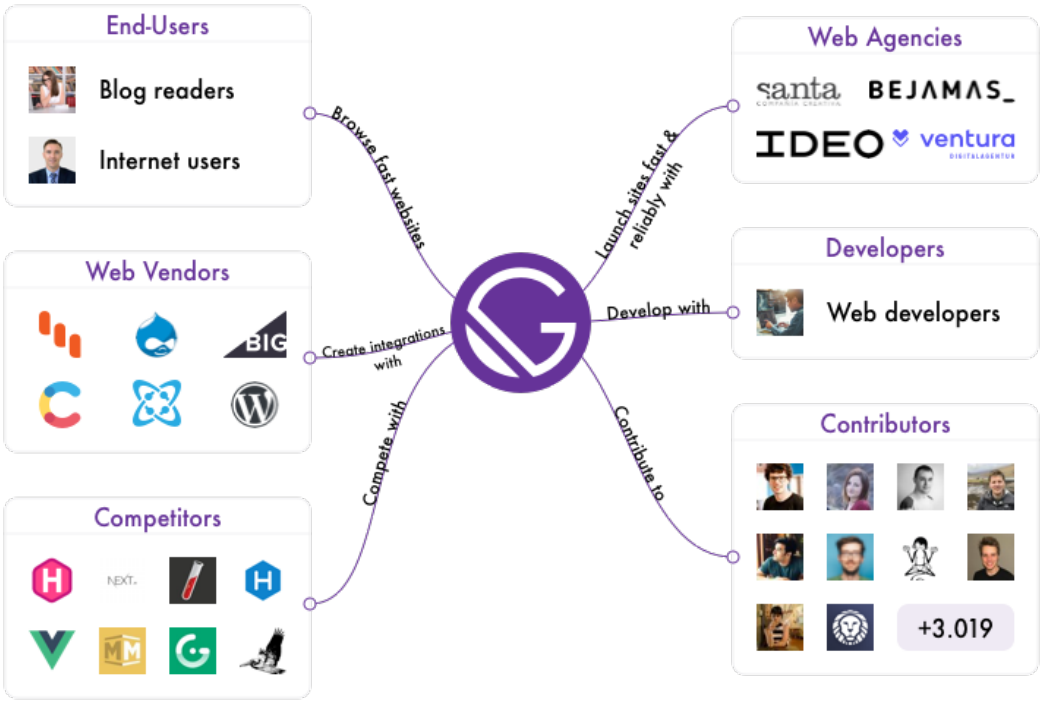


Figure 8.2: Visualization of the environment in which Gatsby fits

this natural benefit, the Gatsby team provides more, such as free swag and community events! Prominent contributors to the project include:

- @KyleAMathews [6]
- @m-allanson [7]
- @pieh [8]
- @sidharthachatterjee [9]
- @DSchau [10]
- @tesseralis [11]

The next group is of course, the web developers. Gatsby is designed with developer experience as a top priority. This shows in the advanced features that come straight out of the box, such as hot reloading and a command line interface with a lot of information. Everything in their APIs is considered and their extensive documentation is being translated in over 20 languages. Gatsby simplifies the whole process of writing, deploying and maintaining React apps. With their wide range of available plugins, Gatsby allows front-end developers to act like full stack developers. That Gatsby works for developers shows in their showcase filled with both personal blogs [12] and high-profile company websites such as Braun.com [13]. After all, who doesn't want a blazing fast website? This brings us to the next stakeholders: the end-users. People browsing the web nowadays expect increasingly high quality and high speed webpages, even on their phones. As stated before, Gatsby delivers on the high speed by default.

Some of the largest web vendors, like Wordpress and Contentful want to tap into the advantages Gatsby provides as well. They understand that data and website won't be one-to-one links anymore, but that it will become a so called "content mesh". Partnering with Gatsby shows developers that a vendor is modern and flexible, and it allows developers to keep using them.

Of course, Gatsby is not the only website generator. Big players in this space are Next.js and Hugo. One might choose to use Next.js when requiring dynamic routes with server side rendering. Hugo, on the other hand, might be your choice if you're looking for a large theme library.

8.2.3 Where is Gatsby going?

After you've created your Gatsby website, you'll want to publish it as quickly as possible. Since a few months, this has become easier with the help of Gatsby Cloud. With Gatsby Cloud, Gatsby aims to create a more complete developer ecosystem. This includes easy collaboration with shareable preview URLs and support for many headless CMSs out-of-the-box. This also helps non-technical users to create new projects in minutes. While Gatsby Cloud is free for personal use, professional use is charged. A few weeks ago Gatsby Builds has been added as an expansion to Gatsby Cloud. Builds is designed to be "the fastest Continuous Deployment solution for Gatsby applications". Builds give feedback immediately during development, allowing developers with less initial experience with hosting and continuous integration to start using Gatsby.

[6] https://github.com/KyleAMathews
[7] https://github.com/m-allanson
[8] https://github.com/pieh
[9] https://github.com/sidharthachatterjee
[10] https://github.com/DSchau
[11] https://github.com/tesseralis
[12] https://lowmess.com/blog
[13] https://ca.braun.com

Gatsby is currently on V2.19. They are working towards a third major version. There is no release date, nor are there definite plans on what to include in this release. However, the development has been kicked off. The initial roadmap was published last July and Gatsby inquired their community what major changes they think would be beneficial. For now this has resulted in the following focus areas:

- Normalizing path handling across the board
- Major updates to dependencies and changes of critical dependencies
- Exposing more low level APIs and APIs that will better support i18n

Presumably this development will still take over a year.

On the shorter term, Gatsby is working hard on improving and localizing its documentation. Translation is a community effort and allows Gatsby to cater to even more users' needs. Also some of their plugins have a clear short-term roadmap.

One of the core contributors is for example working full-time on the WordPress plugin [14]. This will be a rewrite using the WPGraphQL API instead of their legacy REST API.

8.2.4 Conclusion

Gatsby is a community effort in which anyone can contribute. And with this community, they will change the way websites are built. This post covered Gatsby's vision, capabilities, roadmap, product context, and stakeholder analysis. In the next post we will discuss some of the architectural decisions made, a system decomposition, tradeoffs, as well as architectural styles and patterns. Stay tuned!

8.3 Gatsby Through the Eyes of...

Software can be viewed from many angles. Ask a developer and a user what they think about some software product and you'll probably get not only contradictory answers, but even uncomparable answers. This is because they look at the software from such a different point of view.

This is the second part in our four part series on Gatsby. In the first post[15], we wrote about what Gatsby is and where it is going. In this post, we will take a moment to explore the multiple architectural views of Gatsby. If you haven't read the first essay yet and you're trying to get a grasp on the system, this would be a good time to go and read it.

Before we move into the specifics, it's a good idea to get a basic understanding of the architectural style of Gatsby.

The Gatsby project actually is a combination of many small packages bundled in one monorepo using Lerna[16]. These packages can be classified as follows: first there are a couple of core packages, including `gatsby` and `gatsby-link`. Then, there is the command line tool: `gatsby-cli`. Finally, most of the packages are plugins like `gatsby-source-wordpress` and `gatsby-plugin-feed`.

The `gatsby` package itself uses the flux-pattern[17] with redux[18]. A widely used pattern for unidirectional

[14] https://github.com/gatsbyjs/gatsby/tree/master/packages/gatsby-source-wordpress
[15] https://desosa2020.netlify.com/projects/gatsby/2020/03/09/the-great-gatsby.html
[16] https://lerna.js.org/
[17] https://facebook.github.io/flux/
[18] https://redux.js.org/

data flow in applications. One cool side note, it also uses a formal turing machine[19]!

Now that we have the basics in place, let's find some interesting viewpoints. For our viewpoints we are inspired by Rozanski and Woods[20], offering valuable insights into the system. Since Gatsby focusses a lot on developer experience (DX), the development view is a good place to start. Followed by the reason that Gatsby sites are fast, is that it does most of the heavy lifting in its build process, before a user ever gets to see the site, we describe this process from a runtime perspective. Next, as Gatsby wants their sites to be deployable anywhere (and fast), we will look at it from a deployment view. Finally, we will zoom out a bit and look at non-functional properties and the tradeoffs that Gatsby has made.

8.3.1 Development view

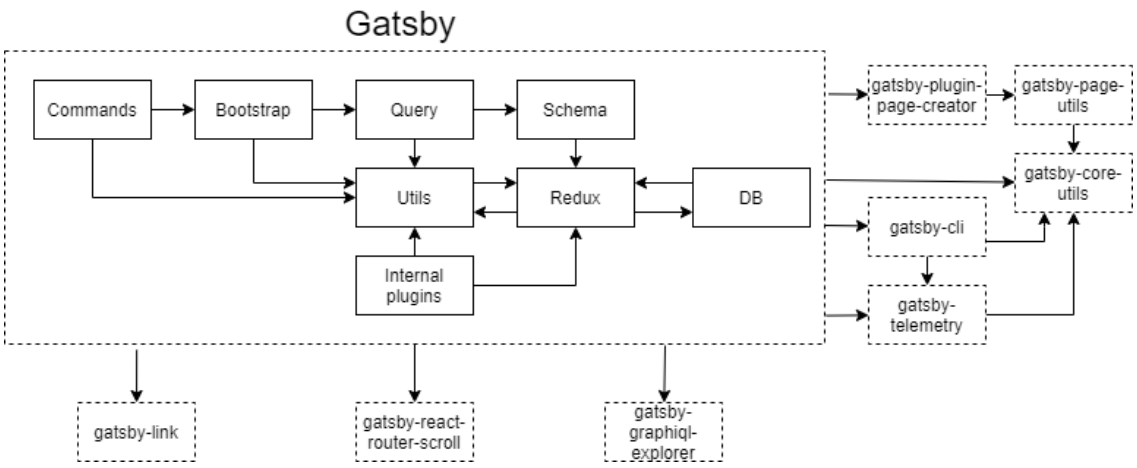


Figure 8.3: Development view

When exploring the Gatsby repository, you will quickly discover that all the code is contained in a directory named packages. This folder currently contains 109 packages. A lot of these are packages are plugins. They will be discussed further in the runtime view section . The most important package, simply called `gatsby`, contains an architecture of its own. We explored this architecture using a tool from Software Improvement Group[21] and found the structure as seen in the diagram above. The `gatsby` package contains core code for the command line interface and uses Redux[22] for state management. The packages outside the box were found by manually inspecting each package `package.json` file.

The `gatsby` package also makes use of other packages within the Gatsby repository, such as `gatsby-link`, which is used by Gatsby sites for internal communication. Some other packages are also worth exploring. Take for example the `gatsby-cli` package which controls the command-line interface. Or `gatsby-telemetry`, collecting analytical data regarding user interactions.

[19] https://github.com/gatsbyjs/gatsby/blob/master/packages/gatsby/src/redux/machines/page-component.js
[20] https://books.google.nl/books?id=ka4QO9kXQFUC&printsec=frontcover&hl=nl&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
[21] https://www.softwareimprovementgroup.com/
[22] https://redux.js.org/

8.3.2 Runtime view

In running Gatsby, there are two key scenarios: `gatsby develop` and `gatsby build`. These command-line commands share a lot of their functionality. Both processes compile your project into static files. A detailed schematic for the build process can be found here[23], but the basic process is as follows:

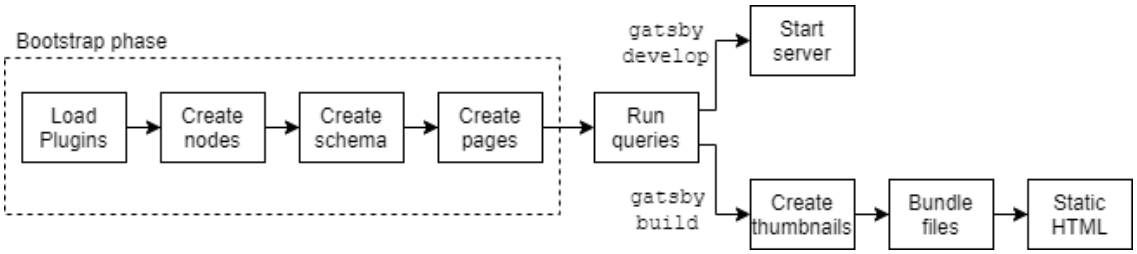


Figure 8.4: Runtime view

First, the gatsby plugins are loaded. Gatsby plugins can be divided into two types: **source plugins** and **transformer plugins**. Source plugins fetch data from some data source, such as the file system, or some REST API. Next transformer plugins are used to transform the raw data into something more useful. In the following step, every plugin creates one or more Nodes, forming the center of Gatsby's data system. The nodes are sewn into a GraphQL[24] schema, and plugins can control which nodes are turned into pages.

Next, all the queries (data requests to the GraphQL schema) defined by all the pages are run. At this point, all the data is at the place it needs to be. With `gatsby develop`, a local web server is started that allows hot reloads whenever the source files change. With `gatsby build`, a production build is created. This includes creating image assets of different sizes, bundling Javascript and CSS files using webpack, and finally producing the static HTML files to be deployed to a static hosting service.

8.3.3 Deployment view

As described in the first essay[25], Gatsby does not host websites made with Gatsby. To deploy Gatsby an external web server is needed. With the `gatsby build` command, as discussed in the runtime view, output is created. The output contains HTML, CSS and JavaScript that is directly usable by the browser. To get these files to the browser, an server system is required. You could get away with an ordinary http server. However there are some more modern ways that are recommended by Gatsby. Gatsby promotes the use of static web servers like Netlify[26], Now[27] and others[28]. These platforms automatically perform builds and distribute the static content over CDNs like Cloudflare[29]]. Via the CDN of choice, the Gatsby project reaches its final destination, being a blazing fast website on a client's web browser.

[23] https://www.gatsbyjs.org/docs/gatsby-internals/
[24] https://graphql.org/
[25] https://desosa2020.netlify.com/projects/gatsby/2020/03/09/the-great-gatsby.html
[26] https://www.gatsbyjs.org/docs/deploying-to-netlify/
[27] https://www.gatsbyjs.org/docs/deploying-to-zeit-now/
[28] https://www.gatsbyjs.org/docs/deploying-and-hosting/
[29] https://www.cloudflare.com

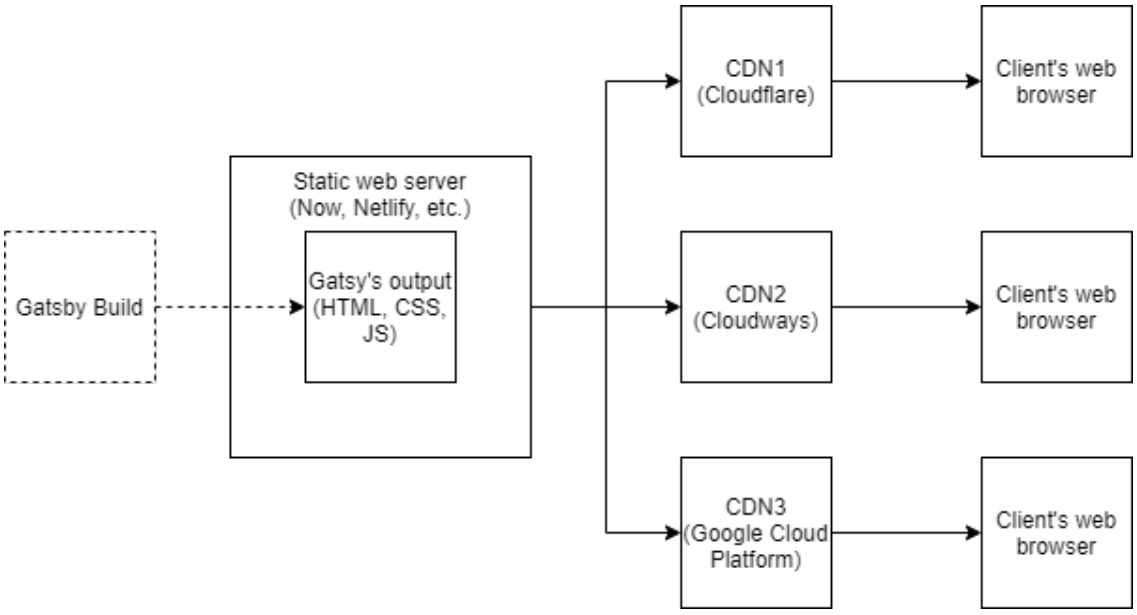


Figure 8.5: Deployment view

8.3.4 Non-functional properties

In the previous sections we have seen how Gatsby compiles dynamic React apps into static websites, with all external data baked into them.

This approach has some large positive consequences: For one, to access a Gatsby website, no database is contacted. This makes Gatsby websites secure by default. Also, and this is a core benefit of any website built using the JAMStack, Gatsby websites can be served over any CDN. This allows for near unlimited scalability in the amount of users.

The approach Gatsby took also has one big disadvantage. The Gatsby compiler currently compiles complete websites. Even when an editor made a single change in a CMS that should only impact one single page. This becomes a problem for large sites, with frequently changing content. Gatsby is however working towards a build time, even for huge websites, of less than 10 seconds[30]. Their largest current effort in this is incremental builds[31], which is currently in private beta.

8.3.5 Conclusion

This exploration of the Gatsby project from different angles gave us a huge insight into the project, and we hope it did the same for you. In the next post, we will dig deep into the system decomposition of Gatsby to assess the project's quality and to uncover the technical debt there… Stay tuned!

[30] https://www.gatsbyjs.org/blog/2020-01-27-announcing-gatsby-builds-and-reports/
[31] https://www.gatsbyjs.com/incremental-builds-beta/

8.4 Gatsby in Debt

Welcome to the third part of our four-part series on Gatsby. The previous essays can be found here: first, second. For a complete understanding we highly recommend reading those first. In this part we will evaluate technical debt and architectural decisions made. To be able to cover this all we will first take a look at the quality assurance systems Gatsby has in place. We'll explain what a contributor needs to do to have their pull request successfully merged and what Gatsby does to protect their code quality. When we have in place how pull requests are merged, we will analyze what pull requests have been merged recently, what changes are coming up, and how these changes impact the technical debt and maintainability of Gatsby. This should give a clear understanding of what is going on inside Gatsby. After getting this understanding, we get a bit more formal and dive into absolute quality assessment, analysing results from SIG and BetterCodeHub. In conclusion we will take all evaluated aspects together. If you want to learn more about these topics, this essay is just for you!

8.4.1 Quality Assessment

Let's start with one of the most important parts of software quality: bugs per square meter... No, testing, of course. To assess the test quality of Gatsby, we first looked at the analysis by SIG. This didn't yield accurate results. Luckily Gatsby uses Jest[32] for testing. Allowing us to generate coverage reports ourselves.

With Jest, you can use a function to specify which directories should be included in the coverage report[33]. This function can easily adhere to the project structure and automatically pick up newly added plugins or packages. A nice configuration aspect that makes assessing the correctness of the coverage report a breeze. The function hasn't changed for over a year, which is remarkable. This might be a good sign, but there is currently no way to validate its correctness.

Diving into the tests themselves, it appears that Gatsby has a quite structured and overall high quality testing policy. With simple unit-tests verifying behavior of single functions, integration-tests verifying component interaction and fully fledged end-to-end-tests. Test structure is very clear, due to elaborate documentation, allowing new developers to maintain the same standard in new code. The testing policy for newly added or changed code is also clearly communicated through the docs[34].

Overall, the codebase has **60%** statement, **55%** branch, **57%** function and **61%** line coverage. While these numbers are low compared to the commonly used 80%, we have to take into account the low coverage of some plugins. On the other hand, it also seems that some core components have low coverage, for instance `gatsby-cli` and some parts of the `gatsby` package. From a coverage point of view, it would be nice to see all core packages to have a coverage greater than 80%. Currently, coverage is already quite satisfactory in our opinion.

As part of sharing the entire codebase with the community, anyone is free to contribute and review new changes to the code. Some sidenotes: Firstly, changes regarding the internal organization structure are reserved for Gatsby employees. Secondly, every PR needs to be approved by the respective code owner in the `CODEOWNERS` file[35], approval is required by someone relevant to that part of the code. Finally, merging a PR is only reserved for the core team and Gatsbot. Gatsbot is an automated bot that merges PRs with the label `bot: merge on green` for which all pipeline checks succeeded[36].

[32] https://jestjs.io/
[33] https://github.com/gatsbyjs/gatsby/blob/master/jest.config.js
[34] https://www.gatsbyjs.org/contributing/managing-pull-requests/#code
[35] https://www.gatsbyjs.org/contributing/managing-pull-requests/#who-can-approve-a-pr
[36] https://www.gatsbyjs.org/contributing/managing-pull-requests/#gatsbot

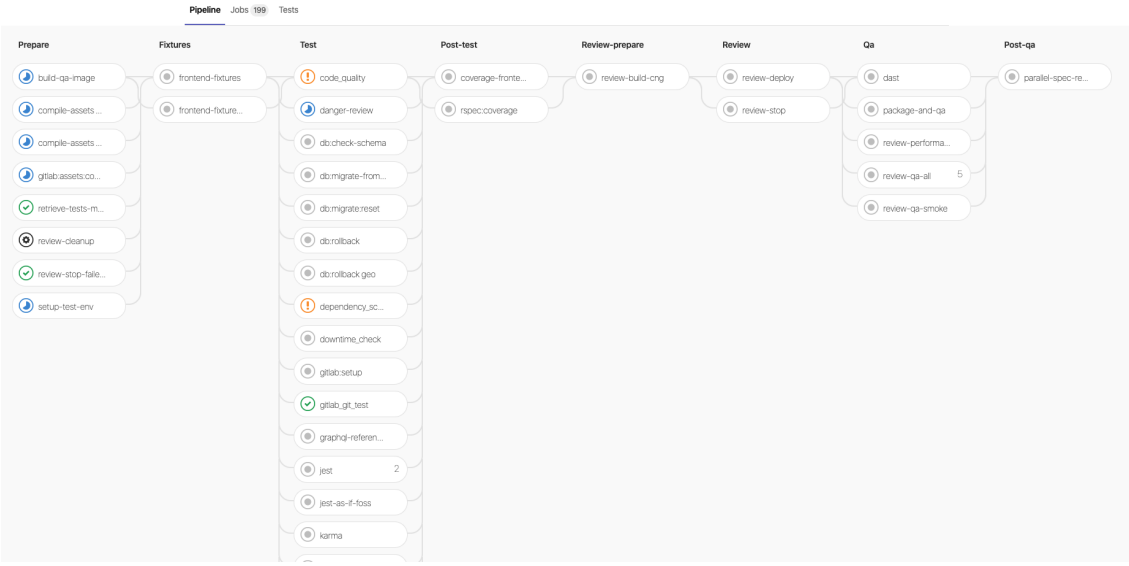


Figure 1: *Pipeline*

Gatsby merges over 100 PRs every week. For documentation specifically it is important to follow their style and formatting guidelines. To code changes some other rules apply: they should include tests asserting implemented behavior and ensuring that fixed bugs can't re-occur. The pipeline (figure 1) runs checks dependent on each other, some checks require others to have passed. Type checks are being done in the pipeline. The code is automatically reviewed by danger.js for simple mistakes.

One important aspect for Gatsby are the guidelines for reviewing a PR in the docs[37]. The main points of these guidelines are:

- Be kind
- Use GitHub suggestions
- Link examples
- Try to avoid bikeshedding

Especially the point on bikeshedding[38] stands out. Bikeshedding is defined as follows: "Futile investment of time and energy in discussion of marginal technical issues". This has led to a lot of lengthy discussions as Go error handling[39][40][41][42]. In python it even led to Guido's (Python's creator) resignation[43]. Since so many people are contributing to Gatsby, this rule should really be taken to heart. If you have the time, read the link for bikeshedding, it provides some interesting insights.

[37] https://www.gatsbyjs.org/contributing/managing-pull-requests/#giving-feedback
[38] http://bikeshed.com/
[39] https://github.com/golang/go/issues/21155
[40] https://github.com/golang/go/issues/21146
[41] https://github.com/golang/go/issues/18721
[42] https://github.com/golang/go/issues/16225
[43] https://mail.python.org/pipermail/python-committers/2018-July/005664.html

8.4.2 System evolution

To see how Gatsby is evolving, we'll first look at its recent history. For this matter we analyzed all pull requests merged in the last month, i.e. from 19 February to 19 March. This yielded **430** pull requests divided over many areas of code. Three hotspots stood out. As expected from the Gatsby community, documentation updates top the list, with 75 PRs. Second place, with 65 pull requests was the TypeScript migration[44] of the core codebase, which only started on March 5th! The community was very active as well with 45 websites added to the showcase. Besides these, support for MDX[45] had significant updates with 21 related pull requests and there were 17 dependency updates by `renovate [bot]`. This covers roughly half of the pull requests. The other half of the pull requests were either focused on one of the many plugins, or sometimes on core features such as GraphQL, yarn 2[46] compatibility and moving from hot-reload to FastRefresh.

In the upcoming time, Gatsby has exciting features ahead. Some of these features will have a positive impact on the project's technical debt: Most notably, the support for yarn 2[47], and the migration of the core package from JavaScript to TypeScript[48]. Some changes, like allowing variables in the StaticQuery component, the implementation of the schema customization API[49] and updating gatsby-plugin-sharp to allow image processing on demand in dev will greatly improve the developer experience. The Gatsby team is also planning to create a Desktop app: a GUI on top of the CLI. This will not impact the current codebase, but as it is an additional product, they have to watch out for introducing a lot of technical debt in the system.

Of the changes mentioned above, we expect two to have the largest impact on Gatsby's architecture and technical debt: the migration from TypeScript to JavaScript and the introduction of variables to the StaticQuery component.

8.4.2.1 JavaScript -> TypeScript

For the last four and a half years, the language of choice for Gatsby has been JavaScript, after having been written in CoffeeScript[50] initially [51]. This has been a solid choice for now. However with the increased size of the project, having untyped code in the core becomes a burden and a source of bugs rather than an opportunity. TypeScript is more commonly used in webdevelopment projects nowadays. TypeScript is a drop in replacement for JavaScript, allows gradual migration of the codebase[52]. Typescript also provides static type checking, allowing developers' development environments to better understand the code and offering auto-completion suggestions. This makes it easier for any of the 3000 contributors to understand and thus to contribute to the code. Fundamentally, this should result in less unexpected testing failures, easier refactorings and less bugs!

8.4.2.2 Variables in StaticQuery

In Gatsby, you can get the correct sized image on the page by querying it from the GraphQL api, with a page query, or a StaticQuery. The StaticQuery component is a React component that allows a developer to specify a GraphQL query and to use the result of this query in its child components. This GraphQL query

[44] https://github.com/gatsbyjs/gatsby/issues/21995
[45] https://mdxjs.com/
[46] https://yarnpkg.com/
[47] https://github.com/gatsbyjs/gatsby/issues/20949
[48] https://github.com/gatsbyjs/gatsby/issues/21995
[49] https://github.com/gatsbyjs/gatsby/issues/20069
[50] https://github.com/gatsbyjs/gatsby/commit/24606f5a2d5c85d7b6661403333f34823409bdf3
[51] https://github.com/gatsbyjs/gatsby/pull/37/files
[52] https://github.com/gatsbyjs/gatsby/issues/21995

gets executed at compile time, this now no longer has to be done when loading the page. Currently, it is not possible to add variables to these queries. This is a problem when multiple components rely on really similar data. In this case, the components either need to be duplicated with slight changes, or the data flow needs to be changed. This leads to a suboptimal development experience with Gatsby, which would be solved by allowing variables in a StaticQuery.

Since Gatsby runs the StaticQuery at compile time, it is hard to see which value the variables in the StaticQuery instances will have upfront. While struggling with the issue, Wes Bos (a well known web developer) made a tweet resulting in a solution[53]. The proposed solution would insert an additional step in Gatsby's build pipeline, compiling files using StaticQuery components into multiple specialized instances without those variables. This additional step in the build process might add some technical debt to the Gatsby system itself, but it will reduce the technical debt in projects created with Gatsby by a lot for sure!

8.4.3 Maintainability

To assess the maintainability of Gatsby, we analyzed the project using tools from SIG. SIG rates code on many different aspects, including code duplication and module coupling, on a scale from 0.5 to 5.5 stars. These scores indicate the project's code quality compared to other codebases: to get 5 stars on a metric you need to be in the top 5%, for 4 stars in the top 5-35%, for 3 stars in the top 35-65%, for 2 stars in the top 65-95% and if you score 1 star you are in the bottom 5%.

These are the results for the full Gatsby repository:



Figure 2: *Scoring by SIG*

Since a big part of the codebase consists of plugins, these numbers don't say too much and we performed an additional analysis of the `gatsby` package using Better Code Hub[54] (also created by SIG). Better Code Hub rates projects using ten simple pass/fail guidelines. The great thing about their method is that they allow for some violations in a small percentage of the code. For example, for "Write Short Units of Code", at most 6.9% of units may contain more than 60 lines, at most 22.3% may contain more than 30 lines of code, etc. [55]. A pass on each guideline corresponds to receiving at least 4 stars for the SIG/TÜViT Evaluation

[53] https://github.com/gatsbyjs/gatsby/issues/10482
[54] https://bettercodehub.com/
[55] http://shop.oreilly.com/product/0636920049159.do

Criteria[56]. Below, some of the results of the Gatsby core package are shown. The vertical bars represent the minimal quality that the codebase should deliver.

8.4.3.1 Components most likely affected by future change?

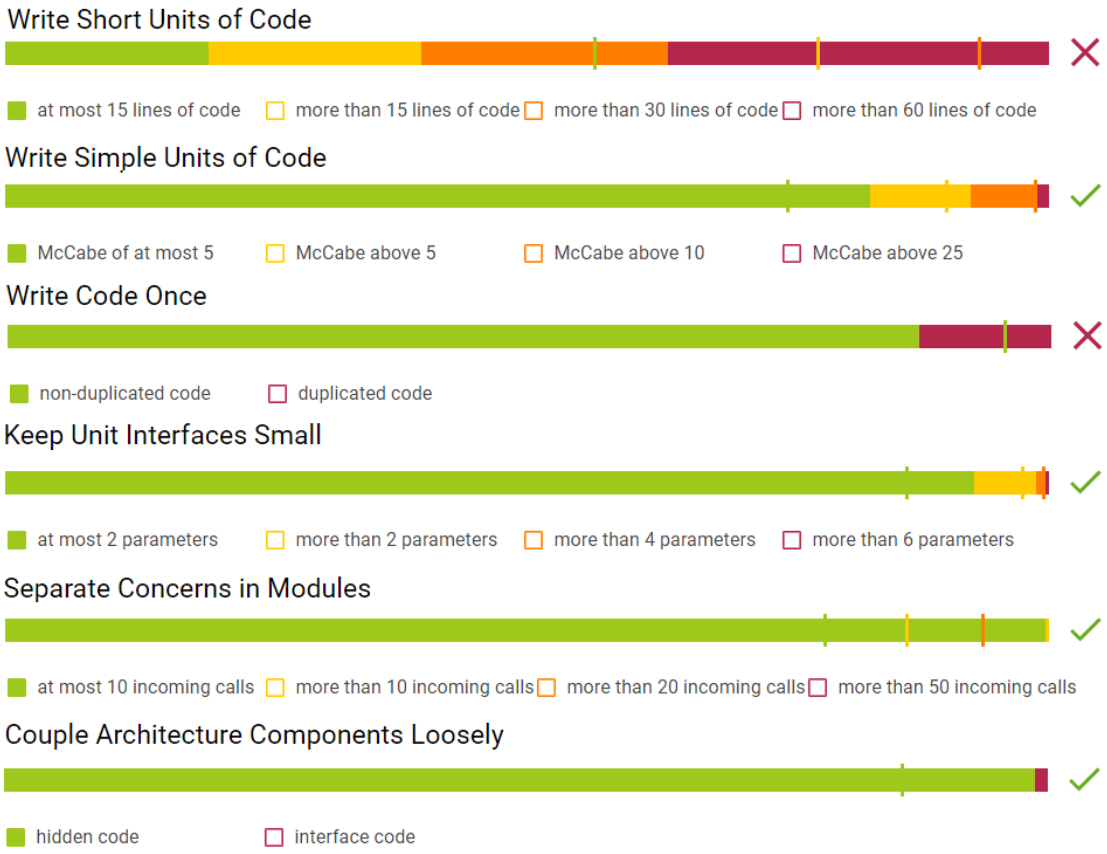


Figure 3: *Sig guidelines evaluation*

As can be seen above, two guidelines are not met: writing short units of code and writing code once (no duplication). In fact, Gatsby includes a function consisting of 867 lines[57]! As for code duplication, one of the refactoring candidates is the comparators file for the loki database[58], which contains a duplicate block of 56 lines of code. Even though it concerns test code, duplication should be avoided.

8.4.4 Conclusion

We analyzed the software quality of Gatsby from various angles. Gatsby tries to be a very open and inclusive community, allowing many people to add their plugins to the system. This does lead to a lowered test

[56] https://www.softwareimprovementgroup.com/wp-content/uploads/2019/11/20190919-SIG-TUViT-Evaluation-Criteria-Trusted-Product-Maintainability-Guidance-for-producers.pdf

[57] https://github.com/gatsbyjs/gatsby/blob/master/packages/gatsby/src/schema/infer/**tests**/inference-metadata.js#L43

[58] https://github.com/gatsbyjs/gatsby/blob/master/packages/gatsby/src/db/loki/custom-comparators.js

coverage. However, the tests that exist, are actually good. With this huge community, comes a lot of communication. Gatsby handles the communication well by applying many guidelines and policies.

Mainly in their plugins, Gatsby contains too much duplicate code, which makes it prone to unnecessary errors. This could be partly solved by rewriting some plugins to minimize duplicate code. However, due to the amount of packages which are not connected with one-another, this will be quite a challenge. Gatsby acknowledges some of its technical debt and is working hard on solving problems. This can for example be seen in their effort transitioning from JavaScript to TypeScript.

Altogether, Gatsby, like all large projects, has some technical debt and has made some tradeoffs. They are working hard on improving their technical debt and have some truly exciting changes ahead. We hope to have guided you well through the technical depths of Gatsby and we hope to see you in our next and final essay!

8.5 To Gatsby and Beyond!

This is our fourth and final post on Gatsby. The first post[59] introduced Gatsby by explaining its product vision. In the second one[60], we discussed several architectural views and the third one[61] explored technical debt. This essay will cover another architectural concept influencing Gatsby's flexibility and maintainability: variability management. Due to Gatsby's plugin architecture with over 1800[62] plugins, it has an "infinite" number of different possible configurations. Still, everything works together surprisingly well!

First, let's get an idea of the main variable features. That is, features that are not hardwired in the product, but that you as a user, can enable, disable and play around with. When you start making a website with Gatsby, you can get a head start using a theme[63] or sub-theme. If that is not to your liking, you can also have a look at the hundreds of starters[64] that are available. Filling your site with data is done with (again hundreds of) data sourcing[65] and transformer[66] plugins, which allow you to fetch data from different sources, to transform that data to different formats and to customize[67] the exposed GraphQL schema. But Gatsby's variability doesn't stop there. Developers can adjust the workflow to their needs with custom linting rules[68], and custom Webpack[69] and Babel[70] configurations . Furthermore, Gatsby supports all long term supported Node.js versions[71] and allows developers to specify on which browsers the website should run using browserlist[72]. Finally, when you are done building your website and its time to ship it, you can choose any static web hosting service[73]. All in all, we would say that there are enough options for most people.

[59] https://desosa2020.netlify.com/projects/gatsby/2020/03/09/the-great-gatsby.html
[60] https://desosa2020.netlify.com/projects/gatsby/2020/03/16/gatsby-through-the-eyes-of.html
[61] https://desosa2020.netlify.com/projects/gatsby/2020/03/25/gatsby-in-debt.html
[62] https://www.gatsbyjs.org/plugins/
[63] https://www.gatsbyjs.org/blog/2018-11-11-introducing-gatsby-themes/
[64] https://www.gatsbyjs.org/starters?v=2
[65] https://www.gatsbyjs.org/plugins/?=gatsby-source
[66] https://www.gatsbyjs.org/plugins/?=gatsby-transformer
[67] https://www.gatsbyjs.org/docs/schema-customization/
[68] https://www.gatsbyjs.org/docs/eslint/
[69] https://www.gatsbyjs.org/docs/add-custom-webpack-config/
[70] https://www.gatsbyjs.org/docs/babel/
[71] https://www.gatsbyjs.org/docs/upgrading-node-js/
[72] https://www.gatsbyjs.org/docs/browser-support/
[73] https://www.gatsbyjs.org/docs/deploying-and-hosting/

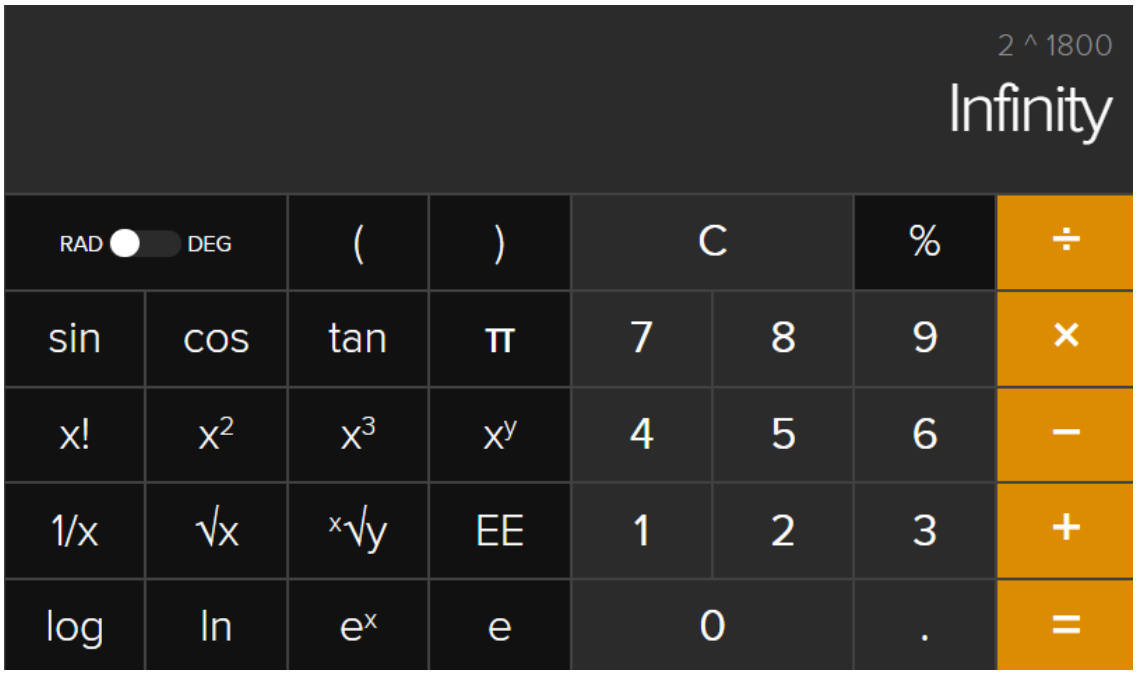


Figure 8.6: 1. Possibilities

8.5.1 Feature model

Since Gatsby has so many variable features, putting all of them together into a feature model wouldn't show you much. Therefore this section looks at the plugins feature only. Each plugin you write or use has access to some specific apis. We'll look into how the features exposed to the plugins are composed. The Gatsby team wants you to be able to write as much custom functionality as you desire, without bothering you about stuff you don't want to use[74]. This makes creating a new plugin the least cumbersome. Gatsby does this by checking whether a plugin implements some pre-specified functions. If such a function is exposed, it will be used in the process. Figure 2 shows that plugins can implement three API's for Gatsby to call when running.

First, there is the Browser API[75], which allows the plugin to interact with the browser. This enables plugins for instance to register a service worker, which handles page caching and lazy-loading, or to perform actions on route updates, such as tracking analytics. Secondly, there is the Node API [76], which is responsible for actions performed in or around the build stage. Cleanup of intermediate plugin outputs or confirming the build has correctly outputted required files can be done with post-processing in the `onPostBuild`. Finally, there is the Server Side Rendering API[77], which allows plugins to change content served to the user. These three APIs together allow the user to implement any functionality in Gatsby.All the APIs are inherently optional, so users don't have to write boilerplate.

An example of a commonly used configuration would use Node.js V12 (LTS) with Yarn for managing npm

[74] https://www.gatsbyjs.org/docs/files-gatsby-looks-for-in-a-plugin
[75] https://www.gatsbyjs.org/docs/browser-apis/
[76] https://www.gatsbyjs.org/docs/node-apis/
[77] https://www.gatsbyjs.org/docs/ssr-apis/

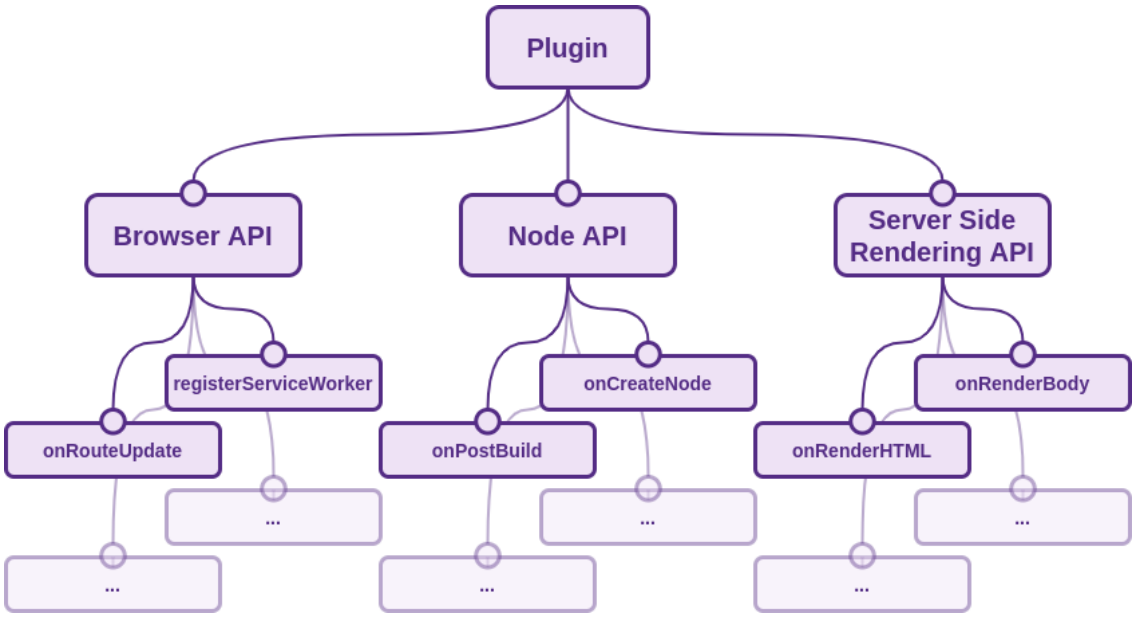


Figure 8.7: 2. Feature model

packages. It would support not dead[https://github.com/browserslist/browserslist] browsers, deploy directly to Netlify and use basic plugins such as:

- `gatsby-source-filesystem` for reading files from local filesystem
- `gatsby-transformer-remark` for transforming Markdown files into GraphQL format
- `gatsby-plugin-sharp` for transforming and optimizing images
- `gatsby-transformer-sharp` for exposing transformed images

A more exotic configuration could for example extend the Babel configuration with `@babel/plugin-proposal-optional-chaining` which allows the new JavaScript `maybeObject?.field` syntax. Additionally it could use the source plugins `gatsby-source-contentful` and `gatsby-source-airtable` and stitch their data together with GraphQL schema customization using the `createResolvers` API in `gatsby-node.js`. Deployment could for example be done via Gatsby Cloud on Zeit Now.

That said, it is near impossible to guarantee that all possible combinations of configurations are bug-free. We analyzed all issues and pull requests labeled `type: bug` and `status: confirmed` in the Gatsby main repo. Surprisingly, this yielded only 5 issues concerning incompatible configurations. This issue[78] describes a failure in data fetching in case of a naming conflict between a theme name and a query name. An example of browser incompatibility is described here[79], showing that images are not properly fading in Safari. The two plugins for KaTeX and MDX are not working together[80] yet. It has also happened that a specific plugin configuration does not work in a specific environment[81] (CSS modules in development using Less.js), or

[78] https://github.com/gatsbyjs/gatsby/issues/19980
[79] https://github.com/gatsbyjs/gatsby/issues/20126
[80] https://github.com/gatsbyjs/gatsby/issues/21866
[81] https://github.com/gatsbyjs/gatsby/issues/17851

that the combination[82] of two plugins and environment gave rise to problems (offline and style components plugins in development). It can be expected that the more variables Gatsby introduces, the more of these issues will arise. How does Gatsby plan on dealing with so many factors?

8.5.2 Variability management

How variability is managed depends on whom you ask. Different stakeholders might give different answers. For example, the end-users (web visitors) will not even know if a page was made with Gatsby or not, let alone in which configuration. To find information about different configurations, web vendors can consult the partnering walkthrough[83] or reach out by mail[84]. Developers and web agencies can make use of Gatsby's extensive documentation, or the documentation of configuration tools like Babel and Webpack. Additionally, each plugin has its own documentation (some even have their own webpage[85]). Gatsby maintainers have their own team on GitHub. Finally, everyone can raise issues on GitHub[86] or ask questions on Discord[87].

However, the different ways to get information or report issues can hardly be called management. How does Gatsby actually prevent issues caused by variability? Right now, for every change, tests are executed using the pipeline discussed in our previous[88] post. This includes tests run on a Windows machine and in the cloud. Although this will certainly catch some exceptions, it can't possibly check all different configurations, due to the enormous amount of them.

However, Gatsby has some tricks up its sleeve! Currently over 179,000[89] open source websites are built with Gatsby, and each has its own configuration. In this issue[90], Gatsby is working on an ambitious project to exploit this fact. The idea is that on each update, one thousand of those websites will be selected at random. Then, they will be built using a new Gatsby version. Since the websites are chosen at random, they will have widely varying configurations. If no error is spotted in building these sites, there is a high chance that the update is free of errors. If some fail, the Gatsby team immediately has a reproducible example that can guide them while bug hunting. Gatsby is working on another[91], more active approach of handling variability. In this approach, they aim to validate options provided to plug-ins, so that if they fail, they fail early with a useful error message.

8.5.3 Variability implementation mechanism

We have seen that there are many ways to configure a Gatsby project. But when are the configurations resolved? And how can you actually use them? The answer to the first question is simple: since Gatsby is essentially a compiler, the binding time of all the variable features is at compile-time. To answer the second question, we need to have a look at the variability implementation mechanism that Gatsby uses. We'll explore the implementation and scalability of Gatby's plugins and of Gatsby Themes.

Plugins are at the core of Gatsby. Each Gatsby application defines the plugins that it uses in a configuration file called `gatsby-config.js`. Internally, Gatsby reads this list of plugins and initializes each one in

[82] https://github.com/gatsbyjs/gatsby/issues/12145
[83] https://www.gatsbyjs.org/docs/gatsby-vendor-partnership/
[84] https://www.gatsbyjs.com/contact-us/
[85] https://using-gatsby-image.gatsbyjs.org
[86] https://github.com/gatsbyjs/gatsby/issues
[87] https://discordapp.com/invite/gatsby
[88] https://desosa2020.netlify.com/projects/gatsby/2020/03/25/gatsby-in-debt.html
[89] https://github.com/gatsbyjs/gatsby/network/dependents
[90] https://github.com/gatsbyjs/gatsby/issues/12300
[91] https://github.com/gatsbyjs/gatsby/pull/16027

order during build time. Adding a plugin to an application is as simple as adding a new line to the `gatsby-config.js` file. Updating a plugin is as simple as installing the new version of the npm package using `yarn add <plugin>@latest`. Removing a plugin from a project takes a bit more time. If there are any queries depending on this plugin. In this case, the developer needs to edit all queries that depend on the plugin.

We haven't found any reason why the overall concept of plugins would not scale. One design choice that is currently in place, however will not scale infinitely. The order in which the plugins are specified, determines if the dependencies on each other are resolved correctly. The Gatsby team is working on a way to specify the dependencies plugins have on other plugins. This change would make it easier to add new plugins to a project already containing a lot of plugins. In major semantic versions of Gatsby, they might drop support for some of their exposed API's. Since plugins make use of the API's Gatsby exposes this means that every plugin that depended on the dropped API, needs to be adapted. Given the amount of plugins in existence, this can become bothersome.

Next up: Themes. Gatsby Themes are just Gatsby websites, but without the content. They contain a predefined configuration, which can be extended by users of the theme. Extending the configuration can be done in a straightforward way using Gatsby's configuration files `gatsby-(config|node|browser|ssr).js`. Themes also provide layouts and page elements as React components. These React components can be 'shadowed'[92] by creating a file with the same name in the project directory. For example, if the component is located at `some-theme/src/components/FooComponent.jsx`, it can be shadowed by creating a file `my-gatsby-website/src/some-theme/components/FooComponent.jsx`. This way, anything in a theme can be overridden. Since Gatsby Themes are just self-contained Gatsby configurations and anything in there can be overridden, makes them as scalable as Gatsby itself.

With that, we've come to the end of our fourth and final post of our series on Gatsby. We really hope you enjoyed reading them, and perhaps even more importantly, that they were useful to you. If you are interested in building websites with Gatsby, or in contributing to the Gatsby project, you should now know where to go. We'll see you there.

[92] https://www.gatsbyjs.org/docs/themes/shadowing/

Chapter 9

GitLab

GitLab is a web-based DevOps lifecycle tool. It is used by developers to store software code, maintain versions, share with other developers, and provides useful features such as CI/CD which help developers focus on their main task: developing software. GitLab consists of two versions: a community edition and an enterprise edition. The former is opensource and free to use, while the latter is paid and holds extra features and customer support. Both versions offer the ability to host projects online or on your own servers. GitLab itself is written in Ruby, using the Ruby on Rails framework, and maintained on its own platform.

In 2016 another team, following the same course, has also analysed Gitlab[1]. The aim is to built upon their work, highlight the differences and progress made since then.

9.1 Team Introduction

We are four Computer Science master students at the Delft University of Technology. We all recently concluded our bachelor Computer Science and Engineering at said university. Most of our team have experience with GitLab and found it interesting that a project hosting service is itself open-sourced. Furthermore, it seemed like a very large project with lots of contributors which makes it interesting to analyse. It also interested us because GitLab is a remote-only company, which we hope means they are used to - and perhaps more open to - contributions and discussions with strangers.

- **Rolf de Vries**
- **Bram van Walraven**
- **Rico Hageman**
- **Hilco van der Wilk**

9.2 GitLab: A single application for the entire DevOps lifecycle

This article is the first in a series of four, where we, four Computer Science master students from Delft University of Technology, will be analysing the open-source project GitLab. In this series we will be

[1] Martijn Pronk, Han Lie, Jasper Denkers, Christian Veenman, Delft Students on Software Architecture DESOSA 2016, GitLab: Code, Test & Deploy Together

conducting an analysis for several aspects about managing an open-source project, both technical and non-technical. Before we dive into these aspects in later articles, we will first provide some context on what Gitlab is and where it is going.

9.2.1 What is GitLab and where did it come from?

GitLab is a web-based DevOps lifecycle tool which, according to their own claims, offers a *"single application for the entire DevOps lifecycle"*. This lifecycle, among other things, includes Git-repository management, issue tracking and CI/CD.

The project was started by Ukrainians Dmitriy Zaporozhets and Valery Sizov around 2011, who at the time needed a tool to better collaborate with their team. The company was officially founded in 2014 with the Dutch Sytse Sijbrandij as CEO and Dymitriy as CTO. Around this time the company had short of ten employees and the tool itself would later be described by Head of Product Mark Pundsack as an "ugly, open source knock-off of GitHub". However, in the years after that GitLab experienced significant growth, it became a so-called unicorn and is now valued at $2.75 billion with over 1150 employees[2].

Interestingly, when you try to visit their address, you will find a UPS store where they have a mailbox. This is because they are a remote-only company, self-proclaiming to be the largest[3]. Having no office has its implications which we will later discover and is one of the reasons they are so reliant on their own tool to develop GitLab itself and various other associated services.

9.2.2 Open-source vs open-core

Despite GitLab being an open-source project, most of its contributions are made by employees of GitLab Inc. This company is the driving force behind the project and ensures continuity. There are two editions of GitLab available to allow the company to pay its employees. First, there is the Community Edition (CE), the free and complete open-source version including, but not limited to, all the contributions of non-employees. Then there is the paid Enterprise Edition (EE) which uses the CE at its core but includes additional features and functionality on top of the core[4].

The source code, issues and pull requests of both editions are publicly available. The difference between open-source and open-core is in essence that an open-source project is completely free to use, while an open-core project is partly free to use and a fee has to be payed for additional functionality. In further research, we will focus mainly on the open-source free-to-use Community Edition of GitLab, unless mentioned otherwise.

As described before, GitLab aims to deliver "a single application for the entire DevOps lifecycle". Source code management, issue tracking, time tracking and continuous integration are the minimum required features to accomplish this. These are all integrated in the Community Editon. Integrations with other tools and features improving the efficiency and quality of the DevOps lifecycle are provided by the Enterprise Edition.

9.2.3 The problem of many stakeholders

There are several different kinds of users involved in the DevOps lifecycle and thus does GitLab maneuvers itself in a difficult position with many different stakeholders involved. Developers, project managers,

[2] https://www.sfchronicle.com/business/article/Where-in-the-world-are-GitLab-s-workers-15018413.php
[3] https://about.gitlab.com/company/culture/all-remote/
[4] https://about.gitlab.com/install/ce-or-ee/

operation managers and DevOps engineers are just a few examples of end users of the application. They all have different demands and different processes and workflows. To accommodate for all these different needs, GitLab had to find a very flexible way of setting up a development process. Their solution to this will be described in the next section.

What makes GitLab very interesting is that they use a strategy called *dogfooding*, which means that the whole company uses GitLab for their own product development. For example software engineers not only build the software for GitLab, they also use the GitLab software for development of GitLab. This double role holds for almost any employee of GitLab, which makes the entire company an end user of their own product. Being an end user themselves, makes it also easier to place themselves into the shoes of their customers and therefore develop for the needs of many different stakeholders.

9.2.4 Different users, different needs

Having so many different stakeholders, GitLab had the problem of many different end-user mental models as well. Each role in the lifecycle will have different ways to interact with the system. To unify these models, GitLab has a few core services, which are simple, but very flexible[5]. These will be described below.

- **Projects** Projects are used as a top level distinction between different systems, applications or company departments.
- **Issues** Issues provide a way to document any changes, improvements or bugs in the project.
- **Repositories** The repository provides a way to store files with version control using Git in the project.
- **Merge requests** Merge requests can be used as a way to discuss, improve and analyze changes to the repository.
- **CI** Provides continuous validation of the repository state and can prevent any bugs covered by automated testing.

Combining these above features, almost all stages in the DevOps lifecycle are covered and each department can use a (sub)set of these features in their daily workflow. As an example we take a software developer who works on project X. The developer is assigned to fix a bug in project X and to see what the bug actually is, he takes a look at the issue. In the issue is documented what the problem is and the developer can ask questions in the issue if something is unclear. Then when he has fixed the bug, he commits it to the repository and opens a merge request. After review by another developer, the merge request is merged and by using CI the developers know this fix did not cause another bug.

As you can see, the developer makes use of most core services and interacts with different people, without having to use different platforms. The product manager who documented the bug can also easily see the progress of the fixes in the same application by looking at the status of the merge request.

9.2.5 The Future

GitLab started in 2013 as a repository management system like Github. It lets teams share code, keep track of issues and collaborate. Over time GitLab grew from an open-source project to an open-core company, providing a single application for the entire DevOps Lifecycle. With their product, GitLab aims for product leadership, already having reached that for Continuous Integration and Source Code Management[6]. GitLab aims to reach this stage in the near future for Continuous Deployment and Release Automation, Project management and Application Security Testing as well.

[5] https://about.gitlab.com/blog/2016/10/25/gitlab-workflow-an-overview
[6] https://about.gitlab.com/direction/

In a 2018 livestream[7], GitLab's Head of Product Mark Pundsack explained that they want to step away from 'Minimum Viable Changes' and towards 'Minimum Lovable Feature'. In essence, this means that they aim to increase the depth of some of their features. Their main features, like Source Code Management and Continuous Integration, are far more mature than most other features. By making all features lovable, GitLab aims to create a product where its users love using all parts of it.

GitLab's roadmap aims at maturing their DevOps categories like Managing, Monitoring and Releasing, with new features like Status Pages, Secrets Management and Code Analytics[8].

Finally, there is a list of moonshot ideas[9] that may take a long time to implement, test and deliver. Ideas are applicable to both the open-source community edition and the proprietary enterprise edition and range from better experience from the command line tools and IDEs to features that better accommodate machine learning projects and microservices.

In this article, we tried to give a brief introduction to GitLab, its history, business model, users, and future plans. We will use this information in the upcoming article where the software architecture is subject to our investigation.

9.3 Architecting for everyone's contribution

In this second article in a series of four, we, four Computer Science master students from Delft University of Technology, continue our pursuit to analyse the open-source project GitLab. In our previous article, we discussed the fundamental concepts and product vision of GitLab. In this article, we will dive deeper into the key architectural elements of the system to gain a better understanding how these concepts and visions were realised.

GitLab's mission is to enable everyone to contribute[10]. But are we able to see this back in its architecture? To find out, we will first take a look at general architectural views on software systems and elaborate why these are relevant for GitLab. Next, we will dive deeper into the architecture of GitLab itself and how the different architectural views influence its design.

9.3.1 Architectural views

An architectural view, as described by Phillipe Kruchten's "*The 4+1 View Model of architecture*"[11] (later standardized by *IEEE Standard 1471*[12]) is used to "*address a specific set of concerns of interest to different stakeholders in the system.*"

In their book[13], Rozanski and Woods continue on Kruchten's work and present "*seven core viewpoints for information system architecture*". They define a view as follows: "*A view is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders*". So, a view should show stakeholders how the architecture

[7] https://about.gitlab.com/blog/2018/10/01/gitlab-product-vision/
[8] https://docs.google.com/presentation/d/19o720CqP9S-xRQoT9y8FF7DgtRxuJhQedaaQQQygYx8/edit
[9] https://about.gitlab.com/direction/#moonshots
[10] https://about.gitlab.com/company/strategy/#mission
[11] Philippe Kruchten. The 4+1 View Model of architecture. IEEE Software 12(6), 1995.
[12] https://standards.ieee.org/standard/1471-2000.html
[13] Rozanski, Nick, and Eoin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012.

addresses their concerns. In short, all viewpoints are relevant to GitLab. To see why, we will briefly look at how each applies to the project.

GitLab is made by and for developers, and developers are part of all the stakeholder groups of the project. The system interacts with developers and other systems, like the CI runners that users can host themselves, to function. Having an overview of GitLab's relationships, dependencies and interactions with its environment is therefore helpful. Within GitLab, different components interact with each other. Databases, web interfaces and Git itself are some of the basic elements of GitLab that help it function. To know how these elements can be interacted with positively influences the system's ability to change. To ensure GitLab functions quickly for everyone in its large userbase, parts of its functionality can run concurrently, such as the web interface. This helps decrease waiting time, and overall increases performance for both the systems and its users. Identifying the components that can run in concurrency is vital for the developers to coordinate. From this we can see that the *Context*, *Functional* and *Concurrency* viewpoints are relevant to GitLab.

Code version control is the cornerstone of GitLab. Whenever a file is created, edited or deleted, the information describing this must be stored and maintained somewhere. Besides this, GitLab itself provides users with the ability to create issues, merge requests, labels and comments, which all must be stored and displayed. As information management is important for the functionality of GitLab, describing how the system does that is essential. The *Information* viewpoints is therefore relevant.

The *Development*, *Deployment* and *Operational* viewpoints are discussed in greater detail in the following sections, where we can see why they too are relevant to the project.

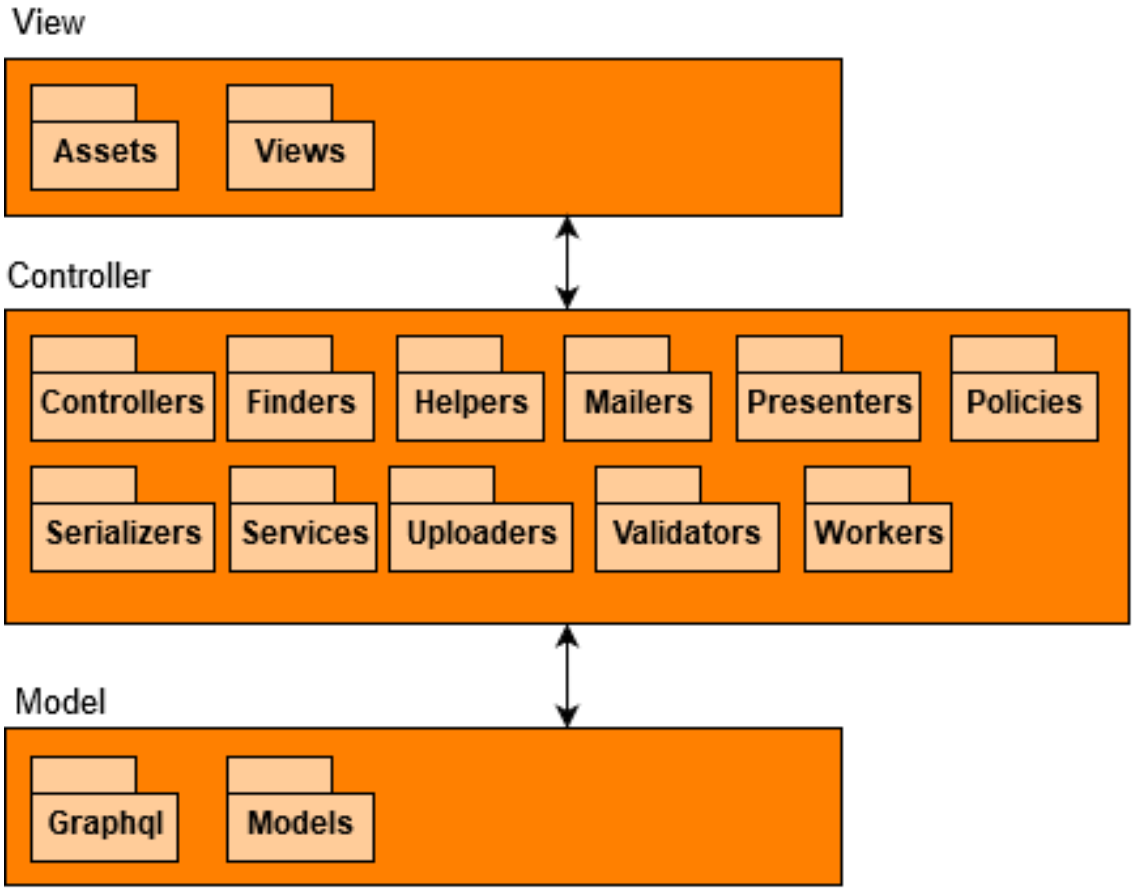
9.3.2 Developers first

The design philosophy of Ruby on Rails, the framework GitLab is built upon, has two main principles; 'Don't Repeat Yourself' (DRY) and 'Convention over Configuration'[14]. The first implies one should not write duplicated code to improve maintainability and stability. The latter comes from the fact that Ruby on Rails has strong assumptions about the needs of developers and enforces certain conventions. As Sid Sijbrandij, CEO of GitLab, illustratively said *"... in every kitchen you enter, you never know where the knives and plates are located. But with Ruby on Rails, you enter the kitchen and it's always in the same place ..."* [15]. A tool like Ruby on Rails is exactly what GitLab needs to accomplish its mission to enable everybody to contribute. In this section we will see how this framework affects the applied architectural patterns and we will decompose the system into its main modules.

9.3.2.1 Software decomposition

In 2016, DESOSA[16] explained and visualised the Model-View-Controller architecture which GitLab inherited from Ruby on Rails. Since then, small changes to the architecture has been made which are visualised in the system decomposition below. The differences are visible in the *Controller* module where *Serializers*, *Uploaders*, *Presenters* and *Policies* are added, and in *Model* where *Graphql* is added.

[14] https://guides.rubyonrails.org/getting_started.html
[15] https://about.gitlab.com/blog/2018/10/29/why-we-use-rails-to-build-gitlab/
[16] https://pure.tudelft.nl/portal/files/8039977/desosa2016.pdf



System decomposition illustrating the Model-View-Controller architecture of GitLab anno 2020.

The addition of the *Graphql* package in the *Model* module is explainable by the current transition from using the current REST API towards the usage of Graphql, an open-source query framework for APIs aiming to reduce the number of requests developed by Facebook.

Each other new module is created to improve the architecture. Uploaders, Policies, and Serializers are designed to apply business logic easier. For example, the latter is used to apply business rules on the conversion from models to exposable JSON. Presenters contain the view related logic extracted from the models to adhere to the single-responsibility principle.

9.3.3 Performance second

Ruby and Ruby on Rails are both designed with ease of use of the developers in mind[17]. Since they are not optimized for performance, systems built upon these technologies have significant overhead. However, to keep contributing to GitLab as convenient as possible, new components are first built in Ruby on Rails. When scaling and/or performance issues arise changes can be made[18]. This leads to the majority of GitLab still being developed in Ruby on Rails and small parts being optimized. Examples of this are the usage

[17] https://phptoruby.io/ruby_history/
[18] https://about.gitlab.com/blog/2018/10/29/why-we-use-rails-to-build-gitlab/

of the Vue JavaScript framework for frequently accessed pages and rewriting parts in more performant or memory-efficient languages like Go.

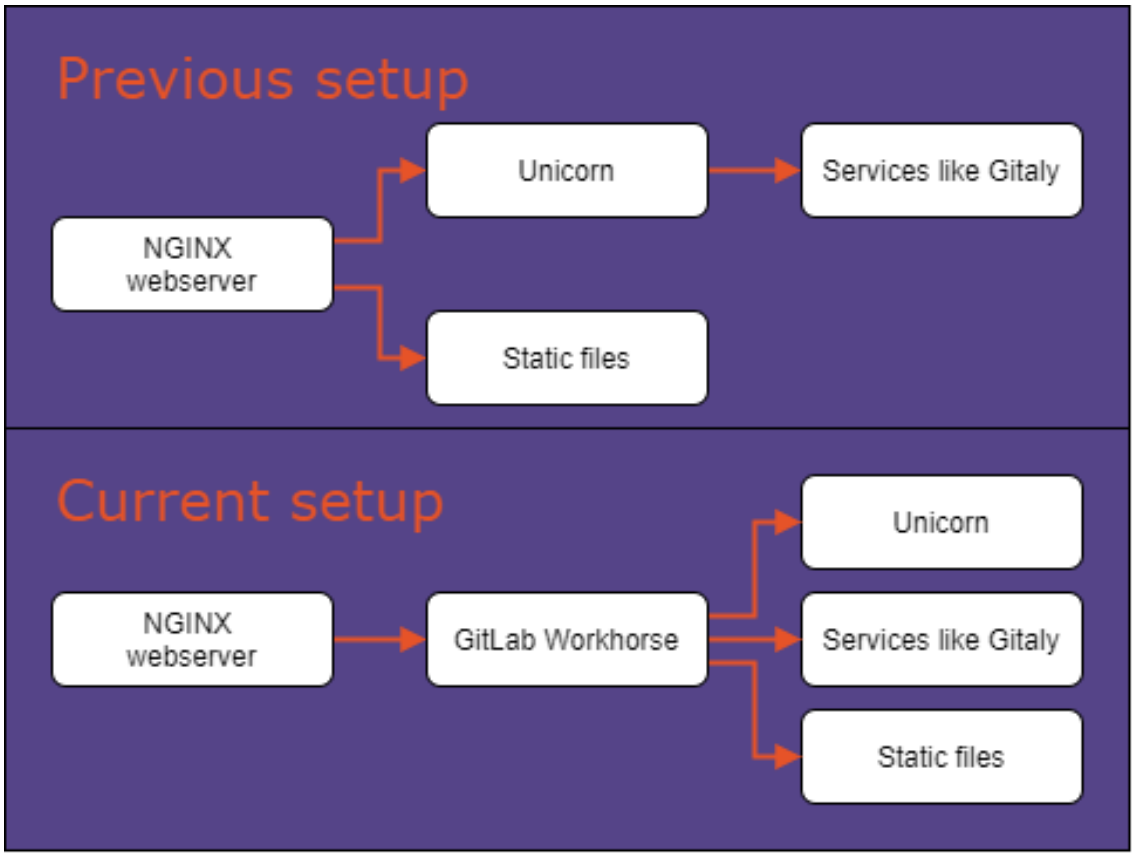
In this section we will provide examples where the non functional requirement of ease of development in Ruby on Rails has been exchanged for performance. The lifecycle of an HTTP request during runtime will be used to illustrate how the system behaves.

9.3.3.1 Scaling horizontally

One concrete example of an architectural decision made to improve the performance of GitLab is the development of Gitaly, which is a service performing all git operations through Remote Procedure Calls (RPC). Making a separate service responsible for the git operations, enables GitLab to scale both the backend and Gitaly separately and horizontally. In the beginning this was only scalable vertically. Because of the use of a complex Network File System (NFS) it became horizontally scalable[19]. However, the usage of a NFS introduced a lot of overhead causing too much latency. Gitaly is developed just like GitLab by the company and the open-source community.

Another example is the development of GitLab Workhorse, which started as a side-project to address timeouts caused by long operations and became the handler of all HTTP requests [20]. Earlier on, the nginx webserver directly served static content and redirected all other requests to the Ruby backend served by Unicorn. Unicorn provides lots of useful features but also requires low timeouts. This became a problem and caused lots of timeouts while cloning a large repository and downloading files.

[19] https://about.gitlab.com/blog/2018/09/12/the-road-to-gitaly-1-0/
[20] https://about.gitlab.com/blog/2016/04/12/a-brief-history-of-gitlab-workhorse/



Schematic overview of the situation before and after the introduction of GitLab Workhorse.

9.3.3.2 Lifecycle of an HTTP request

In the current setup with GitLab Workhorse in place, all HTTP requests are forwarded from the ngnix webserver to the service. It then directly serves the static content, forwards HTTP(S) git commands to Gitaly and preprocesses requests for the GitLab backend. Let's take a detailed look at what happens with the latter two types of request are received by this system.

1) Web requests received by the GitLab Workhorse are forwarded to the Unicorn ruby webserver. This webserver calls the corresponding controller which could interact with the database via the model definitions, services like Gitaly and returns a view.
2) Git commands, sent by for example git clients, are not forwarded to the backend. Instead, solely the credentials are verified with the backend and the commands are directly forwarded to the Gitaly service which processes it.

9.3.4 And finally, deployment

GitLab is a large system composed of many different components, some we have seen in the previous section. It is also deployed by a large variety of users with different hardware. To make the deployment easier, GitLab provides docker images for popular cloud computing providers like AWS and a package

including all services, called Omnibus. Although most of the features of GitLab are dogfooded by the company, it is noticeable that these deploy tools are not used for the public website. This is due to the enormous size of gitlab.com. In this section, we will look at the Omnibus and why GitLab releases every 22nd of each month. We will not go into detail why the deploy tools are not used for the website, but more information about the cloud architecture can be found in the documentation.

9.3.4.1 The Omnibus

From the outside GitLab might be seen as a single software system, but when looking under the hood, you will see it consists of different servers, databases and other (in)dependent components. To manage and configuring these, GitLab has developed the Omnibus. The Omnibus packages all the services GitLab consists of and publishes them as a single application. This provides a much faster development workflow as it prevents spending a lot of time on configuring all components.

9.3.4.2 Monthly release schedule

Another way in which GitLab ensures velocity in it's development, is their monthly release cycle on the 22nd of each month. This time-based release cycle has several advantages. First of all it provides updates via a predictable schedule, which is a must for distributed software systems. Users of the software need to be prepared for possible process breaking updates and should have the choice to stay at a current working version. Another advantage is that a monthly release cycle gives the development a cadence[21]. This ensures that developers can see the results of their hard work within a month, which has a big impact on morale. Furthermore, it helps with planning team capacity and ensures transparency to external contributors to the open-source code base.

9.3.5 Wrapping up

In this article, we looked at the model-view-controller architectural components of GitLab and the way trade-offs between non-functional requirements like performance and ease of development are made. Some case studies, about Gitaly and GitLab Workhorse, where performance is chosen over ease of development are given together with the approach GitLab uses to make deployments easier. Next time, we will look at the way GitLab ensures the quality of their system and the safeguards they have in place to enforce that.

9.4 Staying on the Rails: A Quality Assessment

In our last article, we looked at the architectural decisions made during the development of GitLab. The vision behind most decisions was to enable everybody to contribute. This time, we will look at the different methods used to maintain high-quality standards for a software project where everybody is invited to contribute. Furthermore, we will investigate the project in detail to highlight potential points of improvement.

9.4.1 Quality processes, workflows and guidelines

As we have discussed in previous articles, anything under development at GitLab has this peculiar meta-characteristic to it that it will also be used for developing GitLab itself. A result of this is that they not only build tools to improve the software quality processes for their customers, but also for themselves. In this section, we will give an overview of the software quality processes that are deployed in the development

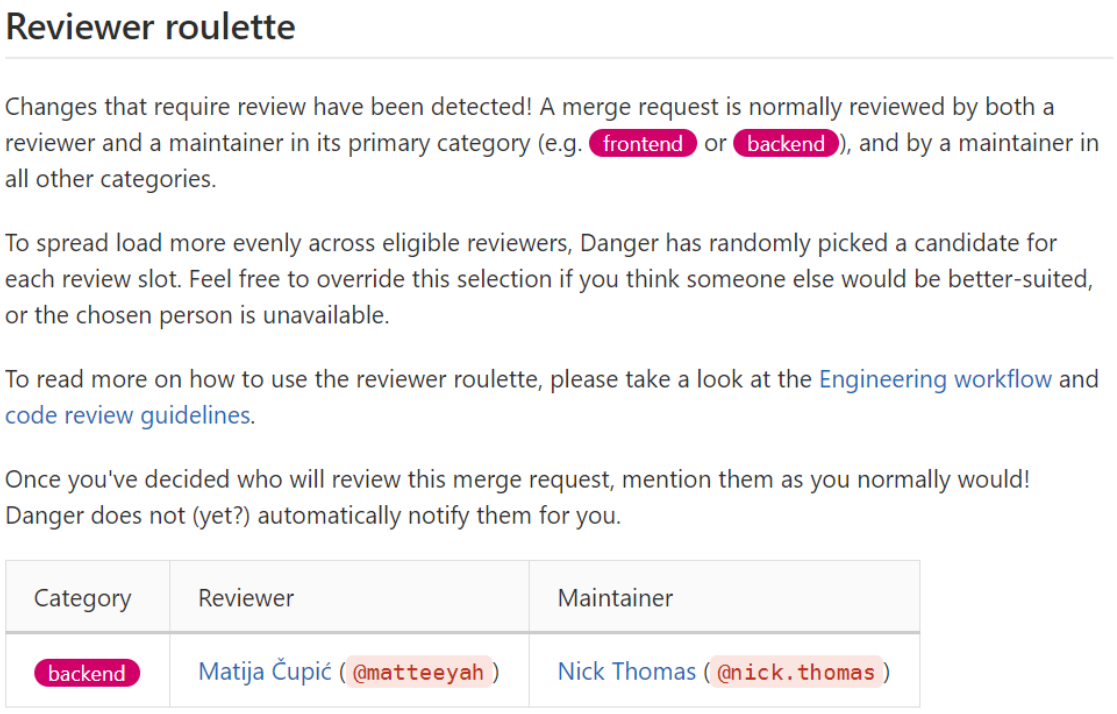
[21] https://about.gitlab.com/blog/2018/11/21/why-gitlab-uses-a-monthly-release-cycle/

of GitLab. This will be divided into their automated processes such as continuous integration and manual processes such as code reviews. This section will furthermore cover GitLab's testing policies and the overall quality that results from these processes.

9.4.1.1 Code reviews

Each contribution, either made by an employee of GitLab or the open-source community, must be reviewed before it is merged to ensure it is up to standard. These reviews are performed inside the GitLab web application and consist of two steps[22]. First, a reviewer is asked to generally review the contribution on the code level. After the feedback of the reviewer is resolved by the contributor, the reviewer approves the changes and assigns the merge request to one or more maintainers. They are responsible for the affected parts of the codebase and are allowed to merge.

As there are several maintainers for different parts of the backend, it can quickly become difficult for a contributor to determine who to ask for a review for all the involved parts. GitLab solved this with the creation of 'Reviewer Roulette'. This tool analyses the merge request and suggests a reviewer and maintainer for each part of the backend with changes. The reviewing workload is spread over all maintainers by taking into account the number of recent reviews.



Reviewer roulette

Changes that require review have been detected! A merge request is normally reviewed by both a reviewer and a maintainer in its primary category (e.g. frontend or backend), and by a maintainer in all other categories.

To spread load more evenly across eligible reviewers, Danger has randomly picked a candidate for each review slot. Feel free to override this selection if you think someone else would be better-suited, or the chosen person is unavailable.

To read more on how to use the reviewer roulette, please take a look at the Engineering workflow and code review guidelines.

Once you've decided who will review this merge request, mention them as you normally would! Danger does not (yet?) automatically notify them for you.

Category	Reviewer	Maintainer
backend	Matija Čupić (@matteeyah)	Nick Thomas (@nick.thomas)

Real-world output of 'Reviewer Roulette' on a recent merge request.

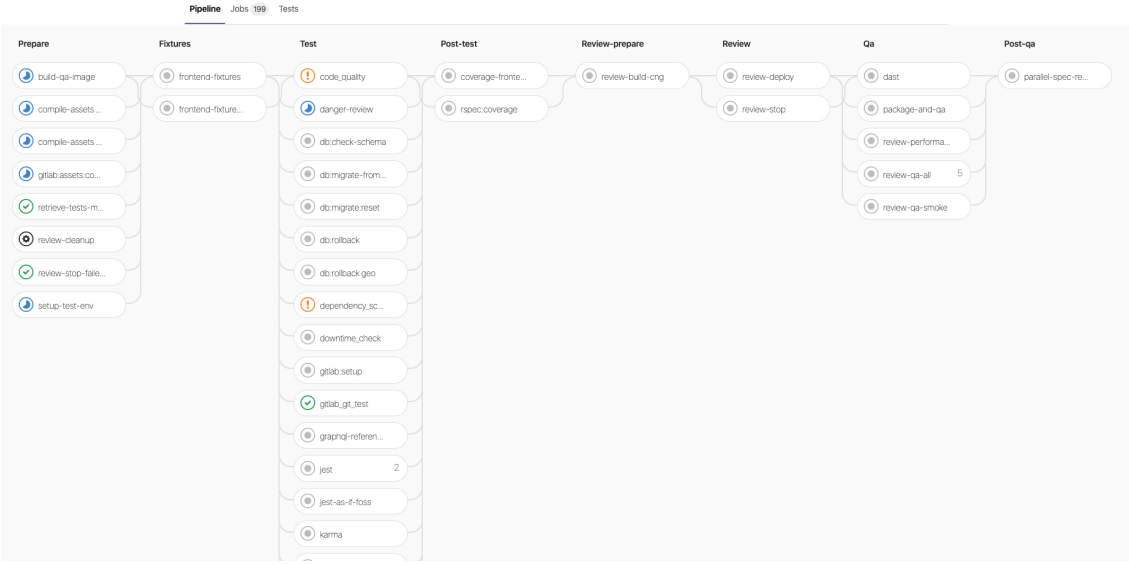
9.4.1.2 Continuous integration

An example of dogfooded tools are the continuous integration and delivery pipelines of GitLab. They assist the reviewers and maintainers by running all tests, create releases for both the community and enterprise

[22] https://docs.gitlab.com/ee/development/code_review.html

editions from the same source[23] and analyse certain branches further. One can think of test coverage measured using simplecov but also the performance in terms of runtime and memory usage of all tests. This is useful to decrease the runtime of the pipeline as it currently takes hours to complete without parallelisation. There are also some rules to speed up the pipeline for certain types of contributions[24]. It is, for example, useless to run the tests when only documentation is updated[25].

The continuous integration pipeline consists of a total of eight stages with a total of 200 jobs. Most of these jobs contain spec tests, but there are also jobs which check the coverage, code quality and run static analyses. An example of the current GitLab pipeline is shown in the image below.



Example of the CI pipeline at GitLab

9.4.1.3 Testing

At GitLab, tests are treated as first-class citizens. The extensive testing guidelines describe best practices to which all contributions should adhere. This includes requirements on which level every architectural component must be tested[26]. It makes a distinction between unit, integration and end-to-end (e2e) tests. The difference between integration and e2e tests is that the former tests the interaction between components but doesn't require the actual app environment. The latter, however, requires a browser introducing a lot of overhead but ensures proper functionality of the complete application.

The cost and value of a test must be taken into account when deciding which type of test should be created. To summarise Noel Rappin, who clearly describes all sorts of costs and values in his blog post[27], costs are related to the time it takes to create, maintain and run the tests while the value comes from the time saved due to early discovery of bugs and running them automatically. Rappin states that "... *the goal in keeping your test suite happy over the long-haul is to minimize the costs of tests and maximizing the value*".

[23] https://about.gitlab.com/blog/2019/02/21/merging-ce-and-ee-codebases/
[24] https://docs.gitlab.com/ee/development/pipelines.html#workflowrules
[25] https://docs.gitlab.com/ee/development/documentation/index.html#branch-naming
[26] https://docs.gitlab.com/ee/development/testing_guide/testing_levels.html
[27] https://medium.com/table-xi/high-cost-tests-and-high-value-tests-a86e27a54df

The table below shows the relative distribution of tests over the different testing levels. The distribution is not surprising as GitLab used the same line of reasoning as described. Unit tests are short, execute quick and are therefore low cost. However, they also provide little value. It are the end-to-end tests which provide a lot of value but also come with a significant cost due to slow pipelines.

Test level	Community Edition	Enterprise Edition
End-to-end	12.1%	7.6%
Integration	18.2%	17.2%
Unit	69.7%	75.1%

The relative distribution of tests over the testing levels measured at 2019-05-01.

An example of these trade-offs can be found with a quick look at the public dashboard containing the testing analytics. *Mailers* are the least tested architectural component with barely 90% line coverage. This can be explained by the fact that mailers should, according to the testing guidelines we mentioned earlier, only be tested with integration tests instead of unit tests.

9.4.1.4 Value of community in processes

GitLab is made by and for developers, which can be seen in its open-core nature. The Community Edition (CE) of GitLab is open-source, so everyone can contribute.

On March 25th, 2020, we had a video call with Community Manager Ray Paik and Senior Education Program Manager Christina Hupy. In this call, we got to discuss the way non-employees influenced the development of GitLab.

GitLab welcomes all contributions, and these contribtions help GitLab employees see the project from a different view. Ray Paik quoted a coworker on merge requests from an outsider, stating every time he sees a merge request from a non-employee, he gets to experience a new perspective on the project.

Ray Paik stated that over 900 non-employees have had at least 1 merge request (MR) merged, and that last year, around 15% of merged MRs were from outside the company. While most of the development is done by GitLab employees, this is still a significant amount, showing the value of GitLab's community.

9.4.2 Quality assurance through architectural choices

Now that we have a view of the quality assurance processes in place for the development of GitLab, we will transcend these processes and take a look at the quality of the overall architecture. What measures have been taken to make the architecture future-proof and what plans are in place to keep it that way? We will also assess the technical debt present in the system.

9.4.2.1 The Rails Doctrine

GitLab relies heavily on Ruby on Rails, a framework which, as the name would suggest, provides a strict format for writing web applications. In the so-called Rails Doctrine, David Heinemeier Hansson states that such a strict framework leaves the programmer with less code to write[28]. This might improve the quality of the code because there is less room for mistakes but this is only the case if such a framework provides the

[28] https://rubyonrails.org/doctrine/#beautiful-code

right tools for the job. In the case of GitLab, this is certainly the case, as it is a very good example of a web application.

Another advantage of using a predefined framework like Ruby on Rails, is the amount of quality assurance tools there are. One of these tools is Rubocop Rails. This is an extension to the popular Rubocop static code analysing tool for Ruby. It provides a way to enforce certain code conventions and best practices, which can then be used as an automatic quality assurance tool within the development process.

9.4.2.2 The Big Merge

Interestingly, not too long ago GitLab made a major architectural desision to maintain the quality of the code base[29]. Before 2019, GitLab had two repositories; GitLab Community Edition (CE) and GitLab Enterprise Edition (EE). This made it easy to separate open-source and proprietary code, which has certain benefits such as licensing.

However, as one of the senior engineering managers wrote in a blog post of early 2019, "*Feature development is difficult and error prone when making any change at GitLab in two similar yet separate repositories that depend on one another*." Even though they automated most concurrency processes over the years, there were still cases were duplicate work was necessary or other conflicts arose. Resultingly, the two projects were merged into one repository.

Originally, GitLab CE and EE were not ready to be merged. Work had to be done, as was stated in a blog post about the merge: "*In total the work involved 55 different engineers submitting more than 600 merge requests, closing just under 400 issues, and changing nearly 1.5 million lines of code*". This clearly was a big investment into the quality of the architecture, but concluded in the post the drawbacks of the seperation began to outweigh the benefits.

9.4.3 What could be improved

Previous sections have analysed various aspects in place to assure the quality of the GitLab project. In this section we will be using the Software Improvement Group (SIG)'s[30] analysis to give an assessment on what could be improved.

9.4.3.1 Software Improvement Group analysis results

At the start of this course, a componentisation of GitLab has been made. Based on this, SIG provided us with an analysis of the system. To see what could be improved, we will look at the refactoring candidates as provided by SIG.

The table below show the SIG score for different system properties.

System Properties	Score (_ /5.0)
Duplication	4.3
Unit size	3.2
Unit complexity	4.1
Unit interfacing	4.0
Module coupling	4.1

[29] https://about.gitlab.com/blog/2019/08/23/a-single-codebase-for-gitlab-community-and-enterprise-edition/
[30] https://www.softwareimprovementgroup.com/

System Properties	Score (_ /5.0)
Component independence	2.8
Component entanglement	1.2

We see that the project scores well on *Duplication, Unit complexity, Unit interfacing* and *Module coupling*, average on *Unit size* and *Component independence*, and poorly on *Component entanglement*. Let us focus on the worst property.

For *Component entanglement*, almost all problems are (indirect) cyclic dependencies between components. Over half of all modules are affected, which sounds bad, but is actually the result of the big merge mentioned in the previous section. GitLab's use of module injection is the cause of this poor score in SIG's analysis. The drawback of module injection is that components are entangled, but from a maintainability perspective, this drawback is worth it.

Based on the analysis, it would be easy, but wrong, to say that GitLab should focus on detangling their EE and CE components. It's the result of having two projects in one repository. However, cyclic dependencies also occur between EE modules, and between CE modules. This is not the result of CE and EE sharing a repository, and would benefit from detangling. The biggest refactoring candidate for EE is `Cyclic dependency between ee/lib and ee/app` with weight 84. For CE, it is `Cyclic dependency between lib and qa/qa` with weight 75. To put these weights into perspective, the total weight of all candidates is 752.

9.4.3.2 Personal experiences

While predefined metrics in tools such as the ones SIG use are useful, we would also like to share our experience with the code quality as we experienced by contributing. One of the major challenges we found as new contributors is that since GitLab is such a broad product with many concepts, the cognitive load to grasp what each class and module stands for is quite high. This is also reflected on in a recently opened issue. Their proposed solution is to reduce top-level classes, which also improves maintainability.

To conclude, in this article we have seen various processes in place and choices made to maintain the quality of GitLab. While this only scratches the surface of all the measures GitLab takes, it provides some useful insights in the architectural decisions of a project of this scale.

9.5 Towards a greener DevOps lifecycle

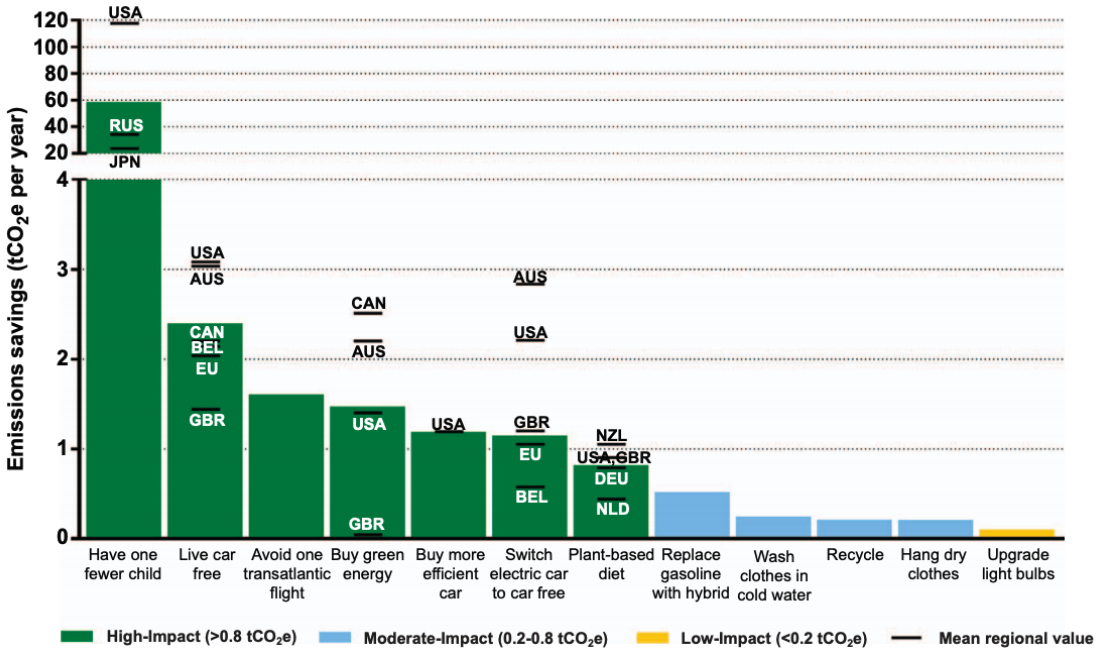
This is the last part of our series about the open-source project GitLab. Previously we have talked about the context, the architecture and the quality assurance processes of GitLab. To finish off, in this article we will make a sustainability analysis of both GitLab as the company and as the project. We will assess the supported and used infrastructure, search for features improving energy efficiency and investigate the impact of being an all-remote company. Additionally, we will see how GitLab employees value sustainability for them and the company.

9.5.1 Staying at home

At the time of writing this, urban areas are rapidly starting to show signs of lower pollution due to people working from home as a result of the 2019 Corona outbreak[31]. In this section we will use available sources to give an estimation on what impact being remote-only has for GitLab's carbon footprint.

There are several aspects of working from home that could influence the carbon footprint of employees. Perhaps the most obvious one is daily commuting. GitLab employees do not have to drive or take public transit to work. Especially in countries where public transit is not widely available, this can make a significant impact.

However, to compensate for the lack of real-life encounters GitLab employees have, they have a yearly summit called Contribute. While this is probably a necessity to keep a healthy work environment, it has the downside that a large number of people are flying all over the world. The figure below indicates of how this compares with not having a car[32].



Comparison of emissions from individual actions from research by Wynes & Nicholas[33]

This leads us to conclude that while it may be beneficial for local pollution as a result from both not having to live in already densely populated areas and not having to travel to work, the total carbon footprint of flying outweighs not having a car on average. While no actions have been taken thus far, according to a comment on one of our issues an effort has been started to perhaps offset the carbon footprint of these flights and to start other sustainability projects.

Lastly, we had a look at the available literature on the environmental impact of an office. There is not a lot

[31] Coronavirus, Lockdowns continue to suppress European pollution, https://www.bbc.com/news/science-environment-52065140, accessed on 09-04-2020

[32] Seth Wynes and Kimberly A Nicholas 2017 Environ. Res. Lett. 12 074024

[33] Seth Wynes and Kimberly A Nicholas 2017 Environ. Res. Lett. 12 074024

of research available, probably due to the various variables that come into such an analysis; construction, isolation and climate are just a few to name. Therefore, the results we will present are meant to give a very rough estimate and could vary greatly, so they should not be taken for absolute truth.

In their analysis of three fifty-year case studies of Finnish offices, the authors found that the buildings had between 3 and 4 ton CO2 equivalent pollution per square metre over their fifty year lifespan[34]. While we couldn't find a study on the average office space needed per employee (which probably also differs per country and company), the Dutch government aims to have around 25m2 office space per employee. Using this indication would result in between 1,5 and 2 ton CO2 equivalent pollution per year per employee[35]. Looking back at the image, this would mean that not having an office compensates a transatlantic flight. However, as mentioned this greatly depends on the circumstances.

9.5.2 Awareness of sustainability at GitLab

In the week of March 30th, we had two calls with GitLab employees. The first was with Zeger-Jan van de Weg, who is the Engineering manager of the Gitaly team. The second was with Marin Jankovski, one of the first employees of GitLab, who currently has the role Senior Engineering Manager, Infrastructure, Delivery & Scalability. During both calls, we asked them about how they experienced sustainability within GitLab.

Zeger-Jan explained that for Gitaly, sustainability is not a priority. The responsibility is partially pushed off to their cloud provider, which we will look into in the next section. In terms of architectural changes and code development, sustainability is not a factor when making decisions.

In our call with Marin, the answer on what GitLab's sustainability policy was, was quite short. He wasn't aware of any. However, we also got to discuss the next big architectural change for GitLab, which is to move from GitLab as a monolith to running on Kubernetes. While this change was not motivated by sustainability, the automatically scaling nature of Kubernetes helps prevent idly running VMs. As a result, less energy is wasted while running GitLab, which is an improvement on the sustainability front.

GitLab's handbook[36] and the issue tracker shows no relevant hits when searching for *sustainability*, *energy consumption*, *energy efficiency* or *carbon footprint*. It would appear that GitLab indeed does not have a policy on sustainability. While that may be true, this does not mean sustainability is not relevant to its employees. In this issue, one can see the spirited discussion among employees on ways GitLab can reduce their carbon footprint. While there aren't a lot of issues that address sustainability at GitLab, this does show that GitLab's employees want to inspire themselves and each other to be conscious about the topic.

9.5.3 Pick your cloud provider

GitLab provides official installation support for different operating systems, container managers and cloud providers. We assume that large customers will most likely opt for a cloud solution or hosting at gitlab.com. In this section, we will asses, based on the Greenpeace Clicking Clean 2017 report[37], the environmental impact of the supported cloud providers and the hosting of gitlab.com. Before assessing the environmental impact of the different cloud provides, we want to stress that usage of the cloud is, for most applications,

[34] Junnila, Seppo. (2004). The Environmental Impact of an Office Building Throughout its Life Cycle. 951-22-7284-9.

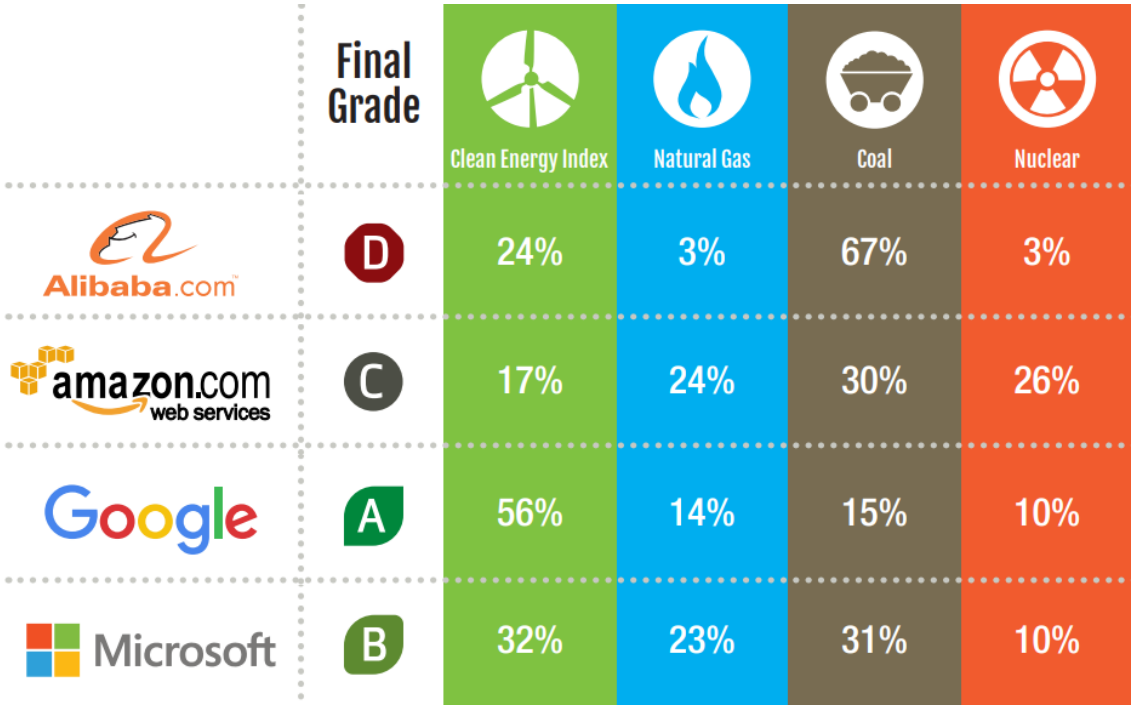
[35] FWR-wijzer, Praktische gids voor de Fysieke werkomgeving Rijk, https://www.cfpb.nl/kennis/publicaties/fwr-wijzer-praktische-gids-voor-de-fysieke-werkomgeving-rijk/

[36] https://about.gitlab.com/handbook/

[37] Greenpeace #ClickClean, http://www.clickclean.org/international/en/

already better than self-hosting, due to the higher utility of the machines[38].

All of the three biggest western cloud providers, namely Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure, come with official support. Both GCP and Azure score relatively well according to Greenpeace as visible in the snippet below. Greenpeace explains in the report that the low score of AWS is partly due to the lack of transparency. Obviously, the report is already three years old and things can be changed. However, AWS is recently accused by Greenpeace to break its commitment[39] and thus it is not apparent that it is getting better.



Snippet from the Greenpeace Clicking Clean report.

GitLab also provides its product as software-as-a-service where one can create public and private repositories on gitlab.com. The infrastructure of this service is expected to handle significantly higher loads than instances of self-managed customers. Recently, GitLab has decided to migrate its complete production environment from Azure to GCP to reduce the costs, failure rate and latency[40][41]. The details about the migration and the openness around it are a textbook example of the values of GitLab and could be the subject of another article. Although sustainability was not part of the objectives of the migration, it was a step in the right direction according to the Greenpeace report.

[38] Cloud Computing, Server Utilization, & the Environment, https://aws.amazon.com/blogs/aws/cloud-computing-server-utilization-the-environment/

[39] Greenpeace Finds Amazon Breaking Commitment to Power Cloud with 100% Renewable Energy, https://www.greenpeace.org/usa/news/greenpeace-finds-amazon-breaking-commitment-to-power-cloud-with-100-renewable-energy/

[40] We're moving from Azure to Google Cloud Platform, https://about.gitlab.com/blog/2018/06/25/moving-to-gcp/

[41] GitLab's journey from Azure to GCP, https://about.gitlab.com/blog/2019/05/02/gitlab-journey-from-azure-to-gcp/

9.5.4 Automate deployment and scaling

At the time of moving from Azure to GCP, GitLab was already in the middle of another migration. Several hundreds of virtual machines were required to host gitlab.com, and the operations part of deploying these will soon become infeasible. Therefore the decision was made to enable GitLab to run on a Kubernetes cluster. This would provide automatic deployment and scaling which would reduce the workload of the operations department. This migration, however, was far from finished at the time of the move and caused the start at GCP with a similar setup as used at Azure.

There are two details important to mention about this migration. First of all, it is an incremental project which is currently still in process. And second, while Kubernetes is mostly used for microservices, this is not what GitLab is after. The whole software suite is split up into a small number, with only seven in the current proposal[42], of distinct services which will be separately scalable.

Usage of Kubernetes does not only decrease the load on the operations department, but it is also able to scale faster than the department can do manually. This reduces the number of idle resources which in turn decrease the cost and energy consumption. In some cases, the average CPU utilization increases with a factor of ten[43].

9.5.5 Conclusion

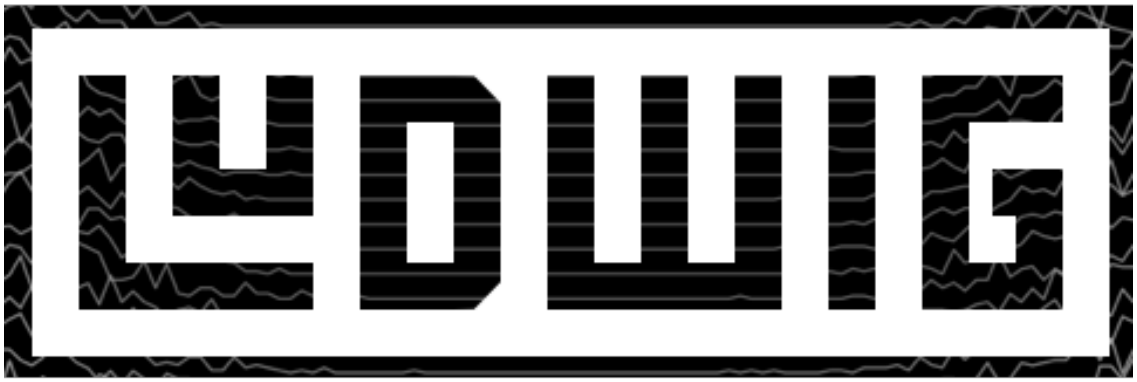
From our calls with GitLab employees and the available literature such as GitLab's handbook, we conclude that GitLab does not have an official policy on sustainability. That being said, they do have a record of taking actions that are beneficial to environmental sustainability. Examples of this are picking the greenest cloud provider, moving their deployment to Kubernetes, but also being a remote-only company. It can also be seen that some GitLab employees value sustainability, which might help GitLab make more choices that are favourable for sustainability in the future. In terms of sustainability at this moment, GitLab does a good job and could keep this up by cutting their most environmentally unfriendly choices, such as transatlantic destinations for its yearly conference.

[42] Production Architecture, https://about.gitlab.com/handbook/engineering/infrastructure/production/architecture/#infra-proposed-archi-pods

[43] How Kubernetes Can Help Reduce the Cloud's Carbon Footprint, https://thenewstack.io/how-kubernetes-can-help-reduce-the-clouds-carbon-footprint

Chapter 10

Ludwig



Ludwig is a toolbox built on top of TensorFlow that allows to train and test deep learning models without the need to write code.

All you need to provide is a CSV file containing your data, a list of columns to use as inputs, and a list of columns to use as outputs, Ludwig will do the rest. Simple commands can be used to train models both locally and in a distributed way, and to use them to predict on new data.

A programmatic API is also available in order to use Ludwig from your python code. A suite of visualization tools allows you to analyze models' training and test performance and to compare them.

It can be used by practitioners to quickly train and test deep learning models as well as by researchers to obtain strong baselines to compare against and have an experimentation setting that ensures comparability by performing standard data preprocessing and visualization.

The following sections analyze Ludwig's structure and architecture from various standpoints. First we examine the vision of the project itself and the stakeholders involved in the Ludwig project. Following this, we attempt to visualize Ludwig's architecture in different perspectives, namely: the context view, the development view, and the deployment view. Next, we look to analyze the project's technical debt and present a code-level view of the project. Finally, we present out conclusions and summarize our findings about the architecture of the Ludwig project.

10.1 The Team

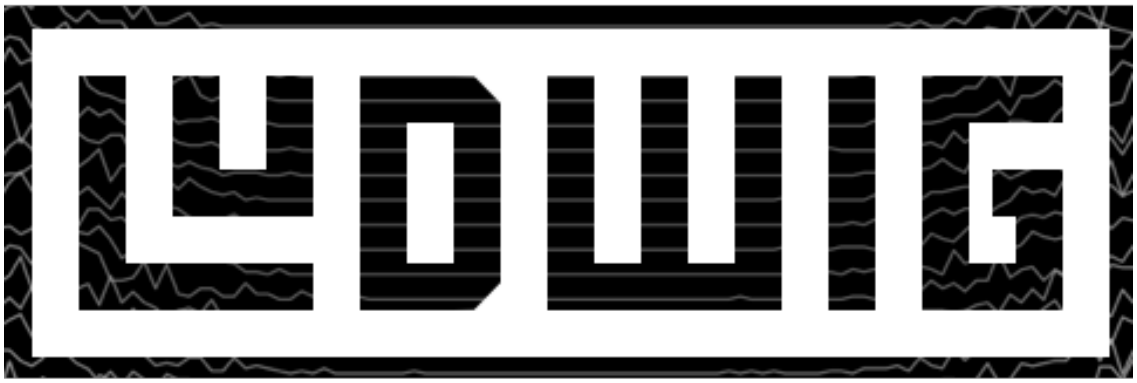
Meet the team:

- Soovam Biswal
- Abhishek Vijay
- Kushal Prakash
- Yüksel Yönsel

10.2 The Vision of Ludwig

Technology should be accessible to everyone - be it an expert in a domain or a novice. Ludwig is such a toolbox that bridges this gap with it's singular motive to make *machine learning* as simple and as accessible as possible.

10.2.1 What is Ludwig?



Ludwig is a machine learning toolbox started by Uber in its endeavour to promote open source development. With its main motive being to make machine learning simpler and easily accessible to everyone, Ludwig attempts to provide you with a toolbox that lets you train your deep learning model without writing code. Furthermore, a suite of visualization tools allows you to analyze models' training and test performance and to compare them. This aims to help experts from other domains without a lot of deep learning or programming knowledge to build and test their own models with minimal effort (However, programmatic (Python) API are also available to use for those so inclined). It can be used by practitioners to quickly train and test deep learning models as well as by researchers to obtain strong baselines to compare against and have an experimentation setting that ensures comparability by performing standard data preprocessing and

visualization.

Ludwig was built with accessibility in mind, meaning that its fundamental values included flexibility, generality, and simplicity of use. Additionally, it was structured so as to be extensible - allowing for easy and methodical addition of support for new data types as well as new model architectures.

10.2.2 End-User Mental Model

Anyone dealing with the application of machine learning need not always be interested to learn about the underlying principles that make the ML algorithm run or implement it from scratch. For instance, for a developer at the very beginning of his/her deep learning career, it may be important for him/her to train, test or visualize ML algorithms without much programming to get a proper understanding. For a researcher, it may be important to obtain accuracy in the results of his/her experiments through the application of sophisticated ML algorithms. Or, for a deep learning practitioner, it may be essential to create fast prototyping and experimentation in order to further develop the model. Ludwig responds to the needs of all these users by implementing the principle of data-abstraction in order to ensure that the user focuses mostly on building his/her model without much coding and debugging. All that the user needs to provide is a model description in a *yaml* file and a dataset in *csv* format, in order to obtain a suitably trained model.

10.2.3 Key Features and Capabilities

In order to understand the capabilities of the Ludwig toolbox, an analogy stated by it's authors is as follows: *If deep learning libraries provide the building blocks to make your building, Ludwig provides the buildings to make your city, and you can chose among the available buildings or add your own building to the set of available ones*.

With this in mind, some of the major features and capabilities of this toolbox is stated below:

- A new data-type based approach to deep learning model design
- A suite of visualization tools enabling users to analyze thier algorithms
- Fast prototyping and experimentation
- Abstraction of data types leading to a more understandable model
- Flexibility over different deep learning models and training procedures

It is also important to keep in mind the limitations of the scopes provided by this toolbox. They are as follows:

- All those data types that require data-type specific implementations are not allowed
- Tensorflow 2 support is not provided
- It does not allow pre-trained models used in the translation of languages

10.2.4 Stakeholder Analysis

A stakeholder is a person, group, or organization that has rights, shares, claims, or interests concerning the realization of the system or its functions to meet their needs. The following section gives an analysis of the stakeholders of Ludwig. The stakeholders are categorized on the categories defined by Rozanski and Woods[1]. In addition, we have analyzed the competitors and facilitators of Ludwig to provide a more complete overview of the interest in the system.

[1] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.

Stakeholder	Description	Definition
Acquirers	According to [2], acquirers oversee the procurement of the system or product. In our case, this clearly indicates *Uber* as the sole acquirer as Ludwig was developed in the Uber AI Labs and has been adopted for building a bunch of products at Uber such as COTA, Uber Eats and many more applications that are currently in production.	Oversee the procurement of the system or product
Assessors	Uber in addition to being the acquirer, can also be considered as the assessor of the Ludwig project. Specifically, Piero Molino is the main architect and maintainer of the Ludwig project	Oversee the system's conformance to standards and legal regulation.
Communicators	The primary source of documentation of the Ludwig toolbox is the Ludwig website. Additionally, [3] describes the operation and framework of the toolbox. (Contributors Pranav Subramani and Emidio Torre developed the documentation and landing pages of the Ludwig website)	Explain the system to other stakeholders via its documentation and training materials.
Developers	Ludwig, being a fairly recent project, has over 60 direct contributors including the Ludwig Core Development team and various members of the github community	Construct and deploy the system from specifications (or lead the teams that do this)
Maintainers	Piero Molino, one of the authors of Ludwig is the maintainer for this project. He oversees all modifications to the system via merge requests.	Manage the evolution of the syPost a question on StackOverflow where other users can answeronce it is operational
Suppliers	The ludwig toolbox is implemented in Python and runs on top of TensorFlow. The suppliers of the Ludwig toolbox includes the various dependencies (Python modules) that make up the Ludwig system and the platforms on which it runs. The ludwig project uses github for code version control, issue tracking, and releases.	Build and/or supply the hardware, software, or infrastructure on which the system will run
Support Staffs	Ludwig does not have staff dedicated to support users with issues. However, they can receive help from the community in the following ways: *(i)* Post a question on StackOverflow where other users can answer *(ii)* Consult the extensive Ludwig toolbox documentation *(iii)* Raise an issue on github where primarily development issues and errors can be raised and answered by the active developer community.	Provide support to users for the product or system when it is running

Stakeholder	Description	Definition
Testers	The ludwig toolbox comes with a set of integration tests (based on pytest module) which ensure end-to-end functionality. All code contributors are expected to perform unit tests on their code before performing a pull request. Therefore, all active contributors to the project can be considered as testers.	Test the system to ensure that it is suitable for use
Users	This toolbox can be used by two different categories of people i.e., non-expert machine learning practitioners and experienced deep learning developers and researchers	Define the system's functionality and ultimately make use of it

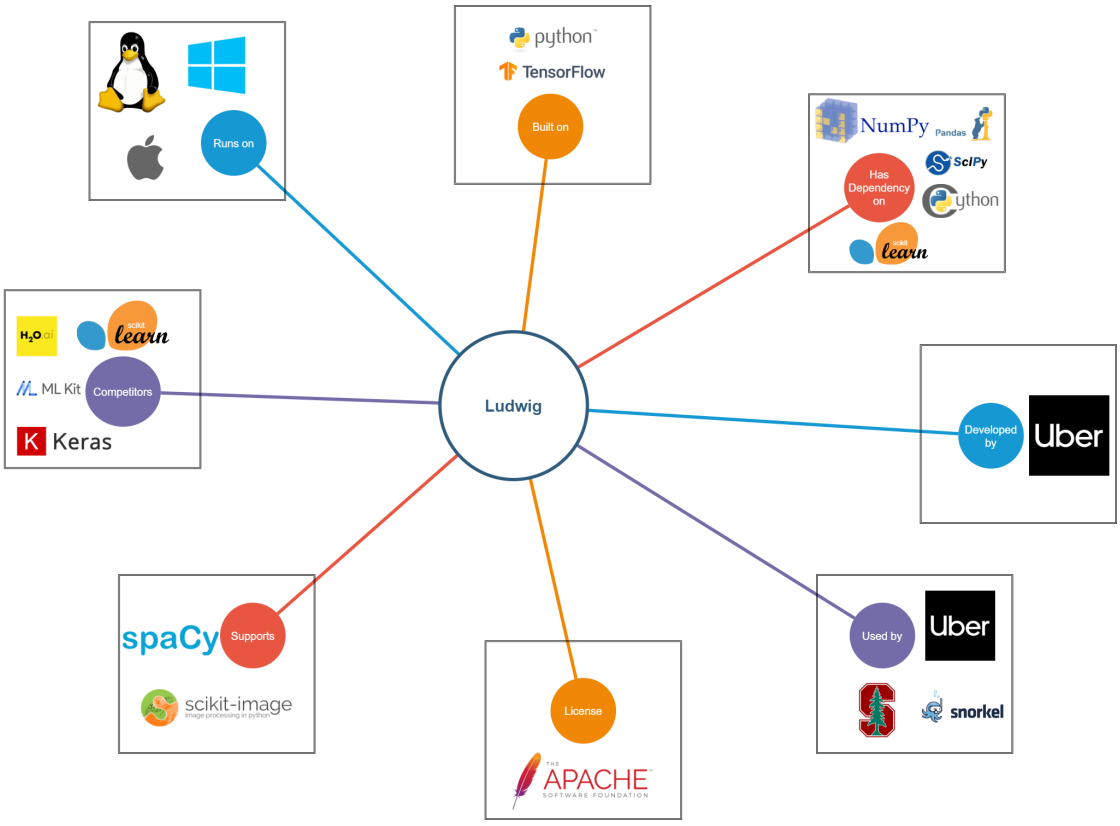
Additional stakeholders	Description	Definition
Competitors	Sonnet, Keras, AllenNLP, and other similar libraries are similar to Lud-wig in the sense that these libraries provide a higher levelof abstraction over TensorFlow and PyTorch. And therefore can be considered direct competitiors.	An organization or product engaged in commercial or economic competition with the system. Related works section of paper [2]
Facilitators and enthusiasts	There are a number of Ludwig tutorials on various media (YouTube, Medium, Reddit etc.) from facilitators and enthusiasts such as Uber Eng, Gilbert Tanner etc.	A person or a group that is involved in creating content relating to the system for educational or leisure reasons.

10.2.5 Current and Future Context

In order to provide a clear understanding of the scope and operation of the Ludwig project, we have an illustration that depicts various dependencies, relationships and intersections of Ludwig as a product with various external entities.

[2] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.

[3] Molino, Piero & Dudin, Yaroslav & Miryala, Sai. (2019). Ludwig: a type-based declarative deep learning toolbox.



Ludwig is solely written in Python and built on top of TensorFlow. It is actively maintained with an open source Apache License 2.0 on Github where it encourages contributions from the dev-community with an exhaustive list of code conventions and how-tos. It has been used in more than ten projects inside Uber, consisting of text classification, entity recognition, image classification, information extraction, dialogue systems, language generation, timeseries forecasting, and many more [4].

The future plan aims at improving the data preprocessing system existing in Ludwig. Currently, hyperparameter optimization is a functionality which is actively being worked upon in order to include it in the toolbox. Moreover, a team at Uber is also working with Prof. Christopher Re's Hazyresearch group at Stanford University to integrate Ludwig with Snorkel, their system for programmatically building and managing training datasets to rapidly and flexibly fuel machine learning models [5].

10.2.6 Product Roadmap

Though the Ludwig team has explicitly stated to consider the addition of new features based on the feedbacks of the community, the following list states the features that they are currently planning to include [6]:

- New text and sequence encoders
- Image Encoders and image decoding

[4] Piero Molino's personal website on Ludwig
[5] Uber Engineering. (2019). Blog post on: Ludwig v0.2 Adds New Features and Other Improvements to its Deep Learning Toolbox
[6] Product Roadmap for Ludwig

- time series decoding
- New feature types (point clouds etc.) and pre-processing support for them

In addition to this, the team has also addressed the current limitations of the product. They are stated as follows:

- Ludwig currently does not support dynamic batch reading of data from memory for different datasets except images
- It does not possess a documentation for lower level functions and UI to provide live demo capability
- It needs to optimize the data I/O to TensorFlow
- It intends to increase the number of supported data formats beyond just CSV and integrating with Petastorm

10.3 Ludwig - Connecting the Vision to Architecture

Architecture is a representation of a system that most if not all of the system's stakeholders can use as a basis for mutual understanding, consensus, and communication. When we talk about the architecture of a software, we refer to a plan that describes aspects of its functionality and the decisions that directly affect these aspects. In this sense, a software's architecture can be viewed as a set of interconnected business and technical decisions.

So, having looked at Ludwig's architecture in a high-level system context, we now aim to delve into its technical structure so as to better understand the choices and decisions that have gone into Ludwig's development.

However, this implies taking into consideration all kinds of requirements: system functionality, system organization, how the system modules interact with each other, whether there are external dependencies, what information needs to be managed or presented or stored, what risks to take into consideration, and much more. To try and answer these questions, we have attempted to split Ludwig's technical aspects with relation to the various architectural views mentioned in Rozanski and Woods[7].

10.3.1 Primary functional elements

The primary functional element - the Ludwig core performs two main functionalities: training models and model based prediction. In addition to this, there exist several visualization and distribution elements. The primary interfaces for Ludwig are the CLI (Command Line Interfaces) entry points that allow users to: train, predict, test/evaluate, visualize and serve.

10.3.2 Information and Data Flow

Ludwig treats user inputs and processed outputs as follows: Ludwig takes user inputs (CSV and model YAML). It then splits and preprocesses the data and builds a model. This is then trained until accuracy stops improving. Finally, Ludwig places the (output) trained model along with its hyperparameters and statistics of the training process in the generated `/results` directory.

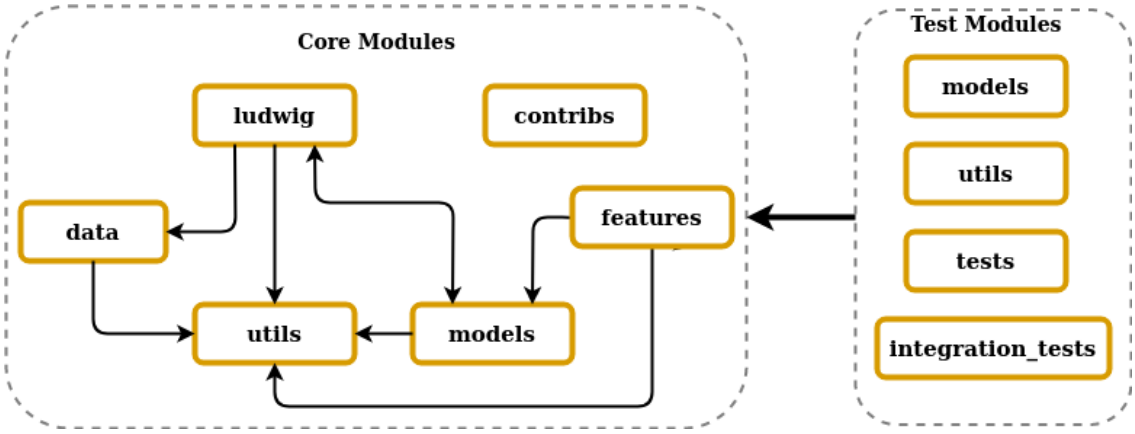
[7] (Views and Viewpoint) Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.

10.3.3 Building Ludwig's code-base from a Development Viewpoint

In order to understand the the planning and design of the development environment that supports Ludwig's software development process, let's dive into Ludwig's codebase[8] and focus on it's Development Viewpoint.

10.3.3.1 Module Organization

The figure below illustrates Ludwig's core and test module organization and their inter-dependencies inspired from Sigrid.



A brief description about each module's role is:

- **Ludwig:** Responsible for integrating and running other modules to perform training, testing, prediction and visualisation for data sets and other configuration parameters.
- **Data:** Deals with managing input data for the entire training process beginning from csv file-read to pre-processing and post-processing.
- **Features:** Includes feature classes that are responsible for data-type-specific logic implementations required to pre(post)-process specific raw data types.
- **Models**: Responsible for combining processing modules such as `encoders`, `decoders`, `loss modules`, `optimization modules`, etc., to provide results to the post-processing unit.
- **Utils:** Includes all minor necessary components and functions that run the system such as `math_utils`, `print_utils`, `time_utils`, etc.
- **Contribs:** Includes third-party-system integration files and ensures easy integration of other systems with Ludwig by adding flags to the CLI.

10.3.3.2 Standardization of design and common processing

Ludwig's code is structured in a modular, feature-centric way to permit streamlined feature addition involving isolated code changes. In addition to this, the structured nature of the codebase is well documented and guidelines for developers can be found in the documentation.

[8] (Development View) Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.

For example, at a module level, logic pertaining to datatype logic is in the `features` module under `ludwig/features/`. Within this module, various feature classes contain pre-processing logic specific to each data type. Each of these classes in turn is required to implement either a `build_input` or `build_output` method in addition to their datatype-specific logic used for computation. So, the codebase structure from the code level to module level is consistently (pre)defined and must be adhered to. Additionally, all contributions to Ludwig are expected to meet stylistic guidelines pertaining to the codebase (i.e., code should be concise, readable, PEP8-compliant, and conforming to 80 character line length limit).

10.3.3.3 Standardization of testing

Ludwig's test coverage is limited to provided integration tests that ensure full functionality when adding new features. They can be found in the `tests/` directory and *pytest* framework is used to run the test set. To run the entire test set, a user can run `python -m pytest` from the root directory. Tests can also be run individually if necessary.

The Ludwig team has also expressed plans to expand the test coverage in the future. This way of standardized testing ensures a reliable and objective way to check contributions with minimal effort as all contributors are required to ensure that added features pass the given set of tests.

10.3.3.4 Codeline Organization

As mentioned previously, Ludwig has a standardized codebase structure to make it easy for developers to locate and modify code. Contributors to the ludwig project are mainly interested in the core `ludwig/` and integration tests `tests/` directories whose folder structures are as shown below.

```
Ludwig core structure:

root/ludwig
├── contribs (Contains classes for third-party system integrations)
├── data (Contains handlers for datasets preprocessing and postprocessing)
├── features (Contains datatype specific logic and feature classes)
├── models
│   ├── model.py (Contains the core training logic)
│   ├── modules (Contains all encoders and decoders)
└── utils

Ludwig test structure:

root/tests/
├── integration_tests (Contains required integration test set)
└── ludwig
    ├── models
    │   └── modules (Contains test encoder)
    └── utils (Contains test utils)
```

10.3.3.4.1 Other Directories

- Additionally, Ludwig's repository contains directories pertaining to documentation and examples.
- The `docs` directory contains the official Ludwig documentation. The documentation is built using *MkDocs* and created using scripts in the `MkDocs` directory.
- The `examples` directory contains example programs used to demonstrate Ludwig's API.
- Finally, the root directory contains build files (primarily, the python script `setup.py` and `Dockerfile`) and additional required documentation (such as `LICENSE` and `README.md`).

10.3.4 Distributed execution

The modular nature of Ludwig's elements allows for implementation of distributed training and prediction by emplying Horovod(another open source software developed by uber). This allows for training and prediction of models to be distributed over multiple machines with multiple GPUs.

10.3.5 Under Ludwig's Hood

To understand how the system modules interact to make Ludwig functional, we illustrate the run-time view of Ludwig in two simple steps. First, the creation and admission of a ***declarative model definition*** and next, the realization of this definition by a ***training - prediction pipeline***.

Declarative Model Definition: This schema allows users to define an instantiation of the ECD architecture (discussed below) to train on their data. This in turn leads to a separation of interests between the end users and the authors implementing these models.

There are 5 types of definition a user can declare namely *input features*, *combiner*, *output features*, *pre-processing* and *training*. Depending on these definitions, every component from the pre-processing to model to training loops in the training pipeline are dynamically built. The parameters used for these definitions can be provided by the users concisely (leading to the apllication of defaults) or in a detailed manner (leading to more user control). How we define these paremeters is illustrated in the figure below.

```
{
  input_features: [
    {
      name: title,
      type: text,
      encoder: rnn,
      cell_type: lstm,
      bidirectional: true,
      state_size: 128,
      Num_layers: 2,
      preprocessing: {
        length_limit: 20
      }
    },
    {
      name: body,
      type: text,
      encoder: stacked_cnn,
      num_filters: 128,
      num_layers: 6,
      preprocessing: {
        length_limit: 1024
      }
    }
  ],
  combiner: {
      type: concat,
      num_fc_layers: 2,
  },
  output_features: [
    {
      name: class,
      type: category
    }, {
      name: tags,
      type: set
    }
  ],
  training: {
    epochs: 100,
    learning_rate: 0.01,
    batch_size: 64,
    early_stop: 10,
    gradient_clipping: 1,
    decay_rate: 0.95,
    optimizer: {
      type: rmsprop
      beta: 0.99
    }
  }
}
```

Figure 10.1: Selection_069

Training - Prediction Pipeline : After defining the model parameters, a data-processing pipeline is implemented as illustrated below.

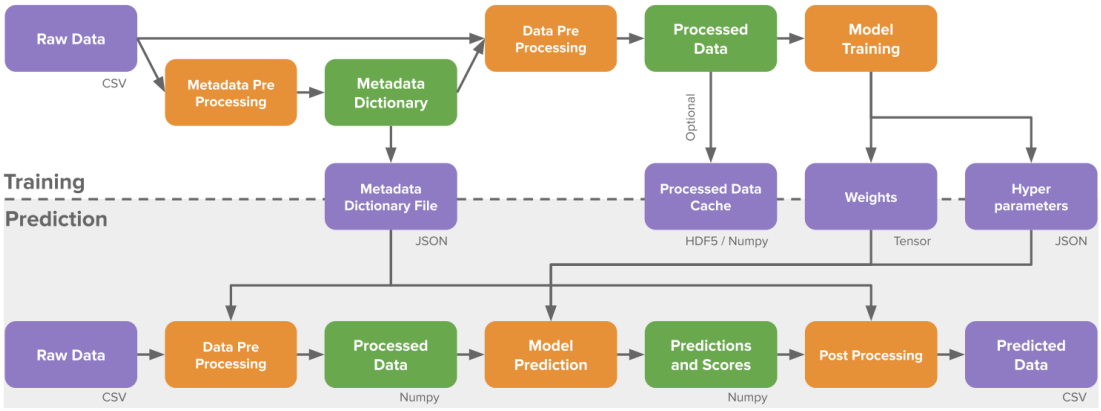


Figure 10.2: image-20200318163211997

It is clear from the figure above that the processing pipeline includes: training and prediction. In the training pipeline, during the metadata collection phase, a `Metadata dictionary` file is used in order to apply the same pre-processing to the inputs during prediction time. A similar reasoning can be provided for storing the model `weights` and `hyperparameters` obtained.

In the prediction pipeline, the metadata stored in the dictionary is implemented on new inputs and prediction is performed by defining the model through the trained `weights` and `hyperparameters`. These predictions are further post-processed into data-space by employing the same mappings obtained during training.

Implementing a training session

1) Model definitions are provided in *yaml* format. `ludwig/api.py` is called to create `LudwigModel` object.

2) Training data provided in *csv* format. `ludwig/feature` component is called and input data is built according to its feature class.

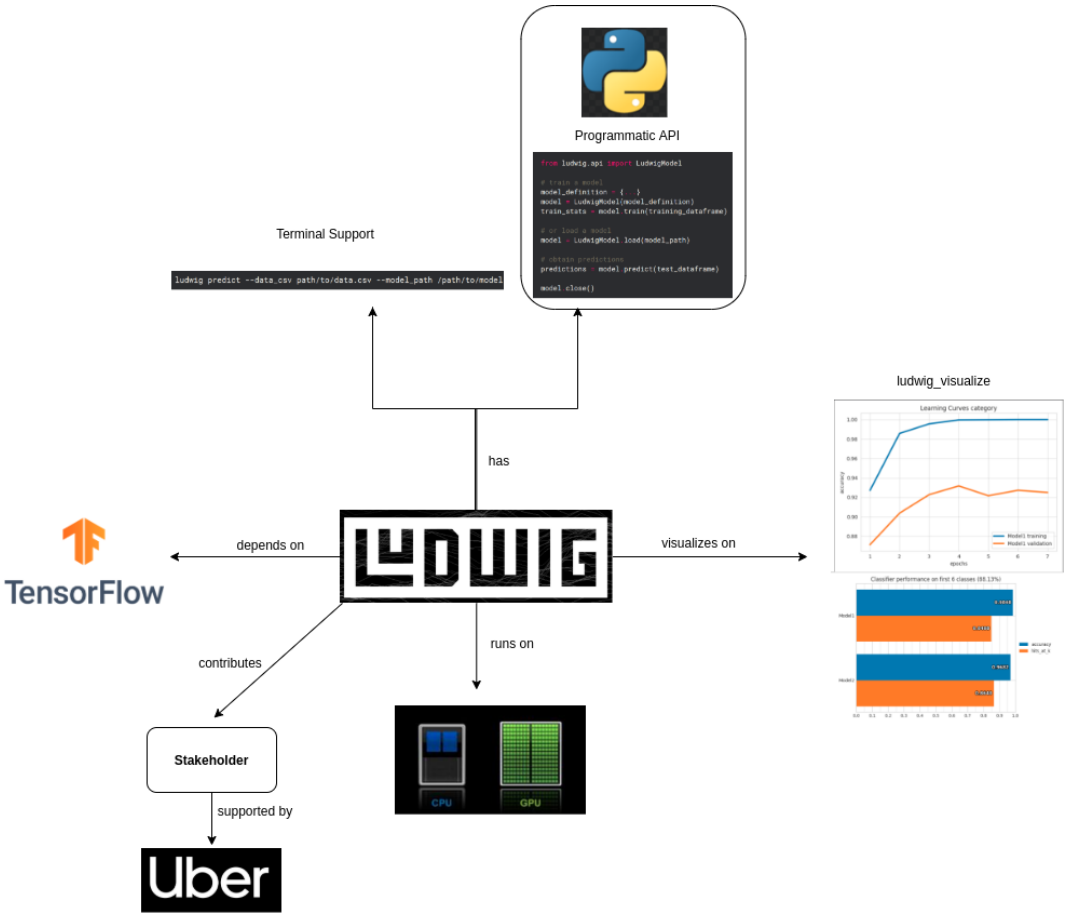
3) `ludwig/models` component initializes the Tensorflow session and Encoder-Combiner-Decoder architecture.

4) During this session, `ludwig/data` component is called to create processed data cache.

5) After finalizing the session `ludwig/feature` component is called in order to build the output.

10.3.6 Deployment and Strategies for Operation

- **Hardware and Infrastructure:** Ludwig runs on Tensorflow. Therefore, it is clear that Ludwig extends Tensorflow and uses it as a base infrastructure. Tensorflow is able to run on both CPU and GPU of users device. Consequently, the hardware requirements of Ludwig are also the same. The following Figure illustrates the Ludwig's requirements.



- **Installation and upgrade:** Ludwig's basic dependencies are as listed here and Ludwig can be installed via `pip` or from the python script `setup.py`. Additionally, extra features are divided into separate packages (installed using: `pip install ludwig[<package>]`) with separate dependencies for convenience.

- **Data Migration/Reuse:** When Ludwig trains a model it creates two files, an HDF5 and a JSON. These are later used for prediction and can also be used for retraining.

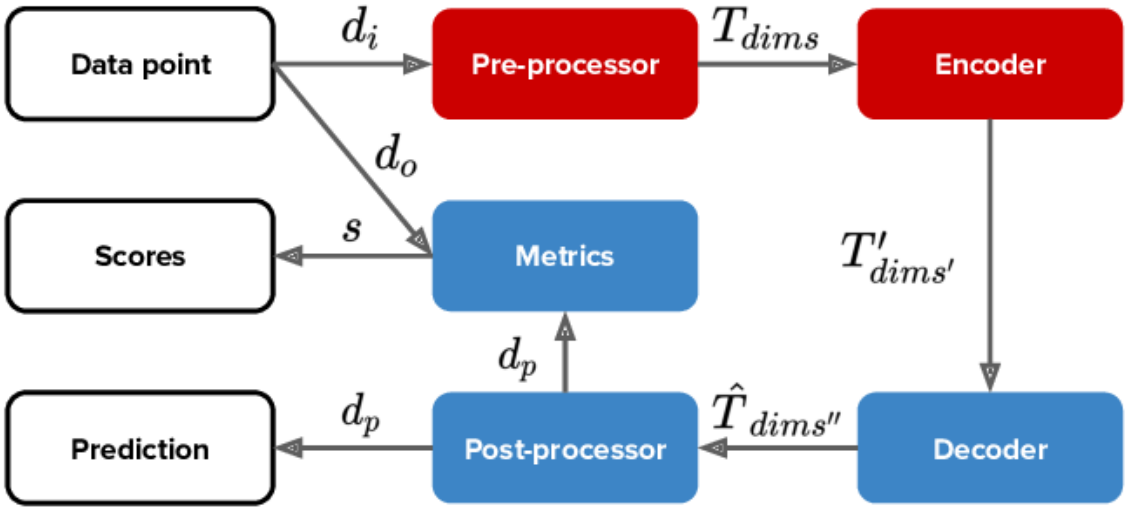
- **Operational Monitoring:** Ludwig's CLI supports multiple logging levels (critical, error, warning, info, debug, and notset(default)) to set the amount of logging displayed during training. Additionally, the `--debug` flag turns on TensorFlow's `tfdbg`.

- **Configuration Management:** Ludwig's CLI supports a wide range of configuration settings independently for each of its CLI entry points.

10.3.7 Architectural patterns

Ludwig's architectural style can be viewed from two different contexts[9]. First, a **Type-based Abstraction** style which involves modularizing the system components based on the data-types of the input features used. Second, a more generic **Encoder-Combiner-Decoder** style that maps most of the deep-learning architectures enabling modular composition.

10.3.7.1 Type-based Abstraction Architecture

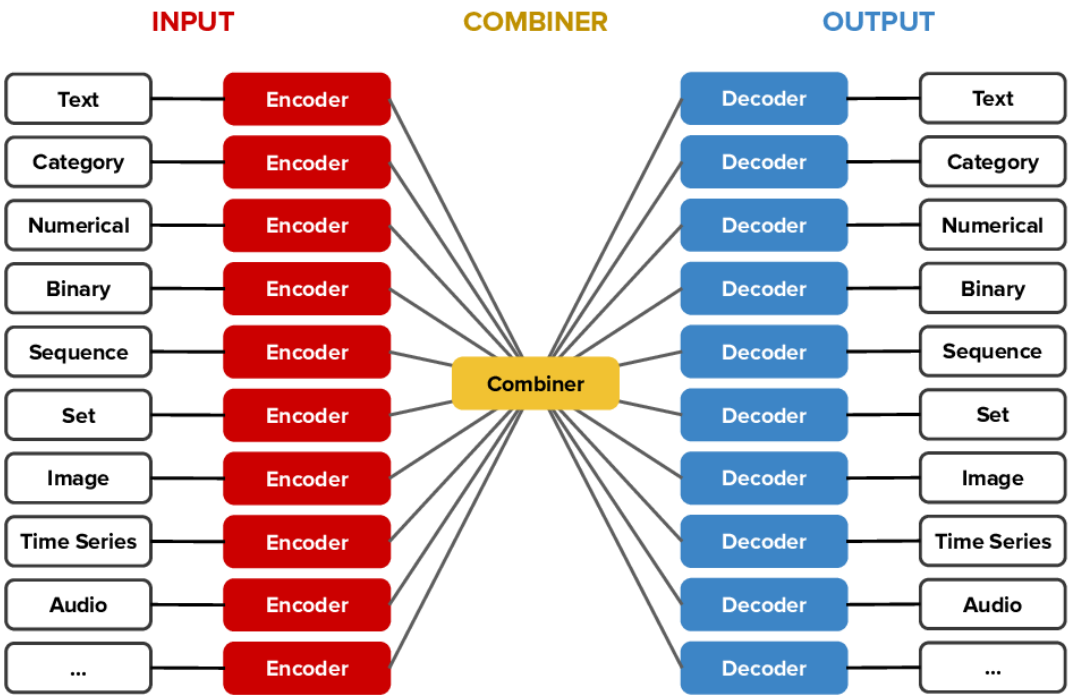


In this architectural style, each data-type is associated with five different function types as depicted in the figure above. - Pre-processor - Encoder - Decoder - Post-processor - Metrics

In order to understand this architecture, let's consider a data point input feature of *text* data-type. This input is pre-processed by tokenizers either by splitting on space or using byte-pair encoding and mapping tokens to integers and forming a tensor. This tensor is further encoded by certain encoding functions such as CNNs, bidirectional LSTMs or Transformers and the output tensor moves on to the decoder or combiner (to be discussed in the next architectural style). After application of the decoding function, the output tensor is post-processed by either mapping integer predictions into tokens and concatenating on space or using byte-pair concatenation to obtain a single string of *text*. Finally, a metric function (loss functions) produces a score based on the ground truth and post-processed data.

[9](Molino, Piero & Dudin, Yaroslav & Miryala, Sai. (2019). Ludwig: a type-based declarative deep learning toolbox.)

10.3.7.2 Encoders-Combiner-Decoders (ECD) Architecture



This architectural style in combination with the *type-based abstraction* is used to define models by just declaring the data types of inputs and output features of a task and assembling suitable standard sub-modules rather than writing the entire model from scratch. Having discussed the functionality of encoders and decoders previously, combiners take up the role as a processing unit (by applying combiner functions) between the two. Combiners such as *concat* perform the function of flattenning and concatenating input tensors into a stack of fully connected layers.

ECD architecture can take up different instantiations by combining input features of different data types with output features of different data-types. For instance, an ECD with an input image feature and text output feature can be trained to perform image captioning where as an ECD with an input text feature and output categorical feature can be trained to perform text classification or sentiment analysis.

10.3.8 Non-functional Properties

Ludwig's non-functional properties include:

- Ease of operation - Well written user manuals coupled with support for all major operating systems, ludwig adopts a minimal coding method - making it simple for anyone to work and debug.

- Acceptable inputs and outputs - Ludwig supports most input and output formats (images, text, etc.).
- Encourage contributions - Ludwig has an abundance of community development, though final decisions are resolved by Uber. Since Ludwig is currently under active development, addition of features and debugging existing features are highly encouraged as is evident from the lightning quick responses to queries and pull requests.

- Integration with external software - Code templates and guidelines have been laid out by Uber regarding integration of software with Ludwig.

To wrap up everything, in this essay we looked at the architecture of Ludwig from different perspectives and how it's various modules are drawn together to make it functional.

10.4 Ludwig's Code Quality and Tests

Studying software architecture is a fantastic way to understand the planning behind a system and how it operates. In our previous posts, we illustrated key aspects of Ludwig's architecture from various perspectives. Now, with this post, we move beyond the building of the system and on to its maintenance and upkeep.

In our analysis of Ludwig's architecture, we briefly touched upon Ludwig's handling of issues such as quality control and standardization. Now, we aim to explore the underlying code that forms the system with a focus on the safeguards and guidelines that uphold the quality and integrity of the system.

10.4.1 Quality Guidelines and Feature Integration

With Ludwig being a young, open-source project, the core development team has put considerable thought and effort into streamlining Ludwig's development process. As a result, Ludwig's documentation is both methodical and thorough - catering to the open-source developer community, offering detailed information about codebase structure as well as code standards/guidelines for feature implementation. This facilitates contributions while ensuring that any contributions to Ludwig must meet these strict standards.

For example, as documented, Ludwig takes a standardized, feature-centric approach to codebase organization from top to bottom. This means that features and corresponding logic(s) that make up certain features follow a standardized implementation. In short, Ludwig's guidelines define features and the (minimum required) methods to implement these features. This feature-centric nature promotes uncomplicated addition of features to the system.

Additionally, components of Ludwig's system ((e.g.) Encoders, Decoders, training logic, etc.) are modularized so that they can be used by multiple features while operating independently. This modular structure also ensures that additions to the codebase only require isolated code changes, thereby insulating the system against failures.

10.4.2 Ludwig's CI Process

Software testing evaluates a system with the intent to find out if it satisfies a set of specified requirements. This is usually done by (either automated or manually) executing the system under controlled conditions and evaluating the outcome. The most efficient way of early-problem-detection in systems is by verifying code check-ins during the CI (Continuous Integration) stage by automating builds with various tests. In Ludwig, the CI process involves two checks which take up around 10 minutes in total to execute.

10.4.2.1 DeepSource's Python Analyzer

The first check that runs in Ludwig's CI pipeline and is implemented over the entire system. DeepSource is a code analysis tool that helps in managing bugs, syntax, performance, anti-pattern issues, etc. present in code. From our observations of Ludwig's pull requests, this check generally lasts from 10 seconds to 2 minutes (depending on the number of issues *resolved* and *introduced* after a commit) and produces its

analysis report which includes detailed insight into: - Security Issues - Performance Issues - Bug Risks - Coverage Issues - Anti-Patterns - Typecheck Issues

Finally, this check is passed only if no blocking issues exist for the commit.

10.4.2.2 Travis CI Build

This check, hosted by Travis CI, involves building the software and performing a set of developed tests. The check's parameters (including programming language, version, build and test environments, etc.) are specified in a `.travis.yml` file in the root directory. Ludwig's CI performs this test twice i.e., in `Travis CI-Pull Request` where the tests are performed in automerge state and in `continuous-integration/travis-ci/pr` where the tests are performed in the current state of the branch that the commits were pushed to.

10.4.2.3 Provided Test Coverage

Ludwig's core developers have provided a set of coverage tests for use by contributors. These tests primarily test interactions between Ludwig's modules to expose possible defects in their interfaces or interactions. For instance, in `test_experiment.py`, one of the test modules, consider the method `test_experiment_seq_seq(csv_filename)` (source). This method tests its functionality by running an experiment with *cnnrnn/stacked_cnn* encoders and *tagger* decoder defined in its input and output feature respectively. The success of this test would indicate the proper functioning of the encoder and decoder modules as well as their interactions.

Several such integration tests are defined that ensure end-to-end functionality. However, the Ludwig team is planning to incorporate more tests to the test suite in coming releases.

10.4.3 Test Quality Assessment

Ludwig currently only supports a set of integration tests in its test suite. The rationale behind this is that with Ludwig being a relatively young project, its current state/structure is expected to undergo alterations in the coming releases and a more restrictive test suite containing extensive unit tests would only hamper the speed of development of the project[10]. However, the core development team has expressed an interest[11] in gradually expanding the test suite and currently do so regularly by adding integration tests ((e.g.) The latest release v0.2.2 added an integration test for SavedModel).

10.4.4 Code Quality Assessment

> A study on "Software Defect Origins and Removal Methods" found that individual programmers are less than 50% efficient at finding bugs in their own software. And most forms of testing are only 35% efficient

(source)

This illustrates how difficult it is to write and maintain *good code* simultaneously. Therefore, it is important to take certain key aspects into consideration while measuring code quality as a part of the inspection process. In order to determine the relevancy of these aspects, we ran Ludwig's code (as of 25th March, 2020) in sonarcloud, a code-analysis tool.

[10] Developer rationale behind existing test suite
[11] Developer interest in expanding the test suite

The figure below illustrates the overview of the key aspects obtained from this analysis.

Now, let us discuss these aspects and examine the corresponding results of our analysis.

10.4.4.1 Reliability

This aspect defines how trustworthy a system is. From our analysis, Ludwig scores a "D" in it's reliability aspect. This is clearly due to its *3003* bugs as denoted in the report. On scrutiny, we found that these bugs are mainly present in documentation modules `docs/.` and `mkdocs/.` and are mostly due to styling mis-alignments, duplications, etc. in HTML and CSS files. As critical sections such as (`ludwig/.`) report fewer bugs, the system can be considered to be more reliabile than the report suggests.

10.4.4.2 Security

An important aspect that curbs vulnerabilities in a system. Ludwig scores an "A" as it does not have any vulnerabilities. However, the report presents 106 security hotspots, indicating codes in those sections have to be reviewed to ensure proper security before changes to the code make them vulnerable. However, further discussion on this issue is beyond the scope of this essay.

10.4.4.3 Maintainability

In the event of system failure, the effort that is required in order to restore the system's functionality is determined by its maintainability. As discussed earlier, Ludwig follows a standardized, feature-centric approach to it's code-base organization due to which its maintainability becomes systematic and the analysis report with a maitainability score "A" aligns well with this fact. On deeper examination, we see that there is a debt ratio of 0.2% and an estimated 13 days of technical debt. Further, out of the *1199* code smells, 80% are present in documentation modules `docs/.` and `mkdocs/.` and the remaining in the source and test codes. The figure below illustrates Ludwig's technical debt. [**Note:** *Bubble size indicates the volume of code smell. And each bubble represents a separate file*]

Our analysis of the files containing code smells indicate that most of these code smells are present due to:

- The number of parameters of a function are greater than the 7 authorized by SonarCloud
- Cognitive complexity of a function is higher than the 15 allowed
- Duplicated code
- The presence of unused variables

10.4.4.4 Duplications

This aspect measures code repetition which is not considered to be beneficial due to its several demerits. Ludwig presents a duplication of 24.5% out of which 7% is present in its source code. This is one of the most important aspects that requires code refactorization.

Although, most of the issues discussed in all the four aspects do not impact Ludwig's functionality, however, they need to be addressed sooner or later in order to avoid any future damage that might arise due to accumulation of these issues.

Now that we have gained some idea about Ludwig's code quality, let's take a look at some on-going developments in its code.

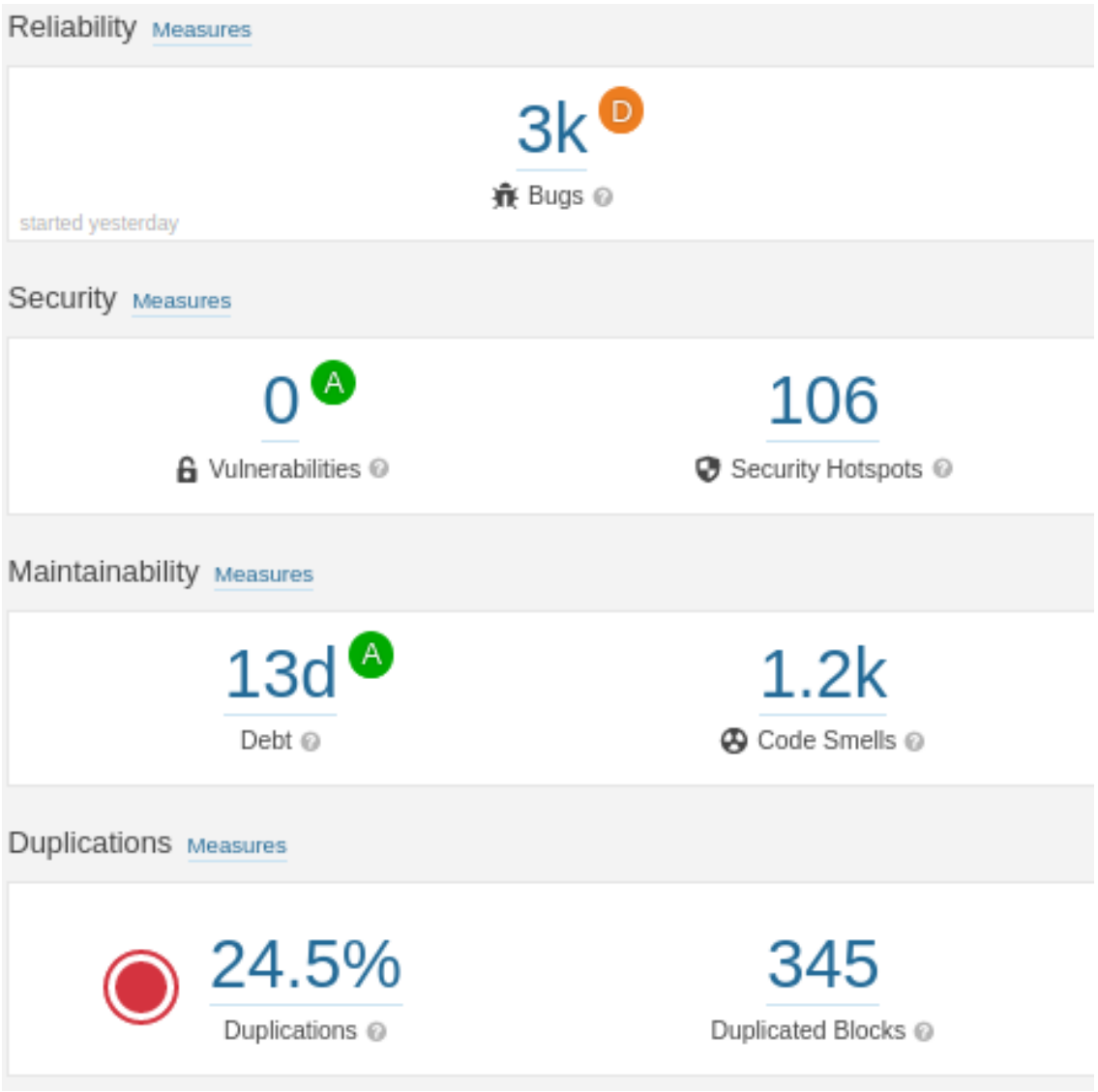


Figure 10.3: Overview

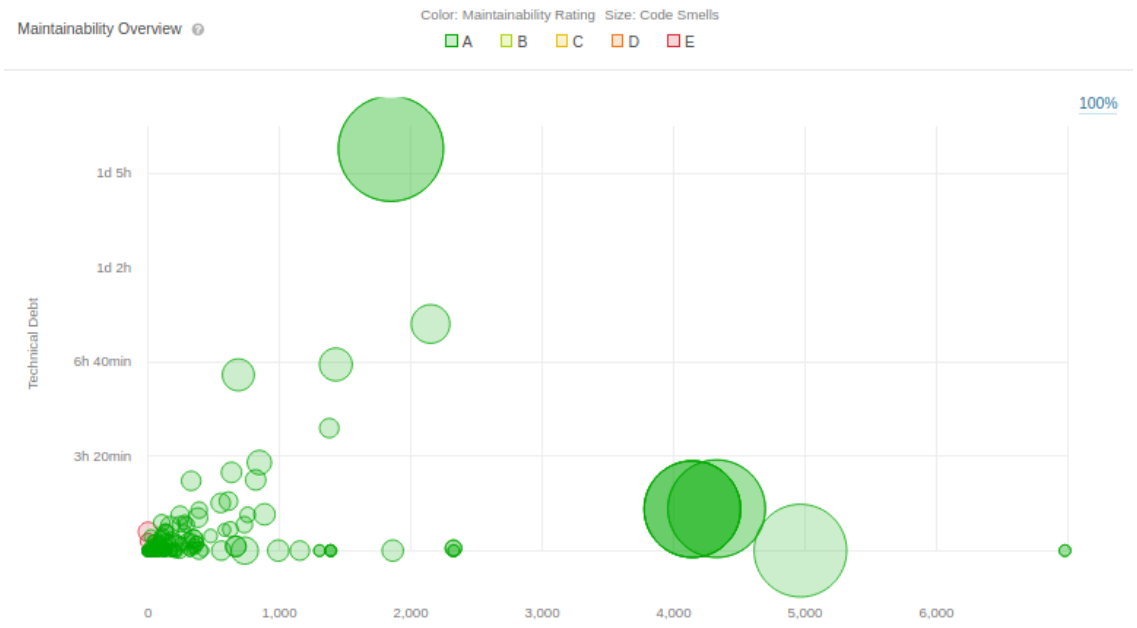


Figure 10.4: Tech_Debt

10.4.5 Ludwig's Hotspots

In order to maintain the quality of the overall system code-base, it is essential to assess the quality of those sections of code or modules which are frequently modified, also known as hotspots. These hotspots present the highest risk of breaching at least one of the quality aspects discussed above which would finally add up to technical debt.

To catalog hotspots, we check the latest commits of the coming release. This will give us an overview of code hotspots in terms of lines of code and as a distribution of commits over the components.

To distinguish these hotspots, we put Ludwig's repo under analysis in CodeScene. The figure below illustrates this. [***Note:*** **The most frequently modified files are the most red ones**].

Now, let us take a look at the quality aspects of some of these hotspots in order of their frequency of modification.

File Name	LOC	Bugs	Security Hotspots	Code Smells	Duplications
ludwig/api.py	636	0	9	2	2.4%
ludwig/train.py	554	0	8	3	5.3%
ludwig/experiment.py	613	0	7	3	22.4%
ludwig/visualize.py	2151	0	28	3	4.6%
ludwig/data/preprocessing.py	848	0	13	1	0.0%
tests/integration_tests/test_experiment.py	420	0	0	0	0.0%

Though most of these modules have duplications as well as code smells, this amount can be fairly assumed to be on the safe side. The code smells in all these modules can be explained with similar reasonings as provided earlier. However, these issues should be addressed as early as possible before they pile up further adding to technical debt.

We can further investigate commits in the latest release for hotspots in Ludwig.

Inspecting the Latest Commits

Ludwig, being a young platforms, has over a 1000 commits. Studying the distribution of these commits to Ludwig's modules is done by sorting the commits with respect module name using the following command:

```
git log --name-only --pretty=format: | sort | uniq -c | sort -nr | grep "name of component" | awk '{ SU
```

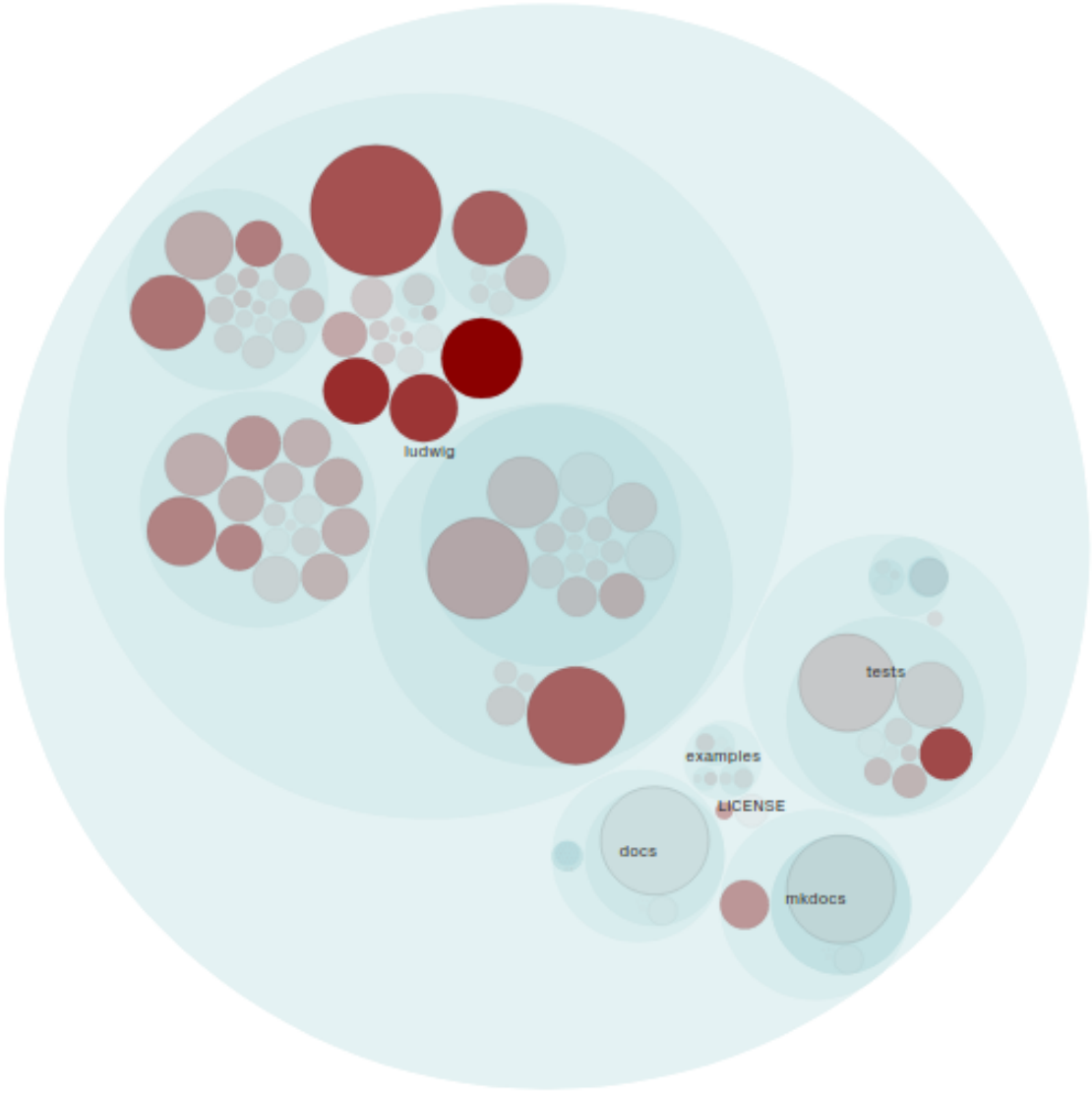


Figure 10.5: Hotspots

are numberous commits involving these improvements and bug fixes due to these changes in the code. Additionally, the documentation is also being updated to keep pace with the additions to the system.

10.4.6 Ludwig's Roadmap

From our analysis above, it is clear that most frequent changes are made to Ludwig's core module. This supports our expectation as the core module houses Ludwig's architectural building blocks which are constantly undergoing change. In accordance to the roadmap stated in our first post, the following modifications are in progress: * Adding new encoders and decoders - New encoders and decoders are added in models directory. #206 #588 * Time series decoding - Should be added in models directory. #244 * Adding new feature types (point clouds etc.) - New feature files are added in features directory. #449 * Pre-processing support #217

Having a look at the merged pull requests on github tells us that some of the planned features like adding audio data type [12], BERT encoder [13] have recently been merged and additional work includes porting to Tf2 [14] and improving Horovod [15] (another open source software for serving) integration with Ludwig. On looking at the pull requests, we see that most pull requests include changes in the code to add more featues and the improvements addressed earlier by uber during the previous release of Ludwig. Since ludwig is a relatively new software, as addressed in our previous essay, it still has a lot of limitations and requires additional functionalities. The blogpost release by uber team during every major release contains the possible additions in their future releases.

10.5 Variability Analysis of Ludwig

Software variability is the ability of a software system to be personalized, customized, or configured to suit a user's needs while still supporting desired properties required for mass production or widespread usage. In the current age of the Internet and Technology, software systems are all-pervasive. Thus, for any software to be effective in today's market, portability, flexibility, and extensibility are more important than ever before. Therefore, software variability is a crucial aspect of any software system that must be addressed within its structure[16].

The idea behind designing for software variability is to put in extra effort during the development stage to provide a base (usually, within the core code) that allows the end-user to generate customizable products with minimal effort. The sources of variability in software are numerous - right from supporting different hardware or software platforms, to functional or even merely executional variability.

To study the variability of a software, we view it as a product line of sorts[17]. This product line consists of many products formed from a single software system that differ from each other in the features that they implement and the platforms they support (platform and architectural variability). Additionally, certain features in these products may have dependencies or implementations that restrict or necessitate combinations of the features with others (variability in installation). Further, the products may differ in user accessibility (variability in interfaces). Now, we aim to study these sources of variability in Ludwig and their implementations in the repository.

[12] PR #396 - Changes for audio feature input
[13] PR #409 - Added Bert Encoder
[14] PR #609 - TF2-porting
[15] PR #12 - Initial implementation of Horovod integration
[16] Software Variability - WISE
[17] Mastering Software Variability with Feature IDE, Authors: Meinicke, J., Thüm, Th., Schröter, R., Benduhn, F., Leich, Th., Saake, G.

10.5.1 Search for Variability

10.5.1.1 In Hardware and Platform support

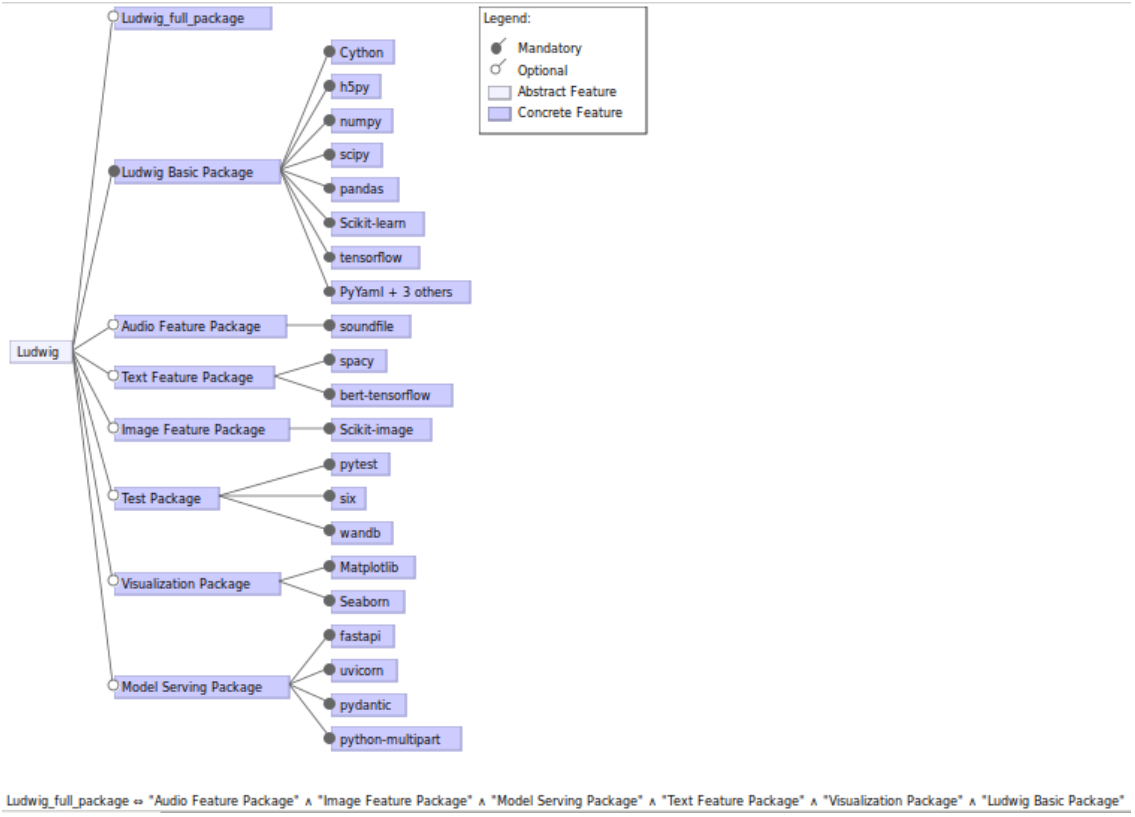
As discussed in our second article, Ludwig runs on top of TensorFlow, using it as base infrastructure. Hence, Ludwig's system requirements match those of TensorFlow and by extension, its variability in hardware/platform support. Thus, Ludwig supports Ubuntu, MacOS, Windows, and Raspbian platforms and for GPU support, CUDA-enabled cards in Windows and Ubuntu are among its hardware requirements.

10.5.1.2 In customized installation

In keeping with its themes of flexibility and ease-of-use, Ludwig also offers some degree of variability in installation. The dependencies of Ludwig's functions are divided into multiple Python packages to afford the end-user freedom to pick and choose packages to cater to their needs. Ludwig consists of a (mandatory) base package and (optional) feature packages.

The base package accounts for dependencies for Ludwig's basic functions (training models and prediction). Other features such as result visualization, serving on HTTP servers, additional datatype support (such as images, audio, and text), and tests have their dependencies conveniently packaged into individual/separate Python packages.

The following feature-model diagram gives an overview of the installation packages and their customizations available in Ludwig. For simplicity, only 8 dependencies for the Ludwig-base package is illustrated.



10.5.1.2.1 Implementation Taking a look at the SetupTools `setup.py` script, we see that Ludwig implements this variability by separately parsing each set dependencies while creating the Ludwig Python package. The script lists the base dependencies as mandatory requirements for the Ludwig package while each of the optional package's dependencies are listed as extra dependencies. So, this allows installation of base Ludwig package using `pip install ludwig`. Additionally, the optional feature dependencies can be installed using pip either individually, or grouped together or in full (i.e., `pip install ludwig[viz]` installs the visualization support; `pip install ludwig[text,viz]` installs both the visualization and the text feature packages; `pip install ludwig[full]` installs the full set of dependencies).

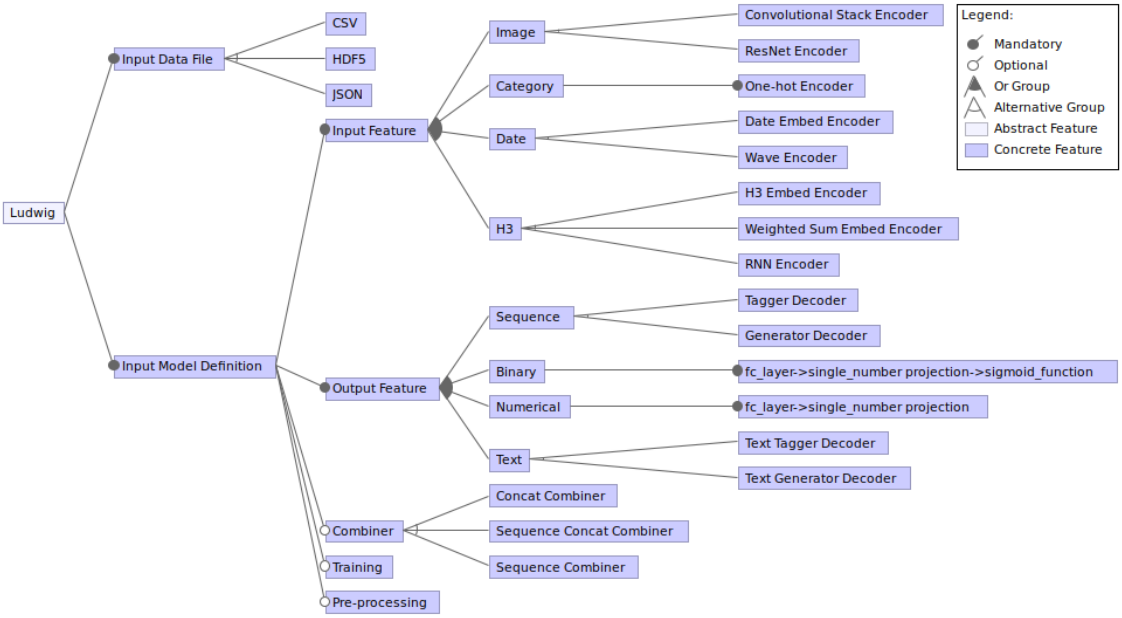
10.5.1.3 In the architecture

From our previous article on Ludwig's *Encoder-Combiner-Decoder (ECD) architecture* and *Input model-definition file* functionality, it can be inferred that variability plays an important role in Ludwig's architecture. A considerable number of feature-types are currently supported in this architecture. Further, each feature-type is presented with one or more encoder(/decoder) types. For instance, *Sequence Features* can support 8 possible encoder types: *Embed, Parallel-CNN, Stacked-CNN, Stacked-Parallel-CNN, RNN, CNN-RNN, BERT* and *Passthrough Encoders* and can be decoded using either *Tagger* or *Generator decoders*. However, the choie of encoders afford more variability as most feature-types do not support more than one decoder option. In fact, some feature-types do not support any decoder yet and hence cannot be used as an output feature, for instance, audio-features. In the case of combiners, three different types are currently implemented in Ludwig: Concat, Sequence Concat and Sequence Combiners which are not necessarily feature-type specific.

Adding to the variability within the architectural elements, the `model definition YAML file` extends it by providing parameters for each of these elements and the training pipeline as listed below:

- **Input Features:** Eight different parameters, namely: *name, type, encoder, cell_type, bidirectional, state_size, num_layers*.
- **Combiners:** Two different parameters, namely: *type* and *layer size of fully connected layers*.
- **Training:** Model hyper-parameter definitions, for example: *number of iterations, learning rate, batch size, optimizer* and *regulators* for their model.
- **Pre-processing:** Feature-specific data pre-processing parameter definitions
- **Output Features:** Similar structure as *Input Features* with *decoders, loss types*, etc.

To clearly understand this variability, let us take a look at the feature-model diagram below. It can be observed that Ludwig basically requires just two input parameters: an input data file to train and a model-definition file to configure the training process. For the sake of simplicity, only 4 feature types per input(output) features are illustrated here. Also only elements of the ECD architecture are expanded further to match the scope of this section.



10.5.1.3.1 Implementation Considering the large number of variabilities in the ECD architecture, one can imagine how difficult it can get to structure the entire code-base specific to each feature-type. Moreover, the process of adding new features would also become cumbersome. Hence, to simplify the process, Ludwig uses a *template-based design pattern* which enables modularity in the code-base design. In this implementation, three different classes: the BaseFeature Class (that deals with the pre-processing of input feature-data), the InputFeature Class (that builds the inputs with corresponding encoders for the model to train) and the OutputFeature Class (that builds the output model with decoders and implements post-processing functionalities) are defined as base classes in `base_feature.py` file. A parsing and mapping of the respective features from the model-definition file to the specific feature module is done in the `feature_registries.py` file. Further, each feature module defines sub-classes of the respective base classes overriding the abstract methods with feature specific ones. And, to implement encoder(decoder) types, mapping registries are used which map the types with corresponding encoder(decoder) modules defined in the modules directory.

10.5.1.4 In the Command Line Interfaces

By this point, it should be clear that Ludwig does not support any Graphical User Interface. Hence, the only method to access Ludwig is through its CLI. To ensure uniformity in access across all operating systems, Ludwig's CLI consists of six different entry point commands:

- **train:** To train a model on the input file specified. No variability within this command exists, which in turn avoids any misunderstanding in its usage. However, different flags are provided and can be optionally used according to the users needs (e.g., *`-kf` for k-fold cross validation, `-uh` for horovod, `-gf` for gpu usage*, etc.)
- **predict:** To predict the results of a previously trained model. Similar to training, multiple flags for prediction are also provided.

- **test:** To predict on a pre-trained model and analyse the prediction performance with respect to a ground truth.
- **experiment:** To run a full experiment comprising of training a model and then testing it.
- **visualize:** To analyse the results for the model on the data-set and to present a variety of plots to understand and evaluate the results. These plots can range from simple learning curves to confusion matrix plots.
- **collect_weights:** To collect the weights for a pre-trained model as a tensor representation.
- **collect_activations:** To collect tensor representations corresponding to each data-point.
- **serve:** To load a pre-trained model and serve it on an *http* server.

The detailed explanation with parameters and usage guidelines of the CLI can be found in its documentation. For all of these commands, variability is presented through the use of different flags. Hence, by shifting the majority of the variability to the underlying architectural domain and maintaining a simple user entry point, Ludwig aims to broaden its consumer base by catering to a wider range of end-users.

10.5.1.4.1 Implementation Taking a look at the code-base implementing the CLI optionalities in `cli.py` file, we found out that the code flow was pretty straight-forward. In this implementation, a CLI class defines all the entry points as methods. A *dispatcher design pattern* is used to invoke a method if the input command matches the corresponding method name. Further, each method on getting invoked, creates the corresponding command object and uses the *"cli"* method to access its functionality along with the input flags. The variability of these input flags are implemented in the target files for each command. For instance, on running the *training command*, ludwig checks all the flags and runs the pre-processing modules (with necessary flag parameters) followed by the training modules. Further, to add a new flag, one can simply add a system argument and modify the module in which the flag would be used. Ludwig also provides an open-ended method through which one can create their own module and add a flag in the `cli.py` file to run the module.

10.5.2 Managing Variability in Relation to Stakeholders

So now that we have explored the many facets of variability in Ludwig, we conclude by attempting to discuss the impact of variability on its stakeholders (primarily the end users and developers) and glance at how Ludwig tries to manage this impact using documentation and information sources.

It is important to keep in mind that Ludwig's end users can range from a novice to an experienced Deep Learning enthusiast. Hence, Ludwig provides different levels of variability depending on the level of expertise of the end-user. For instance, to a novice end-user, Ludwig does not really offer much variability. The user only needs to provide a *CSV* input file and define suitable input and output feature-types in the model-definition file. All the other parameters use their default values/types to train the model. On the other-hand, an experienced user has the opportunity to explore a much wider variety of options available at an architectural as well as processing-level in Ludwig. This includes encoder(/decoder)-types, hyper-parameters, loss types, etc. To provide the user with a better understanding of these variations, a well-documented user guide is maintained by Ludwig's core development team. Moreover, considering the variations in the installation packages, a well-documented installation-guide is also provided that lists all the necessary dependencies and extra packages required to train specific feature types.

Developers can be considered as those experienced deep-learning enthusiasts who play a role in Ludwig's design/development process in addition to their role as end-users. To manage Ludwig's growing open-source developer community, there are uniform guidelines to be followed to contribute to its development via Github. These guidelines ensure that feature additions or implementations follow a specific standard

(refering to how they are called or the way their parameters are defined in the created feature class). This results in an improved cognitive complexity analysed in our previous post.

So, this study of variability in Ludwig lends credence to our previous assesments of Ludwig and show that Ludwig has been designed with extensive variability in mind, both in terms of architecture/implementation as well as in accessibility.

Chapter 11

Material UI

Material-UI is an open source user-interface (UI) library for React components. The vision of Material-UI is to facilitate developers with a UI library for general use, consisting of components that allow developers from different backgrounds to have an easier and faster experience of creating React web or mobile applications.

Material-UI was inspired by Google's Material Design guidelines, originally released in 2014. It contains components ranging from simple ones like a button to more complex components such as the tree view and modal dialogs. Each component has its own page on the documentation website, where some common use cases for that component are highlighted. The source code for these demos is also included in the documentation page. Furthermore, the complete API specification for each component is also documented.

The project is rather active on GitHub. Issues are created frequently, and several pull requests are opened every day. The authors of these pull requests range from the project collaborators, to frequent contributors, to developers who have not contributed to the project before. Having amassed nearly 55,000 stars at the time of writing, and the core package having over a million downloads per week on NPM, we can say that this project is popular and used in many projects.

In this chapter we will take a closer look at the architecture and implementation of Material-UI. We will examine the product vision, the architectural decisions that were made (including decomposition and tradeoff points), the quality control and assessment mechanisms, and finally a deeper analysis based on the course lectures or other relevant material of choice specific to Material-UI.

11.1 About the authors

The authors of the documents pertaining to Material UI are (sorted by last name): Wesley Baartman, Kevin Chong, Paul van der Stel and Erik Wiegel. After having finished our undergraduate programme at Delft University of Technology, we are now taking this software architecture course as part of our graduate programme.

As we use and develop software on a daily basis, we recognise that software projects can have both positive and negative qualities. Naturally, we like to see or create the positive qualities, whereas we prefer to avoid the negative qualities. Although good a software project can be characterised by many different qualities, we will list the positive qualities we find most important below.

- Code is clear and/or self-explanatory. If code is hard to read, more time is required to fully understand the code and its behavior. Code consistency (formatting, code style) is also important here, as inconsistencies make code harder to read.
- Good documentation is available. Tutorials, public API documentation and other guides are an invaluable resource to anyone who consumes the project. Good documentation also requires a good navigation structure, so that relevant information is easy to find.
- Low technical debt. Having lots of technical debt means that it is harder to develop new features in the future. To minimise technical debt, the software project must be well tested, should be kept up to date (dependencies, issues) and temporary solutions should be avoided, since nothing is as permanent as a temporary solution.
- The project should build and run out-of-the-box with either standard build tools or a provided build script. If something is not working properly, the maintainers should be reachable for assistance.
- Permissive licencing. Having a good licence is important, so that outsiders are aware of their rights and duties regarding the software project. We usually prefer open-source solutions over proprietary solutions, if other circumstances are the same. Although viral licences can be beneficial to the world of open-source software, it might mean that the software will not be adopted widely in proprietary solutions. Therefore, having the right licence for a project is very important.

11.2 Materialising Material-UI

In this first post, we will establish a basis for understanding the different aspects of Material-UI to facilitate for more in-depth analyses in future posts. We will do this by taking a look at the purpose of the project, the expectation of the end-users, the stakeholders, the competitors, the key capabilities, the product context, and finally the roadmap.

11.2.1 Purpose

Material-UI aims to be a general purpose UI library and tries to achieve this goal by providing customizable and composable React components[1]. The project's goal is to become both an implementation of **Google**'s Material Design guidelines and provide other useful components not previously defined by **Material Design**, while still following the same principles[2]. The Material Design guidelines define a multitude of different standards for UI design, including a set of common building blocks, called components[3], theming[4] and accessibility[5].

Besides providing the aforementioned UI library, the developers also want to "promote developer joy, a sense of community, and an environment where new and experienced developers can learn from each other"[6]. This can be emphasized by their contribution guidelines[7] and their incredible number of contributors (1668[8] at the time of writing) in the mere 6 years that the repository exists on GitHub. This number is a lot higher than that of the arguably more popular JavaScript library, jQuery, with "only" 277[9] contributors at the time

[1] https://material-ui.com/discover-more/vision/
[2] https://material-ui.com/discover-more/vision/
[3] https://material.io/components/
[4] https://material.io/design/material-theming/implementing-your-theme.html
[5] https://material.io/design/usability/accessibility.html
[6] https://material-ui.com/discover-more/vision/
[7] https://github.com/mui-org/material-ui/blob/master/CONTRIBUTING.md
[8] https://github.com/mui-org/material-ui/graphs/contributors
[9] https://github.com/jquery/jquery/graphs/contributors

of writing, which exists since 2006[10]. Although this only compares two libraries, both are well-known and it shows that Material-UI has a relatively high engagement rate with its users.

11.2.2 End-user mental model

To describe the mental model of the Material-UI's end users, we first need to establish who these end-users are. We have considered two end-users of Material-UI: a front-end developer and the end-user of the application Material-UI is used in.

First, the front-end developer. Since Material-UI is a library consisting of React components, the developer should have all the expectations they normally would have for React components. Thus, for each unique UI component, the public interface consists of declaring that a certain component should be rendered, along with some *props* that specify options for that component[11]. Indeed, Material-UI uses this pattern for their components. It also requires the developer to use a `ThemeProvider` component, which enables the styling of the components[12]. This pattern is also common in React [13]. Thus, we can conclude that the project is aligned with the expectations of developers.

Secondly, we have the users of the applications built with Material-UI. Most users have certain expectations when it comes to the user interface of applications. Buttons should be clickable for example, and tabs[14] should take users to different screens. Most of the user experience research has already been done by Google as they developed the (now widespread) Material design system, and Material-UI aims to implement these guidelines. The project furthermore makes its components compatible with assistive technologies. Thus, the project is also aligned with the expectations of end users.

11.2.3 Key capabilities

As described in the previous section, Material-UI provides developers with easy-to use components that can be used out-of-the box, or can be customized to fit requirements imposed on the project. These components are arguably the main draw for many developers, as these pre-made implementations of components make development easier, especially when the components are complex.

These components are not everything that the Material-UI project offers, however. Although the project comes with a default style that implements Material design, developers would like to use their own custom styling for the components in many cases. The project facilitates this by implementing a theming solution [15]. Components will automatically use the styles that are defined using the `ThemeProvider` component. Since this styling solution is independent of the core component package, third parties can choose to only use this styling package.

Furthermore, Material-UI has another package which contains the icons that are used in Material design. Installing this dependency allows for usage of any of these icons. In addition to the core package, there is also a *lab* package, containing components that are still in beta, and may still have an unstable API. Once these components are well-tested, they are migrated to the core package. There are also some utility packages, such as one containing auxiliary TypeScript types, and another containing codemods, which help with automatically updating to newer package versions.

[10] https://github.com/jquery/jquery/graphs/contributors
[11] https://reactjs.org/docs/components-and-props.html
[12] https://material-ui.com/customization/theming/
[13] https://reactjs.org/docs/context.html
[14] https://material.io/components/tabs
[15] https://material-ui.com/customization/theming/

11.2.4 Stakeholders

The stakeholders that are involved with Material-UI can be classified as follows[16]:

- End users: Both developers and users of the products that use Material-UI can be classified as the end users. In the case of the developers who are working on the front-end, the skills of these users can vary vastly. It can range from users who are just starting to learn the ropes and are creating small prototypes to professional developers who are working within a company on larger projects. These users are interested in the capabilities of Material-UI when it comes to improving their work experience, while the other group of users is interested in the capabilities of the UI when interacting with the application Material-UI is used in. Both groups of users are important to consider, as they have different priorities when interacting with the system.
- The business: While this is an open source project created by a relatively small team, the people on this core team also take on the role of the business in the stakeholder analysis.
- The team is paid through donations/sponsorships with services like Patreon and Open Collective. This means that they as a team will need to take care of using the funds properly as well as getting more users.
- Customers: These are the people who support Material-UI's development, either by investing financially or time wise by trying out new features.
- Domain experts: These are the UX designers within the team as well as any contributors that are involved with the project.
- Developers and testers: The developers include both the core team of Material-UI itself, as well as contributors to the project on GitHub. In Material-UI each MR with changes to the code base needs to be accompanied by tests, thus the developer and the tester are essentially the same group.

11.2.5 Competitors

Material-UI is not the only option when looking for a library for front-end components, several other libraries have been listed by Material-UI themselves to highlight the pros and cons of their library against its competitors[17]. The competitors listed there include:

- Material Components Web and its "predecessor" Material Design Lite
- Materialize
- React Toolbox

These alternatives tend to be less actively maintained compared to Material-UI, as well as being less popular on GitHub (i.e. less stars and downloads/month). A more comparable competitor would be Ant Design.

11.2.6 Product context

For any project, the context in which the project is created and exists in is a major influence on decisions made for the architecture. We will consider this from two viewpoints, the business and technological context[18].

Material-UI is sponsored by various backers through Patreon and OpenCollective, by both individuals and companies, which allows the funding of the core development team. Besides the paid core development

[16] Coplien, J. O., & Bjørnvig, G. (2011). Lean architecture: for agile software development. John Wiley & Sons.
[17] https://v3.material-ui.com/getting-started/comparison/
[18] https://docs.arc42.org/section-3/

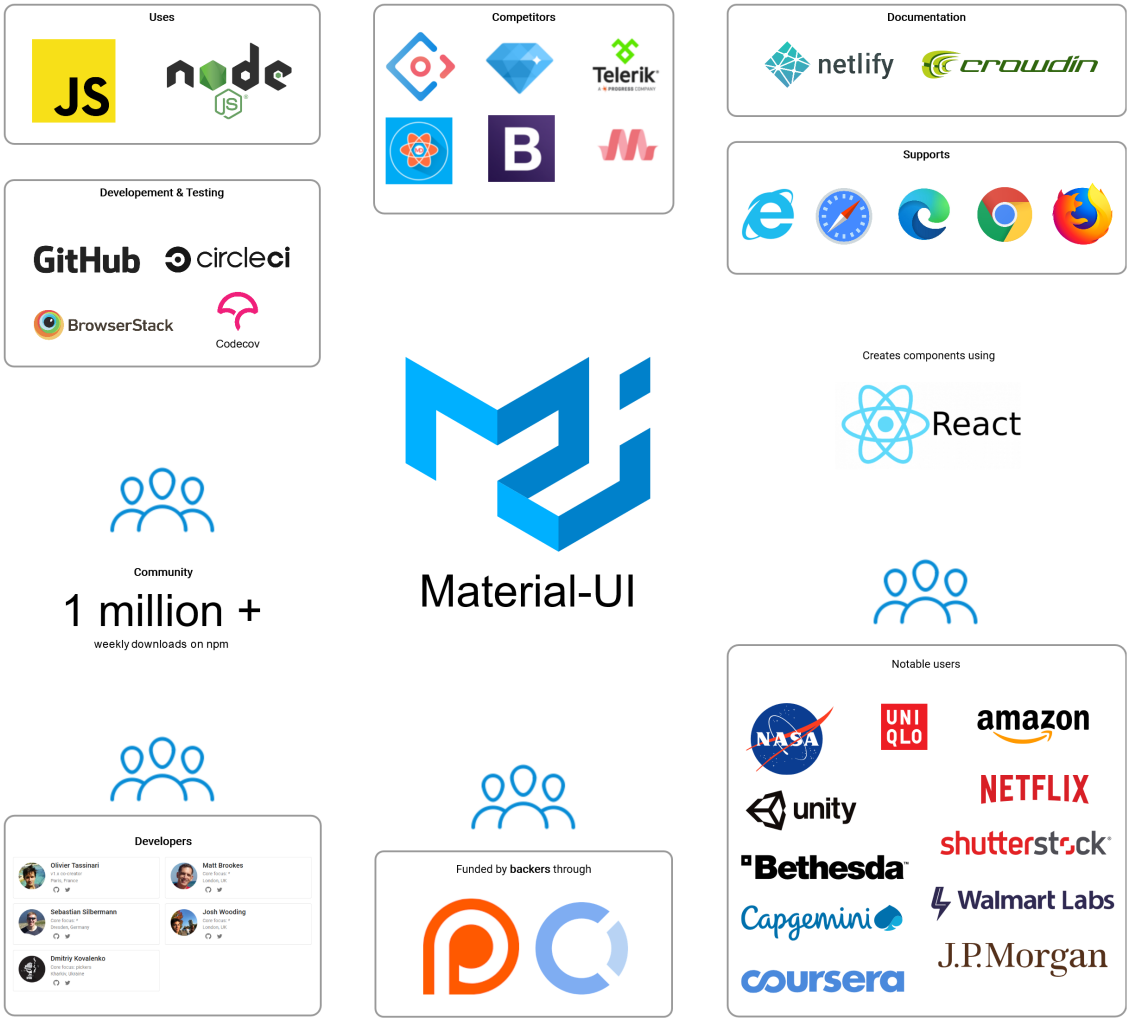


Figure 11.1: Material-UI Context

team individuals also help extend and improve material-UI. The project is used by a large number of users including big companies such as Amazon and Netflix.

Material-UI is hosted on Github and written in JavaScript. Although TypeScript support is mostly there, there are no plans to migrate to TypeScript in the immediate future. The project uses the tools CircleCI, BrowserStack and CodeCov to help with testing the project. For the documentation, Netlify and Crowdin are used. Node.js is supported for server-side rendering.

At the moment Material-UI supports the latest updates of the major browsers and Internet Explorer 11. The current plan is to drop support for IE11 when the number of users will drop beneath 10%[19].

At the moment there is a stable state of being for Material-UI where development is focused on improvement and expansion but no major changes are planned in the short term.

11.2.7 Roadmap

The Material-UI roadmap[20] states the current priorities for improvement and expansion of the project. While the priorities have been scored for importance, the deciding factor is the community's wishes. Even the roadmap encourages readers to express their preferences by adding a thumbs-up to the corresponding issues. In line with this philosophy, the roadmap often links to these specific issues.

The current top priority is identifying and constructing frequently needed components. The core team does, however, want to encourage the use of third party components that already exist and are well maintained. Additionally, they do want to offer developers the option to move their components to the official organization.

The community aspect is again highlighted in the priority to improve the documentation. The developers have decided to integrate a rating system for the documentation to collect data-points about the pages needing the most improvement.

A list of planned components and other priorities is available on the roadmap.

11.3 Dissecting Material-UI

In this second part of our series we will consider the overall architecture of Material-UI in-depth. First, we will consider what viewpoints can be used for an in-depth look. Secondly we will consider the main style of the architecture and then we will use that information to consider some viewpoints. Finally we consider the non-functional properties of Material-UI.

11.3.1 Viewpoints

We will consider possible interesting architectural viewpoints as described in Rozanski and Woods[21]. In the previous post we already have considered the context, the rest we will shortly evaluate for their usefulness in relation to Material-UI

- **Functional** - This is relevant as it is the cornerstone for any project.

[19] https://github.com/mui-org/material-ui/issues/14420#issuecomment-584887038
[20] https://material-ui.com/discover-more/roadmap/
[21] https://www.viewpoints-and-perspectives.info/home/viewpoints/

- **Information** - The project provides no functionality intended to store information and therefore we don't need to consider this viewpoint.
- **Concurrency** - React is currently working on a concurrent mode, and in turn Material-UI developers have worked on preparing compatibility with this new feature. The main coordination and control of this concurrency however is provided through React, and it is therefore more useful to consider this as an additional constraint or requirement for Material UI. We conclude that this viewpoint is not fitting for Material-UI.
- **Development** - We have already seen part of this in the previous post in the context when we saw tools such as GitHub.
  This viewpoint is especially interesting as there is a great focus on community as seen previously in the roadmap.
- **Deployment** - Material-UI does have some technical requirements to use the full functionality, for example the need for Node.js to allow server side rendering, and therefore this view is interesting to consider.
- **Operational** - As Material-UI is essentially a library of components there is no operational support required besides fixing eventual bugs, which can be covered in the development view as maintenance.

11.3.2 Architectural Style

Material-UI's architecture can be characterised best by the layered architectural style[22]. The various UI components extend a core component that provides some base functionality. While this might seem like a very simple architecture, getting such a set-up to function properly is complex and forces many constrainsts on the form of the components.

The goal of the developers is to provide low-level components to maximize composition capabilities[23]. The API of these components are designed to be as similar to one another as they can be, making the library more consistent. Some components are exceptions to this however. As some components use composition to re-use smaller components, their APIs may be inconsistent to simplify their interface, or to improve performance.

11.3.3 Development View

The Material-UI project consists of multiple different components. We shall first give a brief description of each component before giving an overview of how the components depend on each other.

- core: This component provides all React components[24] and is the primary component that end-users will interact with.
- styles: This component provides Material-UI's styling which can used independently from the core component if users want to[25].
- system: This component provides so called *style functions* which can be used to create design systems[26].
- types: This component provides utility types for Material-UI[27].

[22] https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html
[23] https://material-ui.com/guides/api/
[24] https://github.com/mui-org/material-ui/tree/master/packages/material-ui
[25] https://material-ui.com/styles/basics/
[26] https://material-ui.com/system/basics/
[27] https://github.com/mui-org/material-ui/blob/master/packages/material-ui-types/package.json

- utils: This component provides utility functions for Material-UI[28].
- icons: This component provides Google's Material icons as React components which can be used independently of the core package[29].
- lab: This component contains React components that are currently under development[30].
- docs: This component contains documentation building blocks[31].
- codemod: This component is a tool which can help with automatically updating Material-UI's API usages[32].
- eslint-plugin: This component contains numerous ESLint rules specifically for Material-UI[33].
- babel-plugin: This component is a tool used to automatically converts Material-UI's TypeScript demos to JavaScript demos[34].

We will now present a visual overview of how the different components depend on each other:

If a component has an arrow pointing towards another component it means that it depends on the component to which the arrow is pointing. The colour of the component indicates whether a component is meant for direct consumption (blue), not meant for direct consumption (orange), meant for adding support to external tools (green) or meant to help develop or document Material-UI (purple).

11.3.4 Run Time View

At run time, each component can be loaded and used. Depending on the build system of the software project that is using Material-UI, it can either be loaded as one big bundle of components, or it can be loaded as part of the built software project bundle. As many modern React projects are built using a so-called *module bundler* such as webpack or rollup.js, this is the recommended approach for using Material-UI.

It is typically not possible to interact with the provided components directly. In virtually all cases, a developer declares that a component must be rendered, and React will take care of managing the component behind the scenes[35]. Developers can specify *props* that influence how a component behaves or looks[36]. Most of these components are self-contained, and thus fit in nicely with the idiomatic component structure that React recommends.

All of the components can be composed together with the developer's own components to present the user with a complete user interface. Material-UI does this as well, as some components use other components. For example, the `IconButton` component makes an appearance in several other components. There are no circular dependencies however; the architecture of the library remains layered.

Such composed interfaces can be as complex or simple as needed, since React handles most rendering use cases. Examples can be seen on Material-UI's Showcases documentation page[37]. When user interfaces become very large however, the performance of the web page is affected. Such issues can be solved by optimizing component use, or using UI virtualization.

[28] https://github.com/mui-org/material-ui/blob/master/packages/material-ui-utils/package.json
[29] https://github.com/mui-org/material-ui/tree/master/packages/material-ui-icons
[30] https://github.com/mui-org/material-ui/tree/master/packages/material-ui-lab
[31] https://github.com/mui-org/material-ui/tree/master/packages/material-ui-docs
[32] https://github.com/mui-org/material-ui/tree/master/packages/material-ui-codemod
[33] https://github.com/mui-org/material-ui/tree/master/packages/eslint-plugin-material-ui
[34] https://github.com/mui-org/material-ui/tree/master/packages/babel-plugin-unwrap-createstyles
[35] https://reactjs.org/docs/state-and-lifecycle.html
[36] https://reactjs.org/docs/components-and-props.html
[37] https://material-ui.com/discover-more/showcase/

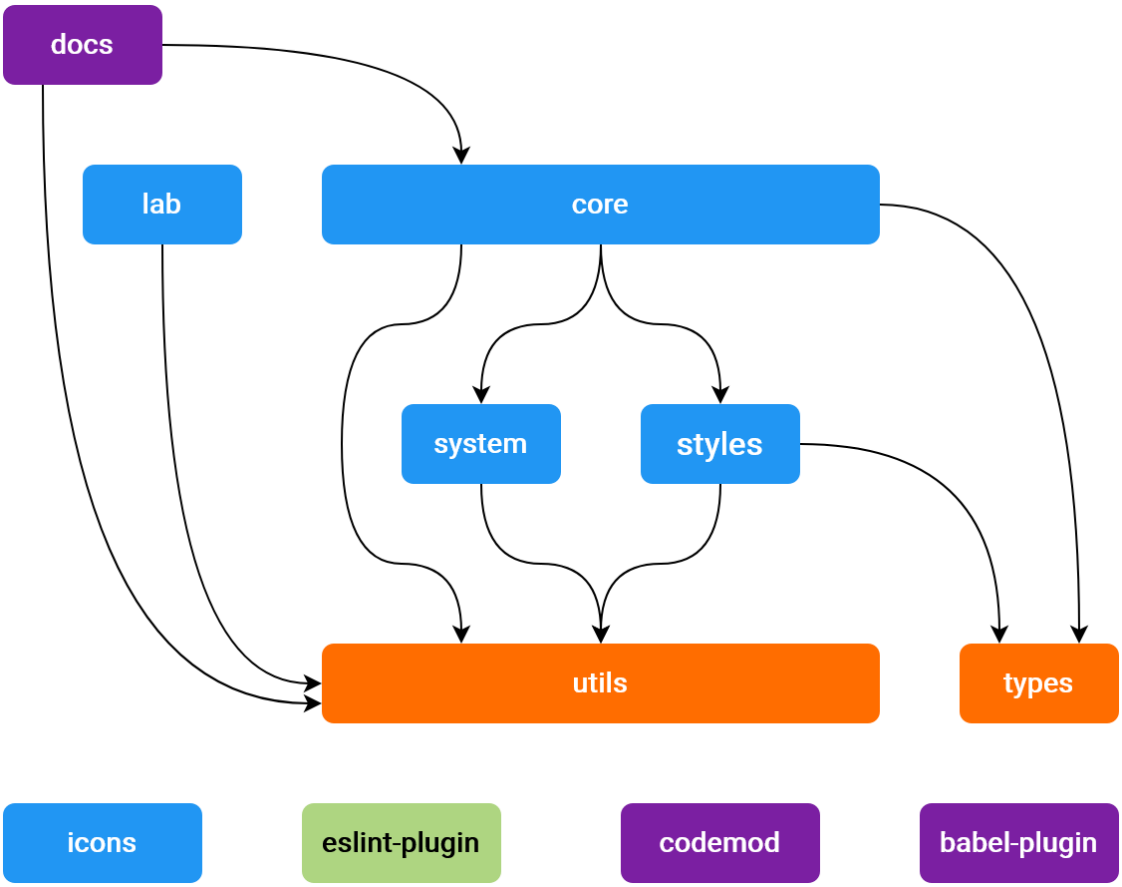


Figure 11.2: Material-UI Components

11.3.5 Deployment View

Because Material-UI is a library, it is not necessary to deploy many different services. However, every pull request to the GitHub repository does trigger a check that automatically deploys a preview of the website using Netlify[38]. Likewise, every merged pull request deploys the updated documentation pages to Material-UI's production website, again using Netlify and GitHub workflows[39].

Furthermore, the actual library itself is published on npm as multiple different packages. Each of the different components corresponds to one of the npm packages[40][41][42][43][44][45][46][47][48][49]. Each npm package is manually and seperately deployed, as can be seen by looking at the difference in release dates between the packages and between the latest merged pull request altering said components.

11.3.6 Non functional trade-offs

There is no free lunch in computer science, which also holds for any software project. Some trade-offs can be found in Material-UI when looking at the non-functional properties that are relevant for this project. One of such property is the number of components. Material-UI would like to include all the components that are commonly used by developers. However, the downside of expanding their library is that the size of the bundle will increase. To mitigate this trade-off the developers of Material-UI have spent their time looking into ways to reduce the bundle size and has set this as one of their priorities. While version 4 had more components in it, its size was reduced by 18% compared to version 3[50].

Material-UI puts a lot of emphasis on having the components work in isolation and they want their API to be as low-level as possible[51]. This means that sometimes their decision will be to include fewer features, rather than including features that do not work in isolation, are not performant, are not easily customizable, or are not accessible. In these cases they would encourage the user to build these extensions on top of the library instead.

11.4 Qualifying Material-UI

The third part of our series will dive into the code to assess the quality of the software. First, we will look at the processes used to guarantee quality in the project and check the quality of the tests used for Material-UI. Then we will connect the roadmap originally described in part 1 of this series to the code.

Besides looking at the code, we will also look at the importance of quality, testing and technical debt in discussions on the Github and the documentation. For this project, we also used various tools to consider the code quality and maintability of the project. Finally, we will evaluate the technical debt in the system.

[38] https://github.com/mui-org/material-ui/pull/20076/checks
[39] https://github.com/mui-org/material-ui/actions/runs/55958265/workflow
[40] https://www.npmjs.com/package/@material-ui/core
[41] https://www.npmjs.com/package/@material-ui/styles
[42] https://www.npmjs.com/package/@material-ui/system
[43] https://www.npmjs.com/package/@material-ui/types
[44] https://www.npmjs.com/package/@material-ui/utils
[45] https://www.npmjs.com/package/@material-ui/icons
[46] https://www.npmjs.com/package/@material-ui/lab
[47] https://www.npmjs.com/package/@material-ui/docs
[48] https://www.npmjs.com/package/@material-ui/codemod
[49] https://www.npmjs.com/package/eslint-plugin-material-ui
[50] https://material-ui.com/discover-more/roadmap/#priorities
[51] https://survivejs.com/blog/material-ui-interview/

11.4.1 Software Quality Assurance Process

To find out in what way quality is maintained in a community project, the best approach is to look at what is required of contributors and how their contributions are checked. This is described in the contribution guide[52] of Material-UI. This guide encourages contributors to create an issue to discuss large changes with the maintainers before working on it. Here we can already get a glance that the paid developers play a critical role in guaranteeing the quality of the project. Creating issues allows them to give some input before someone would even start working on a problem.

The guide continues with the advice to keep pull requests small and a guide on how to properly do a fork and push. While that does not directly improve quality, small and properly pushed pull requests are easier to review. This is relevant because the core team is monitoring the pull requests. They review a pull request and either merge it, request changes to it, or close it with an explanation. We finally see the scope of the role of the core team: they actively review all pull requests and in this way can maintain a high quality of the software project.

The section in the contribution guide finishes with a list of requirements for a merge, which we will limit to some relevant examples. If a feature is added that already can be replicated using the core components it's addition should be justified. In terms of software quality, they want to avoid unnecessary code for features that can already be composed of the existing components. When functionality is introduced or modified, additional tests to confirm the intended behaviour are required. The coding style should be followed, which includes using prettier and linting. Finally, it should succeed all automated checks, which we elaborate on in the following section.

11.4.2 Automated tests and continuous integration

When a change to an open source project is proposed in the form of a pull request, the impact of such a change needs to be assessed. In the case of Material-UI, several aspects of the project are especially important to check for when a contribution is made. Some of them are obvious: bugs that were previously solved should not be introduced again, and the behavior of components should usually not change in such a way that it becomes a breaking change.

There are also aspects that are not as obvious. Examples of these are *visual testing*, which verifies that the appearance of components does not change, which is important for a UI library such as Material-UI. Another is a *bundle size check*, which determines by how much the total size of the build artifacts changes, as these are served to users who use the application built with Material-UI.

Fortunately, all of these inspections have been automated, taking the burden of checking these away from the core development team. Every pull request is checked using several services that automatically build and/or test the project at the state it would be in after the proposed patch is applied.

Many of the required checks are run on CircleCI. These include tests such as unit tests that verify whether components respond appropriately to some props being set to certain values, among other things. There are also specific tests that verify that the TypeScript types match with the JavaScript components, and that the website documentation files are up-to-date with the latest version of the comments in the code, which are used to generate documentation. Finally, CircleCI runs regression tests and in-browser tests.

There are still more checks. Material-UI also utilizes Argos CI, which renders the components, takes a screenshot of the components, and compares these images to previous renders. Any visual difference causes

[52] https://github.com/mui-org/material-ui/blob/master/CONTRIBUTING.md

the check to fail, and a project member must approve the changes manually. Another build is executed on Azure Pipelines. This check will compile the project and the documentation, and figure out how the bundle size has changed. The output is automatically posted as a comment on the associated pull request, making it easily accessible. The last service to mention is Netlify, which automatically deploys the documentation website. It includes review domains, which means that anyone can open the documentation website that was specifically built for any given pull request, to see if it is up to standards.

11.4.2.1 Quality of testing

With all these services, it is easy to assume that Material-UI is well-tested. But to verify that, we need some test coverage tools to gather this information. Luckily, the Material-UI developers know this all too well and have set up the appropriate coverage tools that interface with the JavaScript testing frameworks in use. As the unit, integration and browser tests are run, Istanbul collects the coverage data and outputs it to an HTML report.

We have executed the tests with coverage for the most recent commit on `master` at the time of writing (`6d62842d6f53c1726fc688131af0a16d561e3bd1`). At the end of the test run, nearly 2800 tests have been executed, of which none have failed. The coverage report indicates the following results:

Coverage type	Percentage
Statements	96.6%
Branches	87.58%
Functions	97.79%
Lines	96.58%

These numbers indicate that the code coverage is extensive. One must still be careful with drawing conclusions that the project is well-tested, but this is a good sign. Most files that pull down the percentages are often either very complex, which is not great, but they are still relatively well-tested, or they are so-called index files that include other files. Note that these scores only reflect the stable components and code, as components that are considered *alpha/beta* are not included in the coverage.

Despite the high testing scores, the `bug` label is still a frequent occurrence when browsing the issues list. Typically, such issues are quickly resolved and added as a regression test, which means that the test suite is continuously evolving and improving. After all, Dijkstra's quote applies to this software project as well: "Program testing can be used to show the presence of bugs, but never to show their absence!"[53]

11.4.3 Roadmap to code

As mentioned previously in the first post, main priority right now is to introduce more components[54]. Generally, these new components are first introduced into the the _@material-ui/lab_ package[55] as mentioned in last week's post. By scanning through the merged pull requests, one finds that most changes are actually made to the documentation found in the *docs*. While most other changes, such as bug fixes, performance enhancements, etc. can primarily be found inside the @*material-ui/core* package.

[53] https://en.wikiquote.org/wiki/Edsger_W._Dijkstra
[54] https://material-ui.com/discover-more/roadmap/
[55] https://github.com/mui-org/material-ui/tree/master/packages/material-ui-lab

11.4.4 The importance of testing, technical debt and code quality in discussions & documentation

One way Material-UI avoids technical debt, lack of tests and low quality is the lab package[56]. A component in the lab is allowed to make frequent breaking changes, this way new components can more easily avoid technical debt that would be introduced by avoiding breaking changes. Consumers of Material-UI are warned about these breaking changes before they start using the package.

There is a strict requirement for tests and code quality before a component can be moved from the lab to the core package. This ensures the core components have a standard of high-quality and good test coverage. Moreover, there is the requirement that in the short or medium term future, there are no expected breaking changes for the component. This avoids introducing unnecessary technical debt or breaking changes in the core package.

In many pull requests and issues one will see that the core developers focus on various code quality aspects. See for example original pull request for the Stepper component or the newer Alert component. Additionally, a lot of discussions about the best design decisions take place, with some issues focusing solely design decisions: Stop IE11 support. New tests are often asked for when making certain bug fixes or additions (example. For changes that do not change any functionality, the only requirement is ususally that the existing tests still pass on the new implementation.

11.4.5 Refactoring candidates based on code quality and maintainability assessment

To evaluate the code quality and maintainability we used both SonarQube and Sigrid provided by SIG. SonarQube was overwhelmingly positive about the project's maintainability by giving all components a rating of A. SIG on the other hand, had some concerns regarding the unit complexity and unit size of several components. As the unit size scored on average 1.6 out of 5 stars, while unit complexity on average scored 2.3 out 5 stars.

According to the metrics computed by SIG, the most important risk factors for the maintanability of Material-UI are the unit size (based on LOC) as well as the unit complexity (based on the McCabe complexity). Interestingly, it was found that if a function was a high risk factor for one of the two metrics, it would usually also be a high risk factor for the other metric. Most of these risk factors are due to the arguments given to React.forwardRef, `React.forwardRef` takes a rendering function as argument. However, this rendering function can be insanely large. For example, in Material-UI for the slider this rendering function starts at line 336 and ends at line 779. While this is the largest example in Material-UI, more similar cases can be found in Material-UI. Note that this is not an issue with Material-UI, as this is how React functional components work. In fact Material-UI has made an effort to split up the large rendering functions into smaller functions inside.

A minor refactoring suggestion from SIG was related to duplication of code. Material-UI is a library of components, some components are closely related to others such as Input and FilledInput. Currently, in these components there is a lot of duplications both in defining the styles used as well as the functional properties of the component. In the example of Input and FilledInput, other than the different names used for some variables, the main difference lies in the styles of the two components. Both components make use of InputBase, this already reduces the possible duplication by a large margin, but it could be further decreased.

[56] https://material-ui.com/components/about-the-lab/

11.4.6 Assessment of technical debt

To measure technical debt across the project, we will use SonarQube which has support for JavaScript and React[57]. Since Material-UI themselves do not provide a SonarQube configuration, we will use the standard JavaScript and React configuration for our technical debt analysis.

Using SonarQube we found the following technical debts for the different components of the Material-UI project at the 25th of March 2020:

Component	Technical Debt
core	1d5h
styles	4h
system	5m
types	N/A
utils	20m
icons	0m
lab	2h35m
docs	0m
codemod	1d6h
eslint-plugin	0m
babel-plugin	0m
Total	2d18h

It must be noted that it was not possible to retrieve the technical debt for the *types* component because the component does not contain source code. As can be seen, the two components with the most technical debt are the *core* and the *codemod* component. As noted earlier, the *codemod* component is not really under active development, thus we feel comfortable disregarding the amount of amassed technical debt in said component.

11.4.7 SonarQube detected bugs

Aside from measuring technical debt in software projects, SonarQube is also capable of detecting possible sources of bugs. Using this feature we found two instances of code that could lead to potential bugs in the future. We decided to fix both of these instances and create pull requests for them, which were swiftly merged by the core team.

The first instance was an if-statement that could never be false, thus we simply removed the entire if-statement. The second instance was a function that did not take any arguments being called with arguments from a different location. JavaScript allows this and just disregards the passed argument. Neither of these cases actually caused any errorous behaviour, but we deemed that they might cause confusion and thus, potential bugs, in the future.

[57] https://docs.sonarqube.org/latest/analysis/languages/javascript/

11.5 Varying Material-UI

This is the final part of our four part series on Material-UI. In this post we look at the variability in the project, how this variability is managed and how it is implemented. This is all done to show the range of possibilities in Material-UI. We will then finish with our assessment on how well the variability has been implemented in this project.

11.5.1 Features with variability

We identified the following main variable features for Material-UI:

- Supported browsers. This feature benefits the final end-user the most since it allows them to use their favourite browser to access websites created with Material-UI.
- Themes. This allows developers to quickly use a given theme without having to pay much attention to styling individual components.
- Colour palette[58], allowing developers to alter the colour schemes of their chosen or custom themes.
- Dark mode[59]. With this feature developers can easily create a dark mode for their users, which in turn also benefits the end-users.
- Typography (font, font size, variations)[60], providing ways to alter the way text is rendered.
- CSS overrides[61]. These allow developers to inject CSS overrides for styles.
- Default props[62], which specify the default props of components. Developers can change these to modify default behavior.
- Component variation[63]. This feature allows developers to create variants of existing components with special styling.
- Style using CSS or JSS[64]. With this functionality developers can choose to either use CSS or JSS for styling.
- Material-UI Icons. This optional package allows developers to use a pre-made set of icons that match the Material Design style.

11.5.2 Incompatibilities

Since most of the different variability features in Material-UI are related to spacing, we mainly find different ways of achieving the same result that could cause conflict. The main way this is resolved is by giving precedence to the different ways of varying. A program built using Material-UI may for instance use a custom theme (which already overrides the default theme) which overrides a certain CSS attribute for every style injected by Material-UI, which may override a certain default property override in a theme[65]. Every component can in turn be given its own styling and when an instance of said component is created, it can be given even more styling overriding all previous set styles[66].

All other features that are not directly linked to styling do not interfere with each other and thus can be used with little regard to the configuration of other features.

[58] https://material-ui.com/customization/palette/
[59] https://material-ui.com/customization/palette/
[60] https://material-ui.com/customization/typography/
[61] https://material-ui.com/customization/globals/
[62] https://material-ui.com/customization/globals/
[63] https://material-ui.com/customization/components/
[64] https://material-ui.com/styles/advanced/
[65] https://material-ui.com/customization/globals/
[66] https://material-ui.com/customization/components/

11.5.3 Feature model

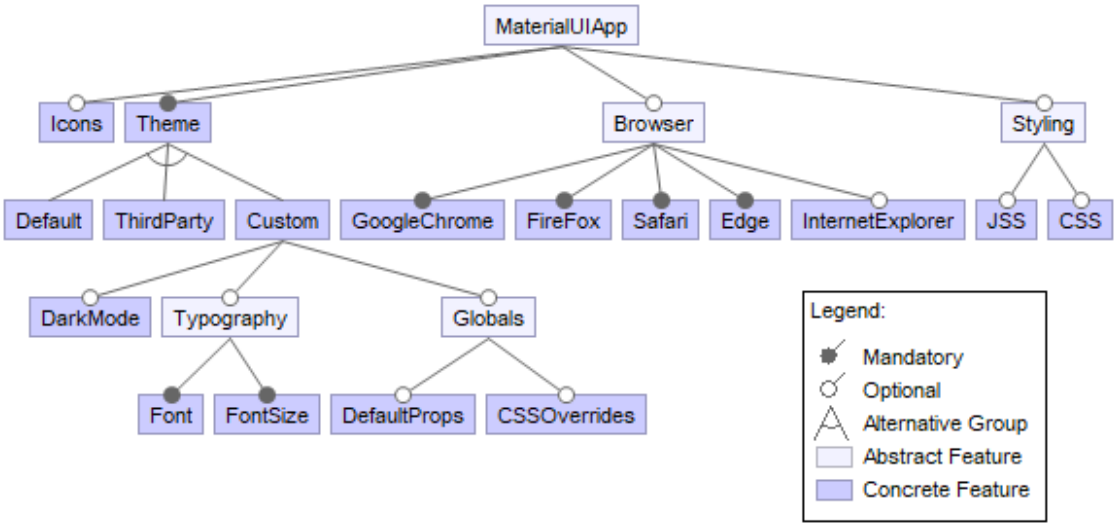


Figure 11.3: Material-UI Features

11.5.4 Variability management for different stakeholders

When analyzing the variability management for different stakeholders, we will focus on the end-users and the developers. We consider the end-users to be the people who use Material-UI in their own projects, and the developers are the people working on Material-UI. The other groups of stakeholders will generally have less direct interactions with the variability of Material-UI.

Material-UI provides the following sources of information to both the end-users as well as the developers:

- The official documentation of Material-UI. Information on how to set up the project to your likings can be found here, as well as more information on how to customize the components that Material-UI offers.
- Several example projects that can be used as references are included in the repository[67].
- Each release of Material-UI will include more information on the changes, as well as a reference to the pull requests that were relevant for the changes. This way both the PR itself as well the summary of the changes can be used as a source of information.
- If the users would run into any issues for which they cannot find an answer in the provided documentation, they can ask the question on Stack Overflow with the tag `[material-ui]`.

11.5.5 Mechanisms to ease variability management

Material-UI has two mechanisms in use that do not directly target variability management, but do ease the process a bit. These are present in the form of checks on pull requests.

[67] https://material-ui.com/getting-started/example-projects/

- CodeSandbox. Each version of a PR is deployed to CodeSandbox which allows the developers to showcase the changes and test the functionality. This enables the reviewers to check different configurations used.
- BrowserStack. Each version of a PR is also tested on multiple platforms. These include Chrome(Headless), Firefox, Safari, and Edge. Each instance is then tested using Karma.

11.5.6 Variability implementations

Let's take a look at how some of these variables are implemented, and when they are bound (set). We will discuss two types of variables here, with one being determined at compile time and one being decided at run time.

First, let's take a look at the platforms that Material-UI supports. This is a characteristich that, in the modern JavaScript ecosystem, is very much dependent on the build configuration that is used. Although JavaScript is standardized by Ecma International under the name ECMAScript[68], there are currently many implementations, which, depending on their age, may or may not support certain modern features.

To combat this problem, many developers use source-to-source compilation techniques. New and modern language features are translated into more primitive versions (when possible), so that older runtimes can run this newer code as well. Material-UI does this through a tool called Babel[69], allowing Material-UI to run on older browsers. There are limitations and drawbacks however. The bundle size increases, and not all language features can be backported. Typically, support for very old or unsupported runtimes is eventually dropped. It is expected that a next major version of Material-UI will no longer support Internet Explorer, for example.[70]

Secondly, we'll discuss how developers can customize the styling of Material-UI. The project provides an API that allows developers to create theme objects. Such theme objects specify what the color palette of the application looks like, how the typography is defined and should behave, and many more things that can impact the visuals of the final application. This is an example of configuration that is not defined at build time, but at run time instead.

The most common way to implement such customization for Material-UI is to first create a theme object using the aforementioned API. A developer can specify the values they'd like to use, and any values that are not specified default to the values set by the project. This object can then be loaded into a `ThemeProvider` component, which will then use React's Context API to create a theme context. All child nodes of the `ThemeProvider` can then use this ambient theme object to apply theming options such as colors. The nature of React contexts also allows such themes to be hot-swapped while the app is running; React will automatically re-render changed components.

11.5.7 Variability and the future

The current future plans are to implement more customization by various means:

- Using styled-components, which would allow more usage of 3rd party libraries[71].
- Allowing the Box props to be used in all core components[72].

[68] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[69] https://github.com/mui-org/material-ui/blob/cc3b03becb98902b027026081e01832b009e7507/babel.config.js
[70] https://github.com/mui-org/material-ui/issues/14420
[71] https://github.com/mui-org/material-ui/issues/6115
[72] https://github.com/mui-org/material-ui/issues/15561

- Allowing the use of dynamic theme variants and colors[73] [74].

- Allowing the use of the components without any styles and instead use themes that decide the style[75].
- Improve the support of custom breakpoints, which will allows more control of behavior with different screen sizes[76].

The nice thing is that these changes independently add functionality, and are therefore good choices for adding more variability. The downside is that these are quite extensive changes that need large overhauls.

The cost of making extensions that allow more variability is relatively low in this project. Because Material-UI uses a component and layer based architecture, adding a new components generally only requires tests for the added components. Therefore it is relatively cheap to maintain a proper test suite during extensions of the project.

The binding time would be determined through how these features will be implementend in the core of the project. Thus, for these extensions it is only needed to implement variability for the specific parts that are added by the extension.

11.5.8 Conclusion

We conclude this essay by establishing that Material-UI allows for a wide range of variability, with the project running on multiple platforms, allowing for many different styles, and supporting many different combinations of component usage. This is achieved by combining compile-time solutions with run-time options developers can make use of. The project is generally unopinionated, and allows consumers of the project to use it in any way they please through the wide range of customizable settings.

Although Material-UI does use its own custom styling solution, it does not force developers to use this solution. While the authors recommend the usage of its own styling system to reduce the final bundle size, it is entirely possible to use a style the user interface with code unrelated to Material-UI. We do not have any recommendations on making the project more variable besides the ones Material-UI lists in their issue tracker.

[73] https://github.com/mui-org/material-ui/issues/15573
[74] https://github.com/mui-org/material-ui/issues/13875
[75] https://github.com/mui-org/material-ui/issues/6218
[76] https://github.com/mui-org/material-ui/issues/11649

Chapter 12

Meteor



Meteor is an open-source JavaScript framework written on top of Node.js, used to develop web and mobile applications. It is a tool for developers who want a responsive and real-time updating application without going through all the trouble of setting one up. Meteor is an isomorphic framework, meaning that all environments (i.e. server, web client and mobile client) are developed using the same language (Javascript), and (partially) run the same code.

One of Meteor's key features is that the developed applications are 'reactive', meaning that if a change is made somewhere in the application, the same change is instantly reflected on all connected clients. Because of this, Meteor is a great tool for developing real-time applications. Meteor uses a MongoDB database on the server, and a Minimongo database on the clients, which serve as a cached version of the database. This makes the client faster, and also allows it to update automatically whenever the server's database is updated.

Meteor is very flexible. Applications can be developed in Javascript, Typescript, Coffeescript, etc. On top of that, different frontend technologies can be used, like Meteor's own frontend Blaze, Angular, React and Vue. Because Meteor is built on top of Node.js, Meteor has full support for NPM. But besides NPM, Meteor also comes with its own package manager called Atmosphere, which has packages specifically created for Meteor by other developers.

12.1 Why investigate Meteor?

We chose to investigate Meteor because for our Bachelor End Project we used Meteor to develop an application to run semi-virtual escape rooms for our client company Raccoon Serious Games.

In short, RSG creates and hosts escape room-like events for companies, where the employees of that company split up into teams and all work on the same physical puzzles. After solving a puzzle, they will have to give their answer to the puzzle on a tablet, and then get up and get the next puzzle. The people who hand out the puzzles have to keep track of which puzzles every team has, which puzzles they've solved and which puzzles they should receive. They do all of this on a tablet as well. Lastly, there are also people who

walk around the room and help the teams if they need it. They can track a team's progress, and send them hints, also from a tablet.

We used Meteor to build the application all of these tablets are running. We were able to test this application in practice at an actual event, and the application was taken into use since then, still being used today.

One thing which we really enjoy about Meteor is that it makes developing a reactive system really easy. A lot of the necessary set-up involved is abstracted away by Meteor, so we don't have to worry about it. But, as you can imagine, we also have a slight love-hate relationship with Meteor. We used it to build our application, but we don't understand its inner workings. There are things that Meteor does which make a lot of sense, and things that don't. Our goal with this project is to get a better understanding of why Meteor is the way that it is, and the decisions that drove this process.

12.2 Who are we?

- Ayrton Braam
- Wouter Morrsink
- Tim Nederveen
- Alexander Sterk

12.3 More information

- Meteor Website
- Meteor Github
- Using Meteor for our BEP

12.4 The vision which makes Meteor shine

Meteor provides a simple environment in which reactive web and mobile applications can be easily written and built. Meteor is open source, but it is still officially being developed by Meteor Software. It's a JavaScript framework built using Node.js. One of its core features is its reactiveness, which means that a change in the application is reflected at once on all the other clients. Meteor is isomorphic, meaning that both client and server side can be written in the same language. Furthermore, Meteor allows for different front-end technologies to be used. This all makes Meteor a great framework to create web and/or mobile applications.

12.4.1 End-users mental model

When end-users, in this case the developers of a web or mobile application, choose Meteor as the platform to use, they do so because they expect the platform will relieve them from work which they should otherwise have done manually, and because they believe that Meteor is a better fit for their project than any other framework. This is partially achieved by giving the right expectations in advance with clear documentation and feature listings, and partially by having a platform which is robust enough to be used for various types of applications, and having support for extending to applications which were not thought of in advance.

In every step of the development process, the benefits of using Meteor should weigh up to the extra effort needed to use Meteor. This holds for time, i.e. learning how to use Meteor should not cost more time than it saves, but also for architectural decisions. When designing the architecture of the Meteor system, indirectly

decisions are made about the applications of the end-users. The Meteor platform guides the users into a particular structure for their application, but this should never result in decisions which the developer feels like they are sub-optimal for their application.

12.4.2 Key capabilities and properties

The intended capabilities and properties of the Meteor platform play a large role in making decisions and trade-offs about the architecture of the platform.

12.4.2.1 Capabilities

There are a few features of Meteor which make it unique and are therefore characteristic for this platform.

- **Multi-platform development** The Meteor platform is designed to develop for web, iOS, Android or desktop apps with one code base.
- **Simplify coupling** Since Meteor is a full-stack platform it allows developers to easily connect different parts of the application to each other, from the database to the end-user's screen. This reduces the amount of code that has to be written to create a working application.
- **Plug-and-play popular frameworks and tools** Meteor allows developers to choose their own frameworks and tools - like front-end framework - to be used in a project and makes it easy to connect these to the other components of the system. Meteor also fully supports NPM.

12.4.2.2 Properties

To make the Meteor platform work a couple of properties are maintained:

- **One language** Every environment, i.e. the application server, web browser and mobile device, uses the same language: JavaScript.
- **Data on the wire** The webserver does not give HTML directly to the client. Instead, the server sends data needed to let the client render the application.
- **Full-stack reactivity** Meteor tries to keep all parts of the application in sync allowing to create a real-time interface which shows the actual state with minimal development effort.

12.4.3 Who built Meteor?

Like most open-source projects, Meteor started small. As 'Skybreak', it was a framework for "weekend projects and small applications."[1] Now, it has grown to a fully-fledged framework that can also be used in enterprise applications. Who are the people that facilitated this growth? In this section, we look at the different stakeholders involved in Meteor's development.

12.4.3.1 Meteor Software: core developers

While Meteor is open-source, it does have main contributors, known as a company called Meteor Software (formerly Meteor Development Group). Ultimately they decide Meteor's future roadmap, as well as what contributions are merged into the project. This is important information for anyone who wants to contribute

[1] M. DeBergalis. First preview

to Meteor, so they know in which areas they can contribute. It is important to them to make sure it is up to date.

Meteor Software follows two rules when it comes to developing Meteor:

- Nothing in Meteor should harm the experience of a new Meteor developer.
- Nothing in Meteor should preclude an expert from doing what they want.

They consider meeting these standards as hard, but incredibly rewarding.

12.4.3.2 Us developers: the end users

It's strange to consider developers as end users of a product. However, Meteor truly is a project by developers, for developers. As people who have used Meteor, we enjoy how easy it is to use, and how little 'boilerplate code' was needed to get our app working (and the real-time updates are just super cool). In turn, this was greatly appreciated by our client, and the end users of the app we developed from scratch!

The advantage of having developers as end users of your project, is that they, in turn, can contribute to it (did you see how we already made our first contribution!). This is of course a win-win for any software project. On top of that, not only does Meteor accept contributions to its core functionality, it also has its own package manager, Atmosphere, where developers can publish their own Meteor packages.

12.4.3.3 Galaxy clients: the customers

We already mentioned how throughout the years Meteor has scaled up to an enterprise level. Large companies like Ikea or Mazda[2] use Meteor for their websites or applications. For this purpose, Meteor Software developed Galaxy, a hosting platform designed for Meteor applications. As someone who has used it, I can tell you it takes *literally* zero effort to deploy your application, and you also get access to a bunch of metrics you can use to improve it.

Galaxy is how Meteor Software makes its money, and it and its clients are therefore very important to the development of Meteor.

12.4.3.4 Silicon Valley: investors

Lastly, while Meteor has been pretty successful since its launch[3], it still needed investors to facilitate its growth[4]. These investors not only provide resources, but also knowledge, for example in the founding of Galaxy. In return, they get a seat at the table and a piece of the pie.

12.4.4 Current and future context

For the context we look at what Meteor provides and what kind of external connections exist, which can be found in the diagram below. The Meteor framework is an open-source Javascript web framework which allows for the fast creation of cross-platform code. The framework intigrates MongoDB and uses the Distributed Data Protocol and uses the publish-subscribe method to automatically push changes in the data to clients without the need for synchronization code. Despite being open-source, Meteor is actually

[2] R. Choudhury and L. van den Oever. How we built a next-gen car configurator for Mazda using Meteor and React.
[3] E. Griffith. First GitHub, Now Meteor: Andreessen Horowitz Backs Another Developer Favorite
[4] G. Schmidt. A New Chapter For Meteor

developed by Meteor Software. The startup intended to become profitable through a hosting environment called Galaxy, which is designed for Meteor applications.

Because Meteor is able to cater to a wide range of applications it follows that a large variety of companies and persons use the framework. Companies such as Mazda and Ikea use Meteor, but also beginning or "ordinary" developers can use Meteor for their apps and web applications. However Meteor faces competition from numerous candidates such as Vaadin, Groovy and Spark. Which are all alternative open-source frameworks. Finally, Meteor prides itself on the amount of help available on StackOverflow, and they have their own section of blog, forums, FAQ, tutorials and guides to help starting developers and keep more experienced ones up-to-date.

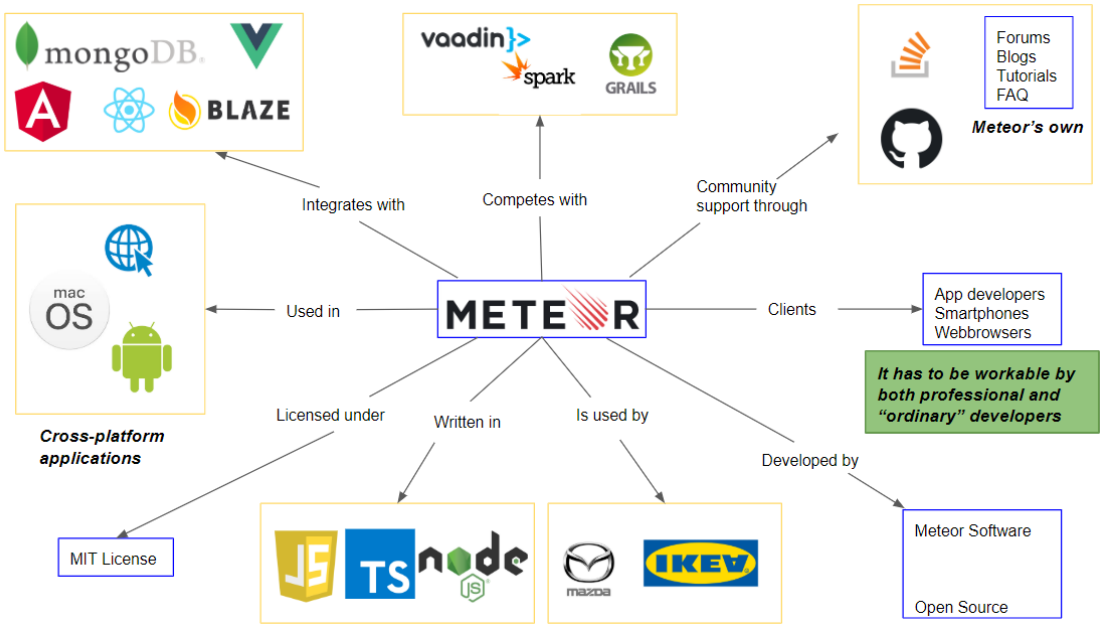


Figure 12.1: Meteor context

12.4.5 Future roadmap

Meteor was purchased in October of 2019 by Tiny Capital[5], which promised that it would continue to invest in Meteor. This can be seen as Meteor Software recently finished their update to Node.js 12, and is working on their tutorials for using Blaze, Vue, Angular, React and others in combination with Meteor.

The planned future updates of Meteor can be found in Meteor's own roadmap[6], in which there are desired features to implemented in the near- to medium-term future. The points of improvement in the roadmap are based on feedback provided from Meteor's users and developers, and are updated accordingly every quarter.

The roadmap is divided into several sections from proposed improvements to the core of Meteor, to documentation and tutorial updates. The idea is that at least two people take the lead on a point of

[5] G. Schmidt. A New Chapter For Meteor
[6] Meteor Software, Roadmap

improvement. Proposed updates to core include ultra-thin Meteor, a minimalistic version of Meteor and performance improvements regarding (re)-building the application.

Furthermore, the future roadmap of Meteor is also influenced by the updates of other frameworks and environment, which Meteor uses. The generally strategy seems to be enhancing existing relations such as updating Cordova to 9 and update the intigrated MongoDB driver. Also platform specific improments are suggested, such as the improving the performance on Windows with ideas like build-in-place.

12.5 The architecture behind Meteor's impact

In 'the vision which makes Meteor shine'[7], we researched the vision behind Meteor. Now the question is, how was this vision realized? For this, the six architectural views have been described by Rozanski and Woods[8], namely: concurrency, deployment, development, functional, information and operational. In this blog post, we will be taking a closer look at which ones of these are relevant to Meteor and go more into detail on those which are.

Viewpoint	Relevance
Concurrency	A concurrency view describes which parts of the system can run concurrently and what is managed. For Meteor, this viewpoint is relevant especially with regards to the Meteor methods[9].
Deployment	The deployment viewpoint is about the environment in which the system is deployed. This is relevant for Meteor, since Meteor has several dependencies.
Development	The development viewpoint is about the software development procedure. Meteor is open source, which makes this view relevant, since both developers from inside and outside Meteor Software work on improving Meteor.
Functional	A functional viewpoint for Meteor is important. It describes the tasks and interactions of Meteor when it is running. This view is useful for all stakeholders, but especially useful for developers who want to understand how Meteor works.
Information	An information view describes the way that Meteor deals with data. The reactiveness of Meteor makes this a relevant view since data needs to be updated fast.
Operational	The operational viewpoint describes how the system will be managed and supported. This is relevant for Meteor since its users need to know how to upgrade to a newer version.

[7] The vision which makes Meteor shine

[8] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

The line between architectural choices in the Meteor framework and architectural choices with regards to how developers are steered to structure their application can be very thin. The Meteor framework's tasks is taking the code written by an application developer to create source files for running a built application. In this essay, we will mainly focus on the architectural choices made for the framework, not the end user's application.

12.5.1 The source code

Looking at Meteor's source code, there is a distinction between the code that is used to develop Meteor web applications and the code which builds, runs and deploys these applications. In the Meteor project, this separation can be summarised in two parts: the Meteor core and the Meteor tool, respectively. In this section, we will look at how these parts are built up and how they depend on and interact with each other. The architectures of both parts differ greatly, as such we will discuss them separately and then compare them.

12.5.1.1 Meteor Core

Meteor's core can be found in the 'packages' directory. Every subdirectory is its own 'package' and provides a part of Meteor's core functionality. Packages are relatively standalone, but they can depend on other Meteor packages, as well as packages from NPM[10]. Packages are loaded by both clients and the server of an application, but they can do different things depending on the platform[11]. Furthermore, most packages are optional and are not even included in a newly created Meteor project.

Not all Meteor packages are written by Meteor Software: Meteor has its own package manager, Atmosphere, where developers can submit custom packages[12]. This is a relatively big feature of the Meteor ecosystem and you will rarely find a Meteor application without any third-party packages.

12.5.1.1.1 Packages There are different types of packages[13].

- Build packages, which run at build/compile time of a Meteor application.
- Runtime packages, which run at runtime of a Meteor application.
- Empty packages / Feature Switches. The presence of these packages can be detected by other packages and it changes their behaviour.
- Wrapper packages, which are wrappers around NPM or other non-Meteor Packages.

While it is infeasible to list every core Meteor package, we will give an overview of the most important/interesting ones.

Package	Function
`meteor`	Defines Meteor environment used throughout the framework and while developing applications. Core package.
`ddp`	Distributed Data Protocol. Websocket communication between server and clients.

[9] Fun with Meteor methods
[10] Meteor Guide: Writing Atmosphere packages
[11] Meteor Guide: Writing Atmosphere packages
[12] Meteor Guide: Writing Atmosphere packages
[13] Package.js: Documentation of Meteor's package API. Build Plugins API

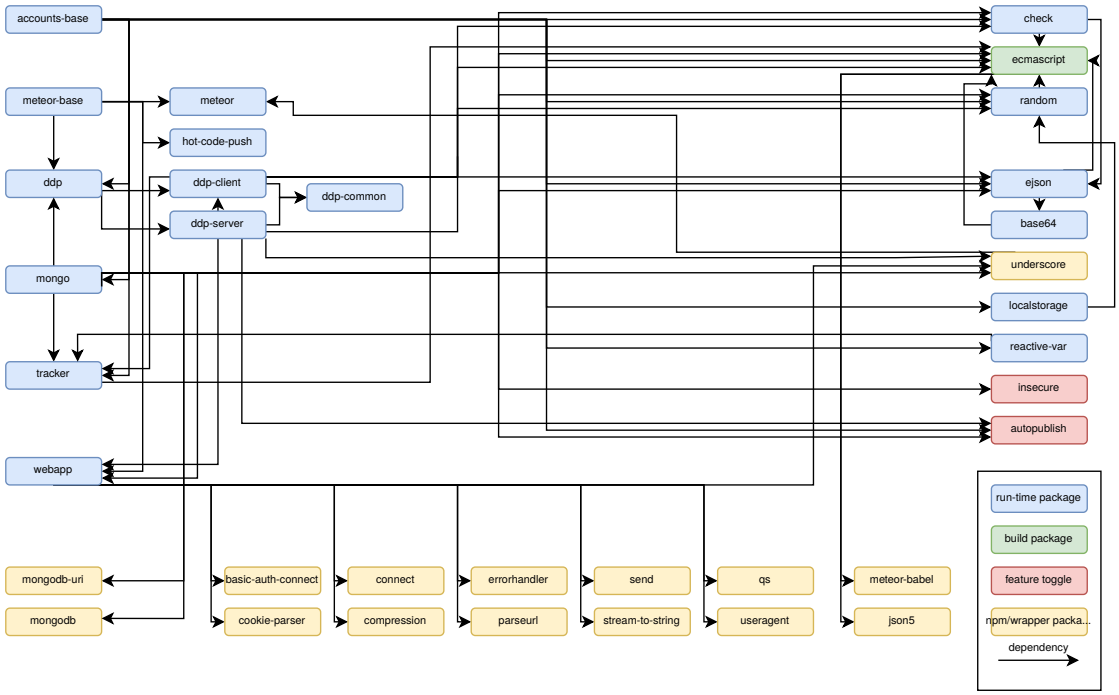
Package	Function
`mongo`	Database driver. Provides database interaction and reactivity to Meteor applications.
`accounts`	User account system. Split up into sub-packages: base functionality, passwords, OAuth, etc.
`tracker`	Enables reactive programming. Reruns computations when data changes.
`webapp`	HTTP Server that turns a Meteor project into a web application.
`tinytest`	Test runner for unit testing local Meteor packages.
`ecmascript`, `typescript`	Enable developing in EcmaScript and/or TypeScript.

Looking at the different packages that exist in Meteor's codebase, it becomes clear that there are higher-level packages, which provide the main functionality, and lower-level packages, which provide more utility and ease of programming. In the following diagram we put some higher-level packages on the left and some lower-level packages on the right. Notice how a lot of higher-level packages have dependencies on the lower-level packages. (Note that this diagram barely covers all the available packages. It is meant to provide a general sense of Meteor's core packages and their dependencies.)

12.5.1.2 Meteor Tool

The Meteor tool can be found in the 'tools' folder. This is Meteor's CLI which builds and runs Meteor applications, which will be discussed in more detail later. It's divided into subcomponents, which will briefly be summarised in the following table.

Component	Function
`cli`	Command line tool. Takes user input and executes commands.
`cordova`	Cordova integration for turning your application into a native Android/iOS app.
`fs`	Communication with filesystem. Enables Meteor to run on Windows.
`isobuild`	Build System. Compiles, bundles, builds and links packages which make up a Meteor application.
`meteor-services`	Deploying to Galaxy.
`packaging`	Managing Packages and Accessing Atmosphere.
`runners`	Running a Meteor app and Mongo instance.
`tests`	Test cases for Meteor Tool.
`tool-testing`	Test runner for `tests`.
`utils`, `console`, `static-assets`, `tool-env`	Miscellaneous.



Viewer does not support full SVG 1.1

Figure 12.2: Overview of Meteor core packages

12.5.1.3 Core vs Tool

When we compare the architectures of the Meteor Core and the Meteor Tool, we find the following.

Firstly, the Tool has a closer resemblance to a 'standard' JavaScript project. Its components are divided into subdirectories and files, which can interact with each other using `require()`, as well as regular imports from NPM. Despite this, it still feels like a 'single' project. Packages, on the other hand, are more like a separate project per package. Each package is declared using a `package.js` file, which appears to be Meteor's version of NPM's `package.json`. It declares imports from other Meteor packages or NPM. Every source file used needs to be explicitly declared within this file, and there can even be separate versions for different platforms (web client, Cordova client, server, etc.).

Secondly, this difference in architecture is also apparent during testing. The Meteor Tool is tested using the `meteor self-test` command. Rather self-explanatory, this runs the tests for the Meteor Tool, using the runner in `tool-testing`. A Meteor Package is tested using `meteor test-packages`, which runs a different type of test runner, TinyTest, on a Meteor Package. For the Meteor Tool, every JavaScript file in the `tests` directory is considered a test suite, for a Meteor Package, test files generally end in `_tests.js` and are explicitly defined in the `package.js` file.

12.5.2 Meteor releases

There are a few things Meteor Software takes into consideration when they deploy the Meteor framework to developers[14]. Note that this is not about how Meteor applications (i.e. applications built using Meteor) are deployed, but rather how developers can obtain and maintain the Meteor framework. First, Meteor needs to be installed. OSX and Linux users can simply use the command-line and run an install script. Windows users, on the other hand, need to install Chocolatey[15]. Developers who like to contribute to Meteor itself, or simply run the development version, can clone the Meteor framework from GitHub and install the dependencies. This is running Meteor from a Git checkout. When this is done, developers automatically download a so-called dev_bundle from Meteor's servers. This dev_bundle is a bundle of code, packages, and tools which provide the functionality of the Meteor tool. This includes but is not limited to:

- Node.js version
- npm version
- MongoDB version
- Packages used by meteor-tool
- Packages used by the server bundle

The standard dev_bundle should be enough for most changes, but more extensive changes might need manual changes to one or more of the fields described above. Should there be damage to your local dev_bundle and you cannot fix it yourself, there is always a way to rebuild it from scratch and re-package the dependencies. There are also security measures for changes to this dev_bundle. Any pull request involving changes to the dev_bundle will be noted by repo collaborators, and a request to have a new dev_bundle built/published will be forwarded to Meteor Software[16].

[14] Meteor development
[15] Meteor install
[16] Meteor Dev Bundle

12.5.3 How Meteor runs

The essential concept behind Meteor is that it creates code for multiple platforms from one codebase. For this, it comes with a build system: Meteor Tool[17]. This build system is used to compile, run, deploy and publish Meteor applications.

The Meteor framework is designed with a "minimal kernel". The largest architectural style of Meteor is that almost all functionality is contained in packages which can be imported when needed. These packages can depend on and interact with each other to deliver functionality to the developer's application when they are imported. Likewise, any code that is placed in an `import/` folder is only loaded when it is used, i.e. (lazily) imported in the JavaScript.

Meteor provides both the client and the server side of applications. It allows developers to provide specific code for each and common code for both[18]. When building the codebase, code that is placed in a `client/` folder (no matter where in the file structure) is not loaded on the server and similarly code placed in a `server/` folder is not loaded on the client. The code that is loaded for the client is executed in the browser or on the device of the users of the application. The code that is generated for the server is executed by NodeJS.

While the Meteor framework does not have specific run time dependencies when building the developers code into an application, a developer can incorporate dependencies like for example databases in their application.

12.5.4 Non-functional properties

Meteor comes with a lot of different packages. This could be an issue in terms of required memory, but Meteor provides an option to create a smaller version of Meteor. This minimalistic version contains less packages, but do not include Mongo or DDP[19].

Another non-functional property is that Meteor should be easy to use for new users. On the other side of this, Meteor does not want to limit the advanced user in any way. It accomplishes this by making it so that tutorials and documentation only introduce new and more advanced concepts when a new user would be ready for them[20].

Furthermore, Meteor should run on Linux, MacOS and Windows. It was originally only built to run on Linux and MacOS, but via putting filesystem calls to path and fs modules via the files.js library it can also run on Windows[21]. The drawback of this implementation is that some file related operations are slower on Windows.

12.6 Comets and commits: the software quality of Meteor

After discussing the vision[22] of Meteor and its architecture[23], it is now time to look at how its quality and architectural integrity is retained during development.

[17] Meteor Tool
[18] Modular Application Structure
[19] Meteor releases
[20] Meteor core changes
[21] Meteor fs
[22] The vision which makes Meteor shine
[23] The architecture behind Meteor's impact

12.6.1 Overall software quality

When talking about software quality in general, a large part of it can be described by the quality of the source code. A good place to start is what Meteor Software themselves say about contributing to the Meteor framework[24]. An option for contributors is reporting a bug in the source code through clear bug reports. Those provide valuable data which can be used to improve the quality of the code, and therefore the framework[25].

Another source of software quality is of course testing such as unit tests and with tools such as Travis CI[26], but this is described in more details in other parts of this essay.

There also exists third party programs that can assist with maintaining the quality of the code. A program called Sigrid provides code analysis which allows you to to measure, evaluate and monitor your software maintainability.

Our Sigrid analysis shows that the Meteor framework does not have terrible maintainability, but it does not have great maintainability either, with an overall score of 2.7. According to Sigrid, the Meteor framework's unit size and complexity as well as component independence and entanglement deserve low ratings. This could theoretically be problematic for developers who want test, as the large unit size and cyclomatic complexity makes it hard for anyone not familiar with the code to write good tests for it, thus unit testing the Meteor framework can be hard and require a lot of time. The low scores could also be a bad sign for maintainability. When a component which a lot of other components depend on gets altered, it is a costly process to ensure and test that the system does not lose functionality.

A closer look at the Sigrid refactoring candidates shows a large issue with the mongoDB_driver, which Meteor Software planned to update, according to their roadmap. It has a really long function with some complicated if-statements, which are hard to test. This could definitely benefit from refactoring, by splitting up some if-statements and breaking up the code into separate functions.

But we should not take Sigrid/SIG as gospel, the tool is still in beta and does not always work properly. To add to this, a lot of files that generated warnings were test files, which have a good reason to be long and complex, and are not part of production code. Matter of fact, a large number of files which contribute to Meteor's mediocre scores are (test) files in optional packages. Nevertheless, looking at the code ourselves, some files, such as `tools/cli/command-packages.js`, could definitely be split up in smaller parts, which should ease oversight and testing.

Other tools, such as Code Climate, provide Meteor with a similar score. But this tool also again takes test, deprecated, and even example code into account, giving a bit of a skewed image as well.

12.6.2 Testing process and pull requests

The main way the Meteor application is tested is via Continuous Integration (CI). For each pull request, the CI must run each test before the changes can be merged. Meteor has its own rules regarding code quality, which is based on the Airbnb guide. The code style is also checked by the CI, by using ESLint[27].

There are several different CI configurations which are used.

[24] Meteor Github: Contributing.md
[25] Meteor GithubL Reporting bugs
[26] Meteor Github: Travis Configuration
[27] Travis log: showing lint

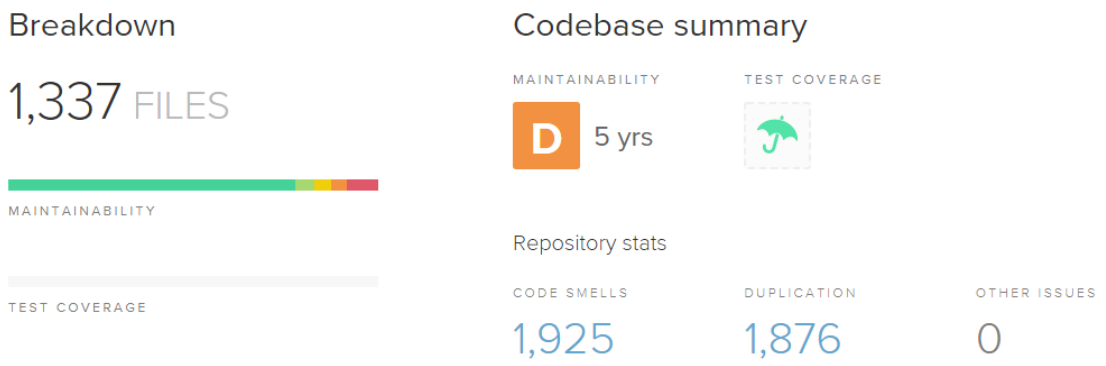


Figure 12.3: CodeClimate results

- Travis is used to run the Package tests. As described in our last post, there is a separation in Meteor's source code. Namely into the Meteor Core (in the form of individual packages) and the Meteor Tool, which is a Node.js application which ultimately builds and runs Meteor Applications as Node Applications. This CI configuration uses Meteor's own test runner for packages, TinyTest, to test the functionality of all the core Meteor packages. In order to test these packages, this configuration instructs to Meteor Tool to launch a webserver, which is then visited using Puppeteer, in order to run the tests for both Meteor client and server.
- AppVeyor is used to test the Meteor Tool on a Windows machine. Starting from Meteor 1.6, the Meteor Tool has support for Windows machines[28], but because the existing CircleCI uses Linux / Docker, there was no CI for Windows. This configuration only tests valuable, hand-picked tests[29]. Furthermore, it also tests installing Meteor (which in this case means running from a Git checkout, not a release installation) on Windows.
- "CricleCI Tests" is used to fully test the Meteor Tool. This is a collection of run configurations made up of different tests, which are ran in parallel in CircleCI. Initially, tests were grouped together/split up by name[30], but nowadays, after each group has run, the CircleCI measures how long each test took to complete, and rebalances the test groups based on this information[31]. Presumably this is done to take full advantage of CircleCI parallel containers. The Meteor Tool is tested using the `meteor self-test` command, as opposed to the `meteor test-packages` command used to test Meteor Packages. This does not invoke TinyTest, but a separate test runner made for testing the Meteor Tool.
- "CircleCI Docs" compares the JSDoc annotations in Meteor's source code, to the information in Meteor's documentation repository. The goal of these checks is to find places where the documentation no longer lines up with the source code, in order to provide a warning that the documentation needs to be updated.

Of course, it is also entirely possible to run tests locally, by executing either `meteor test-packages` or `meteor self-test` in a local console. Note that meteor also has the `meteor test` command, but that is not used for testing the Meteor framework, but applications built using Meteor.

[28] Pull request: Meteor Release 1.6
[29] Commit: Basic Appveyor testing for Windows
[30] Commit: setup circle ci
[31] CircleCI config.yaml



Figure 12.4: Example of CI in Meteor Github

Now being aware of the test process, let's look at how a high standard of code is maintained in the Meteor by going over the discussions in the issues and pull requests.

By going over some open issues, there is not much talk about code quality or testing. Most issues indicate when a problem occurs. There are several issues with no comments at all. In the once that have one or two comments mostly provide some extra information (Issue #10874) or a small update to issue regarding which version is used (Issue #10962). Fewer issues have larger discussion, there is a conversation between the developers and the contributor of the issue. Like in Issue #10850, the developers suggest some things to try and the different parties discuss how to get to the solution. Note that the conversations in issues do not discuss code quality etc. Let see how that happens in the pull requests.

Meteor has a few reviewers who are able to actually confirm a pull request and merge it. These reviewers are part of Meteor Software or frequent contributors[32]. This means that most discussion in a pull request is done between one of Meteor's reviewers and the contributor, but also fellow contributors sometimes take part.

In our own pull request, PR #10960, we used the Underscore library for a check. This created a dependency which could be avoided and a change was requested to improve this. Similar types of comments to reduce technical debt can be found in PR #10896.

In combination with the CI running the tests and the points of improvement coming from frequent reviewers and other contributors, the amount of technical debt is reduced and the code quality is kept high.

12.6.3 Architectural hotspots

Because Meteor uses large files, it is hard to identify hotspots in the code. The fact that a file has been changed a lot, does not mean that the code inside has a lot of architectural problems since each change can be to another part of the code. There are however some pieces of code which stand out in terms of amount of changes.

[32] Meteor reviewers

First of all the `package.js` files in packages that use external sources. For example, the `mongo` package. This makes sense, since these files get updated when there is a new version available. Another file which is changed a lot is the `bundler.js` file which specifies how the tool builds project files. Although this file has a large impact on the working of the framework, this is not specifically an architectural hotspot. The task of this file is combining other files, so it makes sense that this is changed a lot.

12.6.4 Roadmap's impact on architecture

The Meteor Roadmap[33] describes functionality and other changes for Meteor for the near- to medium-term future. Most items in this roadmap relate to updating components and changing or extending functionality available by Meteor application developers. Although this might change the resulting architecture in applications, this does not change the architecture of the framework itself. There are however items which impact the architecture of the framework.

A couple of items are targeted on improving the build performance. This means that the code responsible for building, the Meteor Tool, should be adjusted to be more efficient. For this, parts of the code may need to be simplified or split up, resulting in a better architecture.

Another goal on the roadmap is transitioning as much as possible to NPM. This entails publishing code that does not depend on the core or other Meteor exclusive features to NPM as separate packages. Doing so will result in less code in the main repository and a codebase with less loose components. This mindset of publishing separate NPM-packages is also encouraged for new packages.

12.6.5 Technical Debt

At the start of a software project, it can be hard to imagine how it will fare a couple of years into the feature. What works? What doesn't? Over the years, Meteor has grown quite a bit, and in this section, we will look at what we believe to be the technical debt present in the Meteor architecture, and things we believe that could have been done differently from the start.

12.6.5.1 Atmosphere

Nowadays, it's hard to imagine a JavaScript application which doesn't use NPM. However, up until Meteor version 1.3, Meteor had no full support for NPM[34]. In fact, up until version 0.9.0, the Meteor developers did not even take third party packages from Atmosphere into account either. Adding these packages used a community-developed tool called Meteorite[35].

While Meteor's separation into different packages makes sense, because it allows (experienced) developers to choose which functionality to include, the use of a separate package manager feels a bit dated, given the existence and popularity of NPM. Of course, moving away from Atmosphere and Meteor's current packaging system, and towards NPM, would be a massive undertaking. The current Meteor packages are really the core of the Meteor framework, with the Meteor Tool built around the packaging system. On top of that, Atmosphere packages are designed/implemented almost explicitly for Meteor, so migrating them to NPM might not always make sense. However, the move to NPM has been announced by the Meteor

[33] Meteor Roadmap
[34] Meteor guide: Atmosphere vs. npm
[35] Meteor Github: Packaging - historical background

developers themselves[36][37], and as such we consider this refactoring as the largest amount of technical debt.

12.6.5.2 Blaze

In a similar way, Meteor's own templating language, Blaze, feels a bit less-than as well, with the introduction of React, Angular and Vue. So much so, that it's not even listed on Meteor's homepage anymore, while those three others are. Furthermore, Blaze hasn't seen a proper update or commit in over a year[38]. Since Meteor can be used with other user interface rendering libraries, and developers are not limited to using Blaze, it might be strange to consider phasing it out as technical debt. However, we find that from a documentation point of view, Blaze is still considered Meteor's "main" library. While there are pages on how to use the other libraries (React, Angular, Vue) in combination with Meteor, the other pages in the Meteor Guide still assume Blaze is being used. We would consider reworking the Meteor Guide on including other frond-end technologies as part of the technical debt.

12.7 Exploring the Atmosphere: the variability of Meteor

Two posts ago, we discussed Meteor's architecture and its separation into two parts: the Meteor Tool and Meteor Packages. We explained that Meteor's core functionality is distributed across different, standalone (and sometimes interdependent) packages. There are a number of different packages created by Meteor itself, such as packages for Meteor's account system or packages for Meteor's reactivity and connection to MongoDB. There is also the option of third-party packages designed for Meteor, hosted on a platform called Atmosphere.

Meteor's packaging system allows developers to fully customize their Meteor application as well as to pick and choose the functionality they need. Therefore, Meteor comes with a lot of variability. In this post, we will look in further detail in Meteor's variability. We will discuss several different configurations for using Meteor, how this variability is managed and how this variability is implemented.

12.7.1 Features

Looking at Meteor's `packages` directory, we see that there are over 100 different packages. This implies the existence of many different features. However, it's not that simple. Some features are spread across packages (`accounts-base`, `accounts-password`, `accounts-oauth`) or depend on other packages (`ejson` or `base64`). Furthermore, not every package defines functionality that could we would consider as a 'feature': `ejson` is an extension of JSON which supports more types, but its main purpose is to be used by other packages, not necessarily by developers using Meteor. Consequently, nearly every Meteor package depends on it, making it impossible to not include it in your project.

In this section we will look at features we were able to identify within Meteor. We looked at features in both the Meteor Packages and the Meteor Tool. In the table below we have listed the 10 features we identified:

[36] Meteor guide: Atmosphere vs. npm
[37] Meteor Roadmap
[38] Blaze Github

Feature	Location	Explanation
Development Platform	Tool	A developer can develop Meteor on the three main operating systems: Linux, OSX and Windows[39]. This is implemented by offering a separate dev_bundle for each platform.
Deployment Platform	Tool	When deploying, users are given the choice between using Meteor's own hosting platform Galaxy, using the `meteor deploy` command, or by building a local version of their application using `meteor build`, and deploying this manually. When building a local version, there is an option to specify the target architecture: osx.x86_64, linux.x86_64, windows.x86_64, linux.x86_32, windows.x86_32
Account systems	Packages	The developer can choose from multiple account systems. From an account with a simple password to logging in using your Google, Facebook or GitHub account[40].
Development language	Both	Meteor itself is written in ECMAScript 2015, but is transpiled to ES5 for full compatibility with older systems. This allows developers to also develop their application with other programming languages with transpile to ES5, such as TypeScript or CoffeeScript (or ES2015)[41].
Front-End	Both	Meteor provides several different options regarding the front-end of the application. Developers can choose to use Meteor's own Blaze, but also Angular or React[42], or Vue[43]. Of course, the option of serving static HTML, and not using any front-end framework, is also an option. Theoretically Meteor is compatible with other front-end solutions, but no official instructions are given.

Feature	Location	Explanation
Database choice	Both	Meteor's default database is MongoDB. On clients, minimongo is used. These databases are necessary components for much of Meteor's reactive functionality. While it's technically possible to run Meteor without a database, you would lose a lot of functionality. It is also possible to use databases other than MongoDB (for example AlaSQL[44], RethinkDB or PostgreSQL[45]), but this is made possible by third-party packages from Atmosphere, and not by Meteor Software itself. As such, we do not consider these databases in our feature model.
Packages manager	Tool	Meteor uses the `meteor add` command to add packages to its configuration. This allows for a multitude of different features. Not only can a user add official Meteor Packages, they can also that command to add third-party packages from Atmosphere. The choice between including third-party packages is what we would consider a feature. Note that it is also possible to import NPM packages[46].
Application architecture	Tool	Meteor used to have a recommended way to structure the application files. Although it is still recommended, it is possible to choose your own file structure[47].
Client Platform	Both	One can choose to make a web-based application or a mobile application for iOS and Android using Cordova[48]. Note that a requirement of developing for iOS restricts the development platform to OSX, as XCode is needed.
Support for older browsers	Both	Meteor allows for the support for legacy browsers. It is up to the user to decide whether to support legacy browsers[49].

[39] Meteor website: Installation
[40] Meteor Docs: Accounts API
[41] Meteor Guide: Transpilation
[42] Meteor Guide: Front-ends
[43] Meteor Guide: Vue
[44] DJ Walker-Morgan. 2015. Meteor, SQL & Other Databases
[45] Sam Corcos. 2015. Meteor + Any Database
[46] Meteor Guide: Atmosphere vs. npm

When using the Meteor Tool to create a new Meteor project using `meteor create`, developers can specify different configurations for their new project. Different configurations come with different packages by default. The command currently supports `bare`, `full`, `minimal`, `package`, `react`, `typescript` or just the default[50]. The different configurations therefore come with different settings for the features listed above[51].

12.7.1.1 Feature Model and incompatibilities

We made a feature model for the features given above. Note that we have only listed the configurable options supported officially by Meteor Software. While it is possible to use a front-end other than those listed, the ones we have given in our model are those which are documented by Meteor's developers. This similarly holds for the database feature. We made a conscious choice not to include third-party packages which would allow for other databases, simply because they are not implemented by Meteor Software themselves.

Because of Meteor's architecture, there are very few incompatibilities between our investigated features. In fact, the packages are designed to be compatible by nature. The biggest incompatibility we found was that it is impossible to develop Meteor applications for iOS on a Windows or Linux machine, because XCode is required to build the application[52]. Other than that, we mainly found dependencies within features. For example: the accounts system relies on a MongoDB database, using Angular as a front-end typically implies developing in TypeScript (this is not a hard constraint)[53] as well as running Meteor without clients as a REST API requires a third-party package from Atmosphere or NPM[54] (therefore not listed in the feature model).

12.7.2 Variability Management

Now that we identified several features that vary between Meteor applications, it is time to discuss what Meteor provides to help with the different options and how the variability can be used by the stakeholders.

12.7.2.1 Variability management for stakeholders

Most of the variability features defined are relevant for the developers using Meteor, and not so much to the other stakeholders.

Meteor comes with a very detailed documentation website and guide for developers. If a developer would be wondering on what kind of package manager would fit his/her project, it can be looked up in the guide, where the advantages of both Atmosphere and NPM are described. The same holds for accounts, as both the guide and documentation clarify what the possibilities are. The guide explains what the possibilities are, while the documentation page explains with some example code how different methods regarding accounts work. Regarding the different front-end possibilities, Meteor offers the tutorial for all possible front-end frameworks. This allows the developer to go over them and test which one would fit the application best.

When a new version of Meteor is released, most likely the only thing a developer will need to do is run `meteor update`, and `meteor update --release`. This will update the used packages in a Meteor project,

[47] Meteor Guide: file structure
[48] Meteor Guide: mobile
[49] Meteor Github: support modern/legacy browsers
[50] Meteor Github: Project skeletons
[51] Meteor Docs: meteor create
[52] Meteor Guide: iOS
[53] Meteor create angular-boilerplate
[54] Meteorpedia: REST API

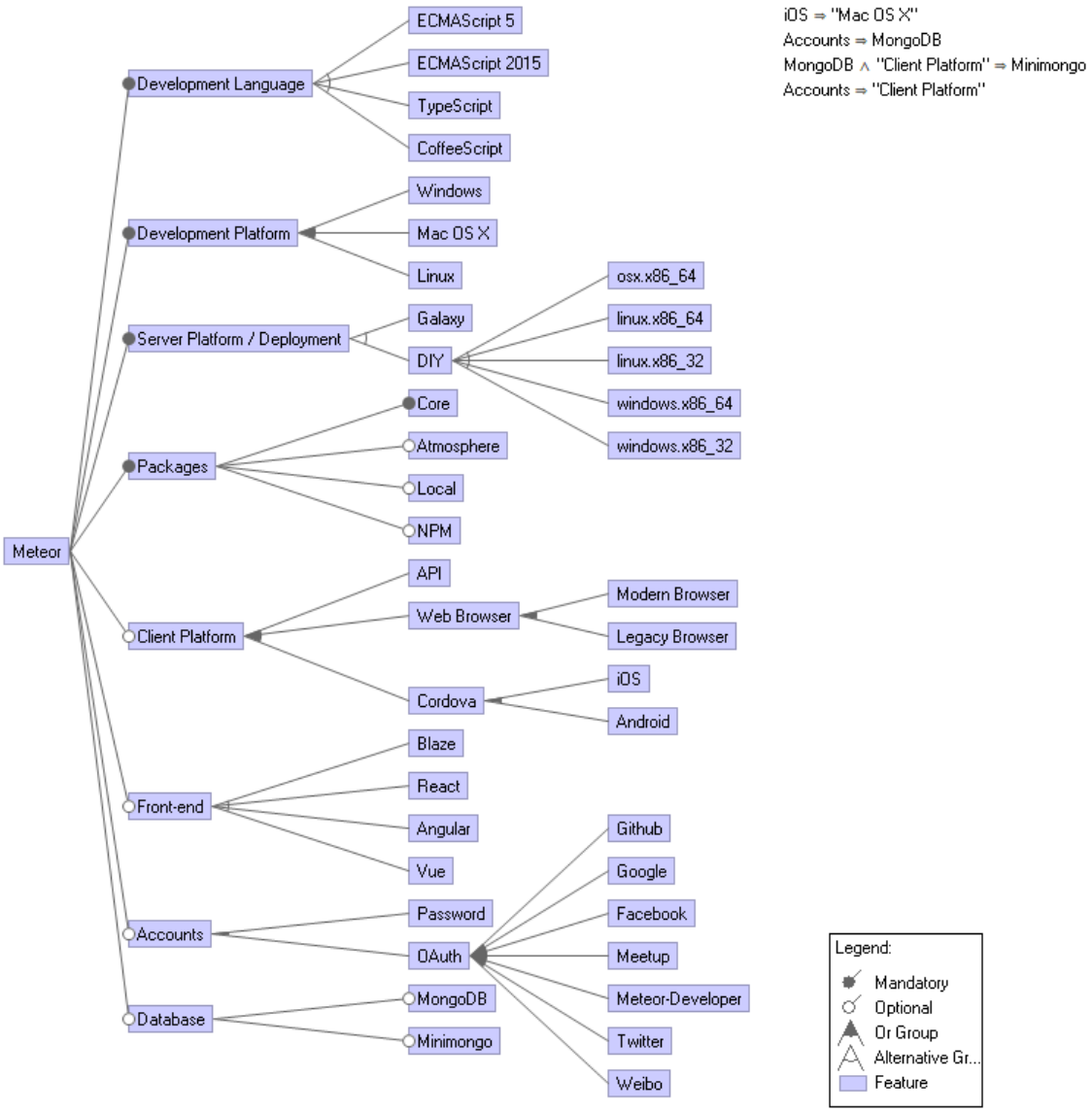


Figure 12.5: Feature model of our 10 identified features

as well as Meteor itself. If a project is a couple of updates behind or unexpected issues arise, it might be necessary to perform some extra migration steps. This are posted in Meteor's changelog.

For the other stakeholders to keep up to date with Meteor, good places to check occasionally are the Meteor blogs and the Meteor forums. These provide information on the latest news and allows users to ask questions about updates and more.

12.7.2.1.1 Alternative variability Imagine that you are a developer (if you are one even better) and you want to use Meteor but, for example, you absolutely hate MongoDB and want to use an alternative product. Where could you find the information on these alternatives?

One answer to this question is Awesome Meteor, a community-curated list of packages and resources. Awesome Meteor gives you an overview of alternatives, which allow you to customize your configuration of Meteor to your own preferences. To get back to our database example, Awesome Meteor provides the following list of alternate databases to MongoDB:

- vlasky:mysql - Reactive MySQL for Meteor
- meteor-pg - New and improved PostgreSQL support for Meteor
- ostrio:neo4jdriver - Neo4j Driver for Meteor, with support of GrapheneDB
- numtel:pg - Reactive PostgreSQL for Meteor
- simple:rethink - RethinkDB integration for Meteor

If you want to explore even more there is also Atmosphere, a repository of over 14.000 community packages that are designed specifically for Meteor.

12.7.2.2 Easy variability management

The main mechanism which eases the variability management is of course Meteor's packaging system. Meteor's functionality is split up across individual packages which are also separately tested and can be individually included into (or excluded from) a user's Meteor project using `meteor add`, `meteor update` or `meteor remove`. It's integration with Atmosphere provides for an extra mechanism of variability in terms of which packages can be used.

Another form of easing the variability management comes in the form of the Meteor Tool itself, which automatically detects the platform of the machine you are running it on, in order to build the correct version of the final application for either development or deployment. It is also possible to pass an extra option to the `meteor build` command, to specify a different target platform.

12.7.3 Implementation

Since the primary goal of Meteor is to compile the developers code into an application that the end-user can use, most of the variability is compile-time binding. Most choices made by the developer are reflected in the generated code. Code for functionality that is not in use, is not present in the created application.

One example on how this is implemented is the package system. A developer can configure the system by importing the desired functionality in their code in the same way a developer would do this with any other external package or code. In the case the developer wants to use functionality of a package which is in Atmosphere or NPM, an entry to a configuration file is added to make it available for import.

Another example is the architecture a developer must use. Developers are free to choose any file structure they want. Meteor recommends placing code in the `imports/` directory, since this code will only be loaded

lazily, meaning it will not be loaded when it isn't imported. This, however, is not a mandatory structure. When files are placed outside of the `imports/` folder, these files are loaded eagerly[55].

The way Meteor is set up is extremely scalable in terms of variability. The package structure makes it very easy for Meteor or external developers to extend the code in the future. Because almost all functionality that can be used in a client application is based on these packages, the developer can extend their application with nearly every change in the future. Furthermore, in most cases, it is quite easy to replace one option with another, for example with the front-end framework, without having to rewrite all other parts of the application.

Meteor has set the goal to move all code that does not depend on other parts of the core code to external package managers. This adds to the clear variability of the system, since it makes clear what choices users have. While developers of packages can test their application against the core code, there is no option to test if two packages can function together. However, since functionality is only used when imported, this will rarely lead to compatibility problems between different packages.

[55] Meteor Guide: file structure

Chapter 13

Micronaut

Micronaut is a modern, JVM-based, full stack microservices framework designed for building modular, easily testable microservice applications.

Micronaut is developed by the creators of the Grails framework and takes inspiration from lessons learnt over the years building real-world applications from monoliths to microservices using Spring, Spring Boot and Grails.

Micronaut aims to provide all the tools necessary to build microservice applications including:

- Dependency Injection and Inversion of Control (IoC)
- Sensible Defaults and Auto-Configuration
- Configuration and Configuration Sharing
- Service Discovery
- HTTP Routing
- HTTP Client with Client-Side Load Balancing

At the same time Micronaut aims to avoid the downsides of frameworks like Spring, Spring Boot and Grails by providing:

- Fast startup time
- Reduced memory footprint
- Minimal use of reflection
- Minimal use of proxies
- Easy Unit Testing

Source: https://github.com/micronaut-projects/micronaut-core

13.1 The Team



Shipra Sharma



Sayra Ranjha



Fabian Nonnenmacher



Héctor Bállega

- **Shipra**: Computer Science masters student @TU Delft, 3 years of experience in Software Development and Testing, interested in Security, Machine Learning and Software Engineering!
- **Sayra**: First-year Computer Science Master's student. Interested in Machine Learning, Software Engineering and Embedded Systems.
- **Fabian**: CS master student @Université de Rennes 1 & @TU Delft, Passioned about: Agile Development, Continuous Delivery, Cloud & Domain Driven Design
- **Héctor**: Cloud Computing and Services master student @Aalto University & @TU Delft, interested in containers, Cloud Native development and good software practices!

13.2 Micronaut - The Product Vision

Micronaut is a full stack framework for JVM developers helping them to build modular, easily testable microservices and cloud-native applications. It provides a familiar development workflow similar to Spring or Grails but with a *minimal startup time and memory usage*. Therefore, Micronaut covers the gap where traditional MVC frameworks are not suitable, such as Android applications, serverless functions and IoT Deployments.

Traditional JVM web applications are based on frameworks that promote the Model-View-Controller pattern, Aspect Oriented Programming and Dependency Injection. Some examples are Spring and Grails, both frameworks heavily relies on reflection to analyze application classes at runtime and then wire them together to build the dependency graph for the application.

The I/O and reflection activities can be be very costly for startup time and memory usage, two important resources to pay attention when we run our application in a cloud platform.

The end goal of Micronaut is to preserve the MVC programming and other features of traditional frameworks

and keep a low memory footprint by implementing a reflection-free approach to Dependency Injection and AOP.

13.2.1 End User mental model

The Micronaut API documention reflects the needs of the software developers which are the central end-user of the system. As stated by Coplien and Bjørnvig in the *Lean Architecture: for Agile Software Development*, its important to capture the key-end user mental models in the code to radically increases its value.

Micronaut boost **software developers** productivity and satisfaction by keeping the annotated conventions from Spring. The main difference is that it performs the dependency injection at compile-time rather than run time.

13.2.2 System's key capabilities and properties

As stated earlier, Micronaut aims to be a *full-stack* framework and therefore it aims to provide all tools necessary for building a JVM-based microservice. The tools and functionalities included into Micronaut are well explained in their User-Guide and in the following an overview is given.

The most important aspect which makes it a framework, is that it provides Inversion of Control (IoC) by implementing a Dependency Injection functionality.

The framework comes with meaningful default configurations so that new applications based can be built quickly. For keeping the business application code free of secondary or supporting functions it also contains Aspect-Oriented Programming (AOP) mechanisms.

For communicating between microservices often HTTP(S) based communication (especially REST) is used. Therefore, Micronaut provides functionality to implement HTTP(S) servers and clients. Of course, Micronaut also includes a security solution for all common security patterns with different authentification providers (e.g. BasicAuth, Json Web Token, and LDAP).

Besides HTTP(S), also lightweight messaging is widely used fro microservice communication. To fullfil this need Micronaut provides integrations for RabbitMQ and Apache Kafka.

Naturally, a web-service without a database often cannot provide real functionality. Therefore, Micronaut comes with connectors to many different databases. Besides the classical SQL databases also many NoSQL databases as Redis, MongoDB and Cassandra are supported.

Micronaut is designed for building Cloud Native applications and supports many best practices from this architecture approach. For example, it allows external configuration and integrates well with widely-used distributed configuration servers. Furthermore, it includes different mechanisms for service discovery and client-side load balancing.

Apart from Micronaut focusing on microservices, it can also be used for building serverless functions and CLI applications with the JVM languages Java, Groovy and Kotlin.

It is not surprising, that the functionality provided by Micronaut is inspired by similar frameworks like Grails or Spring Boot. However Micronaut differentiats from them, especially by focusing on quality capabilities. Micronaut aims that applications build on top of it are having short startup times (*Time Behaviour*), are low on memory consumption (*Resource Utilization*), are easy to test with unit tests (*Testability*) and that they can handle the failure of conencted services (*Resilience*).

13.2.3 Stakeholder Analysis

As stated by Coplien and Bjørnvig in the *Lean Architecture: for Agile Software Development*, half of the software development revolves around the technical stuffs like coding, testing, etc but the other half has to deal with the people and relationships. These people who are directly or indirectly impacted by the use or service of any product are generally termed as stakeholders. Stakeholders derive value from the product. There are multiple stakeholders impacted by Micronaut and we broadly divide them into the below categories:

The Business - Micronaut's parent company *Object Computing* sponsors its development, maintains the framework and employs key members of the Micronaut team, hence becoming the key business stakeholder.Also, Micronaut's collaboration with *Google Cloud Platform* and *Amazon Web Services* for extended cloud based service, makes both of them the business stakeholders.

The Customers - A customer could be any enterprise or company using Micronaut's framework to offer its services to the end customers like companies developing and offering cloud webservices. There are multiple customers using Micronaut namely Minecraft, Mojang, CSS, Skylo Tecnhologies, Target and Samsung SmartThings.

Domain Experts - The team of Micronaut's co-founders and software architects like *Graeme Rocher* the co-founder who created several popular open source projects, including Micronaut, *Jeff Scott Brown* who has been doing JVM application development for as long as the JVM has existed and *Álvaro Sánchez* who is a passionate software architect and agile enthusiast with over 16 years of experience.

Developers - Micronaut has a great technically diversified development team including people like *Nirav Assar* who is an experienced Java developer and *Sergio del Amo* who is an experienced web and mobile developer.

End User - *Application developers* who use the framework to develop any sort of application, for example - Microsoft (Customer) decides to use Micronaut and the developers working for Microsoft are the ones actually using Micronaut, hence, they become the end user. Another end user may be any player playing Minecraft's game based on Micronaut's framework.

13.2.4 Product's usage context

Currently, Micronaut continues to be a path forward for server side Java or serverless applications. Being a complete application framework for any type of application, Micronaut enables one to build multiple applications like microservices, serveless applications, message driven applications with Kafka/Rabit, CLI applications, android applications - basically anything with static void (String []args).

Mirconaut stands out in business because of the various properties and features that it works on. These include *Ahead Of Time* compilation (eliminating run time reflections and performing compiled time annotation processes), *Dependency Injections*, *Aspect Oriented Programming*, *Beans Introspection*, etc. which result in monumental leap in startup time, blazing-fast throughput and minimal memory footprint. In a layman's language, all these features make any application based on Micronaut's framework capable of running in low memory and eliminate reflections, dynamic classloading and runtime proxies which increases the run time efficiency because majority of work is handled in compilation processes, hence, positively impacting the two most valued components of the computational world in today's era - *TIME* and *MEMORY*.

13.2.5 Product's roadmap

The future direction of Micronaut is clearly defined by the product roadmap, in particular the roadmap for the next major release, version 2.0.

From the milestones listed on the roadmap it becomes evident that Micronaut intends to become an even more comprehensive full-stack JVM-based microservice framework.

Specifically, the major features that are planned for the 2.0 release are: Reactive Micronaut Data for SQL, Neo4j and MongoDB; HTTP/2 support; improvements to serverless support; new build plugins for Maven and Gradle; and further performance refinements. The 2.0.0 milestone on GitHub includes some additional objectives, notably some changes to address architectural issues.

The roadmap beyond this version is loosely defined. It is interesting to note, however, that the focus so far lies on adding and improving support for even more protocols and services, such as MQTT, Azure, and Google Cloud Platform. It is clear that the direction Micronaut is aiming for is to become a framework with an ever-increasing support for tools to build JVM-based microservice applications.

13.3 Micronaut - From Vision to architecture

As described in the previous blogpost, Micronaut is a framework that provides many different functionalities and tools to build a Microservice application. This blogpost describes, therefore, how all these different functionalities are organized and which patterns are implemented to provide the best experience for a user developing an application.

13.3.1 Overview

The first section, Architectural Patterns, will give an introduction to high-level ideas on how the framework interacts with the application code.

After that, the second section looks into the System Decomposition. This development view describes how the modules are organized.[1] In the context of Micronaut, this is important because it explains how the functionalities map to different modules and how Micronaut can be easily extended in the future.

The next important view is the runtime view, which describes how the building blocks interact.[2] For Micronaut the most important aspect is the interaction with the user's application code.

Section 4, the Deployment View, describes what happens when Micronaut gets released and how it is delivered to the end-user.

Finally, the last section covers the Non-functional Requirements and describes which trade-offs were necessary to full-fill them.

13.3.2 Architectural Patterns

As described in the introduction Micronaut highly influences the architecture of the user's application. Therefore, Micronaut's internal architecture cannot be seen in isolation.

[1] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

[2] Peter Hruschka and Gernot Starke. arc42: Effective, lean and pragmatic architecture documentation and communication. https://docs.arc42.org/home/

Micronaut uses a *Component-Based Architecture* to decompose the design of the framework into modules or logical components. Another pattern extensively used is *Inversion of Control* which is applied to the internal modules as well as to the application built on top.

- ***Component-Based Architecture:*** offers a way to build software with independent, modular and reusable pieces called components. It provides a higher level of abstraction and divides a problem into sub-problems, each of them associated with a component partition. The choice of this software architectural pattern for Micronaut implies that its development life cycle and software quality get enhanced because its easier for the developers to maintain and improved the vast amount of modules. Additionally, it allows the user to use only the components which are related to the functionality he needs. A more in-depth view of the Micronaut components and its relations is given in the System Decomposition section.[3]

- ***Inversion of Control (IOC):*** inverts the control flow resulting in the framework itself calling the user's application code to perform the required tasks.[4] It is achieved mainly through the Dependency Injection principle, which is a technique where a central instance (the IoC Container) instantiates an object and supplies its dependencies.[5] In Micronaut, the objects instantiated, assembled and managed by the Micronaut IoC container are called *beans*. Together they form the context of an application. Beans, and the dependencies among them, are reflected in the configuration metadata (normally specified with annotations).[6] This pattern influences the whole architecture and is explained in practice in Section 3.

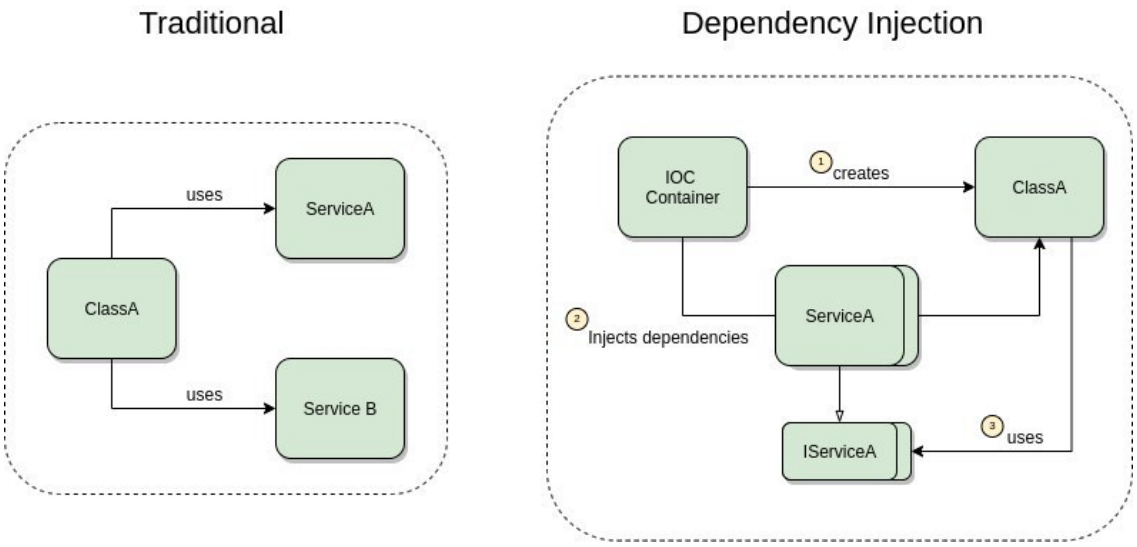


Figure 13.1: Dependency Injection

[3]Rainer Niekamp. Software Component Architecture. http://congress.cimne.upc.edu/cfsi/frontal/doc/ppt/11.pdf
[4]Martin Fowler. Inversion of Control. 2005 https://martinfowler.com/bliki/InversionOfControl.html
[5]Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. 2004 https://martinfowler.com/articles/injection.html
[6]Micronaut Documentation. Inversion of Control. https://docs.micronaut.io/latest/guide/index.html#ioc

13.3.3 System decomposition

Micronaut's code is built and organized with Gradle. Gradle is a common build tool that allows structuring the code into different modules.[7] In total, the code of the micronaut core project is organized in 38 modules. Besides this, the Micronaut framework contains other projects with additional functionality (e.g. Micronaut Data and Micronaut Security).

For narrowing down the scope, this decomposition focuses only on the modules of the core project excluding the tests and the cli project. The following figure shows the remaining 30 modules and the (compile) dependencies on each other. (The direction of the arrow can be read as *depends on*.)

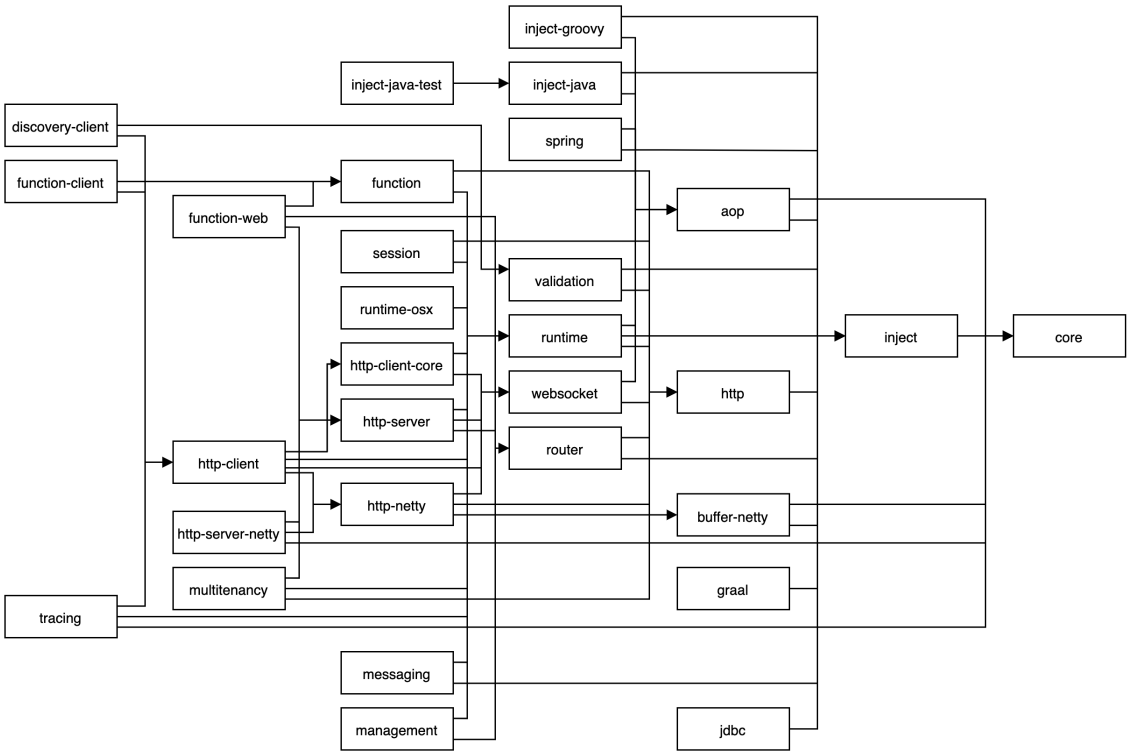


Figure 13.2: Micronaut Core's Modules and their Dependencies

The first notable thing is, that the whole project is organized in many different modules and many of them are independent of each other (e.g. *messaging* and *http-server*). This has the advantage that they can be used independently as well. For example, an application developer can build an Http-server without having to include the *messaging* modules.

Furthermore, we can see that *inject* and *core* are key modules because the remaining modules depend on them.

By reading the code, we figured out that the *core* module contains common utility classes and interfaces. This includes, for example, a set of generic annotations, helpers for reactive programming, IO utils and

[7] Gradle Inc. Gradle User Manual. https://docs.gradle.org/current/userguide/userguide.html

different converters.

The heart of Micronaut is the *inject* module. It contains the main classes for the Inversion of Control (IoC) pattern based on dependency injection, which has been explained in the previous section. This module manages the application context and all its beans. These beans can be configured with annotations. However, processing those annotations is part of a different module (*java-inject* or *groovy-inject*). This is necessary because, in contrast to other frameworks, Micronaut uses ahead-of-time (AOT) compilation to process those annotations (see Section Non-functional Requirements).

Having them as separate modules allows registering them as *annotation processors*, which is a way to extend the java compiler to generate classes at compilation time.[8] It's important to remark that the dependency injection mechanism is not only used by the user's application code. All other Micronaut modules depend on *inject* to supply their beans to the application.

One examples is the module *aop*, which implements Micronauts compiler-based Aspect-Oriented Programming (AOP). AOP allows controlling the program flow with meta-programming and annotations[9]. The *runtime* module which is responsible for starting and managing the life cycle of the application.

For allowing the user to build an Http server or client, the modules *http-server* and *http-client* (and all their transitive dependencies) are required. The modules *function-web* and *function-client* contain the APIs to write serverless functions. The core project only contains a generic implementation, the integration with different cloud-providers is implemented via external modules (e.g. Micronaut AWS).

All Micronaut core modules together depend (transitively) on around 40 libraries. Some of those dependencies are: SL4J for logging, RxJava for reactive programming, Jackson Databind for mapping objects to JSON, SnakeYAML for parsing YAML files, Caffeine for caching and ASM for modifying binary Java classes. To conclude, we can say that Micronaut is not using many 3rd party libraries and the ones it is using are well-known and widely used in the Java community.

13.3.4 Interactions of Micronaut's IoC

As described before, the Micronaut framework includes many different functionalities and therefore the interaction of the components highly depends on the user's configuration. Additionally, Micronaut follows the Convention over Configuration principle and has an extensive predefined behavior that is hidden from the user's application code. Executing the Hello-World Example which only contains a simple REST interface already results in many complex interactions. Nevertheless, to give an idea of the IoC implementation in practice, we take the Hello-World example and describe a few interactions when executing it.

One key functionality of Micronaut is the ahead-of-time compilation. By adding the module *java inject* to the compiler, the annotations are processed at compile-time. The compiler generates classes based on the annotations with the definitions of the beans. Afterward, Micronaut can create beans and read their metadata from the generated classes. Micronaut does not need to use the slow reflection API at run-time.[10]

In the startup of the application the application context with all beans is created (See figure for more details). During this phase, all meta-data of the annotations are evaluated and are then wired together. For example, the IoC Container injects all the REST controller beans into the `Router` bean. Based on them the router can

[8] Hannes Dorfmann. Annotation Processing. 2015 Annotation Processing

[9] Martin Fowler. Domain-Oriented Observability - Aspect-Oriented Programming. https://martinfowler.com/articles/domain-oriented-observability.html#Aspect-orientedProgramming

[10] Oracle. The Java™ Tutorials - The Reflection API https://docs.oracle.com/javase/tutorial/reflect/index.html

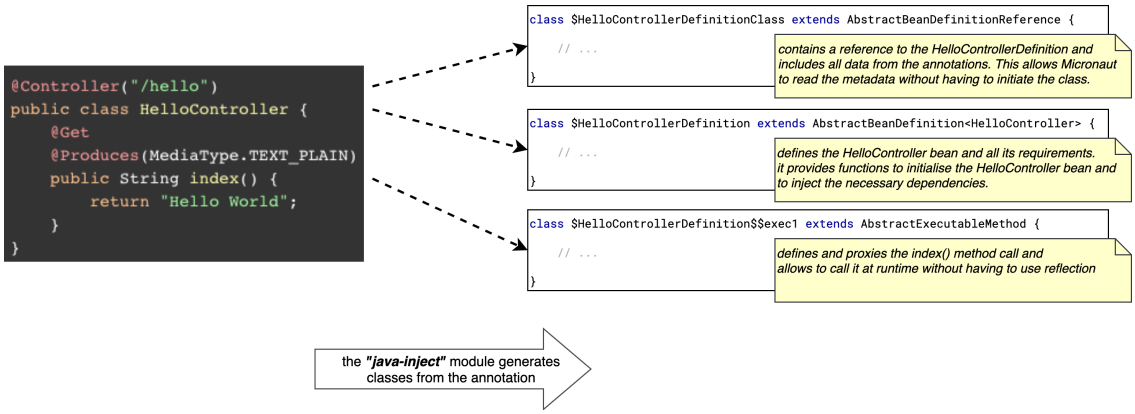


Figure 13.3: Hello World Example: classes generated during compilation

create a mapping table for all defined routes. After creating the application context, the Netty web server is started.

When a client sends an HTTP request to the endpoint `/hello`. This call is processed by the Netty server which triggers a handler in the *netty-http-server* module. This handler uses the `Router` bean to determine the corresponding controller which, in this case, is the `HelloController` bean. It calls the related method and forwards the result (' 'Hello World!") back to the Netty server which sends it back on the HTTP channel.

13.3.5 Deployment view

As stated in Rozanski and Woods, the Deployment view focuses on aspects of the system that are important after the system has been tested and is ready to go into live operation.[11] Going into live operation means for Micronaut either being executed as part of a user's application or the distribution of Micronaut's framework to the user.

In the first case, the execution environment highly depends on the user's configuration. He can execute it, for example, as an Android application, a serverless function or a microservice executed in a cloud environment. Furthermore, Micronaut can be integrated with countless third party services. For example, different databases (Micronaut Data), messaging infrastructure (Micronaut RabbitMQ) and authentication providers (Micronaut Security). All these options can lead to a high number of different execution environments.

Therefore, we are focusing on the second case. When building Micronaut two kinds of artifacts are generated.[12] On one hand, we have the classical Java libraries which can be added as dependencies to an application. These artifacts are deployed to Maven Central and can be downloaded there by the users, either manually or automatically with a build system like Maven or Gradle. It's important to mention, that for every module described in System Decomposition a separate jar library is created. This enhances the component-based architecture (see Architectural Patterns) and allows the user to only download the artifacts he needs.

[11] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

[12] Graeme Rocher. Micronaut Core Release Process https://github.com/micronaut-projects/micronaut-core/blob/1.3.x/RELEASE.adoc

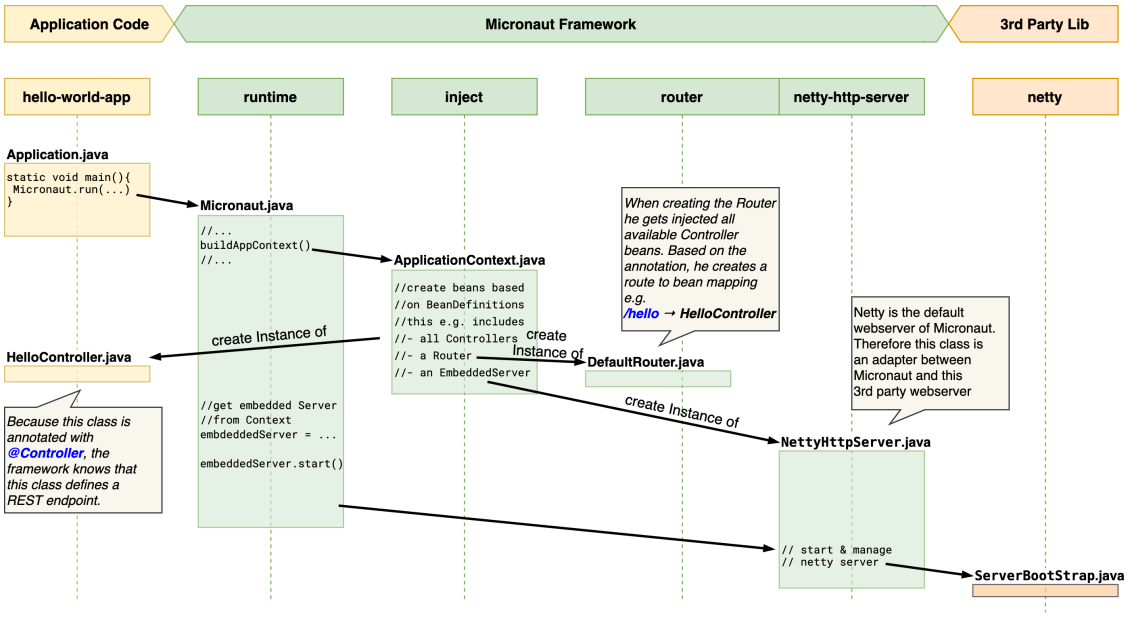


Figure 13.4: Startup phase in the Hello World example

On the other hand, we have the *cli* application which is the recommended way to create new projects.[13] This application is deployed on three different ways depending on the running platform:

1. Using SDKMAN for Unix-like systems.
2. Using Homebrew for macOS.
3. Install through a binary on Windows.

The figure above resumes the main components from the deployment view. First of all, Micronaut is a JVM framework, thus it requires Java and its development kit in the 8 or 11 version. The development team also gives support for other JVM languages such as Groovy or Kotlin. As explained in System Decomposition, Micronaut requires some 3rd party libraries. These libraries are automatically downloaded as transitive dependencies when a build system is used.

Micronaut can run on multiple platforms (Windows, macOS, and Linux) and it doesn't require any higher system requirement than the JDK 8.

13.3.6 Non-functional requirements

Non-functional requirements or also called quality properties are requirements that "do not directly mandate functionality but still have a significant impact on the architecture".[14] Examples of non-functional requirements include the performance, security, usability, and scalability of a system.[15]

[13] Micronaut Documentation. Micronaut CLI. https://docs.micronaut.io/latest/guide/index.html#cli

[14] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

[15] Calidad Software. ISO2500 Software and Data Quality https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

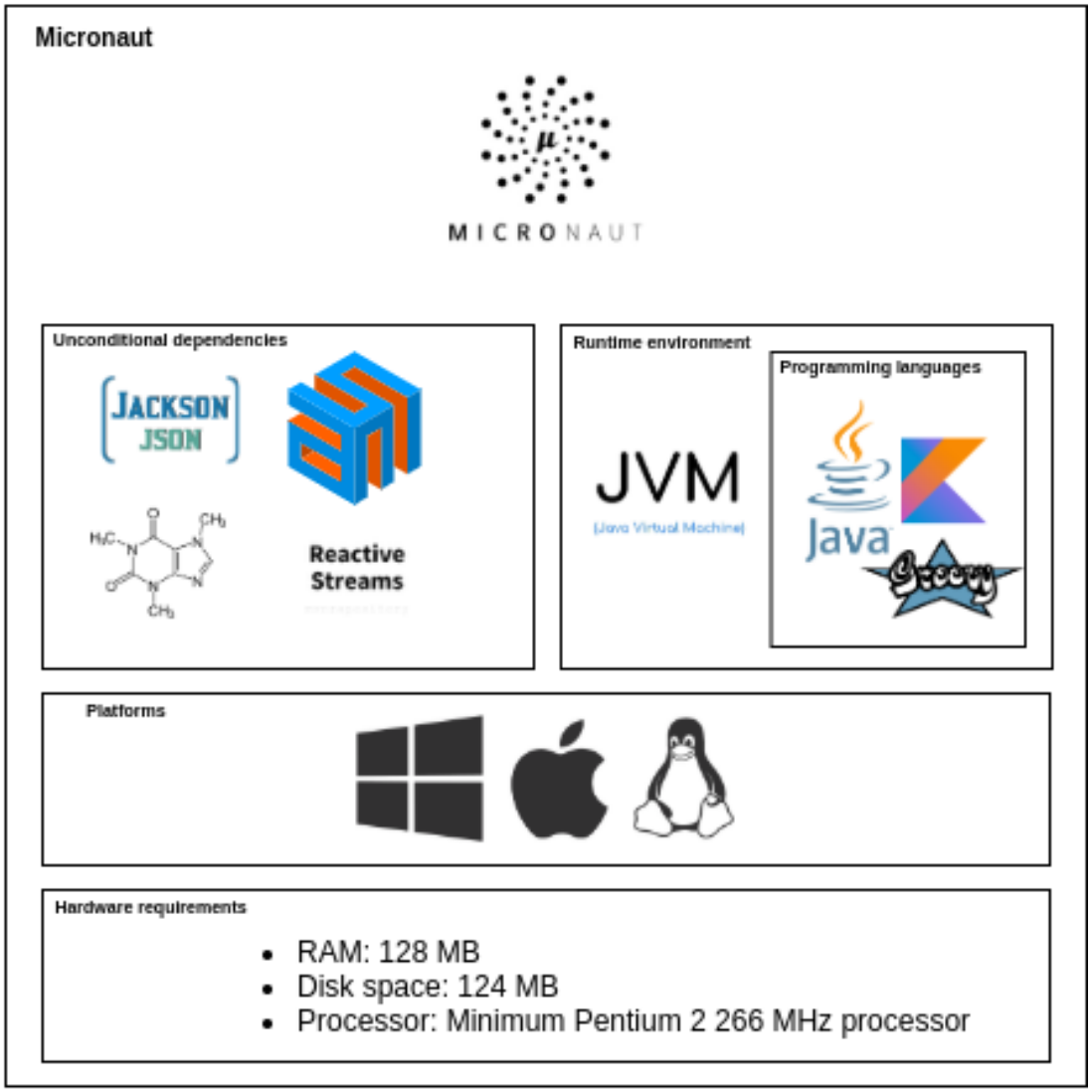


Figure 13.5: A graphical overview of the deployment view in Micronaut

Micronaut's non-functional requirements are focused on performance, specifically fast startup times (timing behavior) and reduced memory footprint (resource utilization).

Traditionally, frameworks do things such as annotation processing, reading byte code[16], invoking endpoints, and performing data binding using reflection and proxies.[17] Reflection happens at runtime and its use has multiple disadvantages.[18]

First of all, reflection leads to large memory consumption.[19] Since reflection is slow, frameworks create a reflection data cache to limit the number of expensive reflection calls. As there is no standard way of defining reflection caches each framework defines its own, causing a lot of redundant data being cached. To make matters worse, reflection data is not garbage collected until the application is low on memory. Secondly, the more beans an application has, the more (slow) reflection calls are needed to analyze the classes, which increases the startup time of an application.[20] [21]

Micronaut eliminates the need for reflection and proxies by using the already mentioned ahead-of-time (AOT) compilation. AOT compilation essentially means that more is done at compile-time and less at runtime. With the classes generated at compile-time, there is no need for using reflection at run-time.[22]

Using AOT compilation does come with its own set of drawbacks. Firstly, it leads to longer compilation times, as everything normally done at runtime using reflection is now done at compile-time.[23] Longer compilation times might slow down development and affect testability, as developers now have to wait longer before they can run or test their code. Additionally, in order to achieve AOT compilation, Micronaut had to create its own abstraction over the Java annotation processor API[24] and implement its own Dependency Injection (see Architectural Pattern) and AOP implementations.[25] All of this is code has to be maintained, which might affect Micronaut's future maintainability.

13.4 Micronaut - Quality and Technical Debt

In our previous blogpost, we discussed about Micronaut's architectural patterns and its different views. In the third blogpost of this series, we analyze the quality of Micronaut's codebase and look into the measures taken to ensure its maintainability.

---

[16] Graeme Rocher. Introduction to Micronaut. 2018 https://objectcomputing.com/files/8415/4220/9027/18-11-14-Intro-Micronaut-Webinar-slide-deck.pdf

[17] Object Computing. Micronaut 1.0 RC and the power of Ahead-Of-Time Compilation https://objectcomputing.com/news/2018/09/30/micronaut-1-rc1

[18] Graeme Rocher. Introduction to Micronaut. GOTO 2019, 2019 https://www.youtube.com/watch?v=RtjSqRZ_md4

[19] Graeme Rocher. Introduction to Micronaut. 2018 https://objectcomputing.com/files/8415/4220/9027/18-11-14-Intro-Micronaut-Webinar-slide-deck.pdf

[20] Graeme Rocher. Introduction to Micronaut. GOTO 2019, 2019 https://www.youtube.com/watch?v=RtjSqRZ_md4

[21] Graeme Rocher. Introduction to Micronaut. JBNCConf 2019, 2019 https://github.com/micronaut-projects/presentations/blob/master/jbcnconf-2019.md

[22] Object Computing. Micronaut 1.0 RC and the power of Ahead-Of-Time Compilation https://objectcomputing.com/news/2018/09/30/micronaut-1-rc1

[23] Moritz Kammerer. Microservices with Micronaut. Cloud-Native Night, 2019, https://www.slideshare.net/QAware/microservices-with-micronaut

[24] Object Computing. Micronaut 1.0 RC and the power of Ahead-Of-Time Compilation https://objectcomputing.com/news/2018/09/30/micronaut-1-rc1

[25] Graeme Rocher. Micronaut Deep Dive. Devoxx, 2019 https://www.youtube.com/watch?v=S5yfTfPeue8

13.4.1 Overview

As other projects analyzed in this course, Micronaut is developed as open-source software. The public availability of the code brings the advantaged of being used and therefore tested by many users. Moreover, it allows the open-source community to effectively peer review and bring new improvements to its codebase.[26]

With such a large community working together, Micronaut needs a solid CI Process to integrate fixed issues and new functionalities proposed by the community.

The Development Activity section discusses the planned roadmap, the ongoing development activities and the key hotspots. The last section, Quality Assessment, provides an analysis of the maintainability, technical debt and potential refactoring strategies of the most active modules.

13.4.2 CI Process

According to Martin Fowler [27] *Continuous Integration* is a development practice where individual team members integrate their work frequently and verify the integration with an automated build.

Every new push to the repository and new pull-requests trigger Micronaut's automatic build, that is defined with Github Actions.

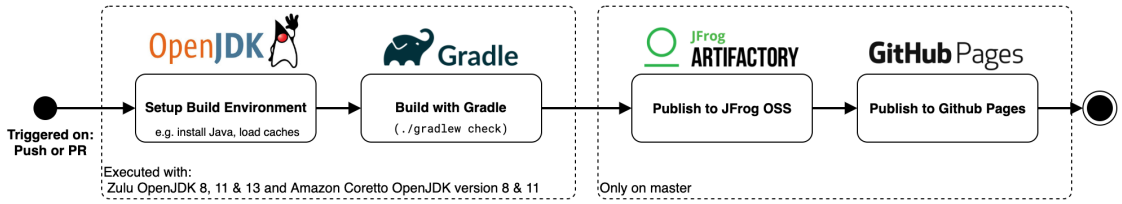


Figure 13.6: Microanut's Automated Build Pipeline

As the Figure shows, the most important part is the Gradle Build which is executed on 5 different Java versions. A deeper look in the project's `gradle.build` file reveals, that this includes compiling, the execution of tests and a static code analysis with checkstyle.

Afterward, on a master build, the generated jar files are uploaded to JFrog's OJO, a maven repository for pre-releases. Finally the pre-release documentation is updated.

Besides this automatic build, there is a release-build, a dependency check and performance tests.

13.4.2.1 Tests

In 2009 Mike Cohn introduced the concept *test pyramid* which suggests organizing tests in three different layers.[28] From a modern point of view, the concept is simplistic and the naming of the layers is not ideal, however, the basic idea of writing tests with different granularity and having few high-level tests, is a good practice for keeping the test suite fast and maintainable. [29]

[26] Open Source Software Development. Open Source Software (OSS) Quality Assurance: A Survey Paper
[27] Martin Fowler. Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html
[28] Mike Cohn. Succeeding with Agile. Addison-Wesley Professional, 2009 https://www.oreilly.com/library/view/succeeding-with-agile/9780321660534/)
[29] Ham Vocke. The Practical Test Pyramid. 2018 https://martinfowler.com/articles/practical-test-pyramid.html

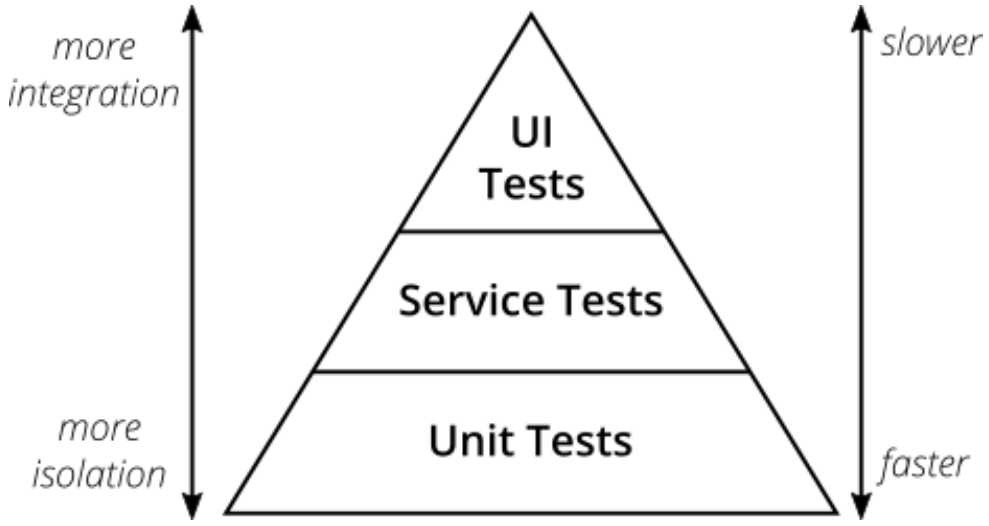


Figure 13.7: The Test Pyramid

The Micronaut team writes its tests mainly with Spock, a Groovy testing framework that provides behavior-driven_development (BDD) compatible syntax. Because the tests are not assigned to different granularity levels, we came up with a simple rule to distinguish between them. We categorize all tests which are either annotated with `@MicronautTest` or contain `Application.run(...)` into the category *class-level integration test* because then an application context is created and beans are wired together (see previous blogpost). Therefore, we assume that those tests cover multiple classes at once. For the remaining tests, we assume that they cover one individual class and are categorized as *unit tests*.

An exception are the `test-suite` modules. They contain tests written in Java, Kotlin, and Groovy which only access APIs available for the end-user. Therefore, we consider them as *end-to-end tests*.

We counted the tests per category and measured there execution time on the CI System for the development state at the time of writing.

- 2353 `unit tests` - 2 min 17sec
- 1358 `class-level integration tests` - 5 min 17 sec
- 504 `end-to-end tests` - 1 min 27

Those results show, that Micronaut's tests are following the idea of the test pyramid. They have many unit tests that are executed quickly and they have fewer high-level tests that are slower. Furthermore, it proves that our naive classification approach is not groundless and *class-level integration tests* are slower.

Because Micronaut is not measuring the test coverage, we have added Jacoco to the Gradle build. Even when this broke 4 tests, we could measure a line coverage of 68.7% for the remaining tests. To evaluate the test coverage further, we uploaded the Jacoco data to a self-hosted Sonarqube instance.

This view illustrates, that the *cli* module is barely covered. As explained in the previous blogpost, the *cli* module is an independent application and, therefore, we excluded it from the overall coverage calculation. Without this module, the line coverage increases to 75.5%.

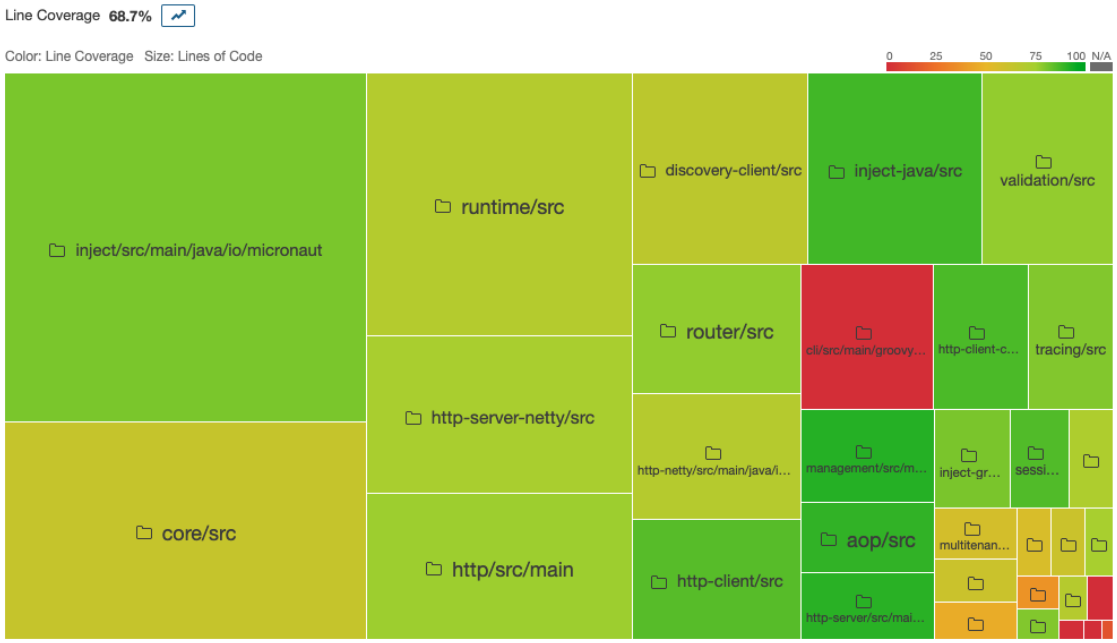


Figure 13.8: Sonarqube: Line coverage per module

According to Martin Fowler[30], the numeric value of test coverage alone provides only little information about the quality of tests. Nevertheless, he expects *"a coverage percentage in the upper 80s or 90s"* for well-thought tests. Micronaut is not measuring the coverage continuously, but with 75.5% coverage, they are close to this suggestion.

We think the careful selection of testing tools, the different granularity of tests and the reached coverage, demonstrate that tests are taken seriously and are an essential part of the development process and maybe, this mindset is more important, than following a metric blindly.

13.4.2.2 Release

The release build is highly automated. When a new git tag is added the following Travis pipeline is executed.

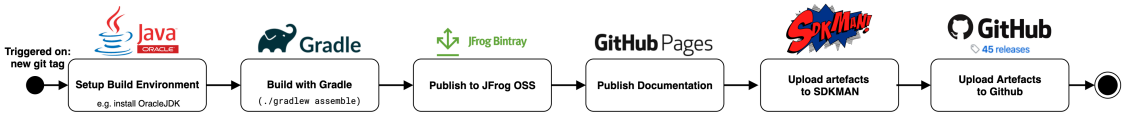


Figure 13.9: Release build

[30] Martin Fowler. TestCoverage. 2012 https://martinfowler.com/bliki/TestCoverage.html

13.4.2.3 Dependency Check

Additionally to the automatic build, Dependabot checks daily the dependencies for outdated and insecure versions and automatically commits the updated ones.

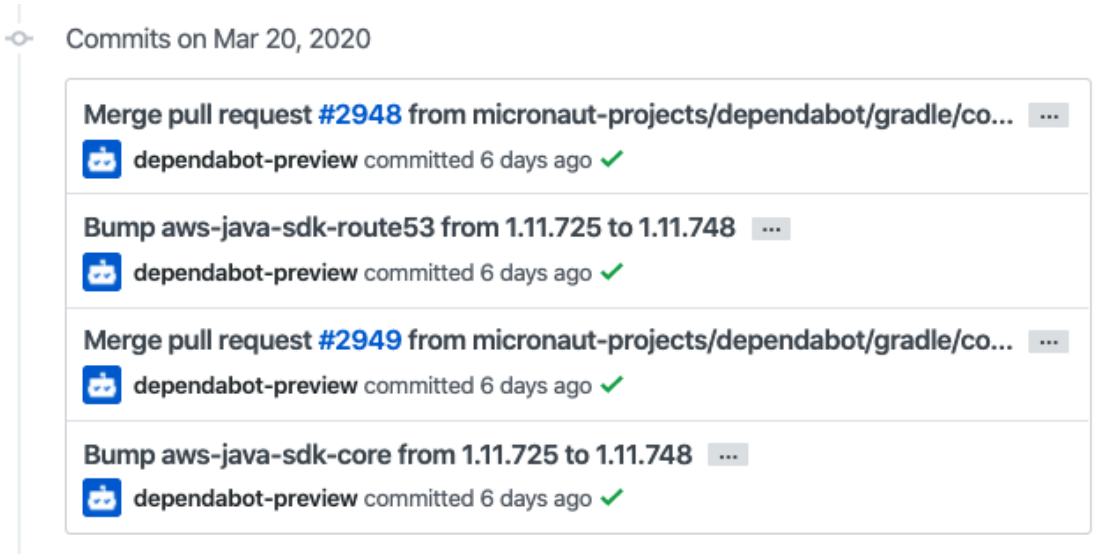


Figure 13.10: Dependabot Commits

13.4.2.4 Performance Tests

As discussed in the previous blogposts, performance is an important quality criterion for Micronaut and, therefore, it must be measured. Especially interesting is the comparision to competitor framweork which the Team has done in this blogpost.

13.4.3 Development Activity

13.4.3.1 Analysis of recent activity

To identify the activity hotspots we analyzed which modules had the most changes in recent coding activity. For that, we measured coding activity in two ways: in terms of the number of files changed, and in terms of the number of lines added and removed.

We use PyDriller[31], a Python framework to traverse commits in a certain period of time. We aggregated the file changes per module for the past three months and compared it to the changes in the last year to see if any hotspots have changed over time.

Looking at the number of file changes per module over the past three months, we see that the `inject-java` module is by far the most active module, followed by `runtime` and `inject`. The rest of the modules have seen significantly less activity.

[31] Spadini, Davide and Aniche, Maurício and Bacchelli, Alberto. PyDriller: Python Framework for Mining Software Repositories. https://pydriller.readthedocs.io/

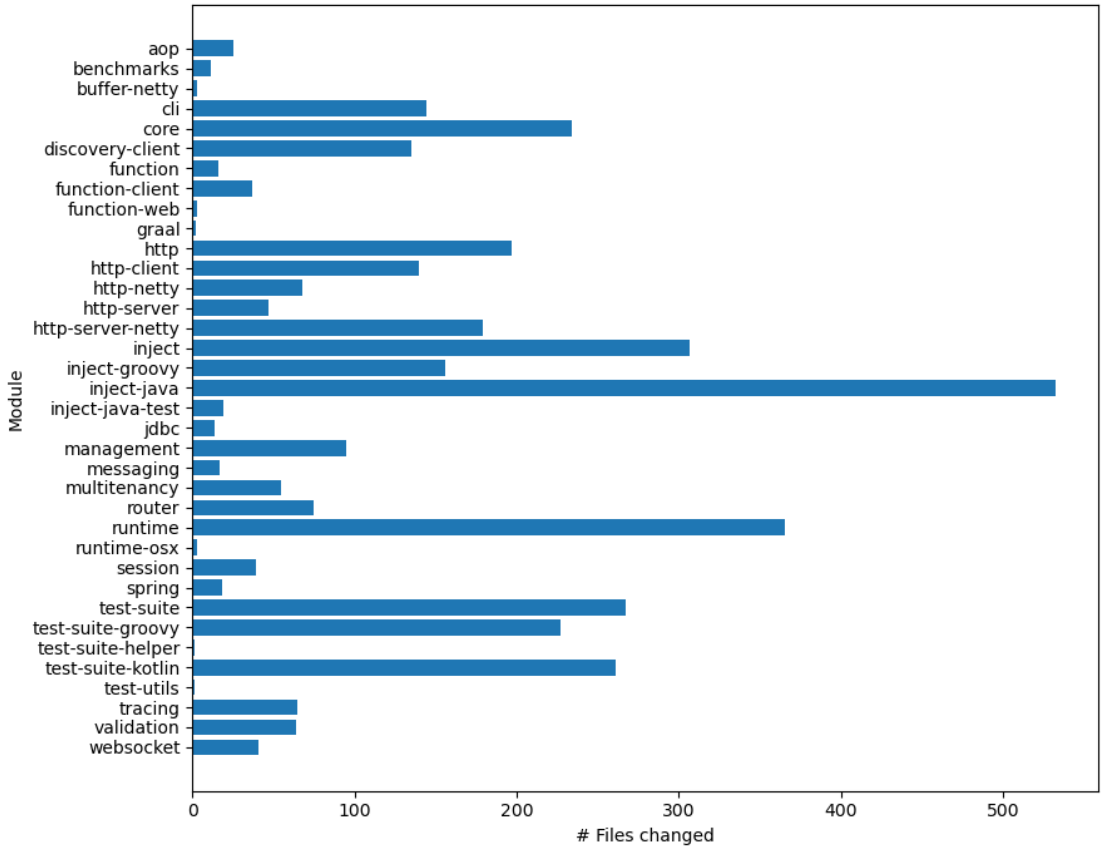


Figure 13.11: The number of file changes per module over the past three months

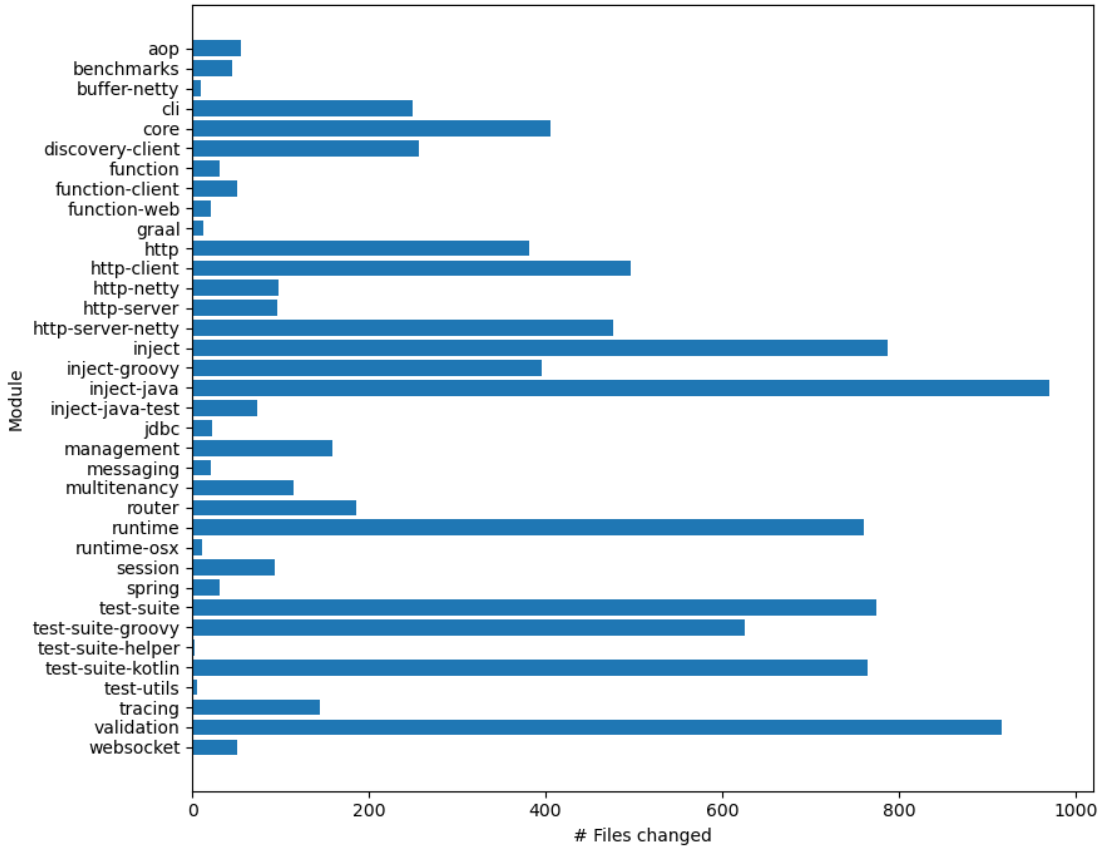


Figure 13.12: The number of file changes per module over the past twelve months

If we compare to the activity of the last year, we see that `inject-java`. `runtime` and `inject` still belong among the most active hotspots. However, they are now accompanied by `test-suite`, `test-suite-kotlin` and `validation`. It is noteworthy that the modules that recently have been the most active are also hotspots over a longer timespan.

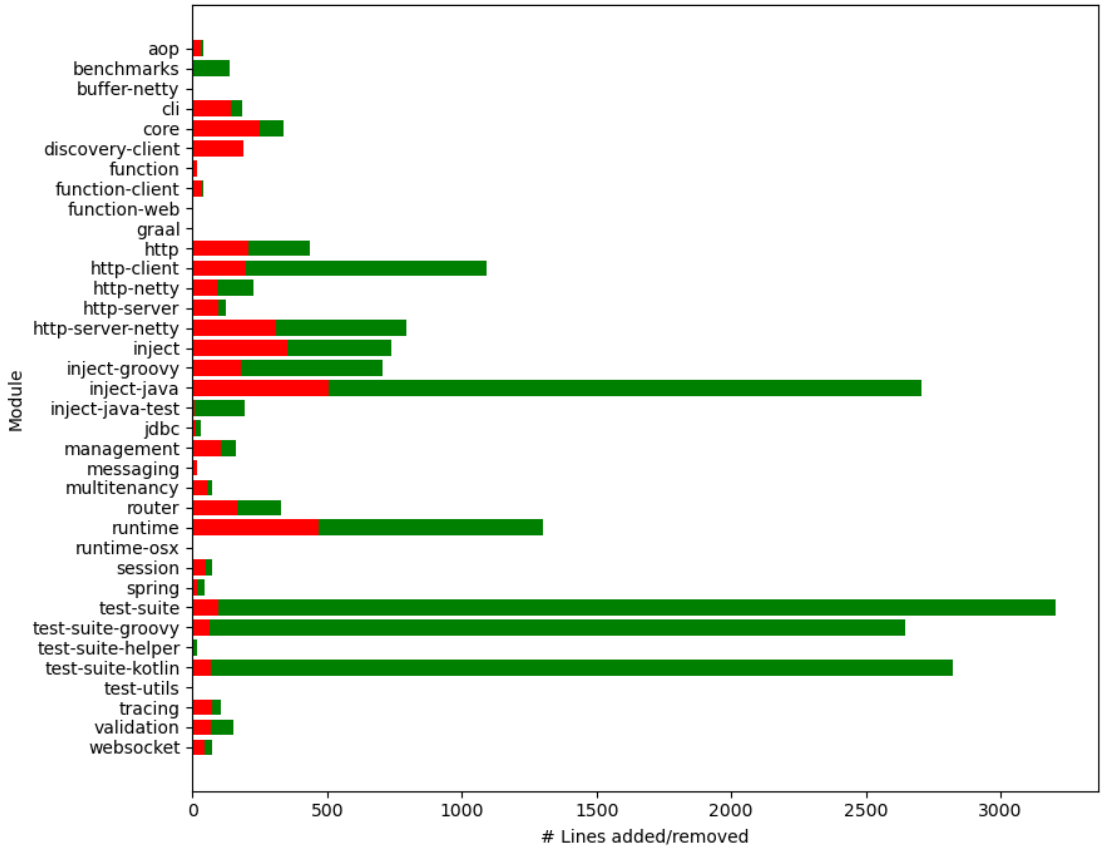


Figure 13.13: The number of line changes per module over the past three months

In contrast, if we consider coding activity from a lines added/removed perspective, the test-suites have been the most active, as well as again the `inject-java` module. Again comparing to the activity over the past 12 months, the same trend holds, but now with `validation` being even more active than the aforementioned modules.

Depending on the metrics, we end up with different recent activity hotspots. For file changes, these are `inject-java`, `runtime`, and `inject`, while for line changes the hotspots are `test-suite`, `test-suite-kotlin`, `inject-java`, and `test-suite-groovy`. However, we can say for certain that `inject-java` is a hotspot as it is amongst the most active modules for both metrics.

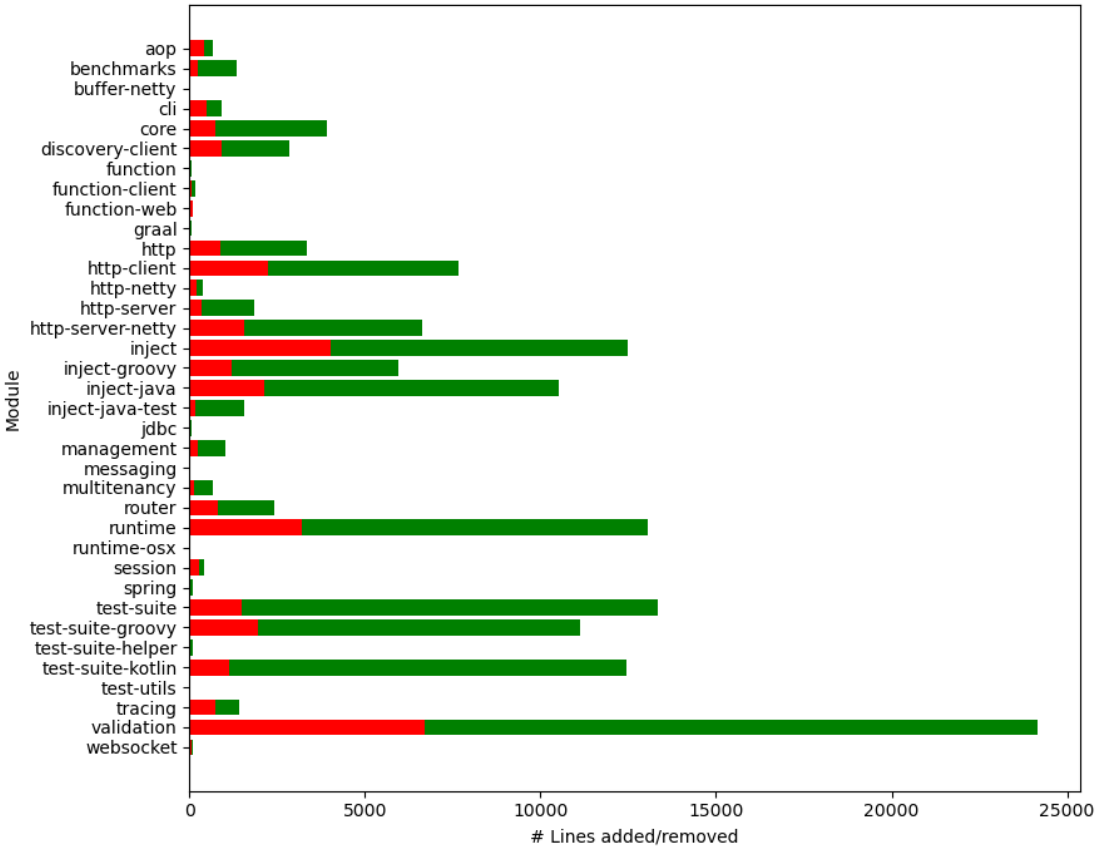


Figure 13.14: The number of line changes per module over the past twelve months

13.4.3.2 Future activity (Roadmap)

Work on the roadmap has progressed rapidly since our first mention of Micronaut's roadmap. At the time of writing, a lot of the features from the discussed 2.0.0 milestone have been implemented. Several issues remain in the newly updated 2.0.0.M2 milestone.

We will identify which architectural components are likely to be affected by the upcoming features on this milestone. For each issue, we list the modules that are likely to be affected. We determined this by identifying if there is a specific class or module mentioned in the issue, and otherwise by searching in the code base for code related to this issue. For instance, the issue #1421 concerns the behavior of the bean context, therefore, the class that is probably affected is `DefaultBeanContext.java`, which belongs to the `inject` module.

- #1418 Fail compilation if a concrete class has an introduction advice annotation: `inject-java`
- #1421 Throw an exception if the bean context is used after its closed: `inject`
- [\#1859 Don't generate beans if @inject is present but the type is not declared a bean](https://github.com/micronaut-projects/micronaut-core/issues/1859): `inject-java`
- #1885 Register beans with external annotations: `inject`
- #1969 Remove the use of Jackson for MapToObjectConverter: `runtime`
- #2177 Separate cache core from runtime: `runtime`
- #2686 Support random available port for management endpoints: `management`
- #2732 Refactor server filters to be registered and found more like client filters: `http-server`
- [\#2790 Don't allow generated property combinations from environment variables to be used by @EachProperty](https://github.com/micronaut-projects/micronaut-core/issues/2790): `inject`
- #2809 Support RxJava 3: `runtime`
- #2811 Support Groovy 3: `inject-groovy`, `test-suite-groovy`
- #2869 Refactor multipart upload route execution: `http-server-netty`
- #2953 Specifying –test spock errors create-app in 2.0.0.M1: `cli`
- [\#2959 Can't introduce own @Client annotation](https://github.com/micronaut-projects/micronaut-core/issues/2959): `http-client`
- [\#2958 Invalid type bound from @ConfigurationProperties](https://github.com/micronaut-projects/micronaut-core/issues/2958): `inject`

The modules that occur most often are `inject`, `inject-java`, and `runtime`. It is interesting to see that this is in line with the hotspots in the activity of the last three months, as identified in the previous section.

13.4.4 Quality Assessment

In the above section we concluded that `inject`, `runtime`, and `inject-java` are likely to be affected soon. As explained in our previous blogpost `runtime` and `inject` are central modules, as well as the `inject-java` module which is executed at compile-time and processes the annotations.

We have analyzed the development state at the time of writing using SonarQube, a static code analysis tool which scans the codebase using a preconfigured set of rules, calculates file-level metrics and display them on a final dashboard. The sonarQube evaluation is similar to the checkstyle analysis mentioned in CI process section but contains a larger set of rules.

13.4.4.1 Technical Debt assessment

Technical debt reflects the extra development cost that arises when code that is easy to implement in the short run is used instead of applying the best overall solution.[32] Reducing technical debt in a system helps to improve the maintainability of the source code as well as to reduce the amount of effort required to develop new functionalities.

The figure below shows the results from the analysis. SonarQube estimates that it takes 81 days to fix all technical debt. The metric "day" is based on the Software Quality Assesment based on Lifecycle Expectations Methodgy.[33]

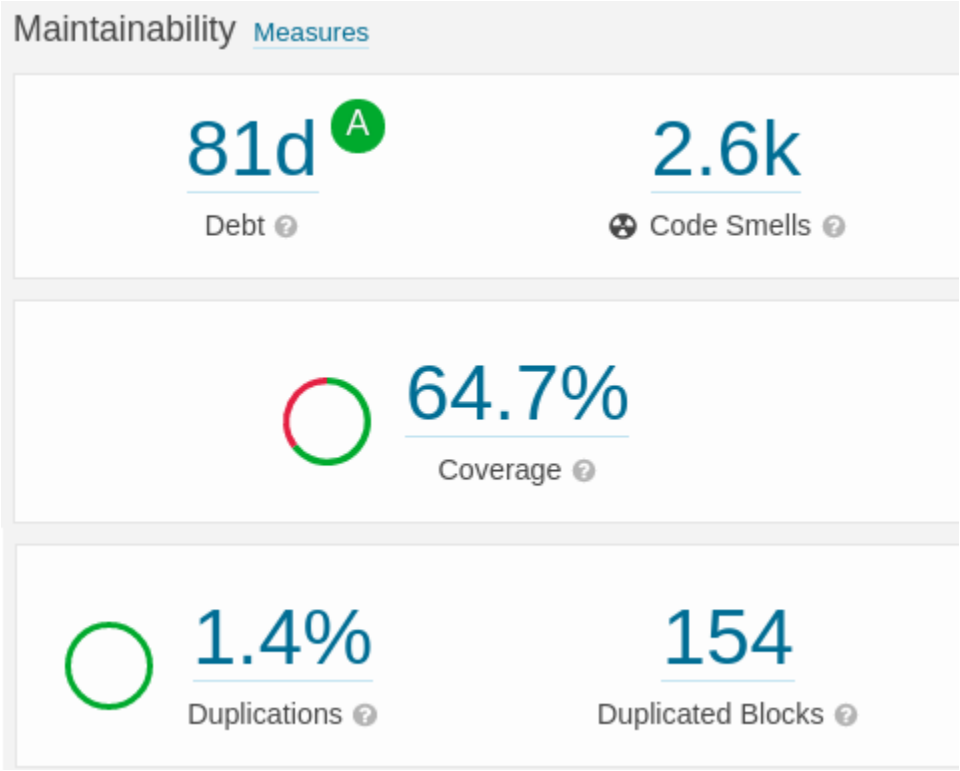


Figure 13.15: Technical debt analysis by SonarQube

SonarQube also gives us some insights about the number of code smells, which is 2.6k, from there 646 are critical and 905 major, however since the set of rules where not customize for Micronaut it might be possible

[32] Martin Fowler. Technical Debt. https://martinfowler.com/bliki/TechnicalDebt.html

[33] Jean-Louis Letouzey. The SQALE Method for Evaluating Technical Debt. In 3rd International Workshop on Managing Technical Debt. Zurich, Switzerland. 2012.

that some of the SonarQube violations are not good predictors of fault-proneness. The number of identical lines of code is about 1.4%, which is almost insignificant. The test coverage of 64.7% was discussed in the previous section.

In the inject `inject` module, which consists of 22346 lines of code, SonarQube has detected 540 code smells and gives an estimation of 14 days to fix them.

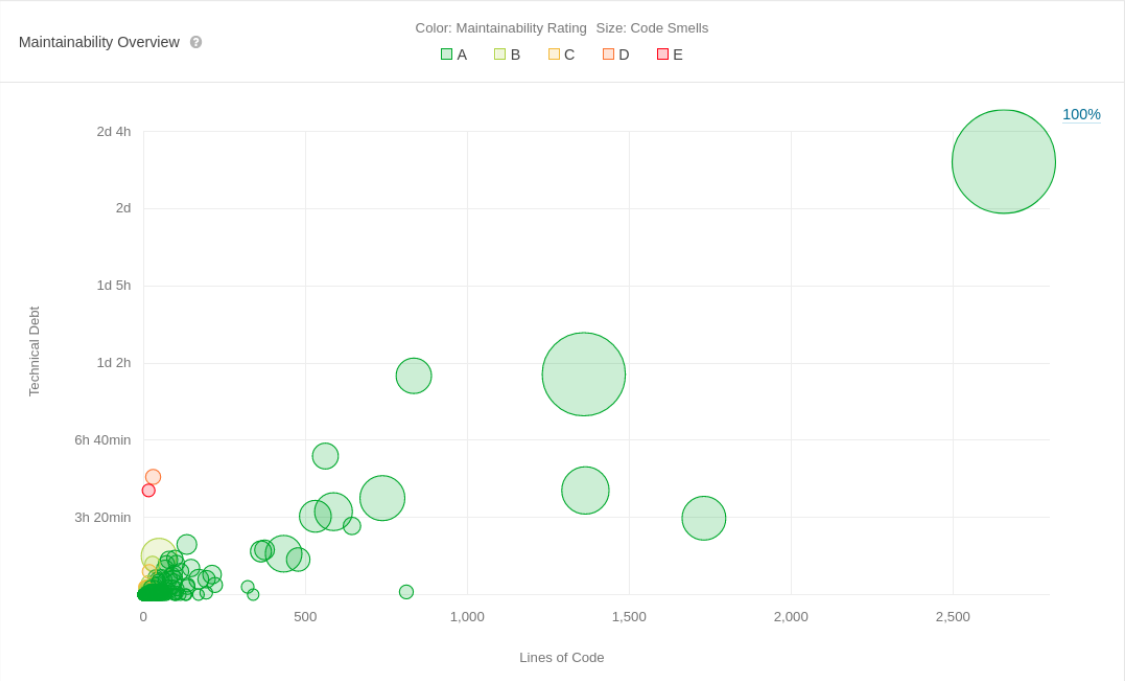


Figure 13.16: Technical debt for the inject module

In the inject `runtime` module, which consists of 12367 lines of code, SonarQube has detected 299 code smells and gives an estimation of 7.5 days to fix them.

In the inject `inject-java` module, which consists of 5876 lines of code, SonarQube has detected 94 code smells and gives an estimation of 8 days to fix them.

13.4.4.2 Potential Refactoring

For improving the quality of the three modules, `inject`, `runtime` and `inject-java` we suggest to solve first the most recurrent code smells which are:

Reduce the cognitive complexity of methods (117 code smells)

To keep code maintainable, the goal is to keep functions simple and readable so they can be understood intuitively.[34] Cognitive complexity is a metric that tries to express if a method is understandable. Therefore, the rule suggests as soon as the complexity is above a certain threshold, the method should be refactored.

[34] KISS principle. Wikipedia https://thevaluable.dev/kiss-principle-explained/

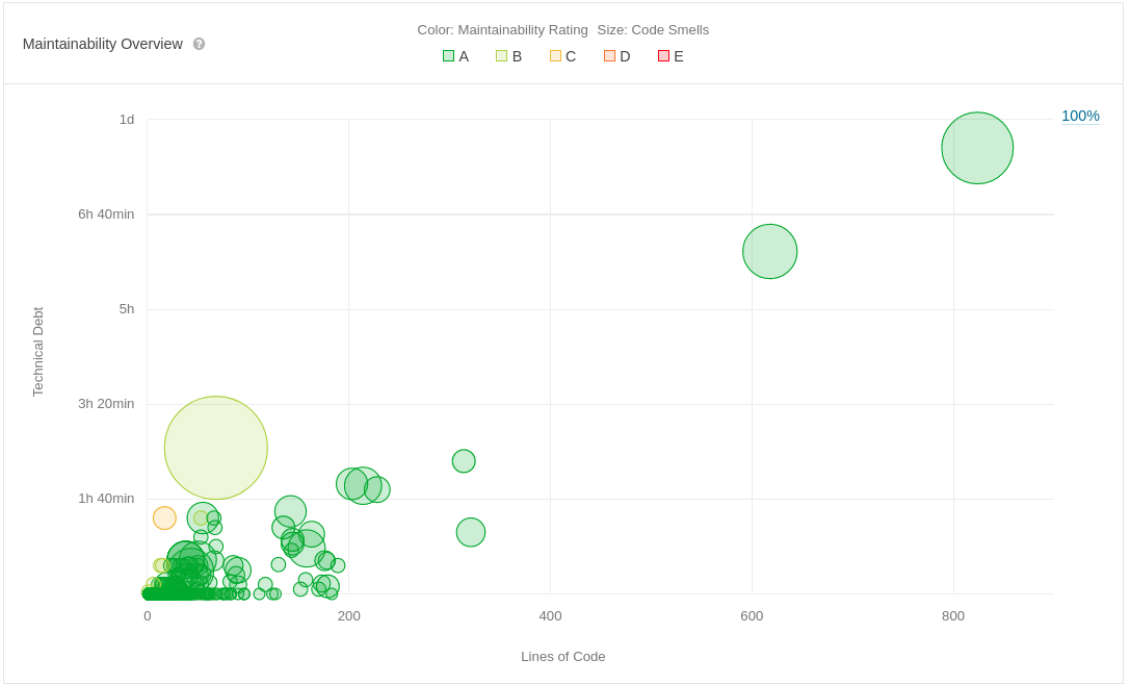


Figure 13.17: Technical debt for the runtime module

Good refactoring strategies are: reducing nestings, extracting parts into other methods and using fewer variables.[35]

Functional interfaces should be as specialized as possible (84 code smells)

It is important to use the specialized functional interfaces from the Java standard library so that the Java compiler can automatically convert the parameters into primitive types.[36] For refactoring it, the generic interface (e.g. `Function<Integer,R>`) must be replaced with the specific one (`IntFunction<R>`).

Local variables should not shadow class fields (65 code smells)

When redefining a variable that is already defined in an outer scope it's hard for the reader to see to which value the variable relates. To avoid this, variables in the inner scope should be given a different name.[37]

In conclusion, the results presented by SonarQube have little room for improvement. This proves that Micronaut CI processes and development mindset have a positive impact on the overall quality.

[35] Cognitive Complexity and its effect on the code. StackOverflow https://stackoverflow.com/questions/46673399/cognitive-complexity-and-its-effect-on-the-code

[36] Autoboxing and Unboxing. Oracle Docs https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html

[37] Do not shadow or obscure identifiers in subscopes.https://wiki.sei.cmu.edu/confluence/display/java/DCL51-J.+Do+not+shadow+or+obscure+identifiers+in+subscopes

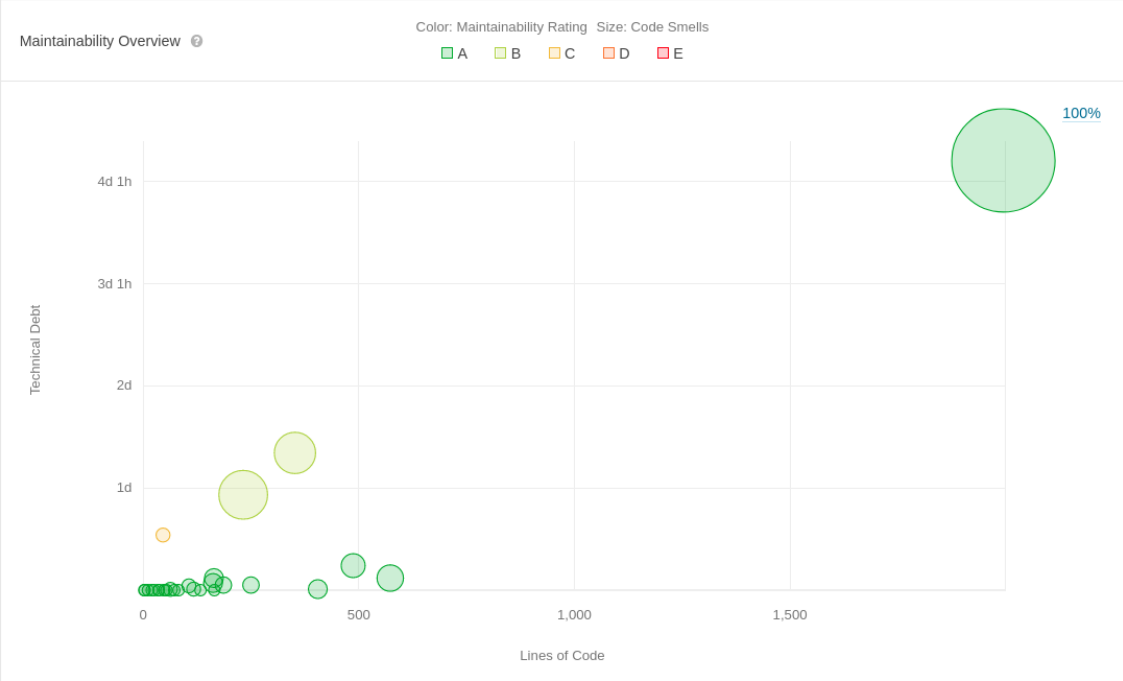


Figure 13.18: Technical debt for the inject-java module

```
// Noncompliant
public class Foo implements Supplier<Integer> {
    @Override
    public Integer get() {
      // ...
    }
}
```

refactor to

```
// Compliant solution
public class Foo implements IntSupplier {
  @Override
  public int getAsInt() {
    // ...
  }
}
```

Figure 13.19: Functional interfaces should be as specialized as possible

```
class Foo {
  public int myField;

  public void doSomething() {
    int myField = 0;
    ...
  }
}
```

refactor to

```
class Foo {
  public int myField;

  public void doSomething() {
    int newVar = 0;
    ...
  }
}
```

Figure 13.20: Rename variable in inner scope to avoid shadowing class fields

13.5 Micronaut - A perfect Microservice framework?

Whereas the previous blogposts focused on the internal architecture view of Micronaut, we decided to change now the perspective in our last blogpost and look at Micronaut through the eyes of a software architect who wants to build a microservice application. Micronaut claims to be *"a modern, JVM-based, full-stack framework for building modular, easily testable microservices and serverless applications"*.[38] Therefore, we explore how Micronaut solves common challenges when developing microservices.

13.5.1 Introduction

"Microservices" are an architectural approach, where a complex system is split into smaller independent applications that can be deployed independently. Ideally, these microservices are small in size and fulfill exactly one aspect of the business domain (Bounded Context). This approach, enables distributed teams to work independently on different microservices, allows to scale microservices individually and enforces strict modularity which increases maintainability in the long run. However, it comes with some downsides. Generally spoken, through the distributed nature of the system the complexity of the interactions increases and managing the complete system becomes more difficult. This results, for example, in more complex integration strategies and challenges in operations. Furthermore, guaranteeing availability and reliability is much more difficult, because the system must be able to handle network latency and failures of individual services.[39]

To really benefit from those advantages and to be able to manage the additional complexity several best practices for implementing microservices exist in the community. Eberhard Wolff describes many of those in Part 3 *Implementing Microservices* of his book *Microservices: Flexible Software Architecture*[40]. In this blogpost, we will take his recommendations and guidelines and how they can be realised with Micronaut. This includes general architecture concepts, communication, integration and operation strategies.

13.5.2 Architecture Concepts of Microservices

As Wolff describes, generally the microservice architecture allows using a different technology stack for every microservice. However, to fulfill certain quality properties in a microservice-based system, it is useful to make certain technical decisions on the level of the entire system. Those decisions influence the individual architecture and enforce specific functionality. This section discusses various, common high-level decisions and explains Micronaut's approaches to fulfill them.

13.5.2.1 Configuration

Typically, configuring a microservice-based system is more tiresome than configuring a monolithic system, because all individual microservices need the right parameters. Furthermore, to be able to run the same application in different environments (e.g. test, production) the configuration parameters should be stored externally from the source code.[41]

[38] Micronaut. Website. https://micronaut.io/

[39] Wikipedia. Microservices. accessed 7th of April 2020 https://en.wikipedia.org/wiki/Microservices

[40] Eberhard Wolff. Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016 https://www.oreilly.com/library/view/microservices-flexible-software/9780134650449/

[41] Alejandro Duarte. Microservices: Externalized Configuration. DZone, 2018 https://dzone.com/articles/microservices-externalized-configuration

Micronaut's mechanism to read external configuration is inspired by Spring Boot and Grails and allows injecting configuration parameters as beans from different sources. Firstly, these sources include property files, environment variables, and Java system properties. Furthermore, it allows users to add new sources by implementing the interface `PropertySourceLoader`.[42]

For microservice-based systems, often a central key/value store is used to make the configuration parameters available to all microservices at runtime. Micronaut provides an interface to implement clients to read configuration properties from those systems. Furthermore, it comes with default integrations to widely-used services as the Spring Cloud Config server and HashiCorp Consul.[43]

13.5.2.2 Service Discovery

Microservice instances can be stopped and started at every time which makes it difficult for services to find each other. Therefore, a discovery mechanism is needed, which resolves a service name to an IP address (and port). From the different available technologies probably DNS is the most well-known. In microservice-based systems often a central service registry is used, where different instances register themselves after startup. A potential client can then request the connection information from this registry.[44]

Likewise, this concept is supported by Micronaut and it has intended interfaces for integrating such discovery services. At the time of writing, this includes the popular service registries Eureka and HashiCorp Consul.[45]

13.5.2.3 Security

Naturally, every microservice needs to know which user has triggered the current call and has to authenticate it. Obviously, it does not make sense to validate the password in every microservice. Instead this should be delegated to a central authentication server. To avoid unnecessary load the authentication server often issues an access token, which is attached to all requests and can be validated by the microservices directly.[46] A well-known protocol which follows this idea, is OAuth2.[47] We all know it from websites, where we can login by using another service (e.g. Google).

The Micronaut framework includes a separate project Micronaut Security which adds customizable security solutions to the application. This also includes the support of OAuth2. From the supported authentication providers JSON Web Token (JWT) is worth to mention.[48] This is an industry standard for transferring signed claims (access tokens).[49] As the following example shows, this access token often includes information about the user, its roles and information about the issuer.

13.5.2.4 Resilience

As discussed, in the introduction the failure of a single microservice should have a minimal impact on the availability and reliability of the overall system. To archive this, the individual microservices should

[42] Micronaut Docs https://docs.micronaut.io/latest/guide/index.html
[43] Micronaut Docs https://docs.micronaut.io/latest/guide/index.html
[44] Eberhard Wolff. Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016 https://www.oreilly.com/library/view/microservices-flexible-software/9780134650449/
[45] Micronaut Docs https://docs.micronaut.io/latest/guide/index.html
[46] Eberhard Wolff. Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016 https://www.oreilly.com/library/view/microservices-flexible-software/9780134650449/
[47] RFC 6749. The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6749
[48] Micronaut Security Docs https://micronaut-projects.github.io/micronaut-security/latest/guide/index.html
[49] RFC 7519. JSON WebToken (JWT) https://tools.ietf.org/html/rfc7519

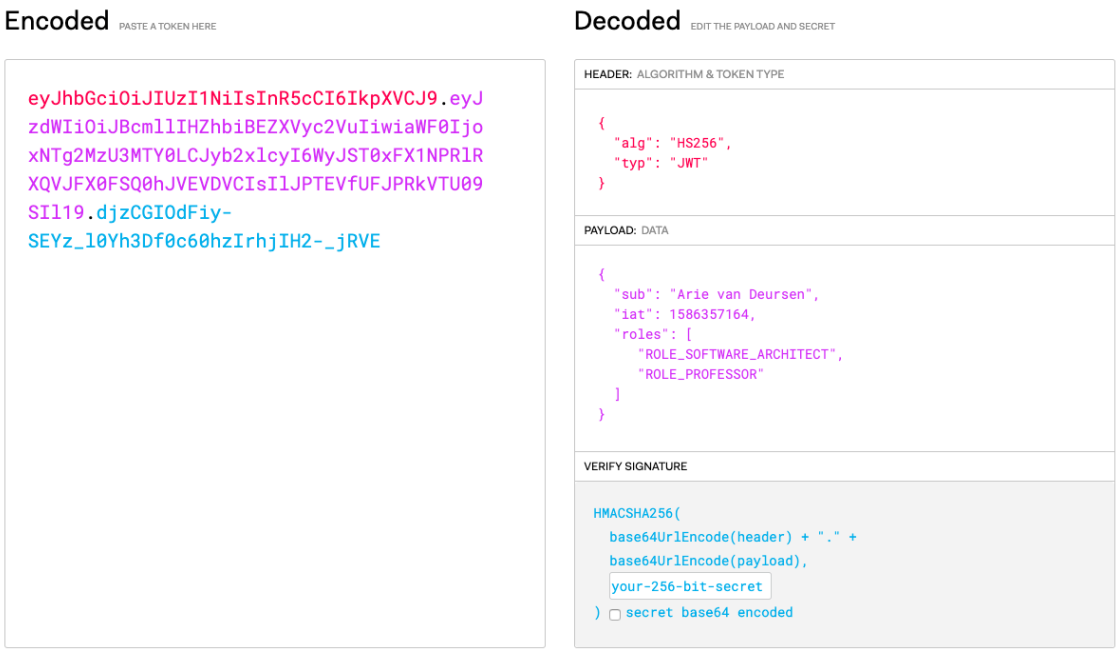


Figure 13.21: Example JSON Web Token with roles

be *resilient* which means that the services accept that failures happen and have mitigation and recovering strategies in place.

One strategy to reach this is the *circuit breaker* pattern. This pattern says, that as soon as a client detects a failure (e.g. Http timeout) it considers the server as down and prevents further calls until it detects the server to be available again.[50] This has the advantage, that the load on the target system is reduced and future calls fail faster. Within Micronaut, this pattern can be enabled with the `@CircuitBreaker` Annotation.[51]

```
@Client("/pets")
@CircuitBreaker
public interface PetClient extends PetOperations {

    @Override
    Single<Pet> save(String name, int age);
}
```

13.5.3 Integration and communication

Microservices should be able to communicate with each other, for which they can be integrated at various levels. *Microservices: Flexible Software Architecture*[52] defines three different levels of integration, namely

[50] Wikipedia. Circuit breaker design pattern. accessed 8th of April 2020 https://en.wikipedia.org/wiki/Circuit_breaker_design_pattern
[51] Micronaut Docs https://docs.micronaut.io/latest/guide/index.html
[52] Eberhard Wolff. Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016 https://www.oreilly.com/library/view/microservices-flexible-software/9780134650449/

the UI level, logic level, and database level.

13.5.3.1 UI level

To integrate microservices at UI level, each microservice provides its own web interface in the form of a single-page-application (SPA), SPA module, or an HTML page. This individual interfaces are then integrated on system level.

13.5.3.2 Logic level

The most noteworthy approaches for logic level integration are REST and messaging.

When integrating through RESTful HTTP, each microservice is identified by its own URI. Microservices can then communicate by invoking each other's endpoints using HTTP methods. The drawback of using RESTful HTTP is its synchronous communication, meaning that an application can be blocked for a long time when waiting for a response to a request.

Communication through messaging is asynchronous, which solves the aforementioned problem. With messaging, a microservice sends a message to a queue or topic. Another microservice can subscribe to the same queue or topic to receive these messages. Using this approach the sender and receiver are decoupled. There are a variety of messaging technologies available, including AMQP, JMS, and Kafka.

13.5.3.3 Database level

There are several approaches to share data between microservices when integrating them at the database level. A widespread approach is to share a database schema between the microservices. The problem of this approach, however, is that microservices are no longer able to change their internal data representation without affecting other microservices.

13.5.3.4 Micronaut's approach

Integration at the UI level happens in the frontend of an application, while integration at the database level using data replication should be supported by the DBMS. Both are beyond the scope of Micronaut and hence Micronaut only supports integration at the logic level, through RESTful HTTP and messaging.

13.5.3.4.1 RESTful HTTP We will illustrate how integration through RESTful HTTP works using an example. We have two microservices, a book catalogue service and a book recommendation service, with the latter requesting books from the former. The book catalogue service can define an endpoint by defining a controller class using the `@Controller` annotation.[53]

```
@Controller("/books")
public class BooksController {

    @Get("/")
    List<Book> index() {
        Book releaseIt = new Book("1680502395", "Release It!");
        Book cd = new Book("0321601912", "Continuous Delivery:");
        return Arrays.asList(releaseIt, cd);
```

[53] Micronaut Docs https://docs.micronaut.io/latest/guide/index.html

```
    }
}
```

Now the book recommendation service can define a client for the book catalogue by defining a client class with the `@Client` annotation specifying the URI or identifier of the microservice.

```
@Client("http://localhost:8081")
interface BookCatalogueClient extends BookCatalogueOperations {

    @Get("/books")
    Flowable<Book> findAll();
}
```

Using this client the book recommendation service can communicate with the book catalogue service, for instance, to request a list of books.

13.5.3.4.2 Messaging For communication through messaging, Micronaut provides integrations with the message brokers Kafka and RabbitMQ, which is an implementation of the AMQP standard. Micronaut's integration with these message brokers makes communication very simple.[54]

Using the same example as before, the book catalogue service defines a Kafka client to send messages using `@KafkaClient`. The `@Topic` annotation specifies the topic the messages are sent to.

```
@KafkaClient
public interface BookClient {

    @Topic("books")
    void sendBook(@KafkaKey String isbn, String title);

    void sendBook(@Topic String topic,
                  @KafkaKey String isbn, String title);
}
```

Now the book recommendation service can receive messages by defining a Kafka listener using `@KafkaListener`. The `@Topic` annotation indicates which topic the listener wants to receive messages from.

```
@KafkaListener(offsetReset = OffsetReset.EARLIEST)
public class BookListener {

    @Topic("books")
    public void receive(@KafkaKey String isbn, String title) {
        System.out.println("Got Book - " + isbn + " " + title);
    }
}
```

Messaging through RabbitMQ works very similar to Kafka. The client is defined using `@RabbitClient`, and the listener using `@RabbitListener`. The queue to send messages to is defined with `@Binding`, and the queue to receive from with `@Queue`.

[54] Micronaut Kafka Docs https://micronaut-projects.github.io/micronaut-kafka/latest/guide/

```
@RabbitClient
public interface BookClient {

    @Binding("books")
    void send(byte[] data);
}

@RabbitListener
public class BookListener {

    @Queue("books")
    public void receive(byte[] data) {
        System.out.println("Received "
                + data.length + " bytes from RabbitMQ");
    }
}
```

13.5.4 Operations and Continuous Deployment of Microservices

Operating an established number of services is one of the central challenges when working with microservices because of the many deployable artifacts that need to be survailled.

13.5.4.1 Logging and Monitoring

To easily retrieve information from the events that occur in our system, like errors, status of the running applications or user-centered statistics we can make use of logging and monitoring strategies. To log our events, the most common solution is to have a central infrastructure that gathers all logs from the running microservices. This avoids checking the logs of every service individually. The continously monitoring of logs provides feedback that is helpful for operations but also for developers and users of the system.

13.5.4.2 Micronaut's approach

The way Micronaut supports logging and monitoring of microservices its heavily inspired by Spring Boot and Rails. By adding the `micronaut-management` dependency we can monitor our service using *endpoints*, which are special URIs that return details about the health and state of the application.

Micronaut support two types of endpoints: custom and built-in endpoints. The custom endpoints are created by annotating a class with the `@Endpoint` annotation and one id. Then, a method from the class can be annotated with `@Read`, `@Write` and `@Delete` to respectively return a respond to `GET`, `POST` and `DELETE` requests. The image below illustrates a custom endpoint `/date` and its different responses.

Besides the custom endpoints, the `micronaut-management` dependency offers built-in endpoints which are shown in the table below.

For our case study the most relevant ones are the `MetricsEndpoint` accessible by the `/metrics` URI. The development team has integrated Micrometer so it can gather the metrics from the `/metrics` endpoint.[55]

[55] Micronaut Micrometer Integration https://micronaut-projects.github.io/micronaut-micrometer/latest/guide/

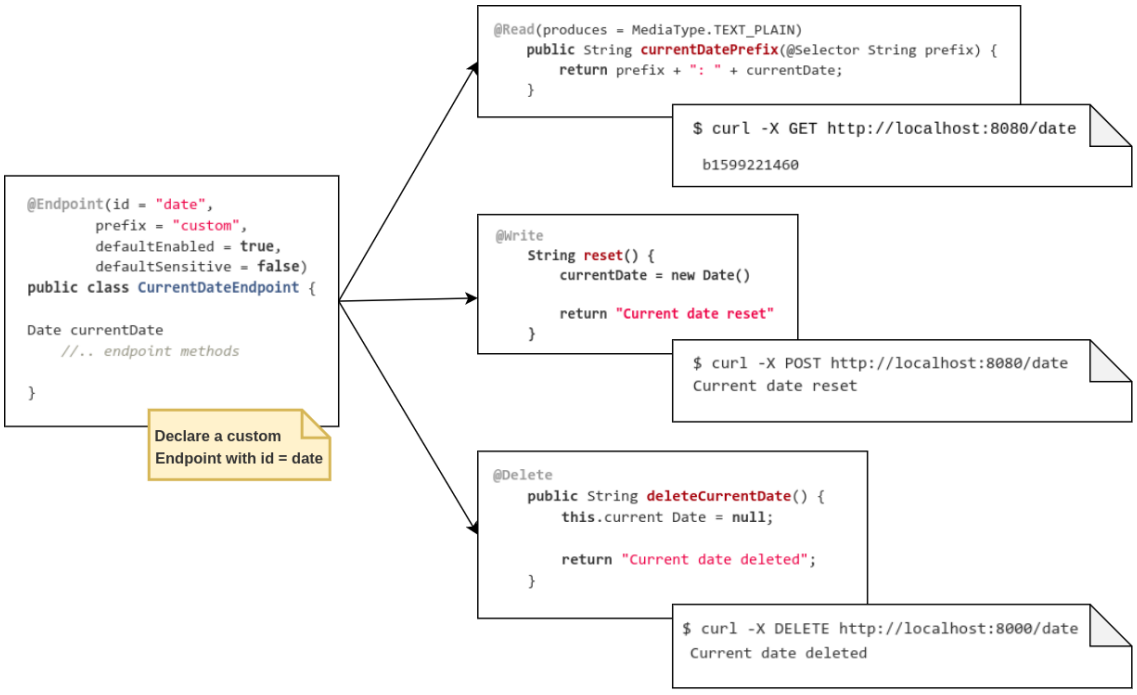


Figure 13.22: Microanut's custom endpoint

Additionally, through the integration with Elasticsearch, Micronaut can forward all logs to this log aggregating system.[56]

13.5.4.3 Deployment of microservices

An independent deployment is a central goal in a microservices architecture. To achieve a good level of isolation, it is best to run each service on a virtual machine. But when this case is not possible and different services are running in the same virtual machine, the deployment of a microservice can generate a high load or introduce changes that also concern other microservices.

There are two approaches to ensure an individual deployment: using virtualization technologies like containers or deploy the microservices in a cloud, where all details of the actual infrastructure are hidden from the application.

13.5.4.4 Deploying a Micronaut Application in a Docker Container

A Micronaut application is packaged in a uber jar and can be therefore executed on every JVM. But, because Micronaut itself does not rely on reflection or any dynamic classloading its deployment can get advantage of running in GraalVM, a Java VM that improves the startup time and reduces the memory footprint.

The best approach following the Micronaut team guideline is to construct a GraalVM native image and deploy it inside a Docker container.

[56] Micronaut Elasticsearch Integration https://micronaut-projects.github.io/micronaut-elasticsearch/latest/guide/index.html

Endpoint	URI	Description	Disabled by default
BeansEndpoint	/beans	Returns information about the loaded bean definitions in the application.	no
HealthEndpoint	/health	Returns information about the “health” of the application.	no
InfoEndpoint	/info	Returns static information from the state of the application.	no
LoggersEndpoint	/loggers	Returns information about available loggers and permits changing the configured log level.	no
MetricsEndpoint	/metrics	Return the application metrics. Requires the micrometer-core configuration on the classpath.	no
RefreshEndpoint	/refresh	Refreshes the application state.	no
RoutesEndpoint	/routes	Returns information about URIs available to be called for your application.	no
EnvironmentEndpoint	/env	Returns information about the environment and its property sources.	no
ThreadDumpEndpoint	/threaddump	Returns information about the current threads in the application.	no
CachesEndpoint	/caches	Returns information about the caches and permits invalidating them.	yes
ServerStopEndpoint	/stop	Shuts down the application server.	yes

Figure 13.23: Microanut’s built-in endpoints

For this, first we need to add the `svm` and `graal` dependencies into our Micronaut application:

```
dependencies {
    ...
    compileOnly "org.graalvm.nativeimage:svm"
    annotationProcessor "io.micronaut:micronaut-graal"
}
```

Then to simplify the building process we need to create a `native-image.properties` file in the directory `src/main/resources/META-INF/native-image` with the following parameters:

```
Args = -H:IncludeResources=logback.xml|application.yml \
       -H:Name=<name> \
       -H:Class=<package.<main-class>>
```

After this, we have to create a Dockerfile to assemble our image with the running application inside:

```
FROM oracle/graalvm-ce:20.0.0-java8 as graalvm
#FROM oracle/graalvm-ce:20.0.0-java11 as graalvm # For JDK 11
RUN gu install native-image

COPY . /home/app/<name>
WORKDIR /home/app/<name>

RUN native-image --no-server -cp build/libs/complete-*-all.jar

FROM frolvlad/alpine-glibc
RUN apk update && apk add libstdc++
EXPOSE 8080
COPY --from=graalvm /home/app/<name>/<name> /<name>/<name>
ENTRYPOINT ["/<name>/<name>", "-Xmx68m"]
```

Finally this docker container can be easily deployed in different cloud-hosting services. Such as AWS EC2 Container Service, or Google Compute Engine.

13.5.5 Conclusion

As this blogpost shows, Micronaut's functionality and architecture has considered all discussed best practices and patterns for implementing microservices. Because it is not necessary that all microservices have the same technology stack, it is easy to implement individual services with Micronaut and integrate them into existing systems. Herby the included 3rd-party integrations help.

Certainly, Micronaut does not provide as much functionality or integrations as Spring Boot, although, it has one key advantage. As discussed in the previous blogposts, its AOT compiling approach leads to a shorter startup time and low memory consumption. These quality properties allow fast and cost-efficient scaling and make the applications ideal for cloud environments.

Concluding, we can say Micronaut can keep its promise and is a good fit for microservice applications. Especially, with its AOT compiling approach it has huge advantages compared to other frameworks on the market.

Chapter 14

MuseScore

Musicians are creators. They engage, individually or in groups, in listening, playing, modifying and writing/creating music. This music is traditionally transcribed on score paper. However, saving, sharing and collaborating those creations is simplified by digitalisation.

MuseScore is an application that allows musicians to create, share and modify digital scores in many ways; even recording them by playing a digital keyboard. Furthermore, it allows musicians to store all their scores digitally and bring them to performances without hassle.

14.1 Team

Martijn van Meerten



Issa Hanou



Toine Hartman



Robert Luijendijk



14.2 MuseScore: Road to reducing paper use in music industry

MuseScore is an application that allows musicians to create, share and modify digital scores in many ways; even recording them by playing a digital keyboard. Furthermore, it allows musicians to store all their scores digitally and bring them to performances without hassle.

This document aims to provide an architectural analysis by identifying the vision underlying MuseScore. To do so, the end goal of MuseScore is identified, together with its use-case and capabilites. The stakeholders are analysed and the roadmap for the product is evaluated. Finally, a context view shows all the relations of the system.

14.2.1 What MuseScore tries to achieve

The vision of MuseScore is "*to create digital products which will replace paper with the digital format in the processes of writing, learning, publishing, and performing sheet music. We focus on replacing paper all over the process from ideation to publishing and performing scores.*"[1]

MuseScore aims to support musicians, whether they are hobbyists, professionals or educators, in their field. This includes the process of creating music and arrangements, but also learning music theory, harmony and notation. Finally, MuseScore strives to assist its users in publishing their works, learning to play instruments and performing.

In order to do so, they offer[2]:

- Cross-platform applications (including tablet devices) which allow creating and editing music faster than using a pen and paper in addition to a realistic playback
- Online services which provide collaborative work, auto synchronization across all devices and the public catalog/marketplace and finally, a better experience for practicing and performing sheet music

Since only the cross-platform application for music score editing is an open-source system, we will only focus on this application. It is important to note that the open-source application MuseScore is created under GPLv2[3], meaning that the application will remain free of cost, and contributors will remain free to change the software. The musescore.com website where scores are shared is not a part of this essay.

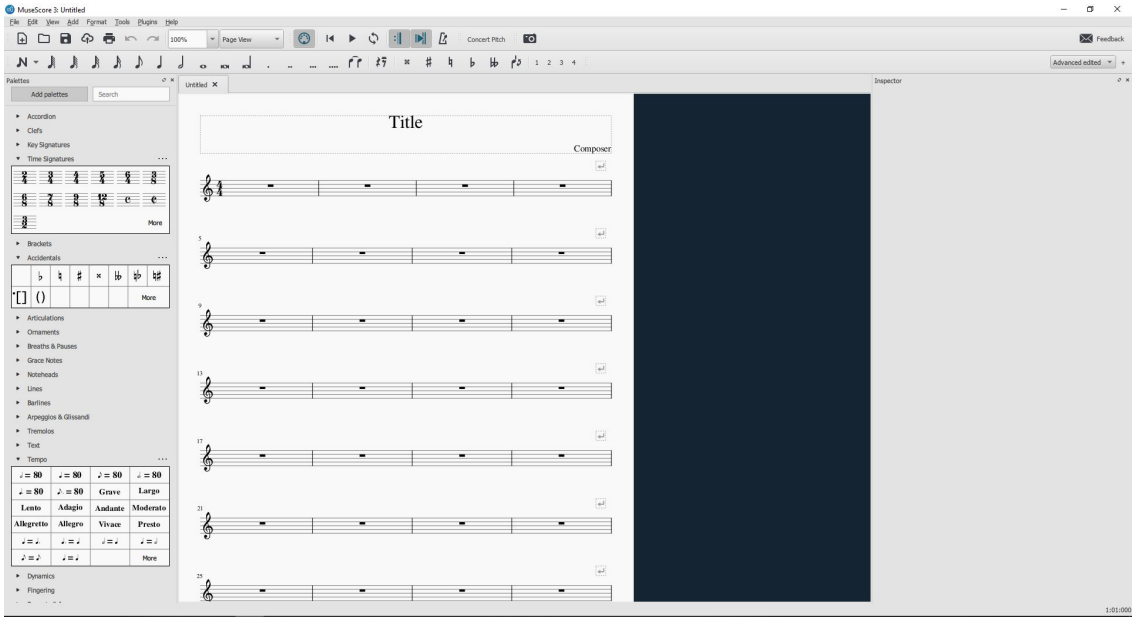
14.2.2 End-user mental model

The MuseScore application is focused on people who want to create their own scores for personal use or to publish them online. This includes both composers who want to put their own music in an actual score; as well as musicians who transcribed, arranged or combined existing music into a new score.

So, the functionality is focused mainly on creating new scores as easily as possible. This entails that users should be able to easily add new notes to the score they are creating. A composer/musician should be able to write music notes in the system just as fast as by hand. MuseScore should offer users a way to freely add music notes to the score, transpose notes in the score and change the length of a note.

The form of the system is a WYSIWYG editor, as seen in the picture below. MuseScore is for writing music scores, like Microsoft Word is for writing text documents. Furthermore, the system should compile to playback the score to the user.

[1] MuseScore. Product Vision. Retrieved February 24, 2020.
[2] MuseScore. Product Vision. Retrieved February 24, 2020.
[3] GNU General Public License, version 2. (website)



Screenshot of the WYSIWYG editor

14.2.3 Key capabilities & properties

MuseScore aims to be a tool that is easy to use for both professionals and amateurs. To establish both goals it should be an all-in-one solution for multiple score-writing tasks; it should integrate well with other tools used by musicians and the software must support multiple workflows.

The main properties that the system strives for, are:

- creating music scores;
- importing and exporting scores;
- handling multiple instruments at the same time;
- supporting notations for different instruments;
- playing back scores;
- allowing MIDI (Musical Instrument Digital Interface, a protocol to easily enter music notes into a computer[4]) inputs

14.2.4 Stakeholders

We describe all the stakeholders of MuseScore by addressing them one by one, and explaining why they are considered a stakeholder of MuseScore.

14.2.4.1 MuseScore & Ultimate Guitar

The main company stakeholder is MuseScore, who have set out the initial vision for the product and the future development of the program. In 2018, the company was acquired by Ultimate Guitar, who can be

[4] no atuhor. *Introduction into MIDI*. 2009. (website).

seen as the main business stakeholder. Their visions of the MuseScore open-source editing system are the same, meaning that the software will continue to be developed under the open-source license GPLv2[5][6][7].

14.2.4.2 Musicians

Musicians are the end-users of the product, we differentiate between hobbyists and professional musicians. Even though both groups use the application in the same way, they have different requirements. A professional musician will uphold the application to a higher standard, than a hobbyist musician. The reason they are still the same stakeholder group is because professional musicians and hobbyists use the application similarly, to create music scores. They are the target audience of MuseScore, and the reason why MuseScore was made.

14.2.4.3 Copyright holders

Many of the music scores that are uploaded are copyrighted, which can lead to illegal use of the music scores. The copyright holder of the score is not necessarily the user uploading the score. Therefore, the actual copyright holder will want to either work together with MuseScore or have influence on MuseScore to see that his/her scores are removed or acknowledged. With the acquisition by Ultimate Guitar, the copyright holders have obtained a bigger stake, because "*Ultimate Guitar has pioneered a successful model for working with music publishers*"[8].

14.2.4.4 GitHub

GitHub is a stakeholder, because the development of MuseScore takes place on GitHub. As successful open-source projects attract many developers to contribute to the product, these people will need to use GitHub to do so, which in turn benefits GitHub.

14.2.4.5 Community

The community of MuseScore is an integral part of the development and use of MuseScore and we have classified the community into three different groups.

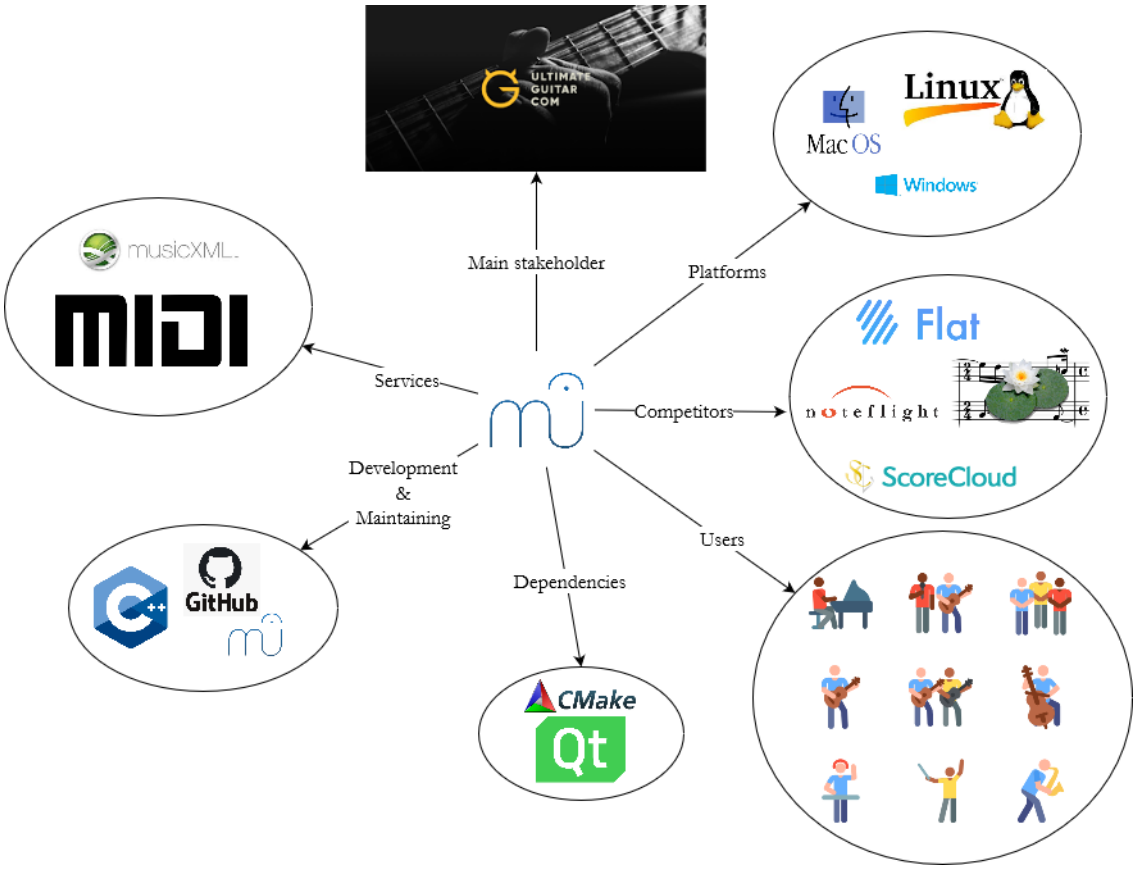
14.2.4.5.1 Contributors These people work to contribute to the project of MuseScore for free. They provide their services for free to MuseScore, thus, MuseScore is partly developed by these contributors.

14.2.4.5.2 Services Services are people or companies that have a business model around MuseScore, this means that they are dependent on the existence of MuseScore and will want MuseScore to do well. These include music educators and people offering trainings and workshops.

14.2.4.5.3 Active Users Active users are different from contributors, because they do not contribute to the software itself, but are helpful online to other users and have built a reputation for themselves on that forum. They might report bugs or recommend new features.

[5] GNU General Public License, version 2. (website)
[6] Eugeny Naidenov. *Welcoming MuseScore to the UG Family!*. February 19, 2018. (link)
[7] Werner, Nicole, Thomas. *MuseScore joins Ultimate Guitar*. February 19, 2018. (link)
[8] Philip Rothman. February 19 (2018). 'Ultimate Guitar acquires MuseScore'. (website)

14.2.5 Current & future context



Context view

MuseScore exists in the context of sheet music, offering a way to keep the scores digitally. In the figure above, we have illustrated the context in which MuseScore operates. This clearly shows the relationships, dependencies and interactions of MuseScore and allows us to easily see connections with these instances.

Where MuseScore is a free application that is available for different platforms, its competitors Flat and NoteFlight only operate as a browser engine. LilyPond differs from MuseScore as music notes are put in like a programming language instead of a drag-and-drop base. ScoreCloud is another application, although it does not support Linux, but offers an App Store application, clearly focusing on Mac users.

The MuseScore code base is written in C++ and maintained on GitHub. However, all communication with the developer community is done on the musescore.org website, where issues are tracked and developer guides are posted.

Having MuseScore interact with tools that listen to the MIDI[9] protocol, greatly improves the usability of the system.

In the future, digital sheet music will most likely become more popular as the use of paper for printing and

[9] no atuhor. *Introduction into MIDI*. 2009. (website).

writing is further diminished[10]. So, tools to create digital scores will be used more frequently. Sheet music has existed for a long time, and the musical notation has hardly changed over the years. Furthermore, the notation is globally accepted, using mostly Italian words. So, as this will probably not change in the future, MuseScore will not need to account for changing its complete base of notating scores.

14.2.6 Roadmap

MuseScore has a detailed roadmap on their website. However, this was last updated two years ago, while MuseScore 3 was released in December of 2018. Therefore, some of the roadmap features in this list could have already been implemented. The most important features that are still in active development are:

- Accessibility
- Notation
- Playback
- Usability

Another document is the known bug list of MuseScore 3. This is more recent, as it was last updated one year ago. However, this does not add any features, but it shows that MuseScore is committed to get MuseScore 3 to the product they envision.

The final document that gives some insights in the roadmap for MuseScore is an interview with their lead designer Martin Keary. In this interview, it becomes evident that he wants to focus more on the user experience of MuseScore, thereby making MuseScore accessible to more people. Accessibility and usability seem to be the two main keypoints in the roadmap that MuseScore is going to focus on in the foreseeable future.

14.3 MuseScore: Views on development

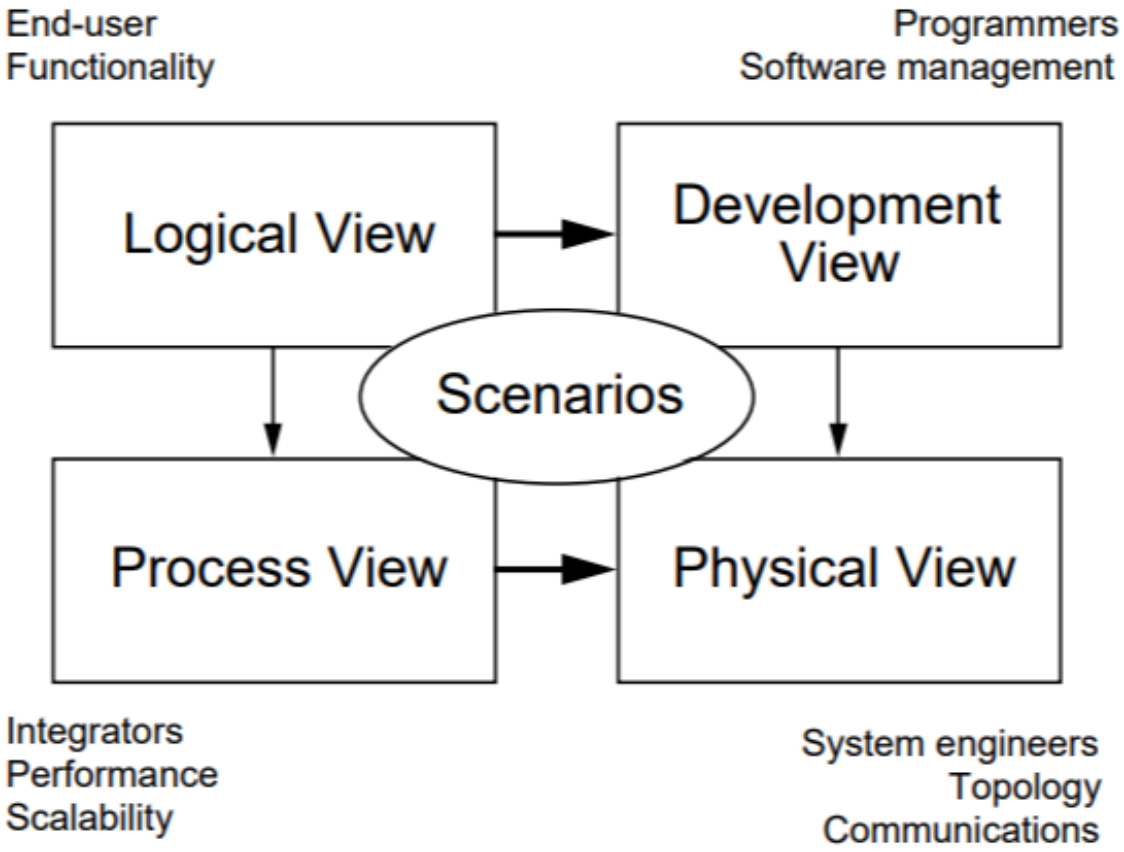
The MuseScore system is based on an architecture that supports all the requirements, both functional and non-functional. Furthermore, each architectural decision made should support the software development process. This essay addresses issues that occur during the design of the architecture and the decisions that were made based on these issues. An outline of the different views of the system will be given, and some will be explored more in-depth.

14.3.1 System views

Several books discuss the architectural views that can be applied to a software system. The ‘classic’ division, as written by Kruchten[11], contains 4+1 views: logical, implementation, process and deployment, which are all dependent on the use-case view.

[10] Latta, G. S., Plantinga, A. J. & Sloggy, M. R. *The Effects of Internet Use on Global Demand for Paper Products*. Journal of Forestry, Volume 114, Issue 4. July 1, 2016. Pages 433–440.

[11] Philippe Kruchten. *The 4+1 View Model of architecture*. IEEE Software 12(6), 1995. (doi, preprint, wikipedia).



The 4+1 views as identified by Kruchten[12]

However, others have argued that more views can be applied to a system: an operational, context, functional, information and concurrency view[13], for example. It is therefore important to determine which views are relevant to the MuseScore system.

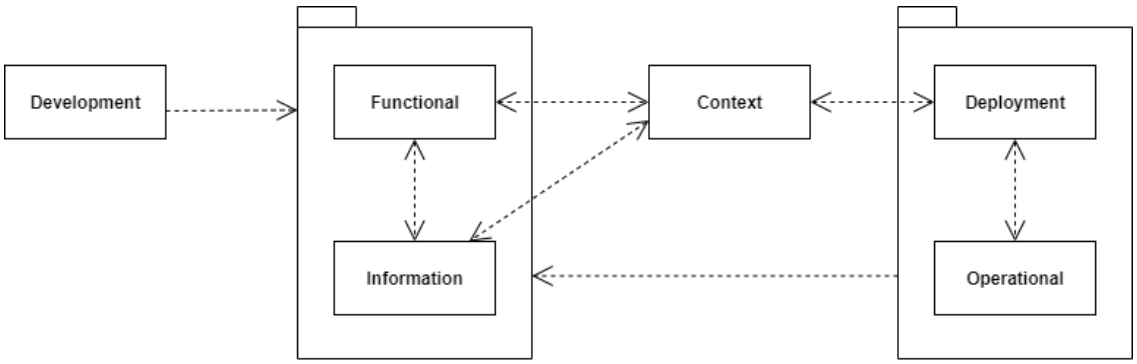
We identify the following views to be relevant to MuseScore:

- Context view, also referred to as use-case view. Without a context, how do you know what you are developing towards, so this is a necessary part of any system architecture. A detailed illustration of this was previously given.
- Development view, which "describes the architecture that supports the software development process"[14].
- Deployment view, which "describes the environment into which the system will be deployed and the dependencies that the system has on elements of it"[15].

[12] Philippe Kruchten. *The 4+1 View Model of architecture*. IEEE Software 12(6), 1995. (doi, preprint, wikipedia).
[13] Nick Rozanski and Eoin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012, 2nd edition.
[14] Nick Rozanski and Eoin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012, 2nd edition.
[15] Nick Rozanski and Eoin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012, 2nd edition.

- Operational view, also called runtime view, which focuses on the production environment when the system is already running.
- Information view, also referred to as the data view, which discusses what and how data is stored and managed in the application.
- Functional view, or logical view. It identifies the functional components of a system and their responsibilities within the system as a whole.

These views are represented in the diagram below, inspired by Rozanski and Woods[16]. We do not identify the concurrency view for the MuseScore system, because we see this as less relevant. The system runs locally, so one user will be using that instance of the system at a time. Therefore, the concurrency is not an issue that needs to be addressed by MuseScore and thus not relevant for the views.



The images and their dependencies sketched

14.3.2 Styles and patterns

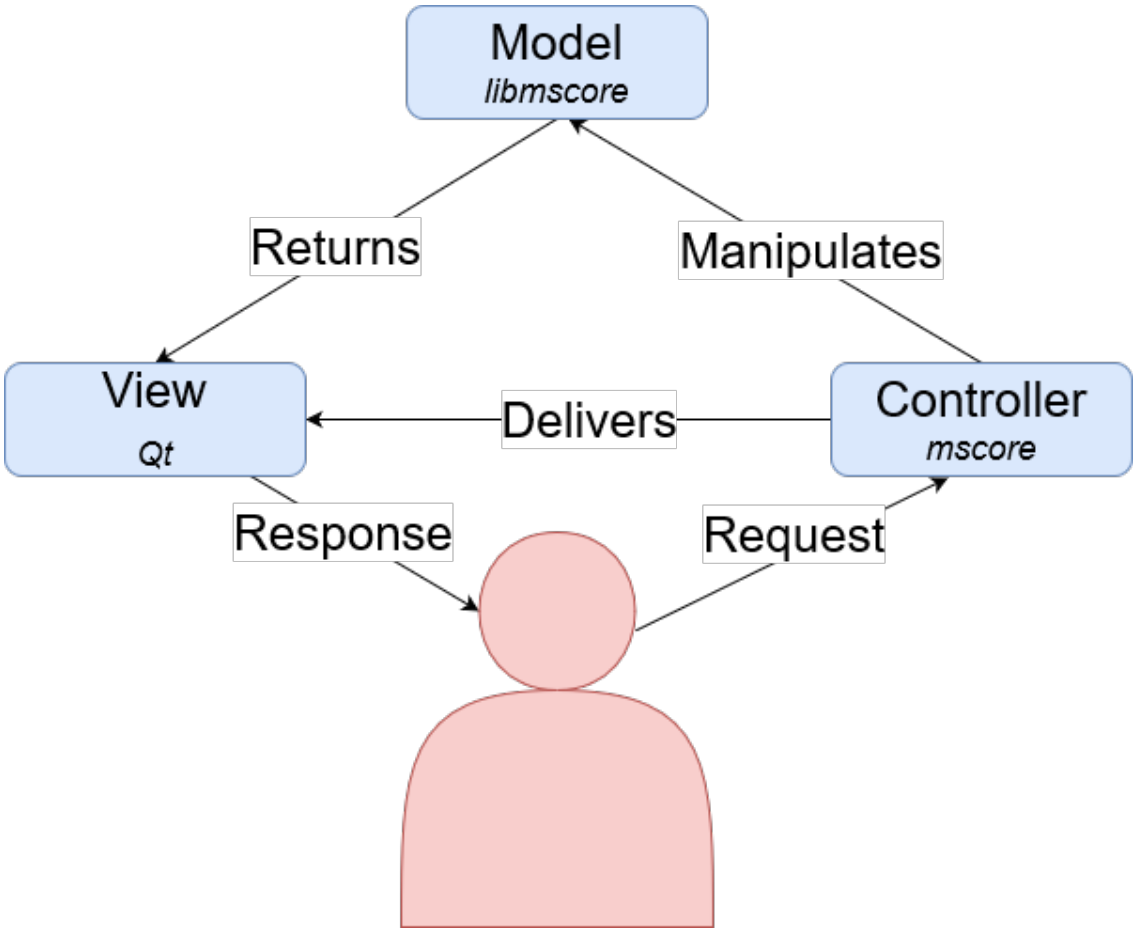
In this section, we identify architectural patterns used to tackle issues in software development. We studied the architecture extensively, but could only find one pattern and no further styles.

MuseScore employs the model-view-controller (MVC) pattern[17]. When using the MVC pattern, the model (data), controller (logic) and view (UI) are loosely coupled. This results in more maintainable software, as changing code in one module does not necessarily affect the code in another module.

Although the MVC pattern should facilitate a loosely coupled codebase, the MuseScore source code has not separated the three rigorously. The view of MuseScore is built in Qt, which is a framework for UI development in C++. The package `mscore` contains most of the controller classes, however, a lot of UI is implemented here as well. The package `libmscore` includes the model classes, but also some controller functions.

[16] Nick Rozanski and Eoin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012, 2nd edition.

[17] Gamma, Erich et al. *Design Patterns*. Addison-Wesley, 1994.



The MVC pattern applied to the MuseScore system

14.3.3 Development view

We will first show how the project is divided into different modules of code in the *module structure organization* section. In the *common design models* section we show how MuseScore maintains a consistent code style despite having multiple contributors. The *codeline model* describes how the code is maintained by a large group of contributors.

14.3.3.1 Module structure organization

The structure of MuseScore has several main components. These are all outlined as packages in the ReadMe on the Github page of MuseScore. There are two main components that we will highlight:

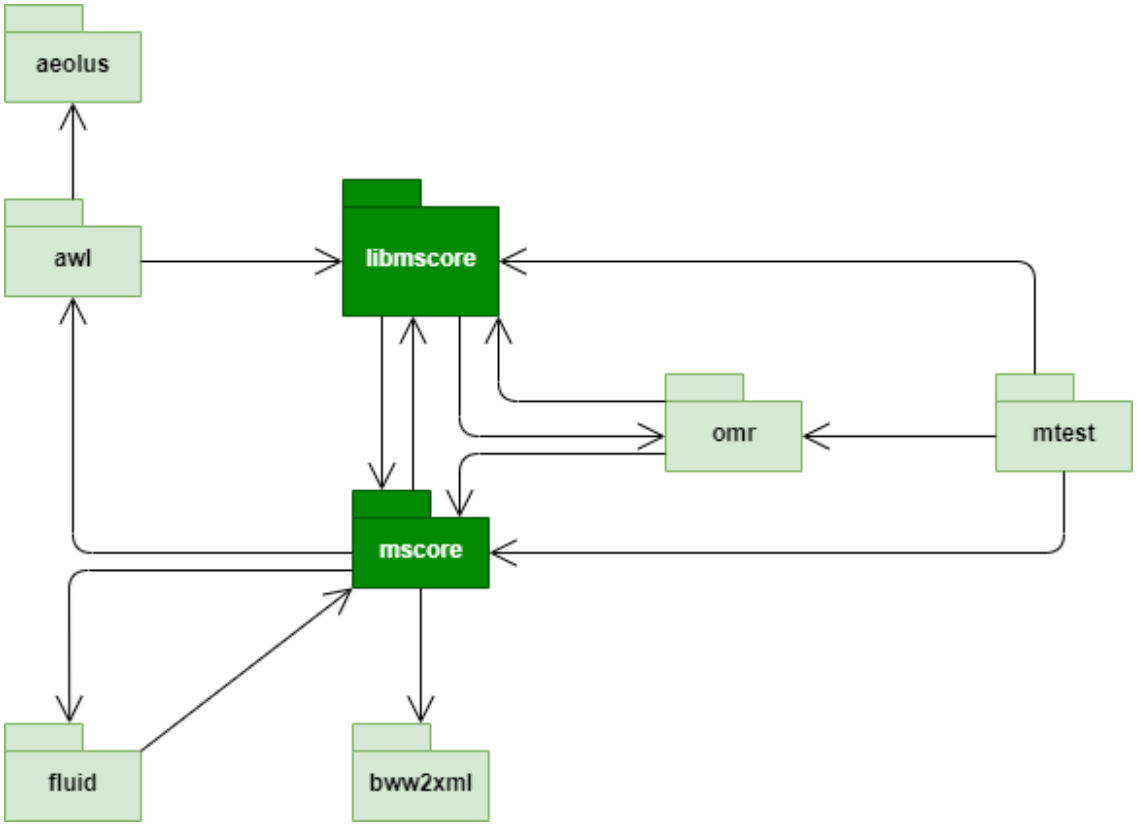
- `libmscore`: the data model for MuseScore.
- `mscore`: the package for the MuseScore UI and controller.

Together these two packages make up for 75% of the code of MuseScore itself, measured in man-year[18].

[18] Kan, Stephen H. *Metrics and models in software quality engineering*. 2003. p. 343.

This percentage is based on the information from the SIG analysis, which calculated the man-years of individual components[19]. These two packages together make all the other packages more of a support package to these two, as seen in the model structure below.

In the model, most of the packages of MuseScore are represented, the ones that are missing are either packages that do not contain relevant code, such as the share package, or third party software.



The model structure of MuseScore. The greener the packages, the bigger they are

14.3.3.2 Common design models

MuseScore has several rules and guidelines on how to work on this project. They have a code rule book that explains how to write your code. For example, it is not allowed to work with tabs for indentation; and they use six spaces instead. All of the rules can be found on their website. Furthermore, the test guidelines explain how to write tests for your code.

14.3.3.3 Codeline models

To maintain the code of the project, MuseScore uses GitHub and git for version control, and their own issue tracker for potential new features and a forum for documenting bugs. It is required to create or choose an

[19] Software Improvement Group. Sigrid Quality Assurance Platform

issue when you want to contribute to the codebase, so other contributors can clearly see your progress. This is clearly defined in the contributors guidebook.

14.3.4 Deployment and operational view

Many ways of publishing or distributing a software system exist. Whether it is a SaaS, a library or a mobile application; each situation is different.

As MuseScore is a stand-alone application, to be downloaded from the website as an executable for your operating system of choice, installation is trivial. Any dependencies, in the form of audio codecs[20], text fonts, and so-called soundfonts[21] are bundled with the application (although MuseScore supports adding third-party soundfonts and plugins).

Once installed, MuseScore has functionality for automatically updating the software. This is a setting that can be toggled in the preferences. When this is turned on, the system checks for updates on every startup. It is also possible to manually check for updates. This allows users to easily have the newest version available.

In terms of the system environment, MuseScore has two strict requirements[22]:

- At least 300MB of free disk space.
- A *recent* version of any major operating system (Windows 7, macOS 10.10, etc).

Because of the simplicity of this system, and the very brief requirements, further aspects of the operational view are not worth mentioning.

In short, the deployment and operational views are rather straightforward. MuseScore is a stand-alone application with all of its dependencies bundled together into one executable. It automatically checks for updates and has system requirements easily met by most systems.

14.3.5 Information and functional view

MuseScore creates data in the form of music scores, and the way this data is handled, stored and shared is important to the users. This can be seen in the way MuseScore handles copyright and how their data is saved.

Copyright can be a big issue in the music industry and the MuseScore system needs to handle this properly. MuseScore has the functionality to add copyright information to the footer of scores. Although it is the responsibility of the user to adhere to any copyrights on scores, MuseScore assists users by providing an easy to follow manual.

While editing, a score is saved in a compressed MuseScore custom format. These can be converted to a MusicXML file, clearly showing all stored information: chord, duration, beam, articulations, volume and instrument.

In terms of functionality, MuseScore supports the use of MIDI conversion to allow users to easily input chords. This is a great feature as it makes sure that chords do not have to be entered note by note, which can quickly become tedious.

[20] Microsoft. *Codecs: frequently asked questions*. March 28, 2019. (link).
[21] MuseScore. March 2, 2020. MuseScore sound fonts.
[22] MuseScore. MuseScore system requirements.

14.3.6 Trade-offs of non-functional properties

In every system, trade-offs have to be made on numerous levels. These trade-offs eventually lead to architectural decisions made during all phases of development. Non-functional properties are usually traded off in favor of functional properties, as architects do not want to limit the core features of their system.

Some non-functional properties that apply to MuseScore include the size of the application, the size of scores when exported, the speed of importing scores and the speed of exporting scores.

The MuseScore 3 application has a size of around 300 MB, once downloaded, it runs smoothly. However, when developing, the compilation does take up a lot of time. Considering the exporting of files, MuseScore supports PDF, PNG, and other graphic formats[23]. The size of these files depends on the length of the score and chosen export format, but they are relatively small. This might suggest that the developers have traded off efficient compilation in favor of smaller files.

Finally, MuseScore has made the accessibility of the application a high priority to support all of its users[24].

To summarize, MuseScore has mostly focused its non-functional trade-offs to benefit the score size and accessibility, as users will mostly use the created score once finished and focus less on the time it takes to get there. The application's compilation and execution time have suffered from this.

14.3.7 Conclusion

To conclude, MuseScore has implemented an MVC pattern to ensure a loosely coupled system for easy maintainability. It is deployed with individual releases, where dependencies are included as well. Once operating, the system has a few dependencies, automatic updating can be enabled for an up-to-date system guarantee. The data for MuseScore is saved in a custom format which stores all the necessary information. In terms of functionality, MuseScore provides support for MIDI conversion. Finally, we saw that MuseScore has made its biggest trade-off to benefit the size of created scores and accessibility, at the cost of compilation and execution time.

14.4 MuseScore: Cost of music

The maintainability of a system can be measured in several different ways. One of these is technical debt, which is a measure used to indicate how much refactoring is necessary within a software project[25].

First, several processes to ensure software quality will be described. Furthermore, we will highlight some aspects of a code base that contribute to the technical debt. Then, the technical debt of the MuseScore software can be analysed to determine the cost of music.

14.4.1 Software quality processes

High software quality can be achieved in many different ways. We will now describe some processes that ensure a higher quality, and thus reduce the technical debt.

[23] MuseScore. March 1, 2020. MuseScore product description.
[24] MuseScore. March, 2019. MuseScore design principles.
[25] Kruchten, P., Nord, R. L. & Ozkaya, I. *Technical Debt: From Metaphor to Theory and Practice*. IEEE Software, vol. 29, no. 6, pp. 18-21, Nov.-Dec. 2012.

As MuseScore is a software project with a codebase of 280,000 lines of code and with close to 200 contributors[26], they need good measures in place to maintain reasonable code quality. In this section, we evaluate the processes in place to maintain the software quality of the MuseScore project. We identify the following four major code quality processes in the development process of MuseScore.

14.4.1.1 Coding rules

The use of coding rules to enforce code consistency across a software project results in higher code quality[27]. The MuseScore website presents an extensive list of coding rules; those include code style guidelines (indentation, notation, other coding practices) as well as submission rules. These rules and other code quality aspects will be checked and enforced by the next quality measure: peer reviews.

14.4.1.2 Peer reviews

Peer reviews are performed on code changes in submitted pull requests. This is not a regulated process, but in general, several senior/experienced MuseScore developers check the submission and ask questions or propose changes. Although everyone can submit peer reviews, only senior developers can actually merge new code. Another advantage of peer reviewing is that other developers can help identify potential bugs in the code.

14.4.1.3 Continuous integration processes

Software quality can be ensured by using continuous integration (CI) processes. These are development practices where developers frequently integrate code into a shared repository. Each integration can then be verified by an automated build and automated tests[28]. This way, new code is tested early and bugs can be discovered as they are introduced. While automated testing is not strictly part of CI, it is typically implied.

MuseScore provides continuous integration with AppVeyor for Windows and Travis-CI for Mac and Linux[29]. These services build the MuseScore application on a virtual machine and upload the result to osuosl. Additionally, Travis-CI runs the test suite as described in the following section.

14.4.1.4 Test processes

One of the most important tools for maintaining software quality is testing. Both the quantity and quality of the tests matter. For significant software projects like MuseScore, writing good tests can be a significant challenge. Therefore, it is imperative to have certain procedures and standards in place that guide contributors in writing and evaluating tests.

MuseScore has several guidelines for testing. These guidelines include details on how to create new tests effectively. On every build, these tests are run automatically by Travis-CI. Furthermore, for every code contribution it is required to create new tests or update existing tests (if necessary).

---

[26] GitHub musescore/MuseScore.

[27] Krishnan, M. S. & Kellner, M. I. *Measuring process consistency: implications for reducing software defects*. IEEE Transactions on Software Engineering, vol. 25, no. 6, pp. 800-815, Nov.-Dec. 1999.

[28] CodeShip. *Continuous integration essentials.* no date. (link).

[29] MuseScore. March, 2018. MuseScore Development Infrastructure

14.4.1.5 Releasing MuseScore

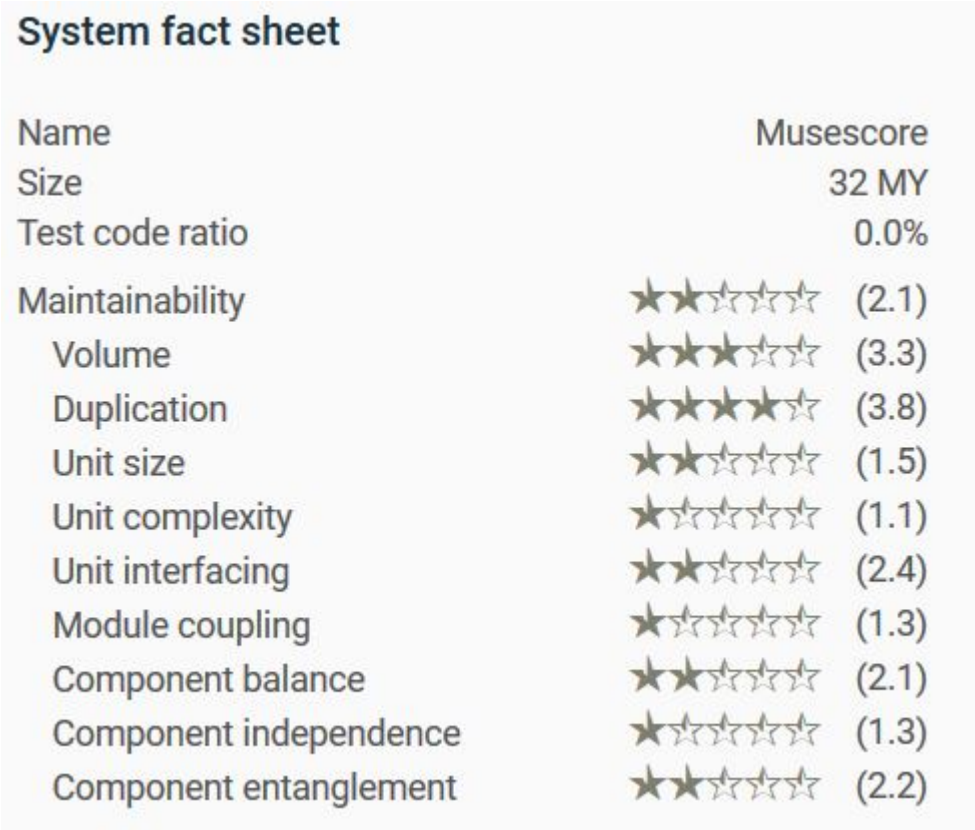
MuseScore uses three different release channels to ensure software quality in different stages of development. These are stable releases, beta releases, and nightly (development) builds. The stable releases contain versions of the software that have been extensively tested by the community. These versions are released to the general public about every nine months. The beta releases come in the months preceding the stable release. These versions may be unstable and are intended for testers and advanced users, to make the stable releases as bug-free as possible. The development builds are released multiple times per day. These builds happen automatically whenever a change is merged to the master branch, and serve as the first test for new functionality.

14.4.2 Code quality and maintainability

The maintainability of a program is the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers[30]. In an open-source project, such a maintainer can be anyone. With a higher maintainability degree, it is easier to make changes to the code base. Thus, high maintainability should be a top priority for MuseScore.

We had a complete analysis of the codebase performed by the Software Improvement Group (SIG). They performed a number of calculations and analyses on the code base as a whole. This was based on a system decomposition, as mentioned in MuseScore: views on development. A fact sheet generated by SIG can be seen below.

[30] Software Improvement Group. *Sigrid manual*. December 24, 2019. (Document)



System fact sheet generated by SIG. Retrieved from MuseScore Sigrid analysis.

The total maintainability score of MuseScore is 2.1 out of 5, where 5 is the best. SIG determines this score by comparing all SIG-evaluated projects to each other, for example, giving the 5% best projects a 5-star rating[31]. This means that MuseScore is in roughly the worst 10% of systems analysed by SIG.

14.4.2.1 Maintainability of the roadmap

In MuseScore: Road to reducing paper use in music industry, we argued that there is a lack of detailed future plans, but that MuseScore has four main features to focus on: accessibility, notation, playback and usability.

When looking at the accessibility and notation features, it is clear that these are both mainly UI-focused features. Improving the maintainability of these features would require working in the `mscore` package, which contains the controller classes for the UI of MuseScore[32]. When looking at the SIG analysis, we see that `mscore` has a maintainability rating of 2.3, which is higher than the overall rating, but still rather low.

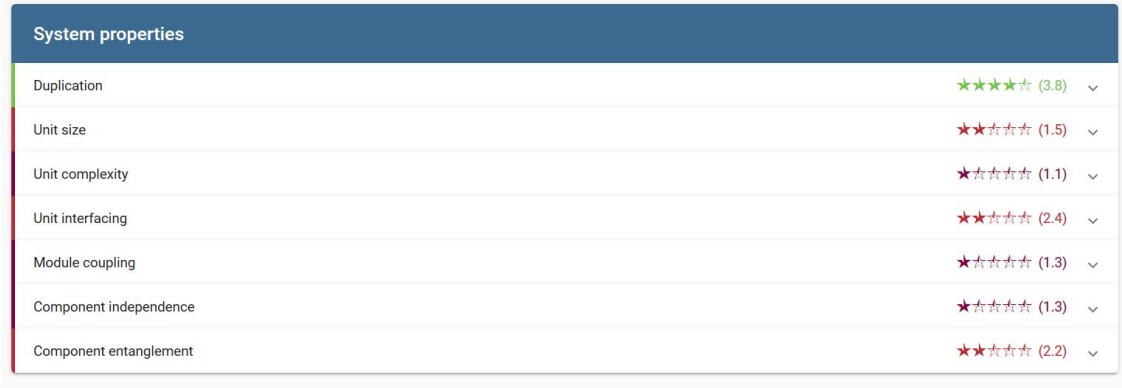
The other two features are playback and usability. These focus more on the technical side of MuseScore, but it is very likely that these also require some UI changes. However, most of the code will be changed in the `libmscore` package, which is the data model for MuseScore. According to SIG, this has a maintainability score of 1.8, which is below the average of MuseScore.

[31] Software Improvement Group. *Sigrid manual*. December 24, 2019. (Document)
[32] GitHub musescore/MuseScore.

It should be noted that these two packages are biggest packages in MuseScore as shown in MuseScore: views on development. So, with these four features being the main focus points for MuseScore in the near future, roadmap-wise, it means that any change in these packages will require more effort than usual, thus refactoring will be costly.

14.4.2.2 Refactoring suggestions

A score of 2.1 stars out of 5 means that there should be plenty of refactoring suggestions. SIG has already put some effort into offering a few. An overview can be seen below.



Overview of the software metrics where most refactoring should be done to improve the maintainability score. Retrieved from MuseScore Sigrid analysis.

14.4.2.2.1 Complexity Although there is some code duplication present, this is not too worrisome. The most important refactoring to be done is in unit complexity, which is based on the McCabe complexity[33]. This metric measures the number of paths that can be taken through a unit of code (usually a method)[34]. These paths are created by the use of loops or conditional statements.

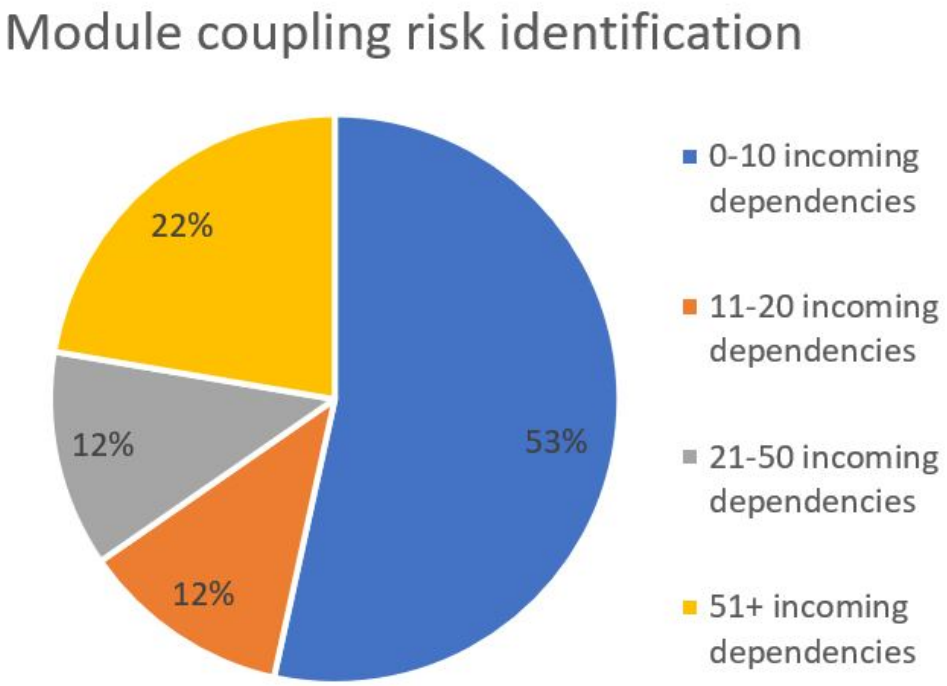
The units of code that were suggested by SIG to improve the score and maintainability can be fixed by splitting methods. This means taking out a piece of complex functionality, which is then called from inside the original method. SIG has identified 100 candidates for complexity refactoring. On top of that, a lot of these candidates currently have a McCabe complexity of almost 100, while the maximum complexity as recommended by SIG is 25[35]. Refactoring these methods is therefore highly recommended.

14.4.2.2.2 Decoupling Another metric which indicates a great need of refactoring, is the module coupling. This is defined as "*the number of incoming dependencies from the modules of the source code*"[36]. The coupling of modules poses a risk to the maintainability of a system, because changing a highly-coupled module also requires changing the dependencies, resulting in more work.

In the figure below, the module coupling problems are identified and rated based on the risk they pose to the maintainability of the code base. Dependency-wise, more than 20% of the code is high risk. High risk is

[33] Software Improvement Group. *Sigrid manual*. December 24, 2019. (Document)
[34] Harrison, W. & Magel, K. *A complexity measure based on nesting level*. ACM SIGPLAN Notices, Volume 16, Issue 3. March, 1981.
[35] Software Improvement Group. *Sigrid manual*. December 24, 2019. (Document)
[36] Software Improvement Group. *Sigrid manual*. December 24, 2019. (Document)

defined as having more than 50 incoming dependencies[37]. As proposed by Michael Ridland (Developer, Consultant and Architect), decoupling can be performed by abstraction and events[38]. This entails using interfaces to define what code belongs together. Events are used to group similar activities together.



The MuseScore module coupling risk identification. Created with data from MuseScore Sigrid analysis.

Decoupling is definitely an important metric for MuseScore to focus refactoring actions on, as was also suggested by other contributors on the developer's forum of MuseScore. User 'shoogle' has proposed to use modularizing to separate out dialects and imports/exports to improve the maintainability and make it easier to add new contributions[39].

14.4.2.3 Coding hotspots

Coding hotspots are locations in the code with a lot of recent coding activity. These are relevant to the maintainability of a project as these might point to parts of code that have a lot of issues. For example, if a new bug is fixed in the same part of code every day, this might suggest there is something inherently wrong with that part of the code.

In terms of recent coding activity, we first define what is recent. MuseScore has pull requests emerging at a rate of several per day and they are merged daily[40]. So, we have chosen to focus on the last month, which is reasonable for a code base of around 280,000 lines of code[41]. Furthermore, we define hotspots to have at

[37] Software Improvement Group. *Sigrid manual*. December 24, 2019. (Document)
[38] Michael Ridland. *Software Architecture: Increasing cohesion and decreasing coupling.* February 24, 2014. (link).
[39] Shoogle. *Modularizing MuseScore*. January 20, 2019. (link).
[40] MuseScore Github. Pull requests. (link).
[41] Software Improvement Group. Sigrid Quality Assurance Platform

least three commits in the last month.

With `libmscore` and `mscore` being the two largest packages in the code base, together making up for about 80% of the code base[42], these are the most interesting to look at as they contain all key architectural components. After studying the code base to look for hotspots, we decided that the `libmscore` hotspots were most relevant for the maintainability of the code base.

As `libmscore` only contains single files and no subpackages, we will identify coding hotspots here in terms of files. One of the main hotspots is the `edit.cpp` file, which was edited in 8 different commits in the last month. This is a file of roughly 2,000 lines of code, focused on editing scores. Most of the recent changes include bug fixes for warnings and crashes, although there were also changes made towards improving the user experience. For example, when switching instruments within a score, the features for certain specific instruments are no longer blindly used with the other instrument. Our own pull request for a bug fix was also located in `edit.cpp`, where we also identified two other bugs.

Another hotspot is `measure.cpp`, which was edited in five different commits in the last month. As a score is built up of measures, this is a key element for a score. The changes were all bug fixes to remove crashes, unnecessary warnings and issues with the PVS-Studio support. The latter being the static analysis tool that MuseScore uses for its code base, which is built-in for Visual Studio.

14.4.3 Technical debt

As explained in the introduction, technical debt is a measure of how much refactoring is required for a software project. We have analysed the software quality processes and maintainability of the MuseScore code. From this, we have learned that MuseScore does have processes in place to ensure high software quality. At the same time, from the analysis performed by SIG, we can see that the code base is not maintainable in the long term, which can make contributing more difficult.

When looking at the maintainability of the MuseScore project, a lot of improvements can be made, as per the refactoring suggestions. As argued by Türk, software quality is determined by a number of factors: functionality, reliability, usability, efficiency, maintainability and portability[43]. Although maintainability is part of the software quality, these other factors are also at play. Therefore, we cannot draw any definite conclusions on MuseScore's software quality.

14.5 MuseScore: Architecting for accessibility and usability

Software can be customized to its users from architecture to the final design. In many different ways, developers can optimize their software for specific users. As mentioned in our first post, the context in which MuseScore operates is based on the usability of the application and the accessibility to the users. The latter was also determined to be functionality that is worthy to be prioritized in trade-offs in our second post.

MuseScore started out as a project from one of the now main developers, Werner Schweer. He wanted to create a high-quality sheet music production tool that was also supported on Linux, which at the time did not exist[44]. Over the course of the first six years of development, the company grew towards 60 employees, including both software engineers as well as translators and tutorial creators.

[42] Software Improvement Group. Sigrid Quality Assurance Platform

[43] Tuna Türk. *The effect of software design patterns on object-oriented software quality and maintainability.* September 2009. Thesis for degree Master of Science for Electrical and Electronics Engineering at Middle East Technical University.

[44] SourceForge. *WYSIWYG music app makes a score*. June 21, 2010. (link).

From the start, the architecture has been focused on the following key features: playing back scores, importing/exporting many different file formats, a WYSIWYG user interface, the notation that musicians are used to, the support of plugins and international support[45].

Early on during development, MIDI (Musical Instrument Digital Interface) support was already recognized as an important feature, as it would greatly improve user experiences for note input. Furthermore, the note input is constantly under evaluation, leading to updates to improve the usability. Besides the actual inputting of the notes, the input functionality for all other notation is also optimized to assist users as much as possible. Finally, MuseScore has shown a big interest in supporting access to MuseScore for everyone, including people with disabilities.

These key focus points for MuseScore will be discussed in how they precisely support users and are then put into perspective with regards to the architecture of the MuseScore project.

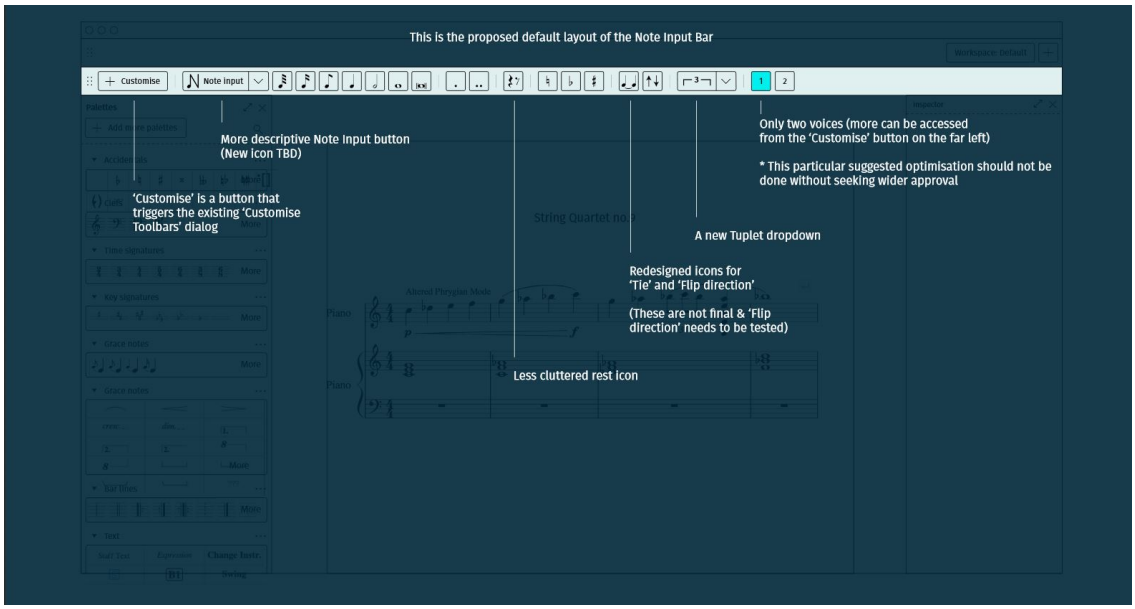
14.5.1 Note input

In March of 2019, popular youtuber, composer and designer Martin Keary[46] published a video on MuseScore[47]. In this video, he gives constructive criticism on the interface design of MuseScore. This video reached a large audience and even sparked MuseScore to hire Martin Keary as the head of design. Martin immediately got to work on redesigning palettes, instrument dialog, drag & drop usability, and also note input.

14.5.1.1 Architecture of note input

The main problem identified in the note input workflow is that inputting notes requires you to press the `note input` button, while this is not always intuitive. The following design document was published and a call to action[48] followed. The image below shows the proposed design for the note input toolbar. Most of the changes that are currently implemented in MuseScore originate from this pull request. Most of the code changes can be found in `input.cpp` and `cmd.cpp`, both in the `libmscore` package. As addressed in our third essay, this package has a low maintainability score, so it comes as no surprise that most of the code changes reside here.

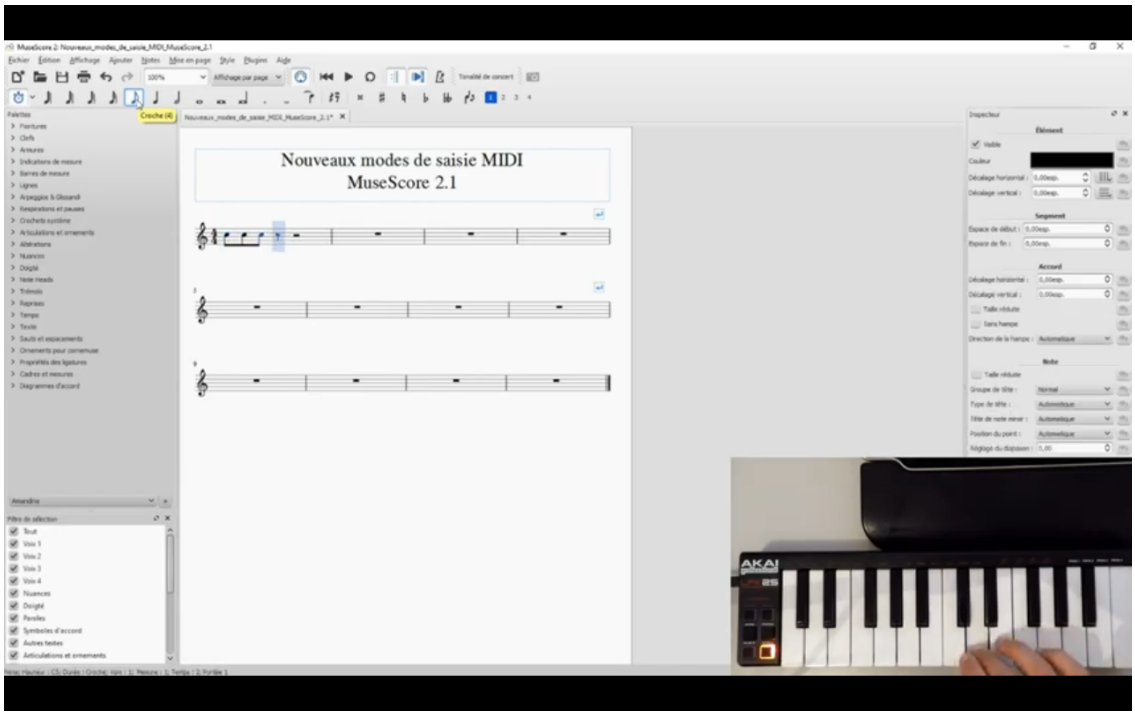
[45] David Bolton. *New features in MuseScore 0.9.5.* Augustus 15, 2009. (link).
[46] Martin Keary. *About me*. Last visited: April 8, 2020. (link).
[47] Youtube. *Music Software & Interface Design: MuseScore.* March 19, 2019. (link)
[48] MuseScore. *Note input design*. September 23, 2019. (link).



Proposed design for the note input toolbar and its functionality.

14.5.2 MIDI support

MuseScore has support for MIDI (as introduced in earlier posts) import/export as well as live input. An example of this can be found in this YouTube video and a cut from the video can be seen below.



An example of live MIDI note input.

MIDI is a digital encoding of musical notes, including expressivity (keypress velocity) and other information. MIDI controllers employ this protocol to send out notes based on key presses.

14.5.2.1 Usability advantages

For this usability analysis, we will focus on the live MIDI input. This enables the musician to play live on a physical keyboard (or other kind of MIDI controller, for example, a guitar). This has several advantages:

- First of all, it allows the musician to do what they do best: play music. If they can just play instead of work with mouse and keyboard, they will more naturally generate sheet music for (their) music.
- The ability to read music scores is not required when using MIDI inputs. If a creator can play music by their own technique or by ear, they can create scores for it. This lowers the threshold for anyone to create them.
- (Rhythmically) complex parts, or freestyle solos, will often be easier to play than to write out in sheet music. By playing those parts live, they can be recorded in scores.

14.5.2.2 Architecture implications

According to the SIG analysis[49], all MIDI functionality is loosely coupled to the core modules. Therefore, this usability feature does not have a high impact on the software architecture. This means that, upon a restructuring of the software architecture, this functionality is easy to re-implement and re-attach to the core functionality. In some sense, it almost resembles a plug-in.

[49] SIG. *MuseScore architecture* (link)

14.5.3 Supporting all instruments

One of the main key features is that MuseScore wants to support all instruments[50]. Every instrument has music notation that is custom to that instrument. For example, a guitar uses tabs to indicate the chords that should be played, while percussion scores include which drum to hit which will define the pitch[51]. Meanwhile, wind instruments are often played in a different key, so the sheet music must be adapted to that key.

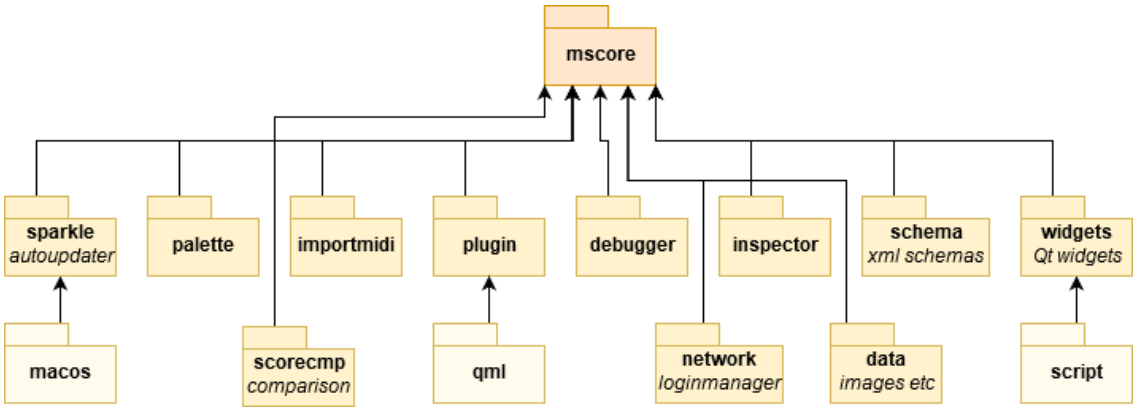
This way, each instrument has its own specific notations that musicians will want to include in their scores. On the other hand, a musician will only be interested in the general notation and specific notation for the instruments in use.

14.5.3.1 Palettes

The solution to this for MuseScore is the use of so-called palettes. Each instrument can have its own palette, including different notations specific to that instrument. These palettes can be seen as workspaces for specific topics[52]. This greatly supports the usability for MuseScore's users, as they can simply find the notation they are looking for under a certain palette. For example, the general 'lines' palette contains different lines to indicate crescendo, decrescendo or legato.

14.5.3.2 Architecture of palettes

Architectural-wise, `palette` is a module in the `mscore` package. The `mscore` decomposition can be seen below. Furthermore, the `palette.h`, `palette.cpp`, `paletteCellProperties.ui` and `paletteProperties.ui` files can also be found in the `mscore` package. The Palette class serves as a general class for all palette implementation, providing methods like `applyPaletteElement` and `append` to use the specific functions from a certain palette.



`mscore` decomposition

Furthermore, the `palette` module provides functionality for users to create specific palettes themselves or customize existing palettes. The UI components for showing palettes are also contained in this module.

[50] SourceForge. *WYSIWYG music app makes a score*. June 21, 2010. (link).
[51] MuseScore. *Drum notation.* March 2, 2020. (link).
[52] MuseScore. *Palettes.* March 5, 2020. (link).

So, MuseScore supports all instruments by allowing users to select certain palettes for score entries. Palettes are implemented in a loosely coupled fashion, so they can easily be altered without having to change all the code.

14.5.4 Visually impaired support

As mentioned earlier in MuseScore: Road to reducing paper use in music, it is said that MuseScore will focus on its accessibility. This also includes people who have impaired vision or no vision at all. While MuseScore had some accessibility features already in version 2, this is improved in version 3 with the addition of using the keyboard to access the palettes[53]. With the release of MuseScore 3.3, most of the features of MuseScore are now accessible to visually impaired people[54].

14.5.4.1 Screen Readers

This is accomplished by supporting NVDA and JAWS for Windows, and Orca for Linux. These programs are helpful tools to aid visually impaired people in using the program, and are called screen readers. Screen readers are programs that read out the text and other features on the page, in the case of MuseScore this could be the notes. MuseScore does not have screen reader support for MacOS, but plans to implement it in the future[55].

14.5.4.2 Helpful features

One of the features that helps visually impaired people is text-to-speech, which will read the text to the user. Another feature is called Modified Stave Notation (MSN). This notation makes the notation in general bigger, thereby making it easier to read for visually impaired people. NVDA also comes with support for Braille keyboards. However, it is unclear whether this currently works in MuseScore. Whether it works or not, it is a goal for MuseScore in general to get Braille keyboards to work[56]. Another feature is the use of shortcuts for all of the navigation within MuseScore. When using shortcuts in combination with the visual aid tools, it will automatically read out the note names when they are selected.

14.5.5 Conclusion

So, how does MuseScore support its users? It is clear that they strongly value the usability of the application, and as a result, have given the accessibility a high priority when it comes to trade-offs.

In order to do so, MuseScore is constantly developing the note input systems, as well as other features. They have clearly shown they put effort into optimizing the note input functionality. Criticism and feedback from users are taken seriously to continue improving the MuseScore application. Besides manual input, MIDI input is also supported to easily allow musicians to enter the music into the digital score by just playing on a digital instrument.

To further support users, MuseScore uses palettes to divide user functionality into groups to keep things organized and clear to users.

Finally, MuseScore wants all users to be able to use their application. Therefore, they have invested in supporting tools that enable visually impaired users to also use the application and create digital scores.

[53] MuseScore. *Impaired vision version 2*. July 19, 2019. (link) .
[54] MuseScore. *Handbook for the visually impaired*. March 9, 2020. (link).
[55] MuseScore. *Impaired vision version 2*. July 19, 2019. (link) .
[56] MuseScore. *Impaired vision version 2*. July 19, 2019. (link) .

Chapter 15

Mypy



[1]

Mypy is an open source static type checker for Python. Users can either use it or not, but by using it, Python code can be checked for common type errors without running it. Furthermore it might help a developer to find bugs (i.e. rounded integer errors) in his own code. It can be used on Python 2.7 and 3

Mypy is used by many companies and projects, where one of the largest companies is Dropbox.

In the upcoming blog posts, an in-depth analysis of mypy will be constructed. First the project vision is investigated, including the stakeholders and the roadmap. The second blog post will consist of the architectural overview constructed from the project vision. Thirdly we will look into the code perspective of the project. Lastly one blog post will be a deeper analysis of the project.

15.1 Authors

The blogposts will be written by four students from the TU Delft:

- Michel Wervers
- Abtin Kerami
- David Zwart
- Niels Hoogerwerf

[1] https://github.com/python/mypy

15.2 The vision behind mypy

In our first blogpost, the main focus will lie on identifying what mypy's properties are and for whom it is created. As a real system architect one could identify this as the vision of the project.

15.2.1 Wait what, static type checking for python?

It is true. Python has often been promoted as being a fast scripting-language. But what is less known, is that Python depends on solving errors in runtime: you have to run code before errors pop up. This can become a huge pain for developers, since they easily lose track of their custom-defined classes and types. Well... that used to be the case! mypy is a library which takes the responsibility of error checking and brings it back to compile time, or as its developers call it:

> *"A python linter on steroids"*.

But why would an extreme linter be a good idea? Look at Javascript, like Python, a dynamically typed and widely used language. Research has shown that type-checking might resolve 15% of all errors up front[2]. Think about how much time you would save by immediately getting these error messages while coding!

As Javascript tackles this problem with Typescript and others, Python offers a solution in mypy. Mypy gives users the possibility to check for type errors, before running the code. The well-known company Dropbox has posted their perspective on integrating mypy into their existing code-base. With integration into well-know IDE's like VS Code, Pycharm and others, mypy might just make you consider using Python for your next project.

15.2.2 Mypy: A linter on steroids

Up until now we have talked a lot about types and type checking. Now we will elaborate on what mypy's capabilities are.

Type annotations are used by the programmer to be able to type a given program variable. These type annotations defined in PEP 484 are treated as plain comments when the Python code is run, but mypy uses them to type check the program.

In mypy we can use different kinds of types which we are able to use in our annotations:

- **Built-in types:** such as the familiar `int`, `char` etc.
- **Class types:** classes re-usable as `type`.
- **Built-in Collection types:** like `List[type]`, a `Tuple[type, type]` or a `Mapping[type, type]`. The `type` represents any generic type (or mapping) within the collection.
- **Any types:** when a type can have more than one type using i.e. `Union` or `Optional`.
- **Generic types:** implement a generic type derive from built-in types like `List[T]`, for example a `Stack`.
- **Miscellaneous types:** described in the documentation. Which define annotation rules for cases where special types are necessary with for instance the async type.

For a complete overview of the kinds of types which can be used the documentation can be consulted. These type declarations, with the exception of some of the miscellaneous types, can all be used in defining the

[2] Z. Gao, C. Bird, and E. T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In 2017 IEEE/ACM 39th International Con-ference on Software Engineering (ICSE), pages 758–769, May 2017

function arguments and the functions return type. It can also be used in classes for its attribute definition. A nice overview of examples of type annotations can be found in the mypy Cheatsheet.

With these type annotations, mypy processes the source code it is given and notifies the user whenever a type mismatch occurs in the expected type versus the type which is encountered at runtime. This can for example happen with variable assignments, function return types and passed function arguments.

15.2.3 Who are the people involved?

Similarly to writing a blog, knowing your audience in software architecture is also important. According to the Lean Architecture book written by James O. Coplien and Gertrud Bjørnvig [3] there are five main stakeholder areas: end users, the business, customers, domain experts and developers. We will describe each of the stakeholder areas with respect to mypy.

15.2.3.1 End users -> developers and Python program users

The end users are both the developers using mypy as well as the users of Python programs. Developers of Python programs are also one of the main stakeholder areas, but a distinction can be made between an end user developer (using mypy to create better Python programs) and a developer as stakeholder (discussed below). The main advantage for the end users of Python programs is that their programs are more likely to work correctly. The main advantage for the developers of Python programs is that their knowledge of their programs is better and they are less likely to create bugs.

15.2.3.2 Customers -> collectives

Collectives (i.e. Companies, project groups and research institutes) using mypy can benefit from its usage. As stated before the usage of a typechecker in a dynamic language could possibly decrease the amount of bugs on average by 15%,[4] which in turn could result in less time searching for bugs and possibly decreases the development and debugging costs. One of the bigger customers in this case is Dropbox.

15.2.3.3 Domain experts -> maintainers

We think the main contributors of the mypy project and the people behind the merge requests can be identified as the domain experts. Looking at the contributors, 4 major contributors can be defined: Jukka Lehtosalo, M. Sullivan, I. Levkivskyi and G. van Rossum are maintainers of the repository. Their view on this product is possibly the most important and they define what the roadmap of mypy is (also look at The Future of mypy).

15.2.3.4 Developers -> contributors

Any developer willing to commit to the project is a stakeholder. For them it is important to have a clear view of what is needed in the project. Therefore the roadmap and the issue tracker are important tools to make sure each person willing to spend time in the project, knows what is left to do or where updates can be made.

[3] James O. Coplien and Gertrud Bjrnvig.Lean Architecture: For Agile Soft-ware Development. Wiley Publishing, 2010.

[4] Z. Gao, C. Bird, and E. T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In 2017 IEEE/ACM 39th International Con-ference on Software Engineering (ICSE), pages 758–769, May 2017

15.2.3.5 The business -> Dropbox

Since mypy is fully free and open source using a creative commons license, no financial or business objective is present. It was started as a PhD project by J. Lehtosalo. One could argue that Dropbox is also the business side behind mypy, since their interest as customer stakeholder could be a source of increased revenue.

15.2.4 What can we expect from mypy in the future?

Although the static typechecking which is provided by the mypy library so far does a fine job in making Python a safer language, a downside is that currently in order to obtain the functionality of mypy, the user needs to go through the steps of installing mypy seperately. Currently mypy is seen more as an extension, seperate from Python making clever use of type declarations introduced in PEP 484.

On the other hand, its future context can be characterized as being integrated with the Python programming language, so that it becomes an *internal* language functionality that users of Python may adopt. The IDE's should in the future context also be able to readily display the users errors when typing mistakes are made.

Besides having a future vision, mypy has a lot of users, which raises the need for a large amount of future extra content and fixes. The developers of mypy therefore have created a roadmap, which clearly states what they are currently working on and what they are planning to work on. The roadmap of mypy consists of bullet points which can be divided into three parts; bug fixes, general improvements and content addition. While one could go over each of the bullet points seperately, this would not provide any insight in the future perspective of mypy. Therefore we will discuss each of the three main parts in more detail.

15.2.4.1 Bug fixes

Logically, bug fixes are a large part of the roadmap. They are important because they make sure that code runs fluently and gives no issues when ran by other users. The bug fixes are tracked in the issues on the github page, where everyone can see and fix them. They focus mainly on making sure the functional requirements are met.

15.2.4.2 General improvements

Secondly, the roadmap of mypy will concentrate on improving the non-functional requirements of among others:

- the speed of mypy analysis
- the daemon, which runs the language server.
- the mypyc compiler, which is a native optimized version of mypy

These projects do not contribute to new features or bug-fixes of mypy, but make it faster and more optimized for use.

15.2.4.3 Feature addition

Lastly, the roadmap specifies what functional requirements are missing in mypy right now. The roadmap states that there are features (i.e. Support user defined variadic generics) which are not supported yet in mypy. Logically mypy would want these features added to fully support the needs of the end-user.

15.2.5 Coming up...

We as writers of this blog were impressed with the capabilities of mypy and the amount of different parties involved in mypy. While it is a simple program with a simple goal, the possibilities are endless! We look forward in researching more about this project for you and are definitely interested in diving deeper into the core of mypy. In our next blog post we will research how the vision of mypy influenced the architecture, so make sure you stay tuned!

15.3 Architecture of mypy

Welcome back all! Today we will dive deeper into the architecture of mypy. As this is our second blogpost, we assume that you have some basic knowledge about what mypy is and what you can do with it. If you have no clue what we are talking about, please check our previous blogpost.

15.3.1 What are your thoughts?

All projects have their respective stakeholders, and the responsibility of the architect is to make sure that all thoughts about what a system should do are conveyed in the project. By subdividing these thoughts into different topics, one can find the different viewpoints for the project.

Rozanski and Woods [5] have described a nice method for subdividing these thoughts into architectural viewpoints, to meet the requirements of the main stakeholders.

Viewpoint	Importance	Explanation
Functional	++	Because functionality is the main reason for starting projects, the functional viewpoint is important
Information	+	How to give the result to the end-user. Readability of the result is important. Less important than the function.
Concurrency	-	Important to notice if programs run concurrently. In the case of mypy, the user should be able to check a file without closing other programs.
Development	+	Mypy is open source, so to contribute, the architecture has to be known.
Deployment	–	Low demands on hardware. As long as python can run, mypy can too.
Operational	+	Mypy should be easy to install, easy to maintain and it has to work on different systems.

Since each architecture is different, each of the viewpoints has a distinct importance to the system. For mypy the table shows this for each of the viewpoints. Furthermore the viewpoints are explained in short. For a description of the functional and information viewpoint, one should read the previous blogpost. The development, deployment and operational views, will be discussed in the next sections.

The coming sections are a lot more technical, so buckle up, because this is going to be one hell of a ride!

[5] Nick Rozanski and Ein Woods.Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional, 2 edition, 2011.

15.3.2 You got style!

Really bad software mostly is made without thinking about the architecture. When you as a developer start a project, you should think about the functional viewpoints and adjust your design accordingly.

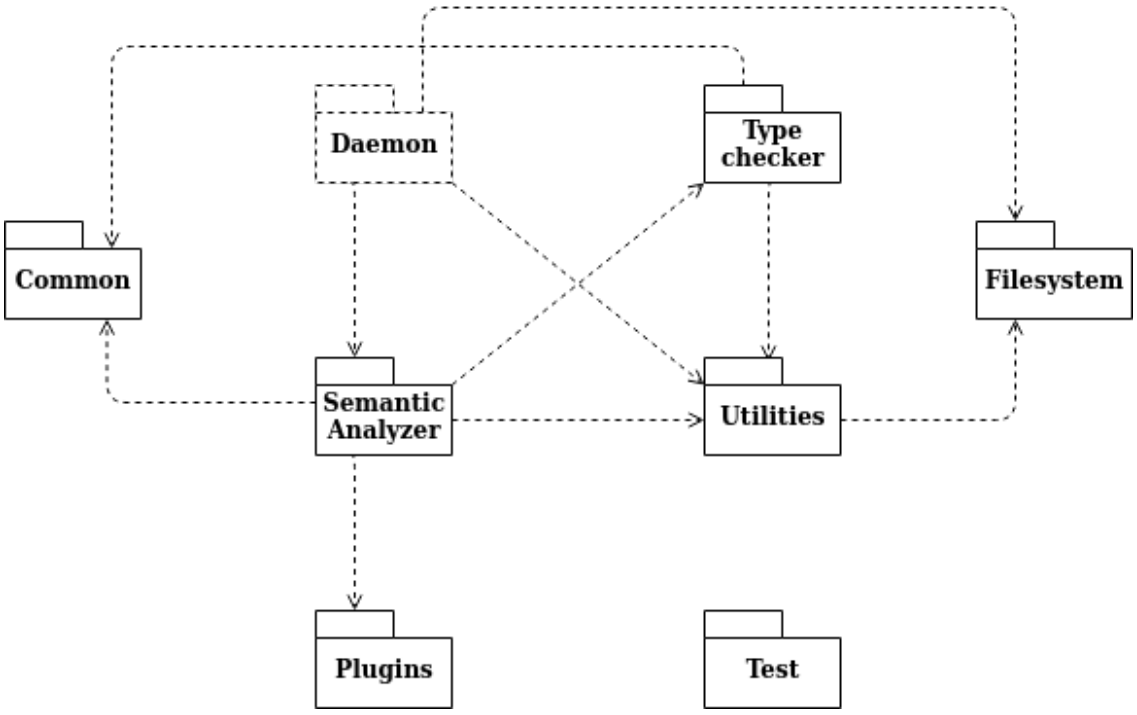
While the functional viewpoint of mypy is mostly described in the previous blogpost, the overall design is not discussed yet. The mypy code base consists mostly of a lot of files in one folder. After scrolling through 1000s lines of code, we know one thing for certain, it all starts with `main.py`.

All jokes aside, the mypy architecture can be described as a pipe and filter architecture. Pipe and filter architectures apply filters sequentially to run it. One can see this pattern in for instance compilers (which coincidentally could also type check).



15.3.3 Analyze this

Are you still with us? Good! Lets start our descent into the complex codebase of mypy and shine some light on it. So what if you want to contribute to mypy, do you know where to start? We would suggest taking a look at the following image.



In this image we can distinguish eight different components. This is our vision on mypy and its dependencies. The main component is the semantic analyzer. This component includes the `main.py` file (called when mypy is run) and the `build.py` file. The `main.py` file calls the build file, which then handles all responsibilities of doing the type checking. As you can see in the image, there are many packages used by the semantic analyzer to do the type checking. An observative reader, could see that the daemon is surrounded by a dotted line. This means that it is optional to use the daemon.

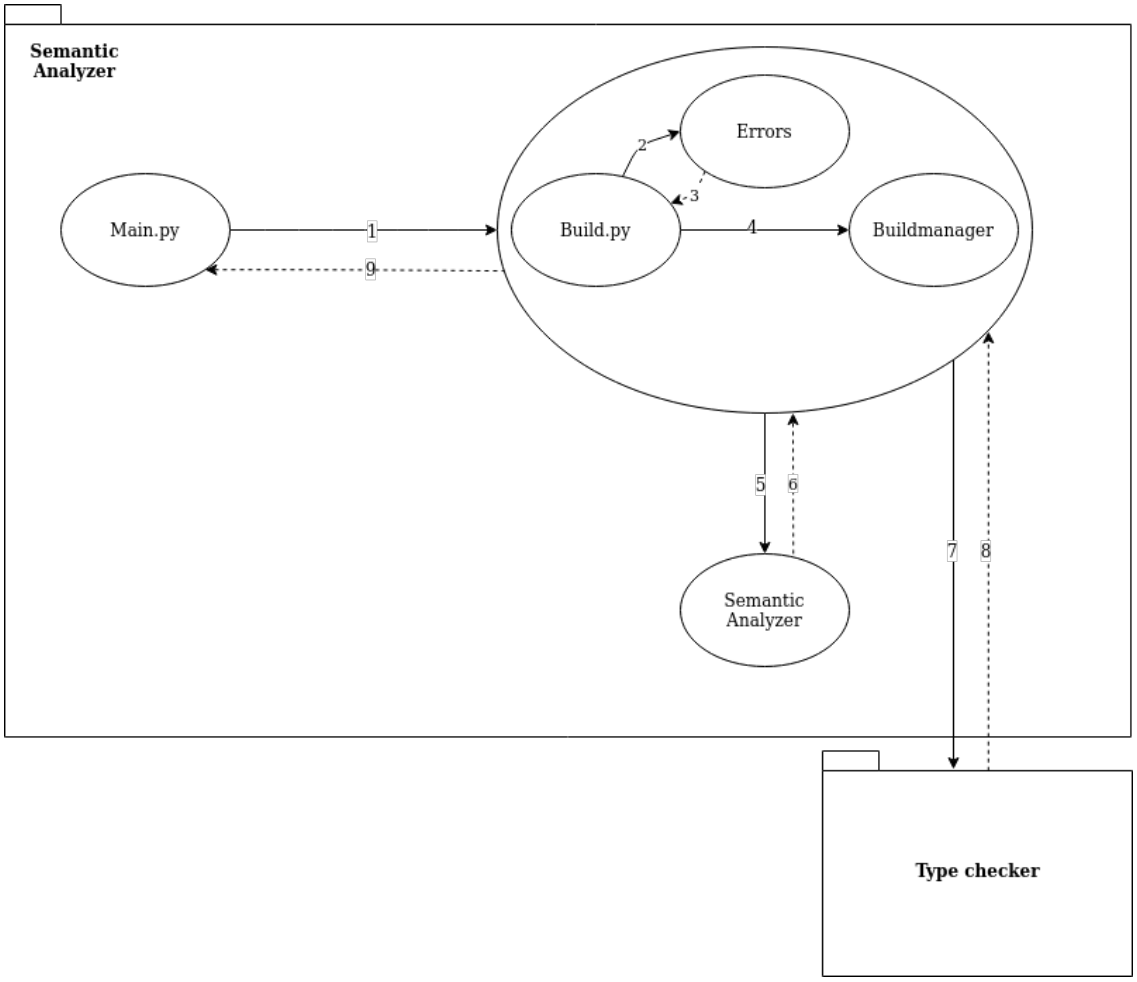
We will go over the different components briefly, in order to give you some feeling about the system. First we start with the *utilities* package, which contains all helper functions for the other packages. The second component is the *filesystem*, which maintains the analysis metadata cache. This analysis metadata is useful for speeding up, and avoiding redundant system calls. When the *semantic analyzer* has constructed the project metadata, a *typechecker* component is instantiated and called. This is where the real types are compared and where an error report is generated. Following, the *common* component contains a definition library of the different errors found. Finally, to be able to have a simple way of testing the mypy program itself, a set of integration and unit tests is available to run mypy against specific scenario's. Mypy is also capable of being extended with (third party) plugins. The *plugins* component handles these extensions. [6]

15.3.4 Mypy as a command-line pipe

There are two main packages that are important for how the pipe flow architecture is composed in mypy. Looking at the image below, we can see the sequential workflow. The numbers represent the order of the calls. Don't get confused by the name of the package and the name of the submodules. We will tear it down

[6] Mypy github for the code base https://github.com/python/mypy

in a runtime view for you:[7]



- `Main.py` is calling the `build.py` file. Since this file consist of many classes we decided to put it in a bigger submodule (step 1).
- `Build.py` calls many different methods to initialize the right configurations for the semantic analyzer (as a package). One of these configurations is what error messages can sent (step 3 and 4).
- With all these settings a Buildmanager is created. The Buildmanager will correctly call the right functions of the semantic analyzer (as a submodule) and handles dependencies between files. (step 5).
- The semantic analyzer checks if the program is so called "valid", so if there are any problems with the code itself. It returns any errors found within the program. (step 6).
- The type checker package does the real typechecking. With the information gathered by the Buildmanager (from among others the semantic analyzer), it type checks a given file. Then it will return found errors (step 7 and 8).
- Lastly the errors are transferred to the end-user (step 9). Depending on the chosen options a report might be created (also see for a more in depth description).

[7] Mypy documentation found on https://mypy.readthedocs.io/en/stable/index.html

15.3.5 Can you run it?

So, now you all know what the mypy architecture looks like, let's talk about deployment. Arc42 [8] talks about the deployment view as the technical infrastructure used to execute the system. The most important thing is the hardware that mypy can be run on. Using that point of view, mypy is a very simple program, that does not require much. It just requires a computer that has python installed on it. Using the command line, it can be used. Alternatively it can be installed via pip, to make it easier to use.

Finally, a daemon is also present, to be able to have continuous type checking with python.

15.3.6 Non-functional is the new functional

Lastly lets talk a bit about some lighter stuff. What if a system architecture is perfectly working, however type checking one file will take a day? This would mean developers will not use mypy and thus the architecture is not correct. To overcome this problem, an architect should always know the NFPs (Non functional properties) of his project. For mypy the following non functional properties are important: [9]

- Strictness. To make sure that developers have freedom in type checking their project, they can add a *mypy.ini* configuration file. In this file they can specify how their whole project should be checked by mypy or certain parts of it. For example: are files without types allowed? Or another example, some developers might want a so called `Any` for some of their functions, which can be allowed or not. Mypy structures the configuration sections in multiple system wide *mypy.ini* or *mypy.cfg* files and finally merges all sources with any given command line arguments.
- Speed. As described before, the need for a fast type checker is important. Mypy deals with this in two ways. First of all the config file can be used to make the type checker faster by changing the behaviour of type checking (for instance only type check a module instead of the full project). Secondly mypy is working on a compiled c version of their type checker to increase speed.
- Documenting errors. As a developer, documenting errors might be important. However having a terminal output might be hard to work with. Therefore mypy has an option to create documentation of the errors in multiple ways (for instance html and xml).
- Documentation. There are many options mypy, therefore starting with it is easier said than done. To make it a bit easier extensive documentation can be found online (or can be build locally).
- Testing. A developer might want to verify the correct functioning of mypy, to make sure that using it is helpful. As described before, mypy has many different tests to verify the correct functioning, which can be run with the `pytest` library.

We hope you all enjoyed this new blogpost about mypy, and hope you have learned something. The next blogpost will contain an exam! Just kidding, we hope to see you again at our next post!

15.4 Checking the typechecker mypy

Analyzing a system which is usually doing the type-checking analysis for us, feels like we're suddenly on the other end of the rope. This time, we'll cover the code quality aspect of the python library mypy and how quality is guaranteed. Read on, if you'd like to know how we think mypy can be improved on an architectural level!

[8] The views as described in https://docs.arc42.org/home/

[9] I-Hsin Chou and Chin-Feng Fan. Regulatory software configuration man-agement system design. volume 4166, pages 99–112, 09 2006.

15.4.1 Maintaining quality mypy code

To ensure software quality, mypy strives to do three things. The first one is very obvious and something which every open software project strives to do: identify and try to solve the open issues. Secondly, refactoring and simplifying parts of the code such as the conditional type binder and the semantic analyzer will decrease the probability of bugs in the future, which is essential for code quality. Finally, by means of an extensive test suite, existing features are always ensured to keep working with the addition of new features or the addition of code to fix open issues. In the following sections, we will guide you through the process of upholding the quality standards of mypy.

15.4.2 Delivering fluent updates (CI)

Mypy delivers its code fast by triggering Continuous Integration on AppVeyor for Windows-based systems and TravisCI for Unix ones. While AppVeyor runs 2 simple tests, TravisCI covers a whole set of different Python versions and configurations at the same time. Furthermore, the compiled version *mypyc* is run there to test the new changes of mypy. Concluding, mypy is tested on OSX, Ubuntu and Windows.

Finally, TravisCI is used to build the documentation, trigger a *wheels* build, checking code style and to perform typechecks of *mypy* commands. Especially the latter is just amazing when thinking about it!

The CI pipeline of mypy triggers a suite of tests, which can be run locally as well. This testing library has the nice feature to be able to specify which test to run. For one of our contributions to mypy (PR #8524), being able to run only one test, for a specific feature, instead of the full test suite, would be preferable. With the mypy testing library this is possible using the following command:

```
python -m pytest .\mypy\test\testpep561.py::TestPEP561::test_mypy_nositepackages_setting_accepted
```

The example is an integration test, which sets up a virtual environment for each test. Let's take a look what testing methods are applied to screen mypy in the next section.

15.4.3 Testing: a layered approach

Mypy implements three different kinds of tests in order to maintain quality. It can be seen as a layering, which you might be familiar with; the division into unit, integration and regression tests. On top of these layers there is one final layer involving a linter, to wrap it all up.

15.4.3.1 Unit tests

First there are the unit tests, which test each individual component or class. They are what new contributors often quickly come in contact with, as we have noticed ourselves in trying to get our first pull request in. The way to write such unit tests is clearly laid out in the developer guide. Developers are expected to write their own unit tests, which are more like test case description files. In these text files, labeled with the `.test` extension, these descriptions are parsed by a class in `mypy/test/data.py` to create the test. Most of the tests, simply perform a type check on the code. However, there is a different type of unit test defined in `mypy/test/testpythoneval.py`, which not just type checks a given program, but also runs it.

15.4.3.2 Integration tests

You can see unit tests as ensuring that small individual components work the way they are intended to (very local in the code). Integration tests on the other hand have a more global scope. They ensure that all the

parts put together, the system as a whole, works the way it is supposed to. Integration tests are defined for mypy in test files located in `mypy/test/test*.py`.

15.4.3.3 Regression tests

Regression tests ensure that when new functionality is added, the unit and integration tests are all rerun. This make sure that everything works as intended after the addition of the new functionality. One could run the regression tests from the mypy repository directly by running `pytest`. Pytest is automatically initialized by `pytest.ini`, which specifies the path and python test files which should be run.

15.4.3.4 The final layer

You might think that this would be enough, but wait, there is more. Another layer is placed on top of the tests, which is defined in `runtest.py`. This is the testing entry point, which the documentation in the main repository points to. It runs the pytest described before with the addition of linting with flake8, to check code quality standards.

15.4.4 Analysis of how mypy upholds quality standards

Wait. Quality is standard right? Unfortunately not always. We will analyze Code quality, testing and technical debt to check the way mypy upholds it quality, by looking at the mypy repository, its documentation, Github issues and Github pull requests.

15.4.4.1 Documentation

The developer guidelines located in the Wiki section of the github repository, states code quality guidelines and coding conventions. These are very simple for contributing to mypy in general: basically enforcing flake8. The requirements for code quality in contributing to typeshed, are far more extensive as described here. Besides code quality, there is a reference on how to write unit tests for implemented functionality[10]. There is no reference to the concept of technical debt in the developer guidelines either, which makes sense as only the core team of developers and not your average contributor can identify cruft: “deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further”[11].

15.4.4.2 Issues

When a user wants to create a new issue, they are given guidelines on what information to specify in order to make the process go more smoothly. The core team labels the issue and readily provides feedback to contributers of the community who might want to take a shot at providing a fix. Overall there isn’t a discussion of code quality / testing /technical debt until the user has submitted a pull request and the core team takes it upon themselves to review the code.

15.4.4.3 Pull requests

Large contributions within pull requests are discouraged. The contributer is instead encouraged to split the contributions into several parts, so the reviewing process moves more quickly. In addition to the existing test suite being passed, the user will be encouraged to write their own tests in case the contribution is substantial

[10] mypy Documentation on testing. https://github.com/python/mypy/tree/master/test-data/unit
[11] Martin Fowler. TechnicalDebt. 2019. https://www.martinfowler.com/bliki/TechnicalDebt.html.

enough. The code quality which mypy requires, is checked purely programmatically, as mentioned already in the documentation section above. With regards to technical debt, it is up to the core contributor who has insights into the code base at a larger scale, whether a contribution can be allowed.

15.4.5 Suggestions on how mypy can be refactored

Okay, now we know what quality standards should be upheld, lets review some stuff! The tools we use to review are Sigrid and SonarQube. We have chosen to perform an analysis using both tools to get an even broader review of points where mypy can be improved.

15.4.5.1 An analysis by Sigrid

We have already given you some analysis that Sigrid has performed with the dependency graph in the last blogpost. We will purely consider the refactoring suggestions provided over the entire code base and not necessarily an analysis per component. These refactoring suggestions in Sigrid are called "violations" and there are 4 identified categories which we will discuss for mypy here briefly.

15.4.5.1.1 Maintainability violations Mypy has a considerable amount of maintainability violations listed in the table below.

Violation type	Instances in mypy	Brief description
Unit complexity violation	330	Code units which are excessively complex.
Unit size violation	277	Code units which are excessively large.
Unit interfacing violation	251	Use of too many parameters in calling a unit of code.
Duplication violations	155	Instances where there is duplication in the same or another file.
Component independence violations	29	The ratio of incoming, outgoing, throughput and internal calls.
Module coupling violations	20	There are several incoming calls to this module.

15.4.5.1.2 Severe violations In this category Sigrid has identified 20 violations. Most of them are in `build.py`, i.e. there is an instance where a catch block in a `try-except` type code structure is missing.

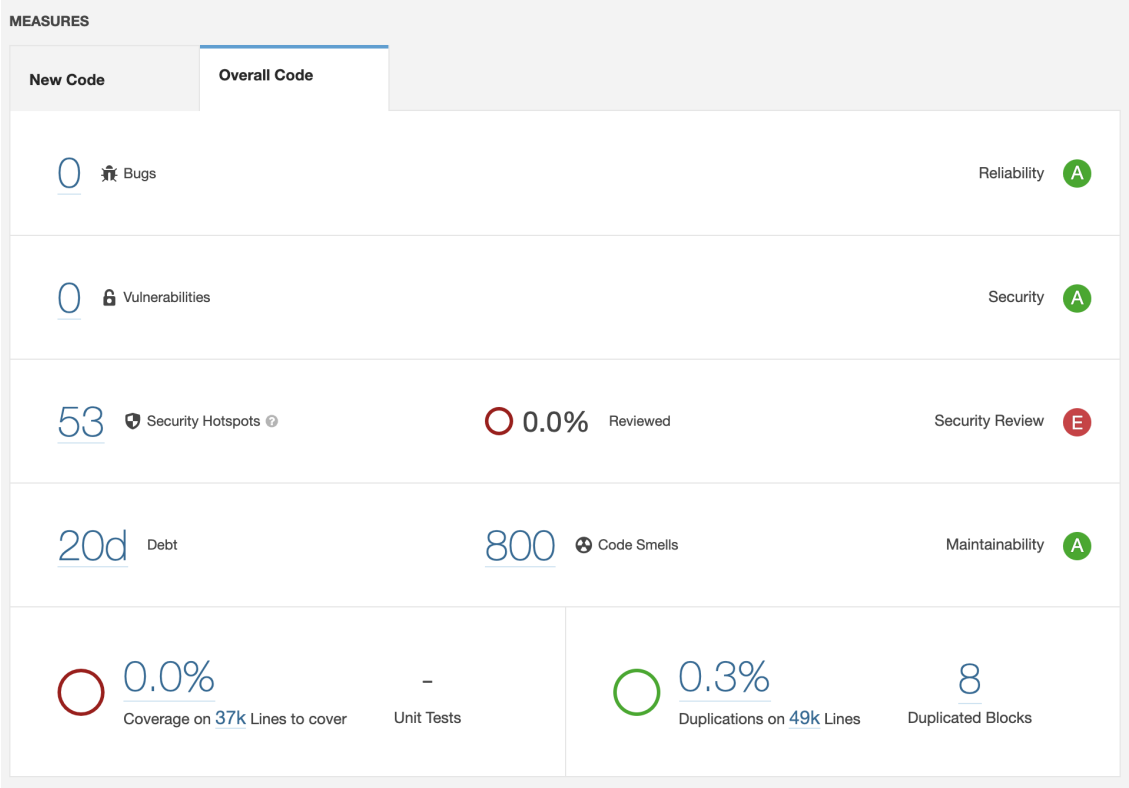
15.4.5.1.3 Warnings Of these Sigrid identifies 335 instances of discovering too many "TODO" and "FIXME" uses in the same file, which indicates poor quality due to incompleteness. Furthermore Sigrid identified 180 instances of lines of code in comments, although by further analysis these have turned out to be (mostly) false positives as they were complementary information that made the code more understandable (i.e. type indication for a variable).

15.4.5.1.4 Code smells Of these mypy has found a considerable amount of violations listed in the table below. This is a selection based on the number of instances, there are other types of violations as well that mypy discovered with a lower frequency.

Violation type	Instances in mypy	Brief description
Redefined symbol	262	Duplicate names in same scope.
Shotgun surgery violation	162	When introducing a small new change would violate the 'Don't Repeat Yourself' principle.
Data clumps violation	46	Data group reappearing as parameter to operations throughout the system.
Extensive coupling violation	43	A class or a module that depends a lot on other classes or modules.
Internal duplication violation	25	Significant duplication within a class or module.
External duplication violation	21	Significant duplication between classes or modules.

15.4.5.2 An analysis by SonarQube

SonarQube Community edition provides the following code analysis results:



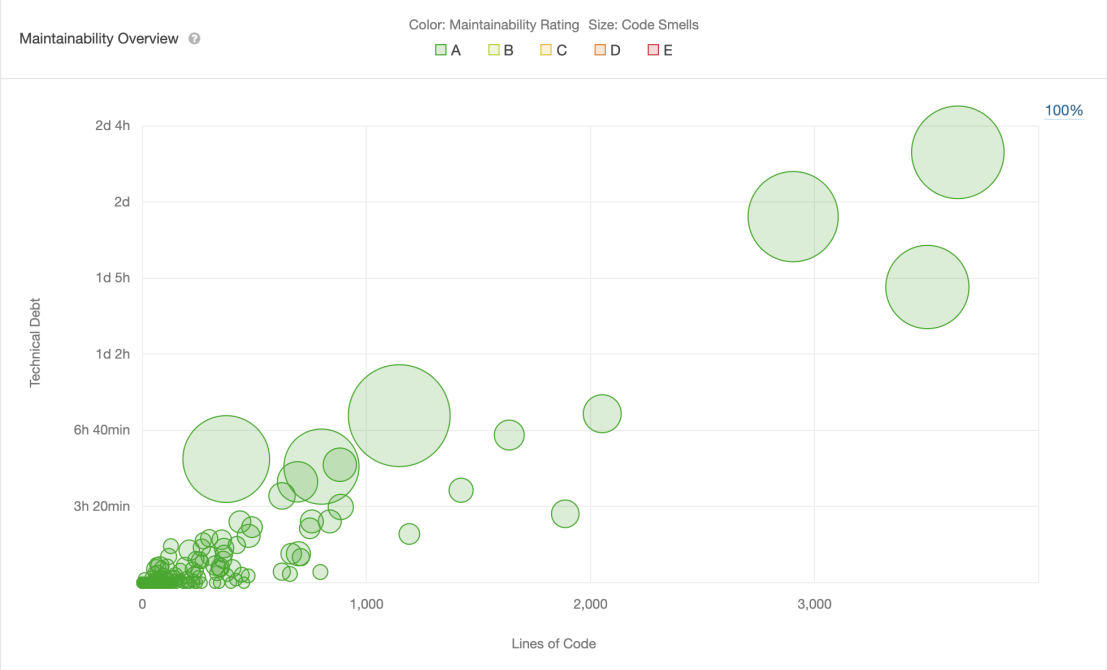
Sonarcube detects no bugs or vulnerabilities. The metric on test coverage isn't very useful. The cause of an indication of 0% coverage is likely due to SonarQube not being able to recognize the tests written for mypy. The most interesting metrics are: Security Hotspots, Technical debt and Code Smells. We will review each

of these more extensively.

15.4.5.2.1 Security Hotspots In terms of security hotspots, SonarQube gives a categorization of the different priorities of possible security threats in the code of mypy.

- **High-priority** threats:
  - Command injections for instances where `sys.argv` is used directly
  - Execution of system commands with `subprocess`
- **Medium-priority** threats:
  - Denial of service attacks by the use of regular expressions
- **Low-priority** threats:
  - This was a false positive where SonarQube analysed a link within the comments and suggested use of `https` over `http`

15.4.5.2.2 Technical Debt In terms of technical debt SonarQube detected an overall debt of 0.7% and a per-file debt of less than 5% for all the source files which qualifies it for a rating of "A". This result is indicated in the chart below:



Even with an "A" rating, improvements are possible. As we will see in the next section, the technical debt can be reduced even further by fixing issues related to code smells.

15.4.5.2.3 Code Smells Code smells aren't bugs in the code, but technical design decisions which might be a contributor to systems technical debt. [12] A total of 800 code smells were detected by SonarQube and

[12] Tufano, Michele, et al. "When and why your code starts to smell bad." 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1. IEEE, 2015.

these are categorized by their severity. Below is a list of the possible classes and examples of the associated issues:

- Blocker code smells: high impact and high likelihood. [0 issues]
- Critical code smells: these have a high impact but low likelihood. [462 issues]
  - refactoring of functions to reduce the cognitive complexity from x to y.
  - defining a constant instead of duplicating a literal. For example the string literal: `builtins.str` which appears in `checker.py`.
- Major code smells: with a low impact but high likelihood. [202 issues]
  - Merging if statements with enclosing ones.
  - Removing or filling a block of code.
    - For instance in build.py line 2388.
- Minor code smells: low impact and low likelihood. [136 issues]
  - Various renaming suggestions to match an expected regex.
  - Removing redundant continue statements.

Concluding the story, we have investigated the mypy code base in search of possible places to improve it. Of course there are many more areas where we could have gone deeper into mypy, but for now we will suspend our writing. No fear, we will return with our last blogpost about the variability of mypy.

15.5 The configuration of mypy

The last blog post was quite a difficult story, wasn't it. In this blog post, we will conclude our mypy investigation series after quite a rollercoaster ride. The final topic will be a bit simpler (but don't underestimate it). This last blogpost will investigate the configurability of mypy and why we would want to configure it.

15.5.1 Huh, Configurability?

So, as stated in the first blog post, mypy is a program that can type check other python files. However, this is not the complete story. The configurability of mypy makes it a nifty tool for many end-users (hey, I've heard that word before!). Because there are many options available for report generation, typing settings and warnings, there is an option for every taste! The upcoming sections will dive deeper into the stakeholders (what is important for them), and what to configure in mypy.

15.5.2 Even more stakeholders?

As said before, people using mypy are the end-users. During the first blog post, we have specified two different sets of end-users. First, the developers (to be more precise the development teams) using mypy for their project. Secondly, the users of type-checked python programs. This last stakeholder group, for instance, includes all people using Dropbox. However, this stakeholder group is not important for configurability. The development teams, in contrast, are highlighted rather vaguely during the first blogpost. A development team of a project has many people with different interests. For instance managers, testers and coders are roles that are important for development teams.

Since it is clear that these roles have different interests in mypy and its capabilities, we will discuss the different variable aspects of mypy first. Afterward, we will again come back to the stakeholders and describe their interest in the variability aspects of mypy.

15.5.3 Lets switch things up

Now that we know a little more about the stakeholders, let's talk about the different options of mypy. As said before, mypy has a very broad series of options available to it, to be able to cater to the needs of many different people. The interesting part of variability is to see what users can switch up. Therefore we have created feature diagrams [13] to discuss the options mypy has, using the documentation of mypy [14]. In the diagram we have merged options into groups, to make the concept easier to grasp.

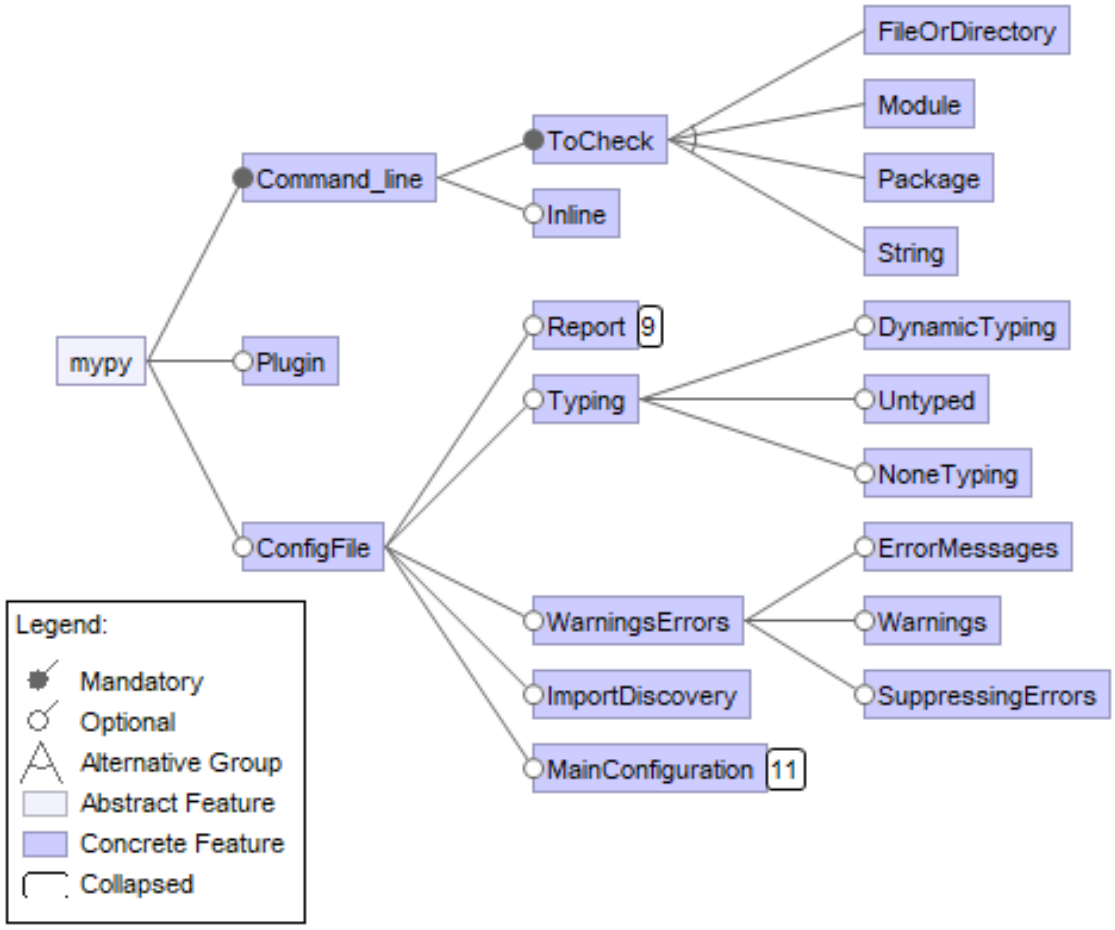


Figure 15.1: Overview of the command line configurable options

In the images, you can see that both the report and main configuration features are hidden. To keep the image clear and readable we have decided to show these parts of the diagram separately. There are two feature diagrams shown above. One of the feature diagrams is for users using the command line interface of mypy, while the other is for users using the daemon.

[13] Feature diagrams as in https://featureide.github.io/
[14] The configuration file as described in the documentation of mypy https://mypy.readthedocs.io/en/stable/config_file.html

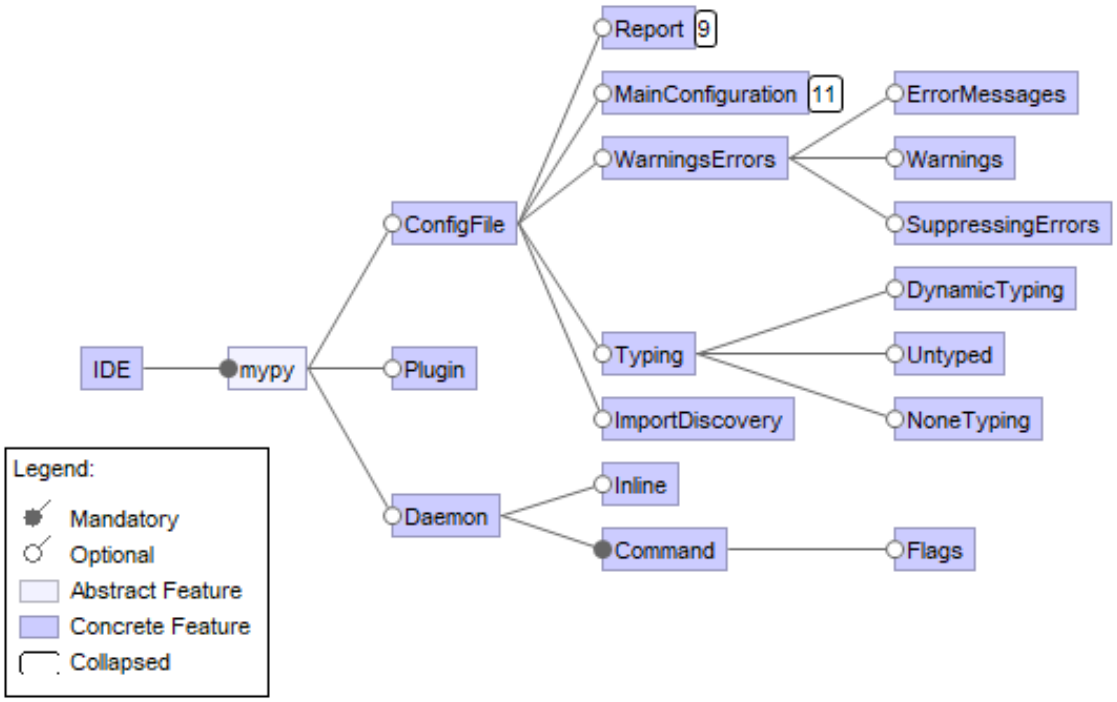


Figure 15.2: Overview of the Daemon configurable options

We will start by discussing some of the higher leaves of the hierarchical tree and gradually descend into the tree.

15.5.3.1 Daemon or command line

So what about those two diagrams. Is one diagram not enough? Well unfortunately not. While some users might want to run mypy directly and check only some files and just once in a while, other users feel the need to type check continuously. To meet the requirements of both users, mypy gives the option to use either the daemon or run the program from the command line.

Both the daemon as well as the command line configurations have the option to use the configuration file or use plugins. However, they do have their distinct advantages, which we will highlight before going into these features.

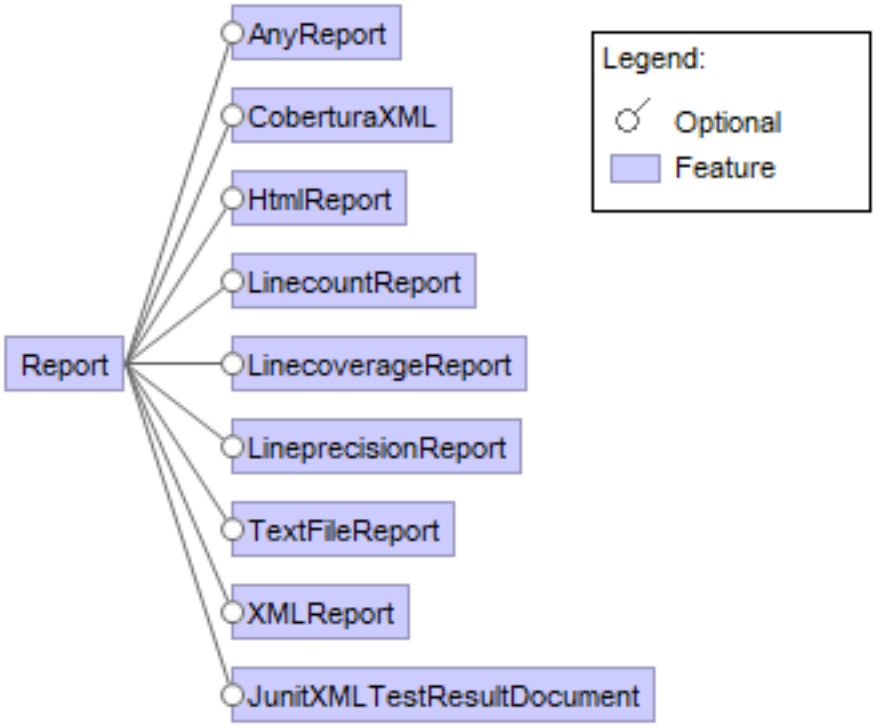
15.5.3.2 Plugins

As the ever-expanding universe, mypy is also possible to expand. The use of plugins makes mypy more flexible. Mainly libraries that do not naturally support the type annotations in PEP484 make it hard to type-check your project. Therefore one could make use of a plugin. A well-known project that can extend mypy is SQLAlchemy. This plugin is useful when working with SQL inside of a project. Mainly the ORM (Object Relation Map) is impossible for mypy to type check correctly and this plugin makes it possible. Very cool!

15.5.3.3 Configuration File

Both the daemon as well as the command line can use a configuration file. This configuration file contains various options that will be elaborated on in the upcoming sections.

15.5.3.3.1 Report generation Mypy supports a variety of different formats for report generation. Since we have collapsed the options in the overview diagrams, an expanded view of the report generation feature will be given in the diagram below.



In the diagram, we can see that there are 9 options for report generation. The options are specified in the documentation. There is a multitude of usages possible for the reports. One could, for instance, be interested in checking the progress of integrating mypy with a line count report (where the lines of typed code and untyped code are counted), but also a nice overview of typing errors could be given in almost any format (XML, HTML, etc).

15.5.3.3.2 Errors/warnings There is also the option to set up how mypy handles errors and warnings. One could turn certain warnings off or can suppress errors. For instance, if a file has a non-fatal error, this can be either reported or ignored. In the feature diagram, we have distinguished between three main features, but each of these features includes a multitude of options [15].

15.5.3.3.3 Typing options Similar to the errors and warning parts of the feature diagram, for each of the endpoint features under the Typing feature, there are many configurable options. Again we tried to combine them for clarity. With the shown options one could change how mypy behaves when encountering code without types, with the `Any` type and which types it can ignore. [16]

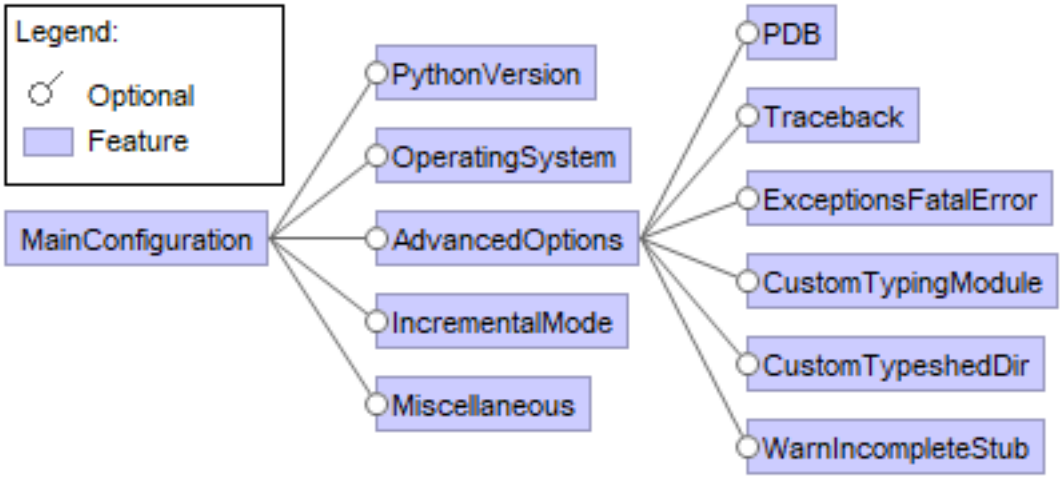
15.5.3.3.4 ImportDiscovery As a developer type checking just one file might be easy. However, if that file uses a lot of different other files via imports, staying on track with the different types in all of these, might be hard. Therefore mypy has options for imports. The behavior can be altered using flags. A lot of previous features are also altered using flags. However, we have chosen to only explain the flags for

[15] The configuration file as described in the documentation of mypy https://mypy.readthedocs.io/en/stable/config_file.html
[16] The configuration file as described in the documentation of mypy https://mypy.readthedocs.io/en/stable/config_file.html

import discovery, to give the reader a feeling for the concept. Getting the idea is more useful than stating the obvious! The following flags can be set

- `--namespace-packages`: enables using namespace packages (explicitly packages without an `__init__.py` or `__init__.pyi`)
- `--ignore-missing-imports`: makes mypy ignore all missing imports
- `--follow-imports {normal,silent,skip,error}`: Adjusts how mypy follows imported modules that are not explicitly passed through the command line
- `--python-executable EXECUTABLE`: If the `EXECUTABLE` provided is PEP 561 compliant mypy will collect type information especially for this executable
- `--no-site-packages`: Disable searching for PEP 561 compliant packages
- `--no-silence-site-packages`: unsuppress error messages generated within PEP 561 packages

15.5.3.3.5 Main configuration Last but not least, we have features which we collected under the name *Main configuration*. The features provided in the diagram below are what we distinguished as primary features.



We will go over each of these features to show what you can do with them.

- `PythonVersion`: Make mypy type check as if the program runs under Python version X.Y
- `OperatingSystem`: Make mypy type check as if the program runs under a specific OS
- `IncrementalMode`: Sets variables to make sure the cache is used properly. The cache speeds up the mypy type checking process
- `Miscellaneous`
  - `--find-occurences CLASS.MEMBER`: Prints out all usages of a class member
  - `--scripts-are-modules`: runs a script (a file without extension `.py`) as if it is a module
- `Advanced Options`
  - `PDB`: will invoke the Python debugger when a fatal error occurs
  - `Traceback`: Displays a full traceback when a fatal error occurs
  - `ExceptionsFatalError`: raises exceptions when a fatal error occurs
  - `CustomTypingModule`: use a custom module instead of the `typing` module
  - `CustomTypeshedDir`: Specifies the directory where typeshed stubs are located

- `WarnIncompleteStub`: gives a warning when included stubs have missing or incomplete type annotations

15.5.4 Melting it all together

Wow, that was a lot to process. While the information given is quite dense and intriguing, the combination with the stakeholders is what makes it interesting. In the following table, we will describe the coupling between them. Please note that some of the features might not occur in the table. This is mainly because they are not relevant to a specific stakeholder. Also, note that there are a lot of stakeholders in a development team (RUP specifies 25 roles according to IBM [17]). We will only cover those that have an interest in one of the variable features.

Stakeholder	Role description	Variability aspect
Software Architect	Designs the global architecture	Main configuration variables, use of plugins, import discovery and typing options
Designer	Design a specific part or module of the architecture	Import discovery and typing options
Implementer	Implements code	Errors/warnings, option to either use the daemon or the command line
Test Manager	Makes sure that testing is complete	Report generation options and use of plugins
Test Analyst	Selects what to test	Report generation options
Test Designer	Designs test and implements them	The complete ConfigFile options
Tester	Runs the tests	Needs documentation only
Deployment Manager	Oversees deployment	Report variability
Project Manager	Oversees the complete project	Report variability
Tool specialist	Creates guidelines how to use mypy or other tools	All possible variability is interesting

Now that we have discussed both the variability options and the stakeholders that benefit from it, we believe the reader will understand our choices made in the table above. Since the main idea behind roles might differ between persons one can always argue whether this table is correct. However, we think it at least gives some insight into why configurability is important.

15.5.5 How it's made

So, now we know all of the possibilities of mypy, we want to give you an idea of how these options are implemented in mypy. The main source of the options is the configuration file. This configuration file will (or will not) be read in `config_parser.py`. This parser file uses standard settings if no configuration file is used, or will load the options specified in the configuration file. We could distinguish an observer pattern in this; options are or are not being set. One could see this pattern back in the way the configuration file is used where flags are set to add or remove functions or options. Since everything is loaded when starting mypy,

[17] Development team roles as specified by IBM https://www.ibm.com/developerworks/rational/library/apr05/crain/

and cannot be changed afterward (except for a few options when using the daemon), we can see this as load time-binding variability. Reconfiguration requires a reboot or restart of mypy.

When developing for mypy one could add options by simply adding functions to the code and adding them in the config parser.

As our last words (we should win the Nobel prize for literature, right?) we want to thank all of you for reading the blogposts. We hope you've learned a lot about software architecture; especially about how it influenced mypy. Sadly everything comes to an end and this was our last blog post. So no next topic and no more joking around. However we have a bright note during this farewell, the website we have posted these blogposts on contains many similar blog posts of all kinds of projects. If you are interested in software architecture, we would recommend going there. Again, lots of thanks for reading!

15.6 Contributor Workflow - an optimization analysis

We're back one extra and final time to explain our experience on working with mypy as an open-source project in hind-sight. Lucky you! Something really interesting happened, while we contributed some functional and documentation-related features to this python typechecking library. We've setup our own workflow, we went through the process of understanding not only the library, but also it's environment and testing system (CI), and now we've got something quite interesting to tell you about what we think `mypy` can improve on!

15.6.1 Workflow: contributing as newcomer

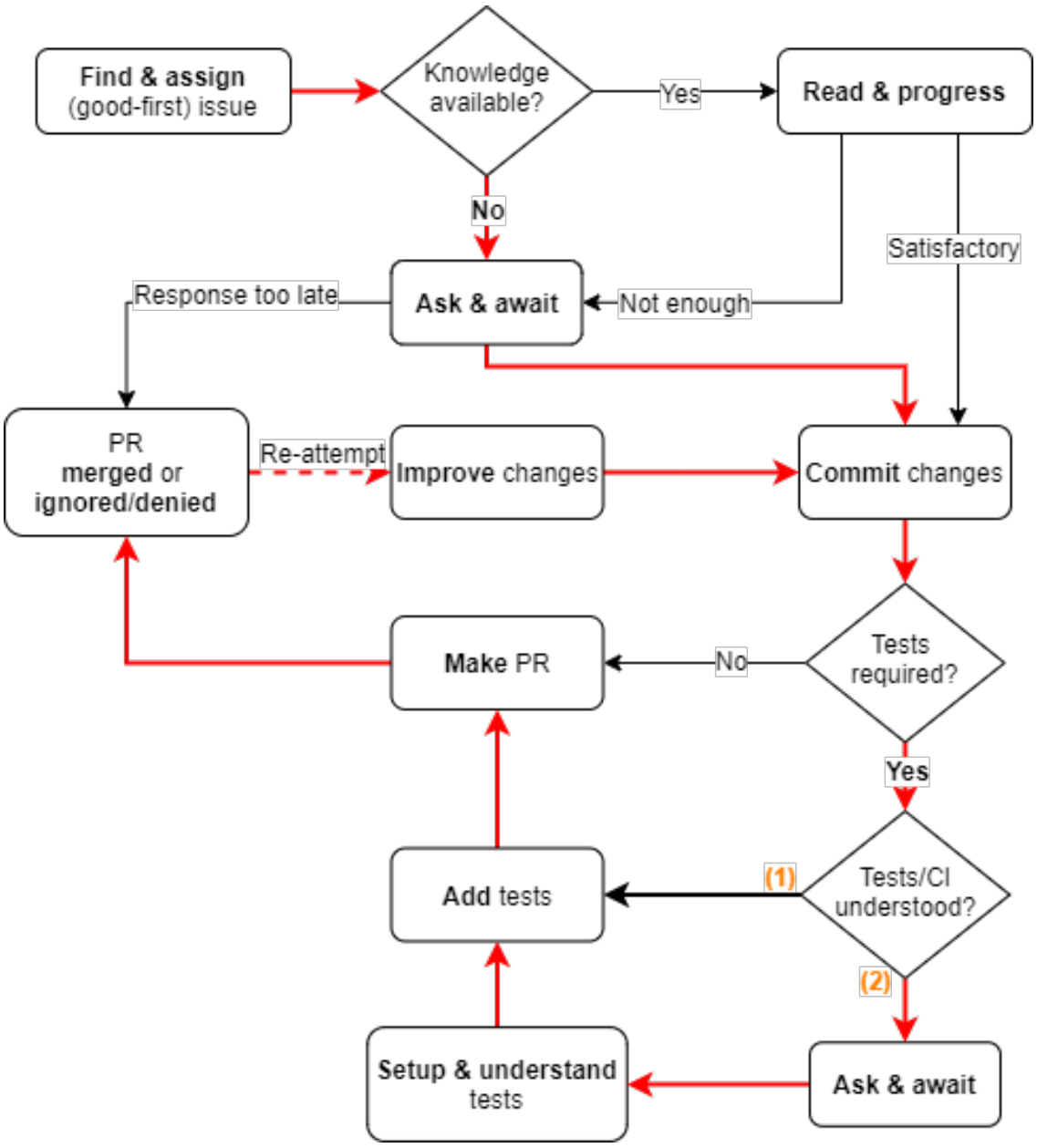
When a developer decides to invest time into an open source project in the form of contributions, there is an incentive to this action. The attention the developer diverts from his/her normal project clearly is halted by some reason. We've found the following reasons to be valid during our past experience on mypy and other software projects:

1) Some feature(s) are missing in the library of choice, making the usage less pleasing.
2) Some bug is holding back the usage, a peer dependency upgrade, or something in the involvement of the library in the original project.
3) Sudden interest is found in the library by an external factor, or a possibility of a great learning experience is met.
4) The library is backed by a company, organisation or group. And a new developer is assigned to the repository's project.

We've found option 1, 2 and 3 to be the most applicable to most of our open-source contributions or usage of libraries. Assuming scenario's 1-3 to actually be the most occurring, it seems of great importance the the open-source project to guide the developer as much as possible. Only proper guidance through **documentation** can result in a compact and effective workflow for contributing changes. We've noticed that mypy tackles this as follows:

- Assume that a developer finds an issue (possibly low-hanging fruit with `good-first-issue` tag.)
- Developer inquires about issue, possibly with or without response.
- Contributions are made through a fork of the repo and a PR.
- (Optionally) tests are added.
- The PR might be: ignored, abandoned or completed.

However, the listing above is too simplistic in some ways. When software contribution is done, the developer might spend more time waiting on feedback and information than actually on programming. The following figure shows how the critical and worst-case path in the contribution cycle for a new-comer might be:



> Critical path of development, with (1) and (2) being critical choices.

The first challenge arises at the `Knowledge available?` question. Since mypy has a lot of documentation, the developer at hand might be able to get further. A big portion of the crowd however, lost their attention at this point. After the a contribution is made, we arrive at the `Tests/CI understood?` challenge. This

process is standardized partially for mypy, its built up in quite a complex manner, but also poorly documented by multiple documents placed in a unfortunate place. We believe that mypy can easily improve its *developer documentation* to reach path **(1)** instead of path **(2)**.

15.6.2 Testing: understanding pytest

As we've discussed in a previous blog, in order to contribute to mypy one has to understand how it uses `pytest`. This library is quite common in the realm of python as a go-to test library. We covered how *pytest* is extensively used for regression testing of which unit tests and integration tests form the basis. However, we did not cover how mypy runs both python-based `TestCase` instances as well as `DataSuite` instances. The latter suite specifies test-data files to run, which provide variability coverage as well as compactness.

We will first explain how the tests are discovered, and then come back to these test suites.

15.6.2.1 Autodiscovery of test

As we've covered, pytest runs the test by scanning the subfolders for any files with the 'test' prefix or postfix in their name. Secondly, `test`-prefixed functions or methods (as well as Test-prefixed classes) are added as testing instances. The pytest runner for mypy starts running multiple workers in parallel to execute the tests.

Now, simply running `pytest` in the folder will take a long time due to a lot of discovered tests, so the developer will have to look up how to filter on their specific tests. This can become quite a complex call, but it is worth the speed: > `python -m pytest .\mypy\test\testpep561.py::PEP561Suite`

We are calling the python file `testpep561.py` and containing the test-suite `PEP561Suite`. This `DataSuite`-extending class specifies a `files` list which refer to one or more *.test files. Those test files have a special syntax, which needs to be parsed, validated and understood by a test-class like `PEP561Suite`.

Now, you might wonder, why are we going into so much detail about this `DataSuite`? The reason is: we are trying to show the many steps a developer needs to discover in order to finally be able to know where to look. Only then can they start learning about the syntax of these compact test files, which is not for the feint of heart. We'll give the simplest example to be found:

```
[case testFunction]
import typing
def f(x: 'A') -> None: pass
f(A())
f(B()) # Fail
class A: pass
class B: pass
[out]
main:4: error: Argument 1 to "f" has incompatible type "B"; expected "A"
```

This file is used as input for one of the many `test*.py` files. It is up to the developer to find which one and based on what criteria (validations)!

15.6.3 Testing, a design debt analysis

The library `mypy` is becoming quite complex by the day on a functional level, but the testing process could and should be simplified more. We've shown that the contribution workflow for a newcomer is possibly

obstructed by lack of knowledge. Secondly, we've hinted at the complexity of the data-driven tests, which is a big portion of the tests run for mypy.

The data-driven tests are definitely the most compact and readable way to understand the mypy tests run. The challenges however are four-fold: - The developer needs to know which python test-file he should adjust/add. - The developer needs to know how mypy runs pytest TestCase instances, let alone tests-suites like DataSuite. - The developer needs to understand how to find the data file(s) for the Data-driven test. - The developer needs to know how to add tests (syntax) as well as how to expand the validation/pre-conditions (in the python DataSuite).

As might be clear, this is a lot to learn before actually being able to contribute!

15.6.4 Advice on more test-driven design

We don't think that mypy is moving in the wrong way by choosing data-driven tests, but we see one primary problem: tests should be as simple as possible and their information should be as contained as possible.

Our advice for mypy is to:

1) Place the data files closer to the data-driven tests near the tested files.
2) Add a very up-front testing documentation (link to github PR) for data-driven tests.
3) Auto-discover test data files like this issue and automatically couple the files to python test-cases.

These options can massively improve the critical path shown in the workflow image presented above. With these tips, we hope you enjoyed this final blog from our side and hope you have become enthousiastic about open-source software, or even about using mypy!

Chapter 16

Next.js



Chris Lemaire, Fabian Mastenbroek, Christian Slothouber
Delft University of Technology

Next.js is a framework to help programmers build fast and modern web applications. Its main features are server-side rendering, easy routing, hot code reloading, automatic code splitting, prefetching of next pages and an easy deployment process. Next.js achieves these goals by being built on React and Node.

During this blog, we will discuss the architecture used by Next.js to achieve its goals. In our first post we will identify these goals in more detail and in later posts we will dive deep into the analysis of the project.

16.1 Next.js: Back to the Future

Traditionally, when browsers were much less capable than today, websites were mostly static with the logic being performed on the server. Back then, most websites served some HTML that was rendered using a templating language such as PHP, which would then be taken over on the client by a *different codebase* (powered by jQuery or similar).

However, with the rise of affordable processing power for many and the increasing capabilities of modern web-browsers, we have seen a paradigm shift to interactive web applications where most work is done in the browser. Consequently, popularized by modern JavaScript frameworks like React, many websites are now built as a Single Page Application (SPA) in which the entire content and logic is encapsulated within a single codebase, improving responsiveness and adaptability.

While Single Page Applications allow developers to embed their application logic into a single codebase, they suffer from some serious issues. Most notably, webpages now need to fully loaded before the browser will be able to show content. This in turn affects search engine crawlers, which might not be able to propely index the website[1], affecting its SEO ranking[2].

Next.js strives to combine the best of both worlds in the form of universal JavaScript applications. By using the power of React, Next.js takes on both roles, by rendering webpages in advance on the server similar to traditional websites, but also resuming the rendering process on the client, enabling intricate code sharing between client and server.

16.1.1 Universal React Made Easy and Simple

To bring this vision forward, Next.js has set out a few primary goals:

- To ease development of server-side rendered React apps.
- To create modern web applications that load as fast as they can.
- To be easy to deploy and test, to speed up the development process.

16.1.2 What Does the User Want?

To understand what Next.js is and how it accomplished its goals, we have to dive into the mind of a user. In this section we investigate who the users of Next.js are and try to understand what they want. For Next.js there are two main types of end users we must keep in mind: the developer and the common web application user.

16.1.2.1 The developer

One may use Next.js to simplify their time spent developing a web app. Next.js provides a very easy setup through `create-next-app` and an immediate development environment with `yarn dev`. Additionally, it takes away difficulty in developing by implementing code splitting and prefetching.

16.1.2.2 The web-user

The common web application user wants any page they wish to see to load quickly. They want to navigate fast and without errors. Next.js helps this by making it easy to have an effortlessly created mapping from routes to pages. By performing server-side rendering, automatic code splitting and prefetching, Next.js tries to load the first page a user sees as fast as possible and fetch other resources in the background.

16.1.3 Six Principles of Next.js

To achieve its vision, Next.js is built around six core principles:

[1] https://www.javascriptstuff.com/server-side-render
[2] Search Engine Optimization, which is the process of increasing quality and quantity of website traffic.

1. **Zero setup**: Next.js believes project do not need setup to get started. Instead, it provides simple filesystem based routing to quickly create new webpages. For more advanced use cases, Next.js allows users to take control.
2. **Automatic server rendering**: Next.js provides by default server side rendering and code splitting. Building the pages on the server reduces the number of round-trip messages which in turn reduces latency. This guarantees that pages are still easily indexed by search engines like Google.
3. **Performance out of the box**: Next.js incorporated features to increase the performance of the created webpages. For example automatic code splitting for faster load times and client-side routing.
4. **Serverless functions**: You can build your API using the routing functionality and serverless functions.
5. **Static exporting**: This feature allows the user to create static website with all the benefits the bring to the table. While doing this it is still able to use modern front-end features and a modern developer workflow.
6. **Styling**: Next.js offers built-in support for CSS-in-JS. The benefits of this feature are that it is component friendly, intuitive, automated and scalable.

16.1.4 Who's in?

The vision of a software product is steered by the stakeholders. To be successful as software means to tend to the requirements of the stakeholders. Why the requirements of the users matter is almost trivial, the users will only use the software if they need the software. It extends however to other stakeholders as well. For instance the development is open source and therefore relies upon the open source community. This means that the code and documentation has to be of a certain standard and quality.

We identified the stakeholders according to the classification of Rozanski and Woods[3]. We did add the End-User classification as we think that their is a significant difference between the "user" of Next.js and the "end-user".

Type	Stakeholder	Description
Acquirers	ZEIT	Next.js is a product owned by ZEIT
Assessors	ZEIT	Since ZEIT owns the software they are also responsible for standards and legal regulation
Communicators	ZEIT, GitHub Community, Bloggers and others	ZEIT offers a range of guides, tutorials and examples on their website and GitHub. These examples are created by the contributors as well. Bloggers and other content creators also write articles and posts on the system.
Developers	ZEIT and GitHub Community	ZEIT has developers working on Next.js and since the project is open source and open for contribution the GitHub community helps in the development as well
Maintainers	Developers	Most notably the GitHub users: @timneutkens, @ijjk and @Timer

[3] N. Rozanski and E. Woods, *Software systems architecture: working with stakeholders using viewpoints and perspectives.* Addison-Wesley, 2012

Type	Stakeholder	Description
Production Engineers	Dependency owners	Next.js is dependent on a lot of other frameworks, libraries and software. Most notably the React framework, Babel and Node.js
Suppliers	npm and GitHub	The Next.js dependency can be retrieved via npm or the source code can be downloaded from GitHub
Support Staff	ZEIT, GitHub Community, Developer Community	Issues and problems with the software are addressable and solved on GitHub. Forums like StackExchange are also actively offering support.
System Administrators	Users	Since Next.js is a framework that means that in principle the responsibility of the working system lies with the user. ZEIT does offer hosting against compensation
Testers	Developers	According to the CONTRIBUTING.md the developers are responsible for documenting and testing the software
Users	Organizations and Individuals that use Next.js to power their website	Some notable companies that use Next.js to power (a part) of their website are: Netflix, Uber, Nike, AT&T, Docker and others
End users	Actual users of website created with Next.js	The requirements of the End-User is given to the respective User. This requirements might reflect on Next.js and are therefore the End-User is a stakeholder in Next.js

16.1.5 Next.js in Perspective

Software never exists in isolation, but instead manifests itself as an element in a larger ecosystem. The stakeholder analysis that we performed in the previous section is just one of the many facets of this ecosystem. To truly understand the Next.js project, it is crucial that we consider how the project operates in its greater context. Formally, Rozanski et al. define this context as the relationships, dependencies and interactions between the system and its environment, consisting of the people, systems, and external entities with which it interacts[4].

We have identified several relevant facets of the Next.js ecosystem and depict them in the figure below:

[4] N. Rozanski and E. Woods, *Software systems architecture: working with stakeholders using viewpoints and perspectives.* Addison-Wesley, 2012

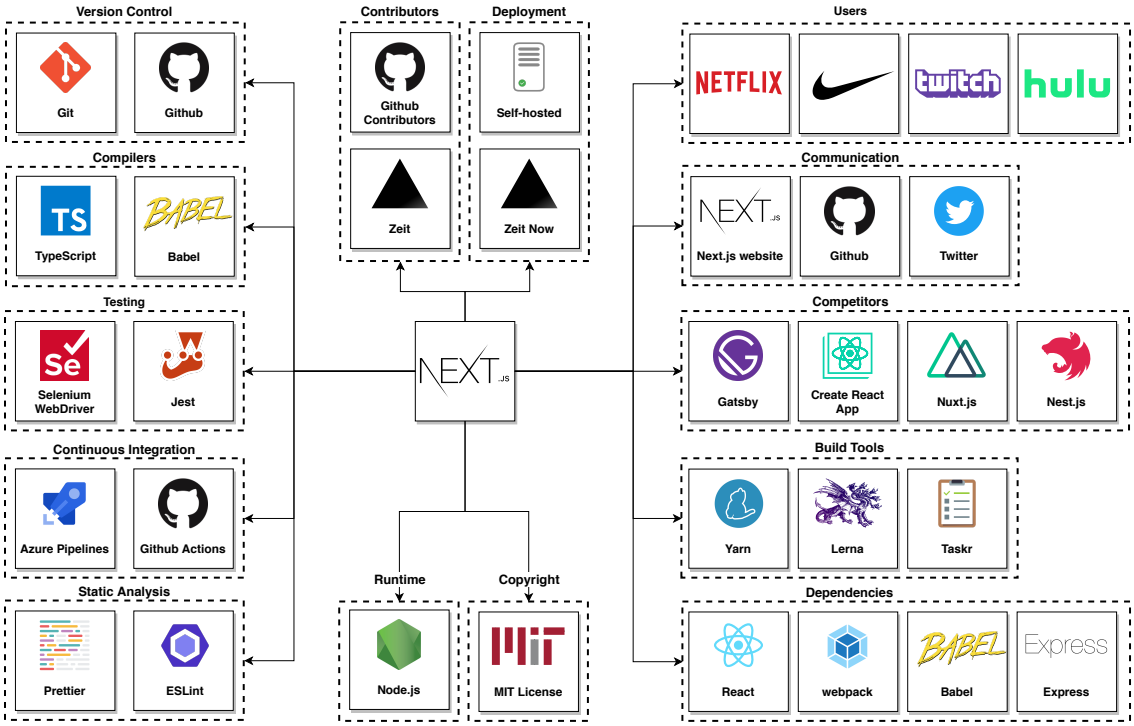


Figure 16.1: The Next.js ecosystem

Next.js is an open-source project developed on Github (**Version Control**), written in JavaScript[5] and TypeScript (**Compilers**) that runs on Node.js (**Runtime**).

The project uses Lerna for organizing and managing its multi-package repository. Yarn and Taskr are then used to manage dependencies and run the build scripts (**Build Tools**). During runtime, Next.js relies on other libraries to provide its core functionality, most notably React and webpack to provide (**Dependencies**).

Next.js is tested using Jest in conjuction with Selenium WebDriver (**Testing**) on both Github Actions and Azure Pipelines (**Continuous Integration**). To keep code consistent and maintainable, Next.js uses ESLint and Prettier for respectively linting and formatting (**Static Analysis**).

The project is developed by both ZEIT Inc. and external Github developers (**Developers**), with the source code made available under the MIT License (**Copyright**). Most communication between developers is done through Github issues and also the recently introduced discussions functionality. The Next.js website hosts the project's documentation and blog, with Twitter being used for announcements (**Communication**).

Next.js is used by a diverse community that also includes large companies such as Nike, Netflix and Hulu (**Users**). These users may choose to deploy their Next.js application using the ZEIT Now platform or decide to host the application themselves (**Deployment**).

Next.js competes with projects such as Gatsby and Create React App that share a similar vision, but approach the problem from different angles (**Competitors**).

16.1.6 What's Next?

Software is never finished, only abandoned[6]. Instead, software evolves as requirements change over time. Projects may use roadmaps to communicate the the key priorities of a project and illustrate how these align with the project's vision and strategy. Roadmaps inform the customers what they can expect from the project in the future. Moreover, they help developers align their contributions with the project's directions.

Unfortunately, the Next.js project does not have a public high-level roadmap. Instead, ZEIT has chosen to maintain a private roadmap and in addition use milestones on Github to disclose any part of the roadmap that can be made public, as issues under these milestones[7].

At the moment, Next.js has milestones for the next minor and patch versions of the project, respectively `9.3.x` and `9.3.0`. Issues and pull requests are linked to a specific milestones to indicate which future release they are targeting. However, a concrete timeline for these changes is missing as the maintainers do not specify a due date for these milestones.

While current approach of ZEIT conveys more detailed information about the changes to appear in a release to external contributors, the high-level goals and overall direction of the project are not clear, possibly hindering external contributions[8]. Moreover, the milestones are not frozen, with as result that new changes may be added to a milestone even after its start.

Though, we can still infer some future plans on the roadmap based on the public communication of maintainers inside the repository. We have identified three important changes that are currently on the roadmap of the project:

[5] Next.js uses the Babel JavaScript transcompiler to convert newer JavaScript features into a backwards compatible version of JavaScript.

[6] J. Saddington, *Software: Never Finished, Only Abandoned.* Blog, 2014

[7] https://github.com/zeit/next.js/issues/8449

[8] https://github.com/zeit/next.js/issues/8449

- Improved Static Site Generation (SSG)[9]
- Support for React Concurrent mode and streaming rendering[10]
- Support for partial hydration[11]

16.2 Architecting Next.js

In our previous essay, we introduced you to Next.js, a modern web framework for building React applications. We examined the vision of Next.js, analyzing the set of fundamental concepts and properties of the project and considering the project in its greater context. With this knowledge in mind, we can now try to understand this vision is realized through its architectural elements and relationships, and the principles of its design and evolution.

16.2.1 The Multiple Faces of System Architecture

The architecture of a complex system usually encompasses a multitude of different aspects, ranging from its functional structure, intercommunication protocols to the development and deployment process.

While it might be tempting to capture and address all of these aspects into a single monolithic model, such models are difficult to understand and unlikely to be useful for highlighting the architecture's key features. They tend to become very complex, operating at multiple abstraction levels at once, and consequently individual stakeholders will struggle to understand the aspects that interests them.

An alternative approach is to instead address the problem from several different directions separately. In this approach, the architecture is comprised of a number of separate but interrelated *views*, each addressing a distinct aspect of the architecture, collectively representing the entire system architecture.

To establish a common language for discussing system architectures, several authors have proposed frameworks for designing such views. Kruchten[12] was first to propose a common set of views, consisting of four reusable views, namely, Logical, Process, Physical and Development. An idea that was later adopted and genericized by IEEE[13] into the concept of a *viewpoint*. They define the principles and language for constructing and analyzing views.

In this post, we will analyze Next.js' architecture of using the seven core viewpoints proposed by Rozanski et al.[14] and depicted below. We will discuss whether these viewpoints are relevant for Next.js and how they apply to the project.

1. The **Context** view considers the broader context of a system, detailing the relationships, dependencies and interactions between the system and its environment. This is important because software does not exist in isolation. This holds true for Next.js as well, which is why we already went into depth into this aspect in our recent post about Next.js' vision.

[9] https://github.com/zeit/next.js/issues/9524
[10] https://github.com/zeit/next.js/discussions/10741#discussioncomment-745
[11] https://github.com/zeit/next.js/issues/10344
[12] P. Kruchten, *Architectural Blueprints - The "4 + 1" View Model of Software Architecture*. IEEE Software 12, 1995
[13] https://standards.ieee.org/standard/1471-2000.html
[14] N. Rozanski and E. Woods, *Software systems architecture: working with stakeholders using viewpoints and perspectives.* Addison-Wesley, 2012
[15] N. Rozanski and E. Woods, *Software systems architecture: working with stakeholders using viewpoints and perspectives.* Addison-Wesley, 2012

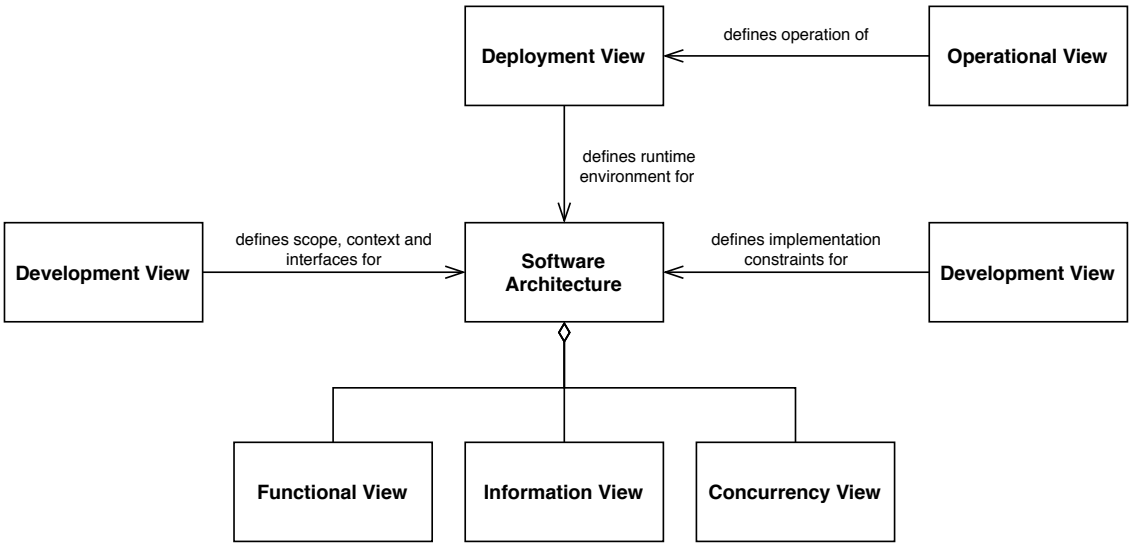


Figure 16.2: A Viewpoint Taxonomy[15]

2. The **Functional** view of a system characterizes the system's functional structure, documenting key functional elements and their responsibilities. This forms the foundation of the architectural description, driving the definition of the other views. Undoubtedly, this also forms the basis for the architecture of Next.js.
3. The **Information** view portrays the functional and non-functional properties of information in a system (such as structure, purpose or ownership) and highlights how the information is stored, managed and manipulated in the system. This is only of small importance for Next.js as it is unopinionated, but more importantly unaware of how information is managed throughout the application, leaving it up to the user and only providing generic assistance in the form of data fetching.
4. The **Concurrency** view describes the concurrent structure of the system and details how concurrent execution is coordinated and controlled. Since Node.js is inherently single-threaded[16], concurrency management within the Next.js code base is greatly simplified. However, this view is not totally irrelevant for Next.js. For example, several processes of building a Next.js application have been parallelized to improve performance and user productivity. Moreover, anticipating the release of Concurrent Mode and the streaming server renderer in React, Next.js needs to support concurrent renders and thus be careful with global state.
5. The **Development** view addresses the aspects of planning and designing the development environment required to support the system development process. For the Next.js project, this is most certainly a relevant view. Considering that Next.js is open source project and relies on many external contributors, it is key that the development process is clear and transparent.
6. The **Deployment** view characterizes the deployment environment and its interaction with the system during runtime. We already briefly touched upon how Next.js applications can be deployed in our last post. Deployment is an integral process of each Next.js application, given that a Next.js application is of no use if it's not deployed and we cannot access it.
7. The **Operational** view describes how the system will be operated, administered, and supported when

[16] Well, technically Node.js now supports multiple threads via the worker_threads module, but shared memory is still rather restricted.

it is running in its production environment. Whether this view is applicable to a Next.js application depends mostly on the particular application. For Next.js itself, this view might not be that interesting as the operational process consists mostly of applying eventual bug fixes and improvements, which we may as well cover in the development view as maintenance component.

16.2.2 Architectural Styles and Patterns

Before we go into depth into the different views of Next.js' architecture, let's briefly touch upon the main architectural styles or patterns that have been applied in the code base.

The Next.js project is architecturally organized as a monorepo. That is, its code base is not a single monolith but is instead comprised of several smaller packages that are developed in a single Git repository. We already saw last week that Next.js utilizes Lerna to optimize the multi-package workflow. Monorepos massively simplifies the organization and tooling of a large project such as Next.js, but also allow for atomic commits or large scale refactoring between packages.

Moreover, Next.js is built on the Node.js platform, which makes heavy use of asynchronous programming due to its single threaded nature. Due to its reliance on callbacks to implement this pattern, Node.js applications tend to suffer from what is known as the callback hell[17], where callbacks are nested within callbacks several levels deep making the code difficult to understand and maintain. Next.js has alleviated this issue by making use of the `async-await` functionality that was introduced in the ES8 standard, which allows developers to "linearize" asynchronous code.

16.2.3 Development View

We have analyzed the Next.js repository's directories and files. In this section this analysis is condensed and explained in an almost neurotically structured way. The goal is that new coming developers and users can use this familiarize themselves with the project.

After analyzing the repository of Next.js we found that every file and directory could be categorized into on of three categories: documentation, testing and actual source code.

16.2.3.1 Documentation

The folders `docs`, `errors` and `examples` all three function as documentation for the developer and the user.

The directory `errors` contains a collection markdown files. The files are used to explain the large variety of error messages one might receive. Each error message has its own file in which is explained what the error is, why it occurred and possible ways to fix the error.

Developers of Next.js are stimulated to create examples showcasing the features developed. The results is that in the folder `examples` over 200 examples are committed. These examples vary from the most basic features like the pages functionality to more advanced examples like progressive rendering.

The documentation in the `docs` folder is more traditional. It contains among other things a getting started guide, a FAQ and an API reference.

[17] http://callbackhell.com/

16.2.3.2 Testing

As the name suggests the folder `test` contains the tests for the project. The folder is subdivided into sub-folders each for different types of tests, for instance unit tests and integration tests.

The folder `bench` contains server-side benchmarks for Next.js. With a few commands the benchmark can be run to test the performance.

16.2.3.3 Source Core

The folder next contains the core components of Next.js.

Name	Description
bin	The main of next
build	Responsible for building next projects. Dependencies like Babel and WebPack are used here.
cli	The command line interface commands are implemented in this component
client	Client side code is implemented in this component. For example linking, page loading and polyfills.
export	The feature to export to a static website is implemented in this package
lib	Universal utilities are implemented in lib. For example finding the config and pages folder, constants and checksum checking.
next-server	Public API of the next-server. The component contains the source for for example rendering and routing.
pages	The pages functionality is implemented here
server	Private code for the server. Functionalities like hot-reloading and html escaping are implemented right here.
telemetry	Next.js collects by default general telemetry data. This data is collected in a anonymous fashion and can be turned of with a single command [^telemetry]. The package telemetry contains the code for this feature.
types	The type definitions for Next.js

16.2.3.4 Source Plug-ins

Name	Status	Description
create-next-app	Production	The folder create-next-app contains the code for the create-next-app command [18]. The command to create a fresh new Next.js project.
next-mdx	Production	The folder next-mdx contains the code for the optional mdx [19] plug-in, a plug-in for using JSX into Markdown.
next-bundle-analyzer	Production	The folder next-bundle-analyzer contains the code for the optional plug-in of the WebPack bundle analyzer [20], to visualize the size of files in an interactive treemap.
next-plugin-google-analytics	Alpha	Plug-in for Google Analytics
next-plugin-material-ui	Alpha	Plug-in for Material UI

Name	Status	Description
next-plugin-sentry	Alpha	Plug-in for Sentry

16.2.4 Runtime View

When building a Next.js project, with for example `next dev` or `next build`, a variety of tasks are performed. Most notable actions taken are that Babel is used to compile the next generation JavaScript source to browser compatible JavaScript.

Secondly, people use a wide variety of different browsers and different versions of those browsers as well. This means that some clients have different functionality in their browsers. To fill some gaps in the functionality polyfills are added to make the project compatible with older browsers.

Thirdly, webpack is used to dynamically process and bundle all source code, dependent styles, assets, and images into static assets.

16.2.5 Deployment View

We also want to take a look at how Next.js is used in reality. How is Next.js distributed and what is needed to deploy a Next.js application? We see that the deployment of Next.js can once again be arranged into two categories:

1. The distribution of Next.js framework and related products over package managers.
2. The deployment of products/websites created with the Next.js framework.

16.2.5.1 Distribution of Next.js

The Next.js project consists of a number of NPM packages organized within a directory in the Next.js repository. These packages are distributed via the npm repositories. Next.js publishes all packages for every release done on Github, which results in about 40 updates to the released NPM packages this month alone.

16.2.5.2 Deploying Websites with Next.js

Next.js is a framework for building websites. To simplify deployment of applications built with Next.js, ZEIT offers a service for hosting such websites called ZEIT Now[21]. Additionally, ZEIT Now offers a significant speedup for Next.js applications by specifically tailoring towards this framework.

Aside from deploying via ZEIT Now, a developer can also choose to host the Next.js application themselves. To host an application yourself, you can just use Node, as a Next.js server can be run like a regular Node server. This mode of deployment lacks the benefits that ZEIT Now gives (the speedup, ease of deployment, etc.), but it remains flexible and gives companies the possibility to host all their services themselves. In addition to this, Next.js offers an option to export your website as static HTML for even easier deployment.

[18] create-next-app
[19] JSX in Markdown
[20] WebPack Bundle Analyzer
[21] ZEIT Now

16.2.6 Non-Functional Features

For this section, we picked a few non-functional aspects from the ISO25000 standard[22] to take a look at. We try to see how Next.js introduces features that improve these aspects.

16.2.6.1 Performance

The performance of Next.js over other web frameworks is largely impacted by its main premise: server-side rendering. Server-side rendering means that a bulky server computer handles assembling web pages before they are served. This means that work is taken away from the browser requesting the web page, resulting in a web page that will load a lot faster in comparison to a client-side rendered application. Additionally, Next.js does another optimization called Automatic Static Optimization[23], which allows pages to be statically rendered when possible.

16.2.6.2 Compatibility

A problem currently existing with client-side rendered applications is that they are not easily search-engine optimized[24], meaning these pages cannot be searched effectively. Using server-side rendering, Next.js is compatible with the current standard.

16.2.6.3 Usability

Next.js heavily depends on the existing React framework to populate its webpages. This framework is well known in the web development community. As such, it provides solid syntax well-known by those who have used it before. This makes it easier to learn Next.js after having previous experiences with React.

16.2.7 Conclusion

This concludes already the second blog post of our series on the architecture of Next.js. We have learned how distinguish between different aspects of a system architecture by means of *viewpoints*. Then, we have analyzed for different viewpoints have analyzed whether and how these viewpoints apply to Next.js. Next week, we will put our focus on the quality and architectural integrity of Next.js and asses the technical debt that's present (if any) in the system.

16.3 Can Next.js Stay Ahead: A Case of Quality Control

Two weeks ago, we took a stab at exploring and identifying the architectural elements and principles that underlie Next.js, a modern web framework for building React applications. Together these elements and principles realize the set of fundamental concepts and properties that form Next.js in its environment. If you haven't done so yet, make sure to also read our first post on the vision of Next.js, in which we go more into depth into the foundations of its architecture.

Software architecture is however not only relevant during a system's construction and establishment, but during its entire lifetime. Software systems are like living organisms: alive and constantly evolving over time as their environmental factors change[25]. And like living organisms, as they attempt to adapt to new

[22] ISO 25000
[23] Automatic Static Optimization
[24] B. Burkholder, *JavaScript SEO: Server Side Rendering vs. Client Side Rendering*
[25] N. Sussman, A software system is a living organism. Blog, 2015

environments, they will inevitably acquire alterations to their architecture. In biological systems, this happens at random, but only the ones that can properly adapt survive. Fortunately, for software systems this scenario is more optimistic since we are (mostly) in control of the changes to our systems. Yet, we must be careful not to accumulate technical debt, which arises when cutting corners during development. While technical debt is not inherently bad, too much of it will lead to decay of the system, threatening architectural integrity and hindering future changes.

Thus, our analysis of Next.js' architecture is not complete without also including in our analysis a discussion about the processes that safeguard the project's quality and architectural integrity, such that its maintainers can anticipate future changes. This aspect is especially important for a project like Next.js because it is a framework for building (web) applications. After all, how can expect developers to uphold the quality of their Next.js application, if the maintainers of Next.js fail to do them themselves.

Hence, today, we will put Next.js' codebase under the magnifying glass and asses whether the maintainers of Next.js have properly upheld the quality standards and architectural integrity that the project strives for. We will do this in four parts:

1. **The Strategy**: We identify and analyse the processes and methods put in place by the maintainers of Next.js to safeguard the quality and architectural integrity of the project.
2. **The Evidence**: We asses the effectiveness of this strategy by evaluating Next.js' codebase from various aspects, such as code complexity, hotspots or test adequacy.
3. **The Prospects**: We attempt to understand how future changes (on the roadmap) could affect architectural components of the project and discuss how we can anticipate these changes.
4. **The Verdict**: We formulate our overall judgment of the quality of the project and conclude our series of essays.

16.3.1 The Strategy

To safeguard quality and architectural integrity, software projects often rely on some form of software quality process, in which they establish what constitutes quality in its context and how its maintainers intend to reach the quality objectives. Some projects (such as Material-UI[26]) do this by defining explicitly in the contributing guide what they expect from contributions, such as Git conventions, documentation, code formatting or tests. However, for many projects, this is not the case and instead they rely on some implicit definition of the software quality process on which the maintainers agree.

This holds for Next.js as well. While Next.js does have a contributing guide, it unfortunately goes into little detail about what the maintainers expect from (external) contributions in terms of quality. Discussion about quality and architectural choices is rather limited in the public repository. There are a few issues[27][28][29] and pull requests[30] that offer some words on the quality and architectural choices of Next.js. However, we expect that most of the discussions are held internally[31]. This does not mean that Next.js does not have software quality process in place. If we scan the Github repository for a bit (see figure), we already observe several hints that Next.js in fact does have some sort of quality process in place.

At a high level, Next.js works with a concept called *canary releases* (in addition to stable releases). Canary

[26] https://github.com/mui-org/material-ui/blob/master/CONTRIBUTING.md
[27] https://github.com/zeit/next.js/issues/8207
[28] https://github.com/zeit/next.js/issues/9310
[29] https://github.com/zeit/next.js/issues/9133
[30] https://github.com/zeit/next.js/issues/8207
[31] https://github.com/zeit/next.js/pull/10722

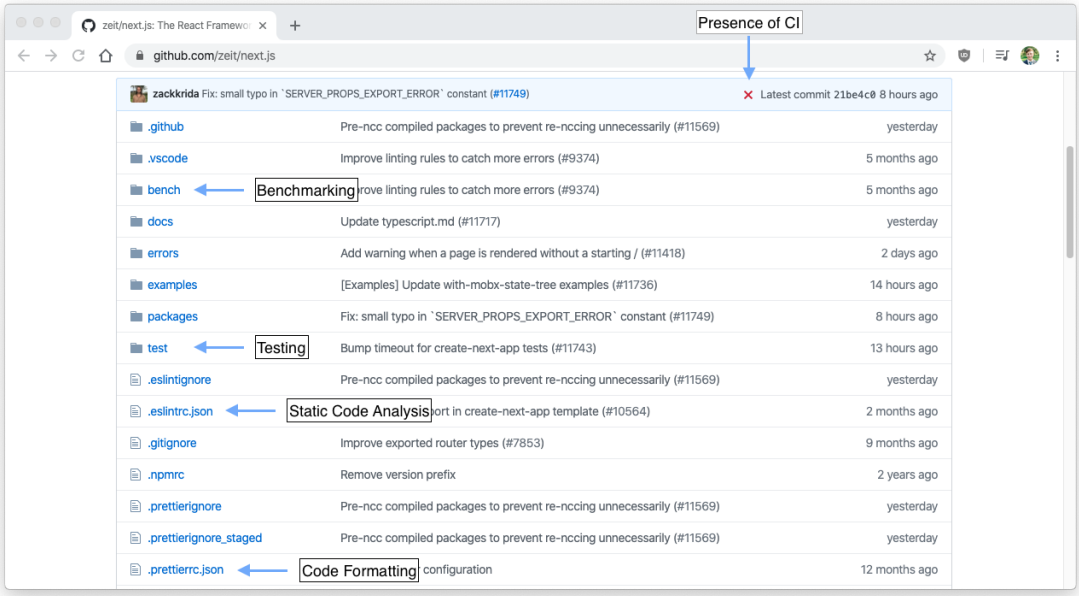


Figure 16.3: Next.js repository hinting the existence of a software quality process

releases always represent the latest changes of the project and give the users the opportunity to experiment with new (and experimental) functionality, while still trying to be backwards compatible and as stable as possible[32]. Consequently, this allows maintainers to test and gather feedback on changes before they are introduced in the next stable release.

In between these canary releases, Next.js has several processes in place to safeguard the quality of the project. Here, we distinguish between processes that need to be performed manually by the maintainers or external developers, and processes that are performed automatically with each commit, pull request or release.

16.3.1.1 Manual Processes

Development of Next.js is organized using a GitHub workflow[33], where contributors may propose changes to the repository by means of a pull request on top of the `canary` branch. Before the changes may be merged into the repository, a maintainer must first approve the changes. This mostly holds for maintainers as well, but this does not seem to be a strict requirement[34].

Whether the maintainers do a full code review before approval is not clear. There are some instances where a maintainer requests changes to the pull request[35], but these pull requests represent less than 5% of the total[36].

[32] Tim Neutkens and Arunoda Susiripala, *Towards Next.js 5: Introducing Canary Updates*. Blog, 2017
[33] Github, *Understanding the GitHub flow*. GitHub Guides, 2017
[34] https://github.com/zeit/next.js/pull/11699
[35] https://github.com/zeit/next.js/pull/10984
[36] https://github.com/zeit/next.js/pulls?page=2&q=is%3Apr+review%3Achanges-requested+is%3Aclosed

Moreover, the contributing guide of Next.js does not outline testing requirements for contributions. While Next.js does have an extensive test suite, it is mostly developed by the maintainers themselves with only a few cases of external contributors contributing tests[37]. This can be explained by the fact that external contributors mostly contribute to the Next.js examples, which do not require tests. Though, the maintainers are certainly appreciative of test cases in pull requests[38].

16.3.1.2 Automated Processes

In addition to the manual work, Next.js runs several automated processes to safeguard the quality of the project. GitHub Actions and Azure Pipelines are used to automatically manage and run the following processes for each pull request:

1. An extensive test harness with over 150 in-browser end-to-end test suites.
2. Static Code Analysis using ESLint to detect common mistakes and code smells.
3. Bundle Size check[39], to determine impact of the contribution on the size of the build artifacts.
4. In case of a release, automatically deploy to the appropriate distribution channels.

In addition to running these processes online in CI/CD environments, Next.js will, before a developer can commit a change, already run static code analysis and format the code according to Next.js' coding style using Prettier.

16.3.2 The Evidence

A strategy is not very useful if it is not reflected in the actual product. So, let's now dive into the codebase of Next.js and evaluate the effectiveness of their strategy. For this, we will attempt to quantify several desirable characteristics associated with software quality in the Next.js.

A starting point could be to measure the amount of tests in the codebase. Next.js has an extensive test harness, consisting of various types of tests. However, the majority of tests are actually end-to-end tests, with over 150 of such test suites. What is more important is how well the tests cover the codebase. While Next.js did at one point in time track test coverage, they removed it as the coverage was wrong due to internal files being bundled. Moreover, the maintainers argue that since the tests are mostly end-to-end and the confidence in their test harness is very high, test coverage is nice to have rather than being extremely useful for the project[40].

We could also utilize static code analysis tools to identify bugs or code smells. However, before we do that, let's first focus attention on the parts of the code that require the most attention, which are the parts with the most development activity. Usually, this code is complicated and susceptible to bugs. As seen in the above figure, for Next.js, most development happens in `next-server.ts` which contains the main server logic and `webpack-config.ts` which builds the configuration for the webpack module bundler.

If we now turn to static code analysis tools, SonarQube in this case, we find that the Next.js codebase is in good condition, as depicted in the figure above, passing all quality characteristics with an A, except for reliability, which scored a D due to a critical bug detected. In a similar fashion, code analysis by SIG shows a maintainability of 3.9 out of 5 stars, where Next.js mainly scores low on unit size and unit complexity.

[37] https://github.com/zeit/next.js/commits/canary/test
[38] https://github.com/zeit/next.js/pull/10592#pullrequestreview-388408075
[39] https://github.com/zeit/next.js/pull/10592#issuecomment-588221211
[40] https://github.com/zeit/next.js/discussions/11512

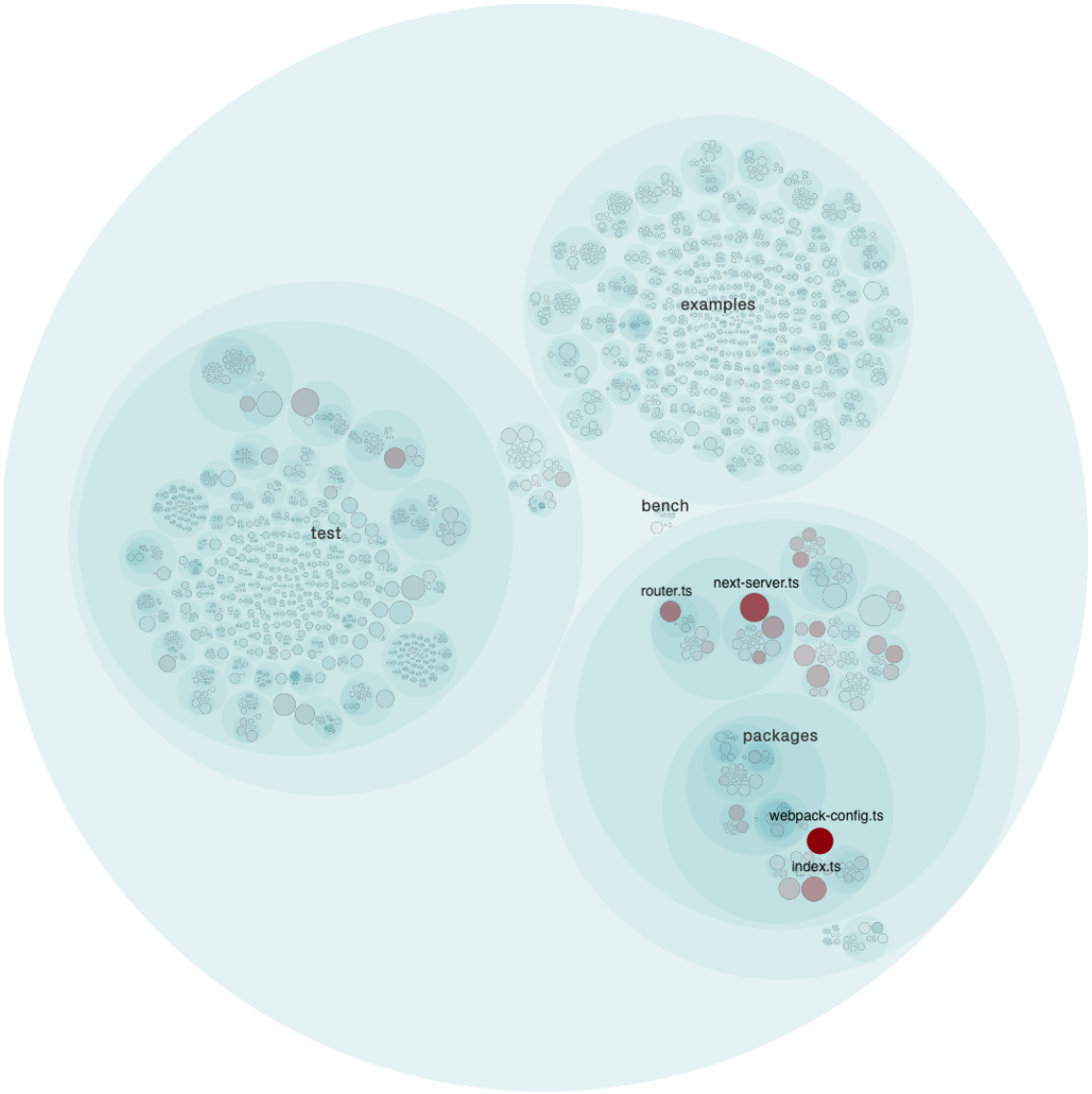


Figure 16.4: Hotspots in the codebase identified by CodeScene

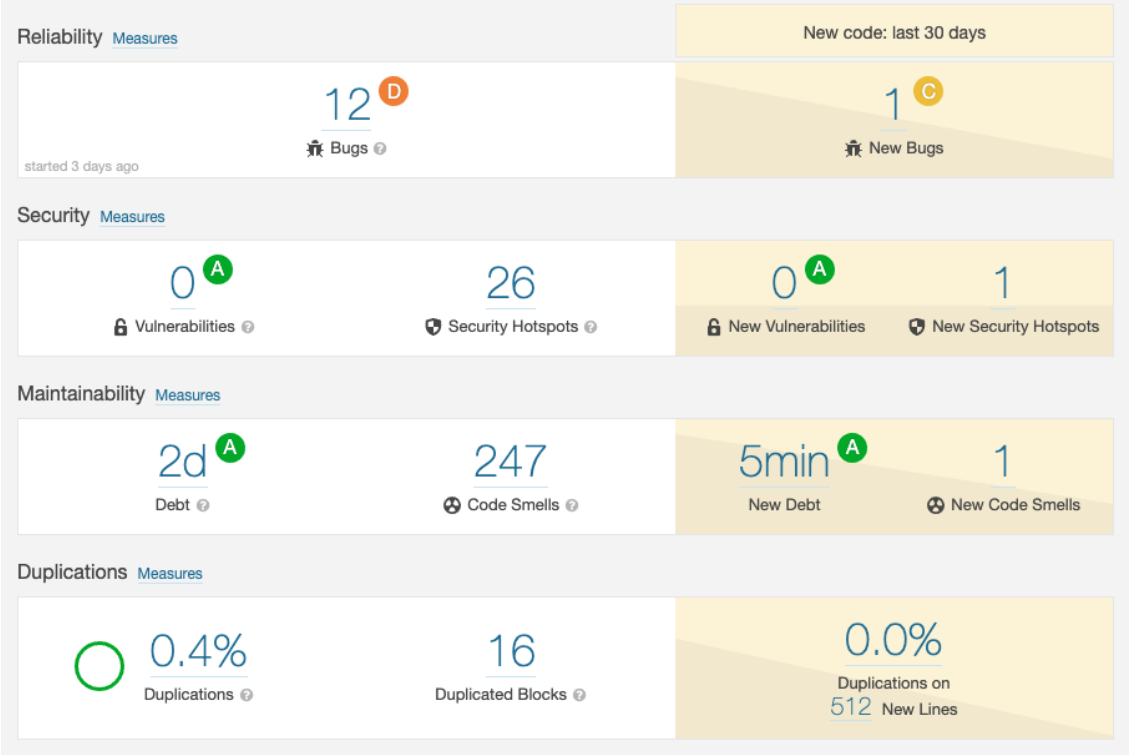


Figure 16.5: Technical Debt measured by SonarQube

While these initial results look promising, still, we observe the presence of bugs, security hotspots and code smells in the analysis. However, upon closer inspection of the results, we find that some of the issues are not applicable to Next.js or are false positives. For example, the security hotspots are caused by the use of regular expressions, which may pose a security risk on untrusted input. This is however not the case for Next.js. Nonetheless, the analysis still contains some valid issues which should be addressed by the maintainers:

1. Variables from an upper scope being shadowed[41]
2. Redundant code[42]

16.3.3 The Prospects

Quality control is not only about safeguarding what we already have, but also about anticipating what is to come. Like we said before, software is alive and constantly evolving. It is important that we take into account the changes of tomorrow in the decisions we make today. In our first post on the vision of Next.js, we already wrote about its private roadmap and what high-level functionality we expected to be on this roadmap:

1. Improved Static Site Generation (SSG)[43]
2. Support for React Concurrent mode and streaming rendering[44]
3. Support for partial hydration[45]

As of writing this post, Next.js has actually released version 9.3 featuring next generation Static Site Generation support[46]. While being fully backwards compatible, this feature required significant additions to Next.js API surface and changes to Next.js internals. From the initial commit[47], we see that the changes are mostly concentrated in the build module (for building Next.js applications) and server module, with a small change needed in the client module.

More of a challenge is adding support for React Concurrent mode and streaming rendering in Next.js. Traditionally, rendering the page on the server required a blocking call using React (in a single-threaded Node.js environment) and consequently many developers kept using global state in their components. However, with the introduction of React Concurrent mode, React is now able to render multiple pages concurrently in a single thread. While this has improved overall throughput tremendously, it presents a challenge for those using global state in their components. Next.js also suffers from this problem, most importantly in the next/head module, for which solutions are now being discussed[48]. We expect mostly the server and client module to be affected by this feature.

Adding support for partial hydration would be a great addition to Next.js, as also acknowledged by the maintainers[49]. However, it is difficult to estimate the impact of this functionality on the codebase as the maintainers have yet to decide how they wish to shape this functionality. Nonetheless, we expect this feature to require a new API interface for the user to specify what part of a page should be rendered on the server

[41] https://sonarcloud.io/project/issues?id=fabianishere_next.js&issues=AXFQ1QI1kkKt-l-TInlI&open=AXFQ1QI1kkKt-l-TInlI
[42] https://sonarcloud.io/project/issues?id=fabianishere_next.js&issues=AXFQ1QGUkkKt-l-TInj-&open=AXFQ1QGUkkKt-l-TInj-
[43] https://github.com/zeit/next.js/issues/9524
[44] https://github.com/zeit/next.js/discussions/10741#discussioncomment-745
[45] https://github.com/zeit/next.js/issues/10344
[46] Tim Neutkens et al., *Next.js 9.3*. Blog, 2020
[47] https://github.com/zeit/next.js/commit/c24daa21722fadfce2d1d561ce65ecc0efa6a7ea
[48] https://github.com/zeit/next.js/issues/8981
[49] https://github.com/zeit/next.js/issues/10344

and what on the client, in addition to some significant changes in the server to able to recognize this and partially defer rendering to the client.

16.3.4 The Verdict

In this post, we have seen the various processes in use by Next.js to maintain the quality of the project and support its architectural integrity. We have assessed the effectiveness of Next.js' strategy and have analyzed where the maintainers' approach seem to be lacking and where it is functioning well. Finally, we have tried to anticipate future changes and how these changes may affect the architectural components of Next.js. All together, these provide very valuable insights of the architecture of Next.js

Overall, we find that the Next.js project is well-maintained and of good quality. Nonetheless, there are still some areas where Next.js could improve. For one, we would like to see the maintainers to be more open about its code quality standards to external contributors, by for example having a more extensive contributing guide. Moreover, while we understand the technical difficulties, we would like to see test coverage being utilized as to prevent new changes from being left untested unintentionally.

Chapter 17

NumPy



NumPy, short for Numerical Python, is the foundation for many widely used Python libraries. Many libraries in the machine learning field depend on it: Tensorflow, PyTorch, scikit-learn etc. The main featureset of NumPy is that it efficiently implements numerical data structures such as matrices and arrays in Python, as the built-in language datastructures don't offer this featureset. Along with these data structures NumPy also implements mathematical operations that can be applied to these datastructures such as matrix inverses or matrix decomposition operations.

NumPy was created by Travis Oliphant in 2006, he merged together `Numeric` with `Numarray` to create the initial 1.0 version of `NumPy`. It was initially part of the larger `SciPy` project, however to avoid installing the entire large SciPy package it was split off from SciPy into its own NumPy package.

17.1 About the team

The team consists of four members:

- Robbert Koning
- Pravesh Moelchand
- Erwin van Thiel
- Jim Verheijde

17.2 NumPy: its Goals, Stakeholders, Use and Future

NumPy, short for Numerical Python, is a Python library that provides functionality for scientific computing. The first versions of the library were initially part of SciPy under the name *Numeric*. As the library became more popular and required more flexibility and speed, *numarray* was created as a replacement by the Space Science Telescope Institute. *Numeric* and *numarray* were eventually split up, but in 2005, Travis Oliphant reunited them, separated everything from SciPy and named the new library NumPy. In 2006, the library was included in Python's standard library[1].

Even though NumPy is described as: "(...) the fundamental package for scientific computing with Python" on its website[2], its core strengths are multidimensional arrays and tools to work with these arrays. It has some functionality for scientific computing, but SciPy would be the go-to library for this.

The library is currently open-sourced on GitHub and is being maintained by a large community of developers. Below, you can read more about NumPy's users, capabilities, stakeholders, context and roadmap.

NOTE: The formatting in this essay has been optimised for the online version.

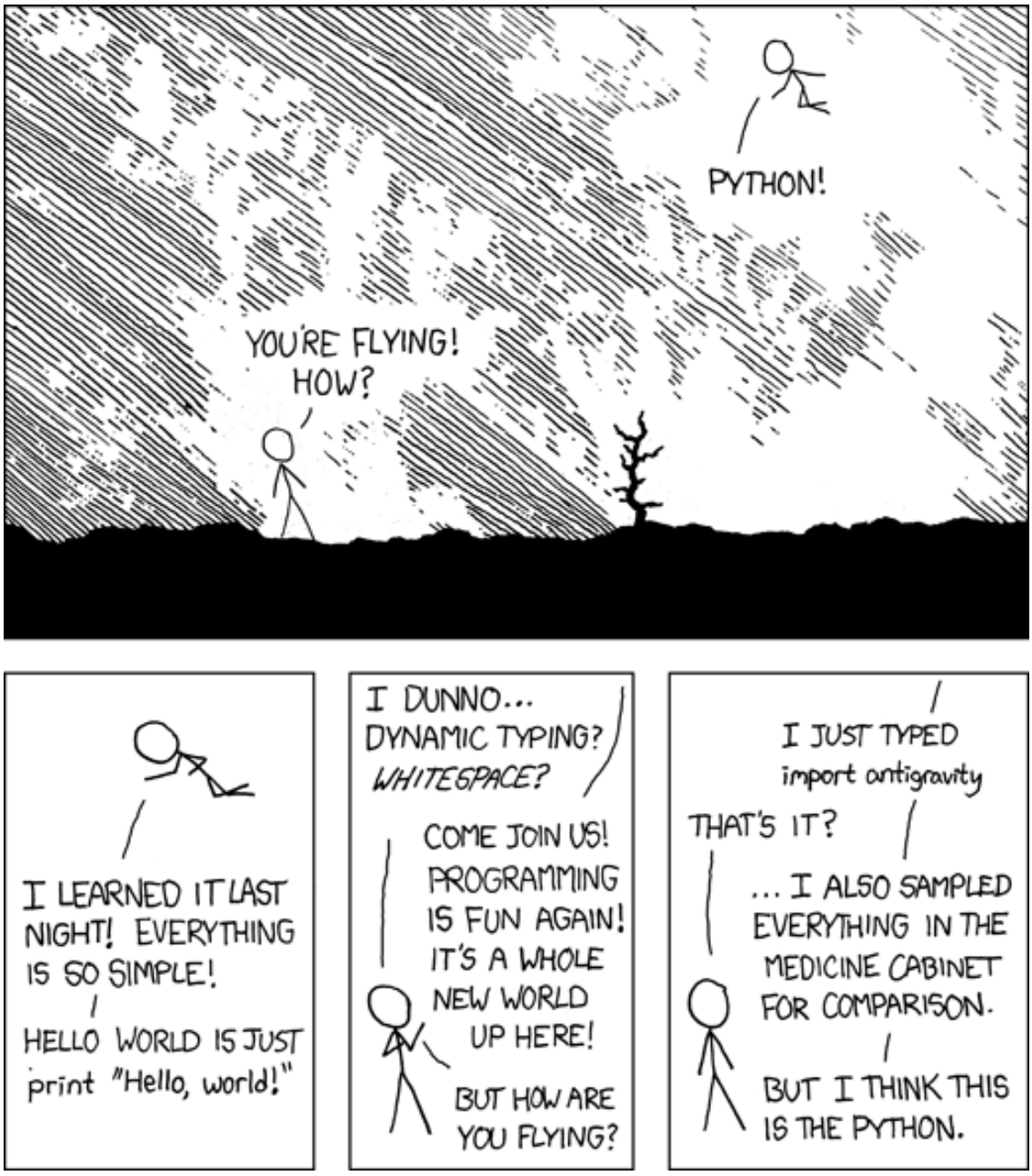
- Mental Model of the User
- Key Capabilities and Properties
- Stakeholder Analysis
- Context
- Roadmap

17.2.1 Mental Model of the User

The book Lean Architecture by Bjørnvig and Coplien states that the mental model of the user consists of the *roles* and *actors* that interact in the system[3]. In the same book, the authors split the view on architecture into two parts: what the system *is* versus what the system *does*. When looking at it from the latter perspective, *use cases* can effectively elicit these end-user world models. This section will list the major use cases of NumPy to capture the mental model of the user.

The main use case for NumPy is offering a way to initialize and apply computations on N-dimensional array structures in Python. Python is a beginner-friendly language and libraries are expected to 'just work' after importing them. This is another assumption of the end user when using Python libraries in general; see a relevant xkcd below:

[1] History of SciPy, retrieved on 2020-03-02. https://scipy.github.io/old-wiki/pages/History_of_SciPy.html
[2] NumPy homepage, retrieved on 2020-02-20. https://numpy.org/
[3] Coplien, J. O., & Bjørnvig, G. (2011). Lean architecture: for agile software development. John Wiley & Sons.



A second use case of NumPy is the speed of the library. Large datasets mean large matrices on which computations need to be done. Therefore NumPy must be able to perform complex mathematical computations quickly on large data structures. For users, NumPy can be accessed entirely by using Python code.

The end-user model thus consists of two main expectations: the user must be able to simply import the NumPy library and start using it immediately and intuitively; additionally the library is expected to be fast, even on large data structures.

17.2.2 Key Capabilities and Properties

The capabilities of NumPy all serve the main goal of the framework, namely being the fundamental tool for scientific computing within Python.

According to its own website, NumPy aims to provide[4]:

> "A powerful N-dimensional array object called a NumPy array, a selection of derived objects and functionality for a range of fast operations on these data structures. These operations include mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations."

This NumPy array has a fixed length and contains only one datatype, unlike a regular Python list. NumPy also contains tools for simulation of random numbers. In short, NumPy allows for fast computations when using array-like data structures.

As mentioned before, using NumPy allows for programming more efficiently in terms of faster computations and smarter memory management. This performance is partially due to its implementation being C code, which is faster than Python[5].

17.2.3 Stakeholder Analysis

The stakeholders of NumPy can be divided into several categories: **developers**, **main sponsor**, **institutional partners**, **funders**, and **end users**. These categories will be further described below.

17.2.3.1 Developers

The NumPy repository is part of the NumPy organisation on GitHub. This group consists of 23 members[6] and has administrative control over the NumPy repository. As of early March 2020, there are 877 contributors to the NumPy repository[7]. The most active maintainers in the time range January 2019 - March 2020 can be found below. Probably not by coincidence, they also took part in one of our pull requests.

List of most active maintainers according to contributors in the time range January 2019 - March 2020 and their commits[8]:

- Matti Picus
- Eric Wieser
- Sebastion Berg
- Ralf Gommers
- Charles Harris
- Warren Weckesser

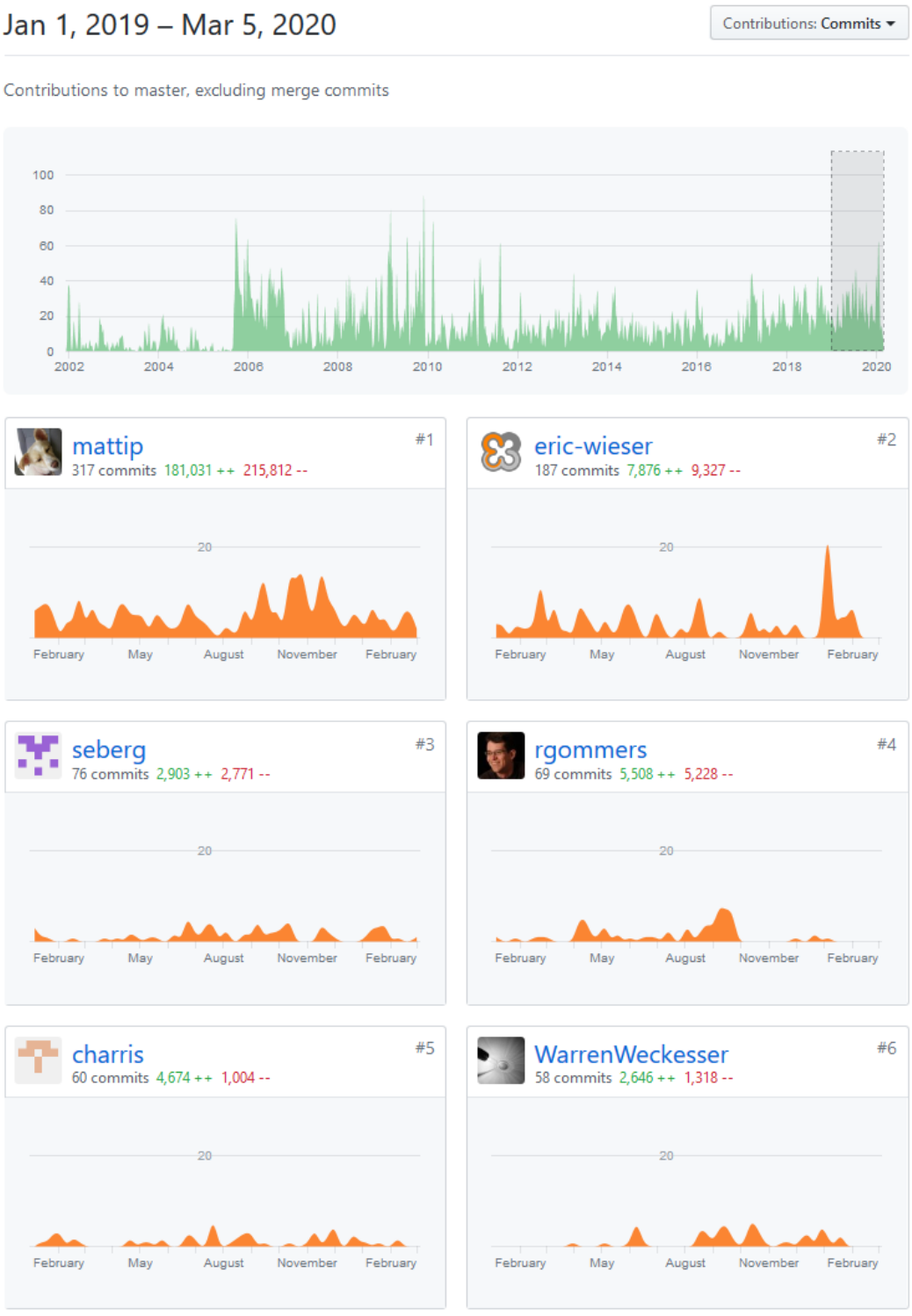
[4] NumPy homepage, retrieved on 2020-02-20. https://numpy.org/

[5] Rossant, C. (2018). Chapter 4.5 Understanding the internals of NumPy to avoid unnecessary array copying. In IPython Cookbook (Second Edition). Retrieved from https://ipython-books.github.io/45-understanding-the-internals-of-numpy-to-avoid-unnecessary-array-copying/

[6] NumPy Organization members on GitHub, retrieved on 2020-03-05. https://github.com/orgs/numpy/people

[7] Contributors to NumPy, retrieved on 2020-03-05. https://github.com/numpy/numpy/graphs/contributors

[8] Contributors to NumPy from January 2019 to March 2020, retrieved on 2020-03-05. https://github.com/numpy/numpy/graphs/contributors?from=2019-01-01&to=2020-03-05&type=c


Jan 1, 2019 – Mar 5, 2020
Contributions: Commits
Contributions to master, excluding merge commits
100
80
60
40
20
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
mattip
#1
317 commits 181,031 ++ 215,812 --
eric-wieser
#2
187 commits 7,876 ++ 9,327 --
seberg
#3
76 commits 2,903 ++ 2,771 --
rgommers
#4
69 commits 5,508 ++ 5,228 --
charris
#5
60 commits 4,674 ++ 1,004 --
WarrenWeckesser
#6
58 commits 2,646 ++ 1,318 --
20
February
May
August
November
February

17.2.3.2 Main sponsor

NumPy's main sponsors is NumFOCUS, a 501(c)(3) nonprofit charity in the United States. This status comes with quite some limitations and regulations, ensuring that the organisation is focused on charitable purposes[9], in this case: "Open Code for Better Science"[10]. NumFOCUS provides NumPy with fiscal, legal, and administrative support to help ensure the health and sustainability of the project.

17.2.3.3 Institutional partners

NumPy's institutional partners are organisations that support the project by employing NumPy contributors, with contributing to the project as part of their official duties. Current institutional partners include Quansight (headed by Travis Oliphant) and Berkeley University of California.

17.2.3.4 Funders

NumPy receives direct funding from the following sources: Gordon and Betty Moore Foundation, Alfred P. Sloan Foundation and Tidelift. We were interested to what extent these funders were involved in the decision-making process of NumPy. The Moore Foundation and Alfred P. Sloan Foundation are both not involved in any decision-making process of their fundees[11] [12]. It is unclear to what extent Tidelift is involved in NumPy, as there is no statement whatsoever and they seem to be more commercially-oriented. We contacted Tidelift about their involvement in NumPy, but unfortunately we only received an automated reaction.

17.2.3.5 End users

The end users of NumPy are also stakeholders in the sense that they, too, benefit from NumPy functioning and being managed properly. Some people use NumPy's computational power in their personal projects, but there are also widely-used frameworks that base their functionality on NumPy. These include Tensorflow, PyTorch and sci-kit learn, among many others. On GitHub alone NumPy is already used by over 300k projects[13].

17.2.4 Context

NumPy is, and in the future will be, used in the context of a Python source code file. As described earlier, the NumPy library provides functionality for working with arrays in Python. There is no GUI and this will probably never be created, as it does not match the environment in which NumPy is used. Below, an example of some NumPy code can be found[14]:

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
```

[9] What is a 501(c)(3), retrieved 2020-03-05. https://www.501c3.org/what-is-a-501c3/
[10] NumFOCUS website. retrieved 2020-03-05. https://numfocus.org/
[11] Moore Foundation, Founder's Intent, retrieved on 2020-03-02. https://www.moore.org/about/founders-intent
[12] Alfred P. Sloan Foundation, Governance and Policies, retrieved 2020-03-05. https://sloan.org/about/documents#tab-governance-and-policies
[13] GitHub repositories depending on NumPy, retrieved on 2020-03-02. https://github.com/numpy/numpy/network/dependents
[14] Numpy code example, retrieved on 2020-03-05. https://docs.scipy.org/doc/numpy/user/quickstart.html#an-example

```
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

17.2.5 Roadmap

NumPy hosts an elaborate roadmap on its website, as well as a template for proposing new plans. These plans are called NumPy Enhancement Proposals (NEPs) and a list of previous and current NEPs can be found on the following webpage.

A solid example of such a NEP is *NEP 14 - Plan for dropping Python 2.7 support*[15]. The final stage of this NEP was done on January 1st, 2020, when NumPy's community dropped support for Python 2 entirely. The NEP for dropping Python 2.7 was already suggested in November 2017[16].

However, NEPs do not cover NumPy's roadmap entirely and hence, we will summarise NumPy's roadmap below[17]:

- Interoperability
  - The NumPy developers want to make it easier for other libraries to interoperate with NumPy. They want to provide better interoperability protocols and better array subclass handling, among other things.
- Extensibility
  - The NumPy developers want to simplify NumPy's internal datatypes ('dtypes') to simplify extending Numpy's functionality. For instance, they want to simplify the creation of custom dtypes and want to add new 'string' dtypes for dealing with textual data.
- Performance
  - The NumPy developers want to improve NumPy's performance through for instance SIMD instructions and optimisations within functions.
- Website and documentation

[15] NumPy's NEP 14, retrieved on 2020-03-02. https://numpy.org/neps/nep-0014-dropping-python2.7-proposal.html

[16] Initial discussion of dropping support for Python 2.7, retrieved on 2020-03-02. https://mail.python.org/pipermail/numpy-discussion/2017-November/077419.html

[17] NumPy's roadmap, retrieved on 2020-03-02. https://numpy.org/neps/roadmap.html

  - The website needs to be rewritten completely and the documentation is of 'varying quality'[18].
- Random number generation policy & rewrite
  - At the time of writing, the developers are close to completing a new random number generation framework.

17.2.6 Conclusion

Many developers use NumPy and are familiar with the library itself. However, they most likely haven't looked at NumPy from a more historical and contextual perspective, as this is not strictly needed to use the library. In this essay we hope to have given NumPy users or anyone else interested in the NumPy project insight into these overlooked aspects of one of the most iconic Python packages to date.

17.3 NumPy: Awesome Architecting for Amazing Arrays

This is the second installment of our 4-essay-long series about the NumPy project. For the first essay about the stakeholders and project in general, please visit this page. In this second essay, we will take look at NumPy from different architectural perspectives which are based on literature. These different views aim to give the reader insight into how NumPy implements it's key properties.

NOTE: The formatting in this essay has been optimised for the online version.

- Architectural Views
- Development View
- Runtime View
- Deployment View
- Non-functional properties

17.3.1 Architectural Views

17.3.1.1 Kruchten

In his 1995 IEEE paper[19], Kruchten describes four architectural views on software architecture:

- **The logical view**, which describes the design's object model when an object oriented design method is used. To design an application that is very data-driven, you can use an alternative approach to develop some other form of logical view, such as an entity relationship diagram.
- **The process view**, which describes the design's concurrency and synchronization aspects.
- **The physical view**, which describes the mapping of the software onto the hardware and reflects its distributed aspect.
- **The development view**, which describes the software's static organization in its development environment.

Since NumPy is a library for numerical computations and not a software program, a view that is concerned with dynamic system-components and communication between components does not seem very applicable and because of this, a process view is not relevant to the architecture of NumPy. For the same reason, a

[18] Section about website and documentation of NumPy's roadmap, retrieved on 2020-03-05. https://numpy.org/neps/roadmap.html#website-and-documentation

[19] P. B. Kruchten, "The 4+1 View Model of architecture," in IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995. DOI: 10.1109/52.469759

physical view is irrelevant: you cannot discuss the system in terms of a physical topology, since it is only a library consisting of code modules.

Considering a development view makes more sense for this project, as the development view focuses on modules in the development environment. NumPy contains multiple different modules in different layers. Besides this, reasoning about reuse and portability is inherent to a library like NumPy. The development view on NumPy is discussed here.

Furthermore, a logical architectural view could also be of good use. On one hand, a logical view is concerned with the system's functional requirements, which are very clear for a system like NumPy. On the other hand, a logical view describes a system in terms of objects and their relations and since NumPy's core functionality is written in C language, which is not an object oriented- but a procedural language, defining objects does not really make sense. As the functional requirements have already been discussed in our previous essay, we will not discuss the logical view here.

17.3.1.2 Rozanski and Woods

Rozanski and Woods present some other views in their *Software Systems Architecture* book[20]. We will consider the views that don't overlap with Kruchten's views:

- **The deployment view**, which describes the environment in which the system will be deployed.
- **Operational view**, which describes how the system will be operated once it is deployed.

The deployment view has some relevance as the library is used with Python, which is available on many different platforms. Therefore, it will need to be able to run on these different platforms without causing the users trouble. The deployment view on NumPy is discussed here.

The operational view is less appropriate, as NumPy is not some kind of service that needs to be monitored or administered. It is simply a library that provides data structures (arrays) and tools to work with them, therefore, we will not look at the operational view.

17.3.1.3 arc42 Documentation

The arc42 documentation provides similar views to the views mentioned above. Included is the runtime view[21], which is also relevant for NumPy. This view is relevant because, contrary to popular belief, the performance of NumPy varies a lot depending on the runtime dependencies used. The runtime view on NumPy is discussed here.

17.3.2 Development View

As described by Kruchten, the development view focuses on the organization of the actual software modules in the software-development environment[22]. This view is is also relevant to the architectural style of NumPy, which is component-based. Therefore, we will also discuss the architectural style of the library in this section.

[20] N. Rozanski and E. Woods. 2005. Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.

[21] arc42 Documentation Runtime view. https://docs.arc42.org/section-6/

[22] P. B. Kruchten, "The 4+1 View Model of architecture," in IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995. DOI: 10.1109/52.469759

17.3.2.1 Architectural Style

At the start of the project, we analysed the NumPy repository and created an overview of the different components. This overview was then sent to SIG and the codebase was further analysed by them. We noticed that the architecture of the NumPy library is fairly simple, it is component-based and there are not a lot of dependencies between the different components. In the section below, you can find an overview of the components as created by SIG.

17.3.2.2 System decomposition

The following diagram was created by SIG using our analysis of the component structure:



Figure 17.1: Component structure, provided by SIG. ('-new' is an artifact from the analysis)

As can be seen from the diagram, the `core-src` component is the largest. This module contains most of the functionality in NumPy and has been written in C for speed. The `core` module contains Python wrappers that allow the library to be used as a Python library. This structure can be seen in most of the modules: source code in C in combination with Python wrappers.

The `f2py` component provides functionality for converting Fortran code to Python code. Fortran is an older languages for scientific computing and as NumPy deems itself "the fundamental package for scientific computing with Python"[23], it is not strange that they provide compatibility with Fortran.

The other components provide tools for specific mathematical applications, such as `random` for generating random numbers, `linalg` for linear algebra and `fft` for (fast) Fourier transforms.

The last noteworthy module is the `distutils` module, which is used for overhead and making NumPy compatible. It provides support for different compilers, more about these can be read here

17.3.2.3 Relationships

Using the component analysis, SIG also created an overview of the relationships between the components in NumPy, which can be found below:

[23] NumPy homepage, retrieved on 2020-03-19. https://numpy.org/

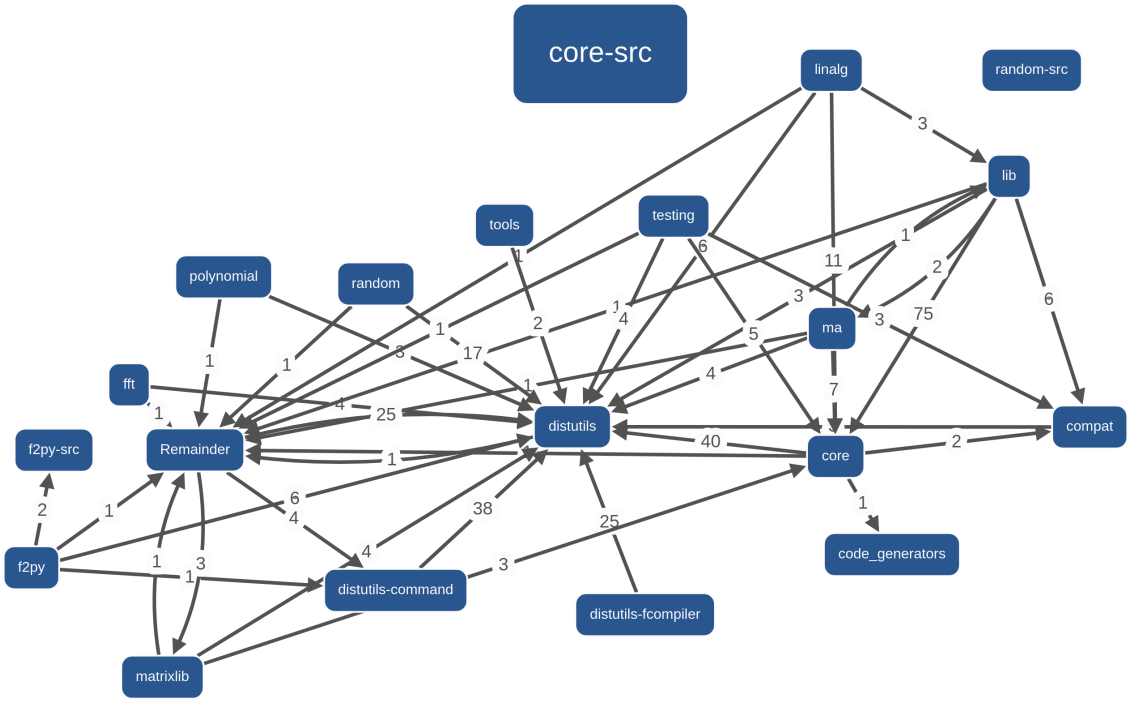


Figure 17.2: Component relationships of NumPy, provided by SIG

As can be seen, the `core-src` and `random-src` are decoupled from the rest of the system. The other components are slightly coupled and the `distutils` component has an important place in the structure. This is most likely due to the fact that it manages overhead and compilation.

The component-based structure can clearly be seen in the diagrams presented above. This structure allows contributors to contribute more easily to the system, as functionality is split into different modules.

17.3.3 Runtime View

As was mentioned in the section about architectural views, NumPy can have a large range of performance, even on the same system and OS! The reason behind this discrepancy in performance is in the underlying BLAS/LAPACK implementations used by NumPy. BLAS (**B**asic **L**inear **A**lgebra **S**ubprograms) is a specification for vector and matrix operations such as multiplications, dot products among others[24]. The reference implementation was done by Netlib in Fortran. LAPACK (**L**inear **A**lgebra **Pack**age) is both a specification and reference implementation of routines for least squares solutions, eigenvalue problems, matrix factorization, singular value decomposition among others[25]. LAPACK tries to move most of the computational effort to BLAS routines in order to gain performance. There exist other implementations for BLAS and LAPACK that are tuned for specific systems/processors[26]. An example is the Intel MKL library, as you might have guessed, tuned specifically for Intel processors.

NumPy can use different implementations of BLAS/LAPACK. For BLAS the order of preference is[27]:

Preference rank (1=highest preference)	BLAS	LAPACK
1	MKL	MKL
2	BLIS	OpenBLAS
3	OpenBLAS	libFLAME
4	ATLAS	ATLAS
5	Accelerate (MacOS)	Accelerate (MacOS)
6	BLAS (Netlib)	LAPACK (NetLIB)

Benchmarks that compare these different libraries show that there is a large difference in performance. Below are the results of a benchmark ran by Markus Beuckelmann for different implementations of BLAS/LAPACK on an Intel i5 processor[28].

	Default BLAS & LAPACK	ATLAS	OpenBLAS	Intel MKL
Dot product of two 4096x4096 matrices	64.22 s	3.46 s	3.97 s	**2.44 s**
Dot product of two 524288 vectors	0.80 ms	**0.73 ms**	0.74 ms	0.75 ms
SVD of a 2048x2048 matrix	10.31 s	2.02 s	1.96 s	**1.34 s**
Cholesky decomposition of a 2048x2048 matrix	6.74 s	0.51 s	0.46 s	**0.40 s**
Eigendecomposition of a 2048x2048 matrix	53.77 s	29.90 s	32.95 s	**10.07 s**

[24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Softw. 5, 3 (September 1979), 308–323. DOI:https://doi.org/10.1145/355841.355847

[25] LAPACK website. http://www.netlib.org/lapack/

[26] LAPACK vendor implementations. http://www.netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations

[27] NumPy build instructions. https://numpy.org/devdocs/user/building.html#accelerated-blas-lapack-libraries

[28] Boosting NumPy: Why BLAS matters. https://markus-beuckelmann.de/blog/boosting-numpy-blas.html

As can be seen the results for this benchmark vary wildly, with speedups of up to 25x. This shows that there is a significant difference in use of different runtime dependencies. Now one might think that this difference is only visible if you actively change the order of preference, but this is not the case. The Anaconda distribution for example, installs the Intel MKL library by default, while installing NumPy via pip uses OpenBLAS. Therefore this subtle difference in runtime dependencies can have major implications on the runtime of your NumPy program.

17.3.4 Deployment View

The deployment view describes the environment into which the system will be deployed and the dependencies of the deployment process. From the GitHub repository multiple files related to CI (Continuous Integration) are present. More on this will be discussed in our next blog post.

As for the CD (Continuous Deployment) part of the pipeline there doesn't seem to be a central location that automatically pushes the new version of NumPy to PyPI (Python package index). From PyPI users can download the package using pip. These releases seem to be deployed manually (on PyPI), the same goes for the releases on GitHub[29]. The versions are tagged but they are posted manually by maintainers.

17.3.5 Non-functional properties

Finally this last section will not so much be about another architectural view (we already had quite a lot of those), instead we will discuss the the non-functional properties of NumPy. Non-functional properties are the properties regarding the operation of a system rather than the functionality of it[30]. These are also important to highlight as they heavily influence the usability and effectiveness of the system. In the case of NumPy we focus on three non-functional properties: performance, test coverage and compatibility.

17.3.5.1 Performance

Performance is undoubtedly one of the most important non-functional requirements for NumPy as it is a library focused on the efficient representation of n-dimensional data-structures and mathematical operations on those. NumPy satisfies this requirement by using C code and using efficient implementations of the BLAS/LAPACK protocols mentioned previously in the runtime view.

17.3.5.2 Test coverage

The SIG analysis points out that the test code ratio of NumPy is `55.1%`, meaning 55.1% of the total code is consisting of test code. This is a good example of how important testing and test coverage is for the NumPy library. As described earlier, multiple CI pipelines are run to check the code and its new additions.

17.3.5.3 Compatibility

One of NumPy's non-functional requirements is that it should be compatible with many software projects. Running with as few dependencies as possible helps in achieving this requirement and it is realised as NumPy only requires Python. It also offers compatibility with Fortran as described earlier. Another aspect is that it should be relatively intuitive to use. While this is a subjective manner, we think that NumPy is indeed

[29] Releases on GitHub. https://github.com/numpy/numpy/releases

[30] L. Chen, M. Ali Babar and B. Nuseibeh, "Characterizing Architecturally Significant Requirements," in IEEE Software, vol. 30, no. 2, pp. 38-45, March-April 2013. 10.1109/MS.2012.174

quite intuitive: if you can work with Python, you'll be able to work with NumPy. NumPy is, however, not compatible with Python 2.7 anymore, as elaborated upon in essay 1. Reason for this is that the team: "has found that supporting Python 2 is an increasing burden on our limited resource."[31]

17.3.6 Conclusion

In this essay we analysed NumPy from different architectural points of view. This approach can give new insights one would not gain from just using NumPy or any piece of software in general. For example, most NumPy users will never look deep into the runtime dependencies of NumPy. However, when explicitly viewing it from a runtime perspective, one discovers that NumPy is more intricate than just a combination of Python and C and can actually make use of high speed linear algebra libraries to satisfy non-functional properties such as performance. This is of course only one of the views. All the mentioned architectural views conglomerated, assert that the functional and non-functional key capabilities of NumPy are implemented.

17.4 NumPy: Software Quality by the Numbers

Software quality plays an important role in an open-source library that is being used in software projects worldwide. It helps in keeping the code maintainable and the releases stable. In this third part of our essay series about the NumPy project we will have a look at the software quality of the NumPy source code and how this quality is assured by the developers of the project.

We start out by looking at the software quality assurance process in general and the steps that are taken in this process. Then, we have a look at the testing process that is implemented in NumPy. After that, we compare the coding activity and roadmap to the the actual architectural components. Finally, we have a look at the improvements which are proposed by the Software Improvement group (SIG) and if there is any technical debt.

NOTE: The formatting in this essay has been optimised for the online version.

- Software Quality Assurance
- Continuous Integration
- Testing
- Coding Activity in Architectural Components
- Evaluation of Code Quality
- Technical Debt

17.4.1 Software Quality Assurance

As described in our first essay the NumPy project relies on a large group of developers from around the world for its development. When working with almost 900 contributors[32] from around the world on a project, it is import to have some sort of software quality assurance process in place. The development process of contributing to NumPy is described as follows on their website[33]:

- Select an issue you want to fix and fork the main repository
- Develop the contribution locally

[31] NumPy proposal to drop Python 2.7. https://mail.python.org/pipermail/numpy-discussion/2017-November/077419.html
[32] NumPy GitHub, retrieved on 2020-03-26. https://github.com/numpy/numpy
[33] Contributing to NumPy, retrieved on 2020-03-26. https://numpy.org/devdocs/dev/index.html

- Push the changes and create a pull request to the main repository
- Now you pull request will be reviewed by one of the core developers
- If any changes are requested, these should be applied before the pull request is approved
- In addition, several continuous integration (CI) checks are performed, these are further described below
- If all changes are approved and the CI passes, the pull request will be merged to the NumPy master branch

All these steps together ensure that new contributions are checked and reviewed by the core developers before they are merged to the master. In addition, the CI helps with checking the code. By performing all these checks before merging to master, the quality of the source code of NumPy is maintained.

17.4.2 Continuous Integration

To test the source code, multiple CI services are used, such as Travis CI, Azure pipelines, CircleCI, LGTM and Codecov. These are configured to run all tests on different platforms: `Windows`, `Linux` and `MacOS` are used for CI[34]. Different versions of Python, ranging from Python 3.5 to 3.8 are used for running the entire test suite[35]. NumPy also uses Codecov as a service to keep track of test coverage and changes. LGTM is used as a code analysis tool that runs security analyses [36]. This is very useful for NumPy as there is a lot of C code (just over 50% of the entire codebase). C code is relatively prone to security vulnerabilities, therefore making this a very valuable tool.

The CI is run for every PR. On the PR page there is a clear overview of the results of these CI pipelines including checks:

There are also other tools used which are not directly visible in the PR checks. One of these tools is the dependabot[37]. This bot is configured to check the dependencies used by NumPy and bumps the versions if there is a new one available. This is important for security updates that otherwise could be missed by the maintainers. By using this automated system the maintainers can focus on the product itself without having to worry about checking the security vulnerabilties of all their dependencies every day. The dependabot can be seen in action below, the whole PR can be found here.

The CI is very extensive, as there are many jobs that run for just one PR. However, this does not come without its cons. The largest disadvantage is that the CI is flaky, our experience in submitting PRs to the NumPy repository was not ideal as jobs would fail and users cannot retrigger specific jobs as this would probably overload the CI pipelines. The second disadvantage was the long time it takes to run the CI. It takes around 10-15 minutes for the entire CI pipeline to complete, which discourages making small commits.

17.4.3 Testing

NumPy makes use of an extensive test suite that, according to its own website, aims to achieve that every module and package is unit tested thoroughly[38]. This means that every NumPy module has its own suite of unit test. The goal of these unit tests is specified as follows:

[34] Azure pipelines config. https://github.com/numpy/numpy/blob/master/azure-pipelines.yml
[35] Travis CI config. https://github.com/numpy/numpy/blob/master/.travis.yml
[36] LGTM GitHub page. https://github.com/marketplace/lgtm
[37] Dependabot config. https://github.com/numpy/numpy/blob/master/.dependabot/config.yml
[38] NumPy testing guidelines, retrieved on 2020-03-26. https://docs.scipy.org/doc/numpy/reference/testing.html

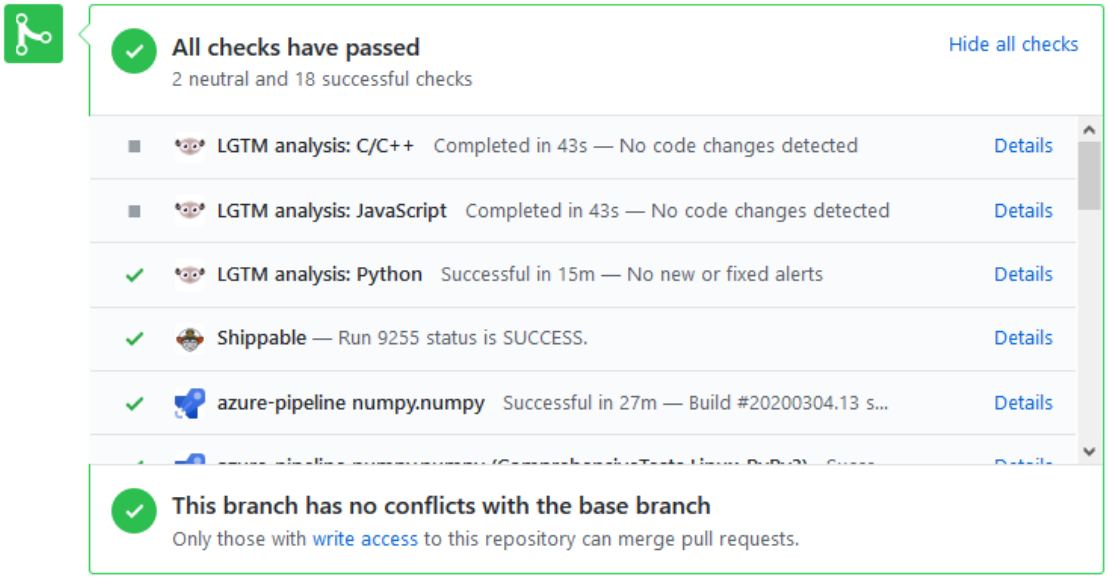


Figure 17.3: Pull request checks

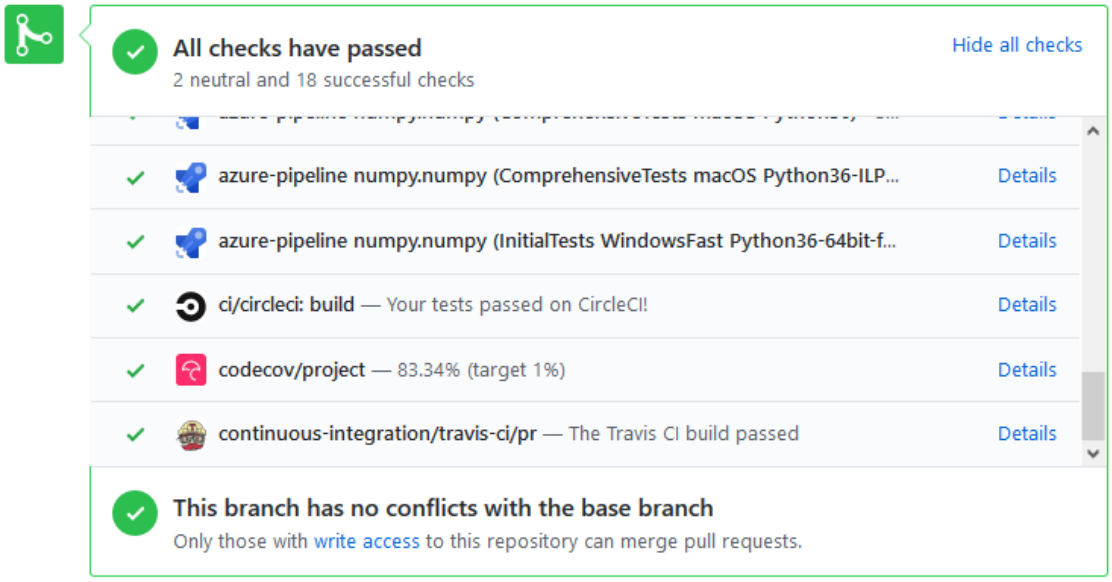


Figure 17.4: Pull request continued

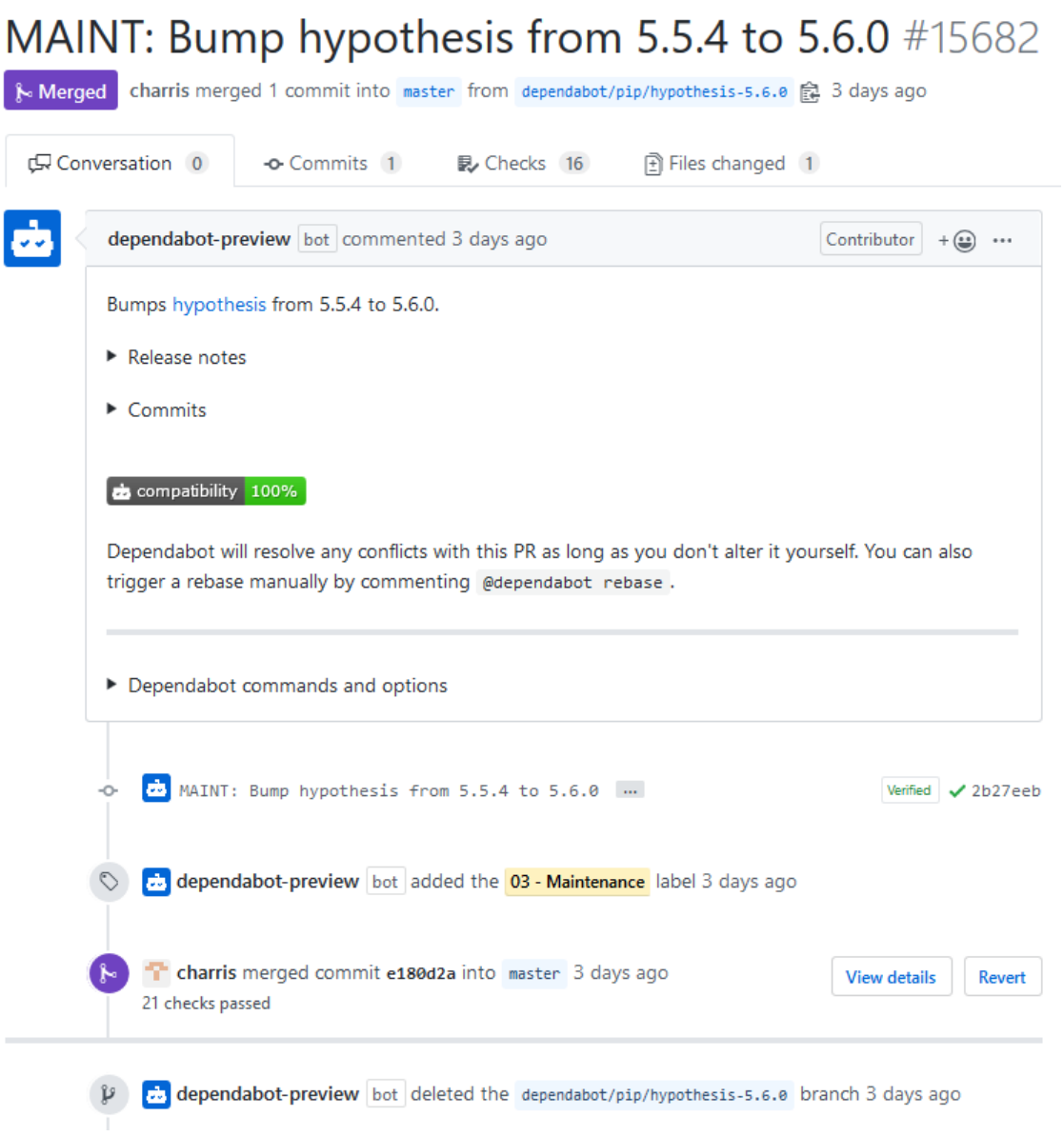


Figure 17.5: Changes made by the dependabot

> "These tests should exercise the full functionality of a given routine as well as its robustness to erroneous or unexpected input arguments."

To accomplish this level code testing, NumPy asks its contributors to write code in a test-driven-development fashion. To run the tests, the NumPy source code should be pulled and the necessary dependencies should be installed. Then, the command `python3 -v -t runtests.py --coverage` should be run. We ran the test suite on the most recent version of NumPy, which includes our contribution, and got the following output:

```
10248 passed, 520 skipped, 107 deselected, 15 xfailed, 3 xpassed in 145.62s (0:02:25)
```

Figure 17.6: NumPy test result

The output results are listed in a generated HTML file containing coverage information of all different modules. The coverage results of NumPy as a whole are displayed below.

Module ↑	*statements*	*missing*	*excluded*	*branches*	*partial*	*coverage*
Total	**90746**	**16254**	**0**	**21891**	**1395**	**78%**

Figure 17.7: Test results

NumPy does not perform any tests other than unit tests, which is understandable because NumPy is merely a library containing tools for computations on arrays and not a system with components that interact intensively.

17.4.4 Coding Activity in Architectural Components

17.4.4.1 Hotspots

Hotspots are a good indicator of main activity in your project. According to Tornhill, in his book about software, there is a strong correlation between hotspots, maintenance costs, and software defects[39]. Therefore, it is a good idea to focus on hotspots to discover potential problems in the code or even the architecture itself.

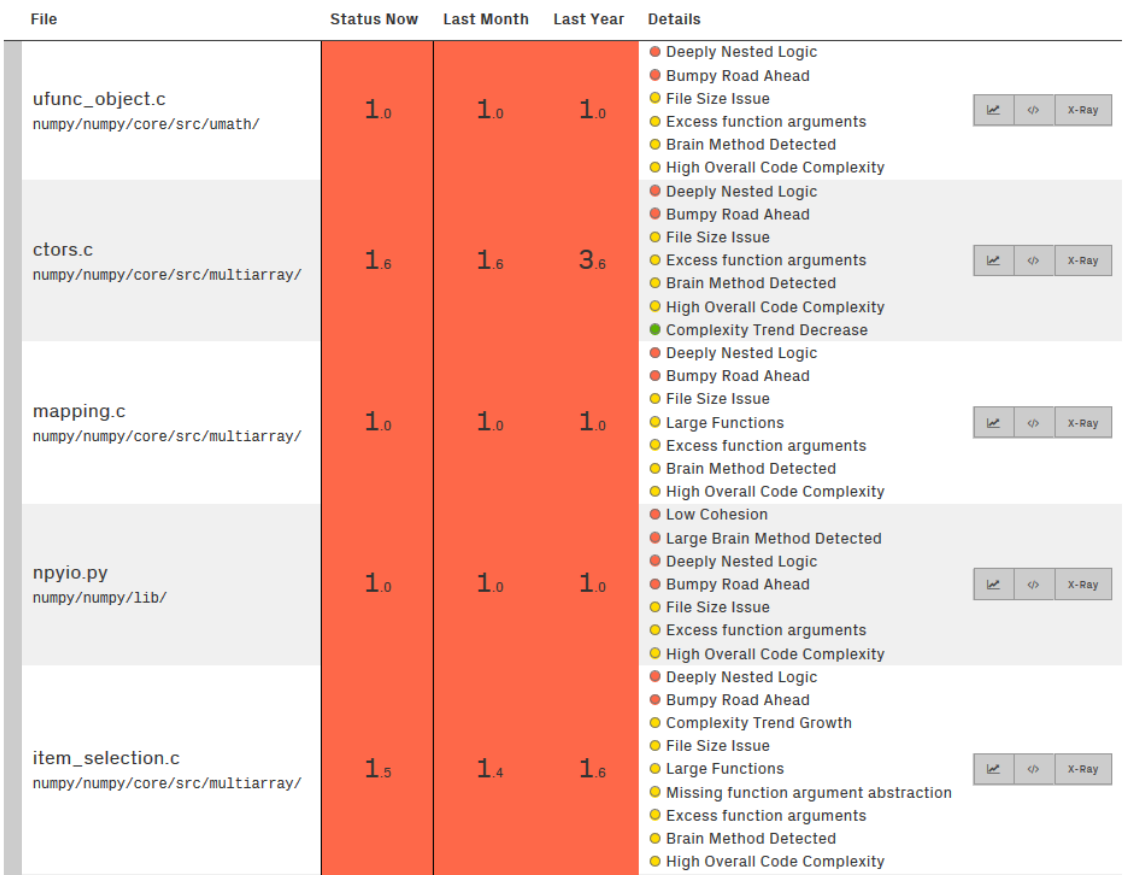
In order to detect development hotspots in the repository we used CodeScene. CodeScene is a company that analyses source code from git repositories to detect a wide range of metrics from several categories. For example: technical debt, hotspots, code health, coupling and complexity. Hotspots, which is what we are interested in specifically, are determined using the frequency of code changes in that file over a period of time[40]. The results of the analysis can be seen below. In the figure is a list with source files, starting with the file that is changed most frequently. The coloured status columns in the table denote The Code Health metric which ranges from 1 (code with severe quality issues) to 10 (healthy code that is relatively easy to understand and evolve)[41]. We will come back to this further on.

To get a better view of where these files reside in the project there is another figure below which show the files that are changed most frequently inside their folder structure. This shows that the files that are changed the most are in the `core`, `lib`, and `ma` modules. The `core` module is evidently the most active module, as the 4 out of 5 files in the top 5 files are in the `core` module.

[39] Tornhill, A. (2015). Your code as a crime scene: use forensic techniques to arrest defects, bottlenecks, and bad design in your programs. Dallas, TX: The Pragmatic Bookshelf.

[40] Hotspots documentation from CodeScene. https://codescene.io/docs/guides/technical/hotspots.html

[41] Code Health documentation from CodeScene, retrieved on 2020-03-26. https://codescene.io/docs/guides/technical/biomarkers.html



File	Status Now	Last Month	Last Year	Details
ufunc_object.c numpy/numpy/core/src/umath/	1.0	1.0	1.0	Deeply Nested Logic Bumpy Road Ahead File Size Issue Excess function arguments Brain Method Detected High Overall Code Complexity
ctors.c numpy/numpy/core/src/multiarray/	1.6	1.6	3.6	Deeply Nested Logic Bumpy Road Ahead File Size Issue Excess function arguments Brain Method Detected High Overall Code Complexity Complexity Trend Decrease
mapping.c numpy/numpy/core/src/multiarray/	1.0	1.0	1.0	Deeply Nested Logic Bumpy Road Ahead File Size Issue Large Functions Excess function arguments Brain Method Detected High Overall Code Complexity
npyio.py numpy/numpy/lib/	1.0	1.0	1.0	Low Cohesion Large Brain Method Detected Deeply Nested Logic Bumpy Road Ahead File Size Issue Excess function arguments High Overall Code Complexity
item_selection.c numpy/numpy/core/src/multiarray/	1.5	1.4	1.6	Deeply Nested Logic Bumpy Road Ahead Complexity Trend Growth File Size Issue Large Functions Missing function argument abstraction Excess function arguments Brain Method Detected High Overall Code Complexity

Figure 17.8: Hotspot file list

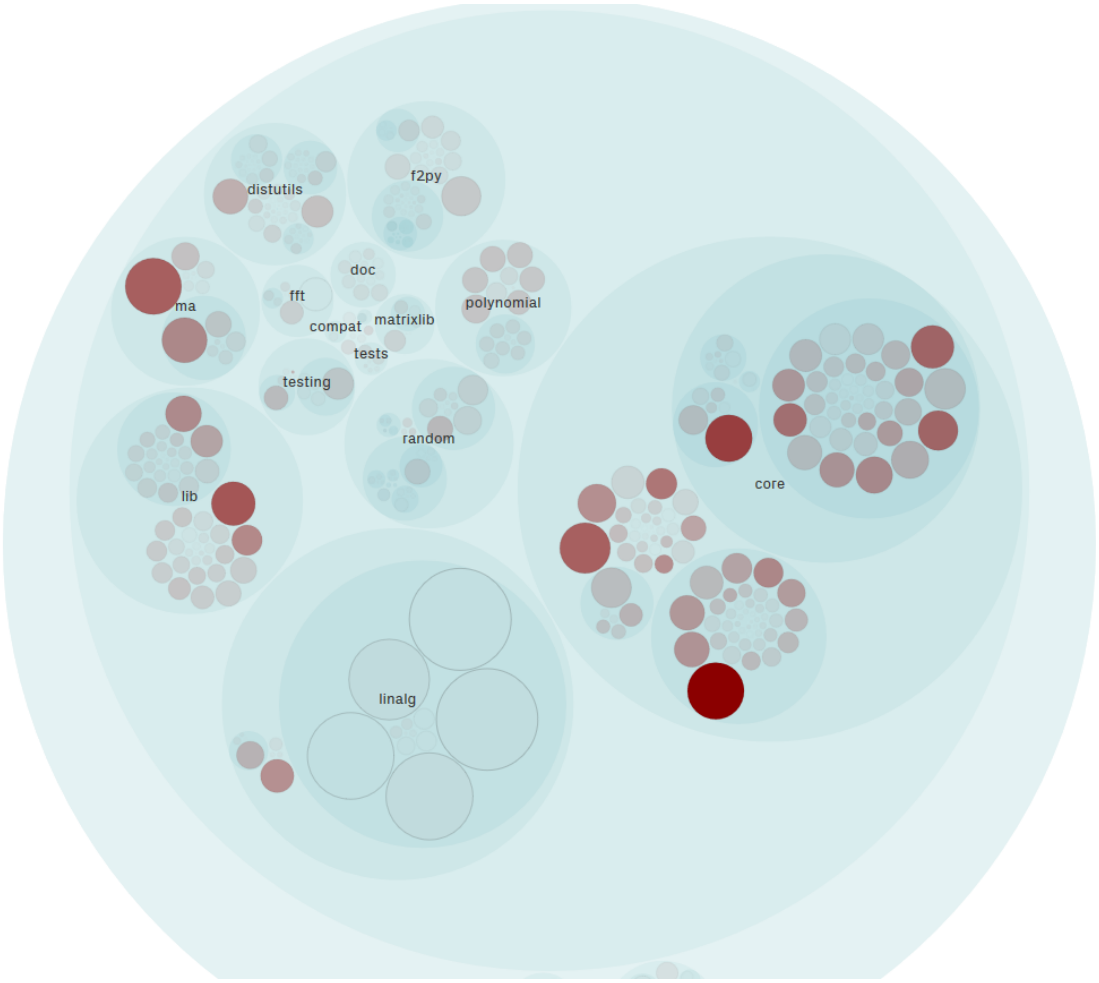


Figure 17.9: Hotspots in the project

17.4.4.2 Roadmap

As has become clear from previous parts of this essay, NumPy's hotspots are mostly placed in its core module. This is not surprising, because NumPy's core module houses the implementation of its array object which is the building block for all of NumPy's other functionality. Therefore, it is also not surprising that most of NumPy's major roadmap items pertain to this core component. The following descriptions were copied from our previous essay and modified to indicate their respective components[42]:

- Interoperability
  - *The NumPy developers want to make it easier for other libraries to interoperate with NumPy. They want to provide better interoperability protocols and better array subclass handling, among other things.*
  - These features relate to how the array object can be subclassed and how libraries with their own adapted versions of NumPy can cooperate within the same source code. All roadmap items related to interoperability will therefore relate to the core component.
- Extensibility
  - *The NumPy developers want to simplify NumPy's internal datatypes (`dtypes`) to simplify extending Numpy's functionality. For instance, they want to simplify the reation of custom dtypes and want to add new 'string' dtypes for dealing with textual data.*
  - As with interoperability, these roadmap items deal with extending or modifying NumPy's internal datatypes, which are housed in its core component.
- Performance
  - *The NumPy developers want to improve NumPy's performance through for instance SIMD instructions and optimisations within functions.*
  - These performance features will be implemented using low-level code, and not Python, so they too will relate to the core component.
- Website and documentation
  - *The website needs to be rewritten completely and the documentation is of 'varying quality'*[43].
  - The website is not part of the NumPy library and does not belong to any component.
- Random number generation policy & rewrite
  - *At the time of writing, the developers are close to completing a new random number generation framework.*
  - Random number generation does have its own component, called 'random'. This is where random number generation frameworks are added as well.

17.4.5 Evaluation of Code Quality

As mentioned before, we have used CodeScene to analyse the code of NumPy. In addition, we have access to the analysis that has been made by the SIG. We will now look at the outcomes of these analyses and possible areas for improvement.

17.4.5.1 CodeScene Analysis

As can be seen from the CodeScene analysis before, NumPy scores very poorly on The Code Health metric of CodeScene. CodeScene indicates that most files have: deeply nested logic, large file and function sizes,

[42] NumPy's roadmap, retrieved on 2020-03-26. https://numpy.org/neps/roadmap.html

[43] Section about website and documentation of NumPy's roadmap, retrieved on 2020-03-05. https://numpy.org/neps/roadmap.html#website-and-documentation

high complexity and contain brain methods, which are functions with too much important behaviour. All these flags that are raised by CodeScene can be attributed to the fact that NumPy simply provides a lot of different functions for working with arrays. The library contains files which consist of different functions that can be used to manipulate and work with arrays. Therefore, we do not consider the raised warnings as problems to the source code.

17.4.5.2 Software Improvement Group Analysis

Through their portal Sigrid, the SIG has provided us with insightful metrics about the NumPy repository.

The system fact sheet provides a high-level overview of how the NumPy repository scores on different metrics defined by the SIG[44]. It can be seen that NumPy scores relatively well on `Volume`, `Duplication` and `Component balance`. The repository is thus not too large, avoids duplication and splits its code well into different components, in accordance with their architectural style which is component-based. You can read more about this in our previous essay.

On the other side, just like CodeScene, the SIG rates the `Unit size`, `Unit complexity` and `Unit interfacing` as quite poor. This can again be contributed to the fact that NumPy simply provides a whole lot of different functions as mentioned before.

Below, some more statistics can be found which have been provided by the SIG. As can be seen, NumPy scores well on code duplication, with only `8%` of duplicated code. Most of this resides in the `core-src` library and are probably boilerplate C code.

NumPy scores clearly poor on unit size and complexity, with respectively `16%` and `32%` of the code falling within the acceptable ranges. The unit size and complexity are not scoring well across the whole system as can be seen in the bar graphs.

17.4.6 Technical Debt

According to Martin Fowler technical debt in a system is a deficiency, or as he calls it, *cruft* in quality which makes it harder to change the system[45]. In this section we will discuss occurrences of technical debt in the NumPy project.

The SIG provides a large number of refactoring candidates through Sigrid. These are however almost all related to unit size or complexity as NumPy scores poorly in these areas. If these refactorings would actually be done, a lot of functions would be moved to separate files, making the repository a lot less structured and much more unclear.

Other technical debt that we came across were quite some lines of legacy codes. This is is however inherent to a library that provides functionality for many different versions of Python. With the removal of support for Python 2.7, a lot of this legacy code has already been taken care of however.

17.4.7 Conclusion

Being an open-source project, the NumPy project has multiple checks in place for assuring code quality. These involve manual checks by core developers but also quite some CI checks for different platforms and versions.

[44] Software Improvement Group user manual, retrieved 2020-03-26. https://sigrid-says.com/assets/sigrid_user_manual_20191224.pdf
[45] Martin Fowler: *Technical Debt*. https://www.martinfowler.com/bliki/TechnicalDebt.html

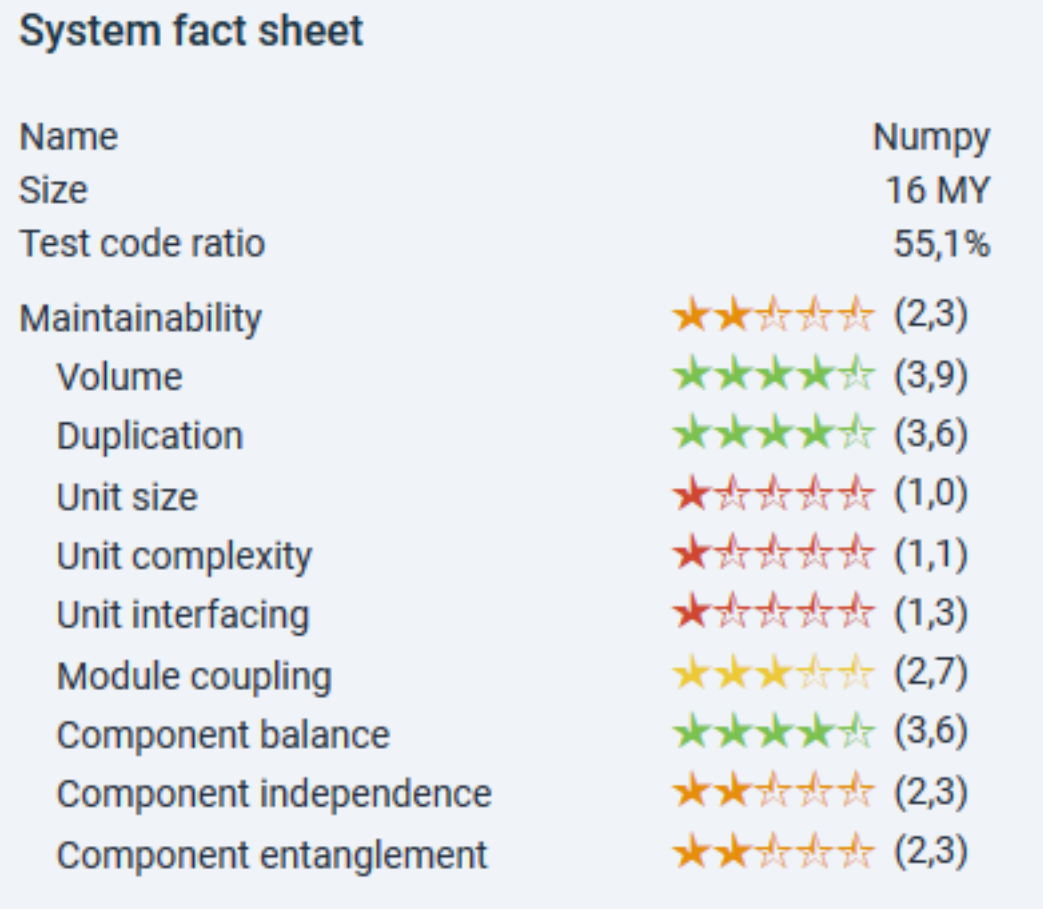


Figure 17.10: Fact sheet of the NumPy library, provided by SIG

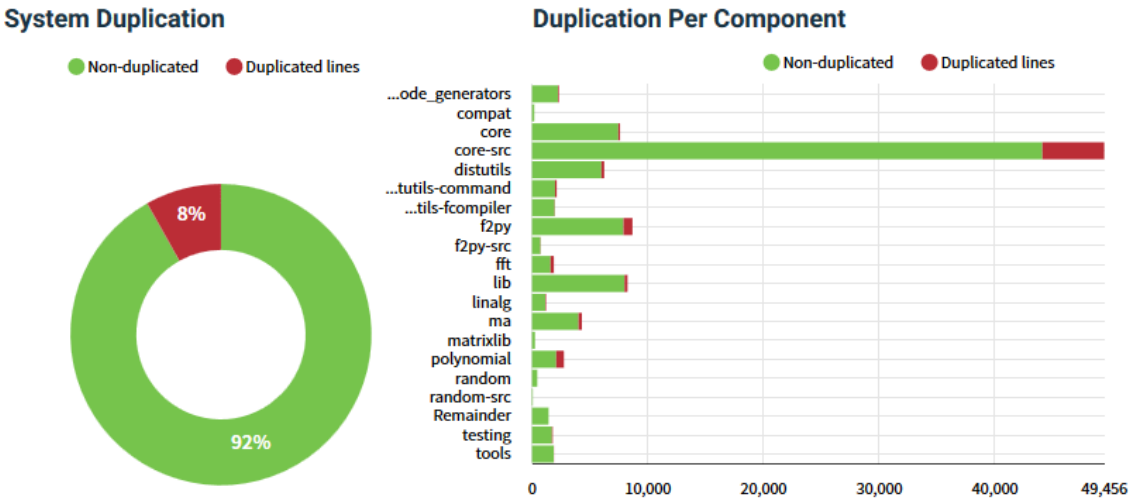


Figure 17.11: Code duplication in the NumPy library, provided by SIG

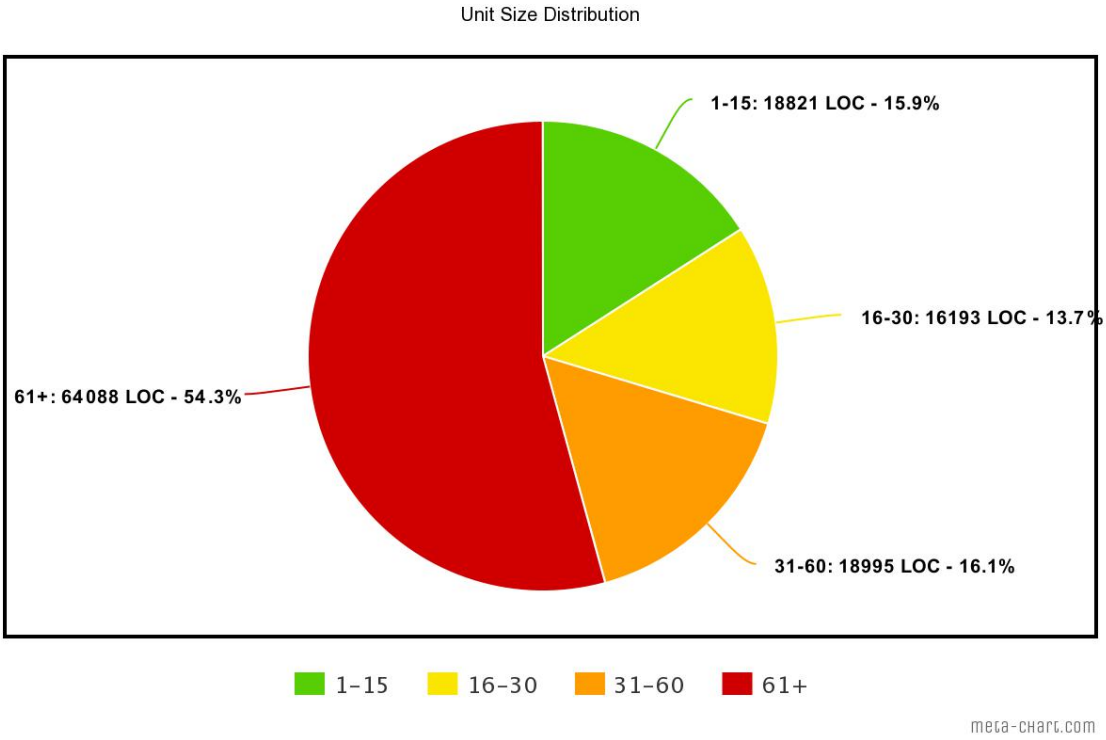


Figure 17.12: Unit size of the NumPy library, based on raw data provided by SIG

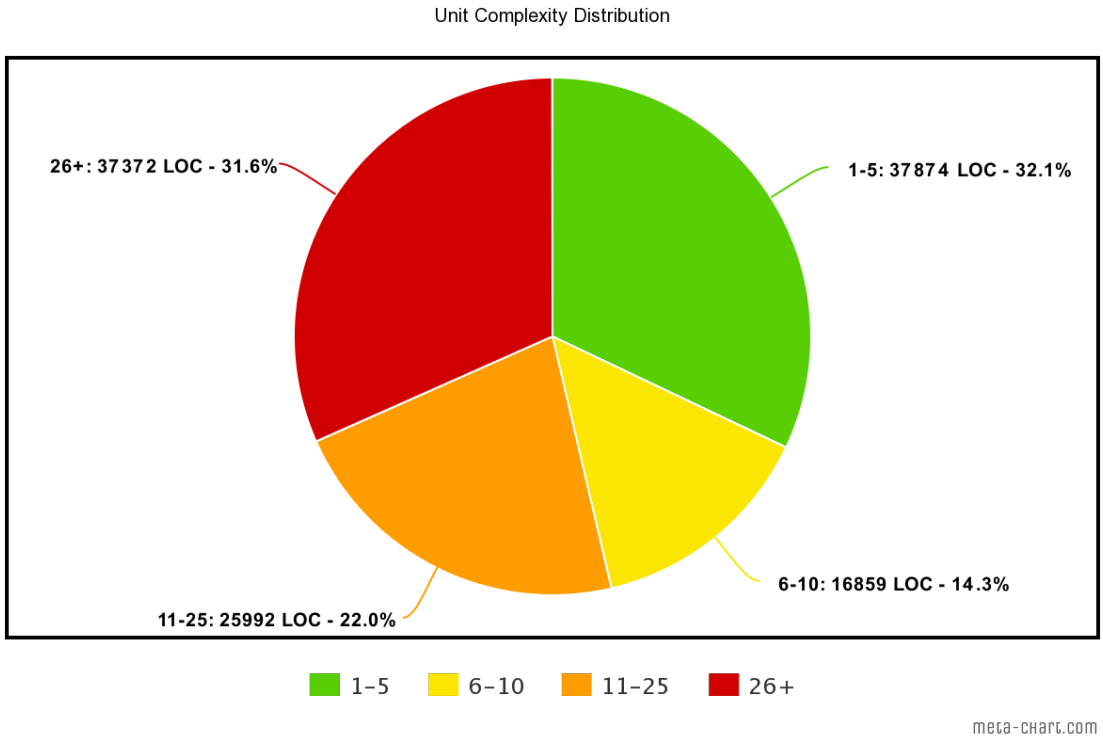


Figure 17.13: Unit complexity of the NumPy library, based on raw data provided by SIG

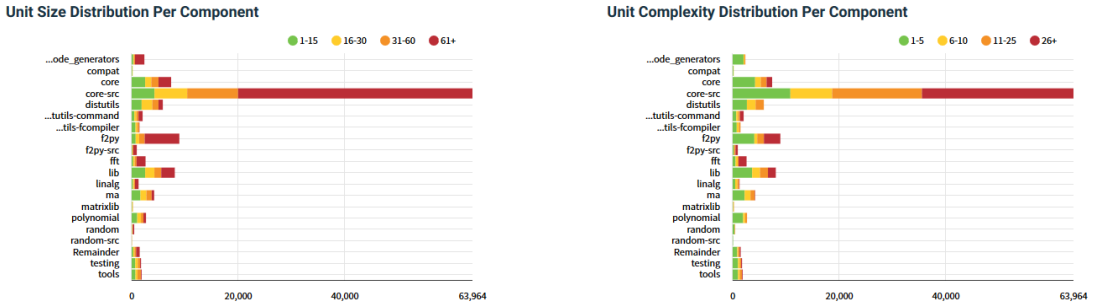


Figure 17.14: Unit size and complexity of the NumPy library per component, provided by SIG

The repository does a good job in following its component-based architecture, but contains many large units with high complexity. This can be attributed to the fact that NumPy simply provides a whole lot of different functions to work with arrays. These functions usually work by themselves and are grouped by category through the separation in different components.

17.5 NumPy: Carefully Crafting Components through Collaboration

In this fourth and final essay of our series on the NumPy project we will combine two very important aspects of developing a (software) product: the technological aspect and the social aspect. As an architecture is designed, developed and built-upon by humans, the social side of the process should not be overlooked. Conway's Law will play a central role in this techno-social analysis of the NumPy project.

We start out by explaining the relevance of the techno-social aspect of a project. Then, we continue by identifying loosely and tightly coupled components in the NumPy repository using information from previous essays and tools like Sigrid and CodeScene. After that, we will analyse the communication between the developers of NumPy using the GitHub GraphQL API Explorer and a custom-made Python script. We close off by assessing the results in relation to Conway's Law.

NOTE: The formatting in this essay has been optimised for the online version.

- Techno-Social Congruence
- Component Coupling
- Developer Discussion
- Computing congruence and connection to Conway's law

17.5.1 Techno-Social Congruence

You might think: 'what even is Techno-Social Congruence and why does it matter?'. Well, Cataldo et al. propose the following:

> We define socio-technical congruence as the match between the coordination requirements established by the dependencies among tasks and the actual coordination activities carried out by the engineers.[46]

This means that the congruence is determined by the technical dimension as well as the social dimension of a project. This is still a bit abstract, so let's dive a bit deeper.

In software development, not only the engineering decisions have an impact on the result, organising the developers behind the project is also of importance. As argued by Conway there is a relation between the design of a system and the structure within the organisation behind it[47]. In the domain of software engineering, this can in general be seen as the software architecture relating to the structure in which the developers are organised.

[46] Cataldo, Herbsleb, and Carley. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. ICSE 2008. https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf

[47] Conway, M.E. 1968. How do committees invent? Datamation, 14, 5, 28-31. http://www.melconway.com/Home/Committees_Paper.html

When multiple developers are working on a project there is a chance that their code will interact or multiple developers are working on the same component. This is where techno-social congruence matters, if the communication between developers is improved, their output will likely also improve.

Cataldo et al. already propose a way to compute the congruence between the coordination that is needed while developing a component and the actual coordination happening between the developers[48]. The proposed method requires a large amount of data and manual analysis, so we decided to simply review the coupling between the different components and then compare this to the amount of communication between the developers while working on the components.

17.5.2 Component coupling

In this section we analyse the dependency relationships between the components that NumPy consists of. We do this by looking at the dependency graph provided by SIG through Sigrid. This graph shows the dependencies between components in terms of call relationships, inheritance relationships and C implement relationships[49].

One dependency that catches the eye is the dependency between the components `lib` and `core`. The `core` component contains the core of the NumPy functionality i.e. *ndarray*, *ufuncs* and *dtypes*. `lib` is mostly a space for implementing functions that don't belong in `core` or in another NumPy submodule with a clear purpose such as `fft`, `linalg` or `random`. It is not a big surprise that `lib` depends so heavily on `core` because NumPy obviously stores a lot of functionality in the `lib` module.

Another striking feature of the image is that `core-src` and `random-src` are included as components, though do not appear to be coupled to other components. This is, of course,rather unlikely but can be explained by the way NumPy is set up and the origin of these components. For performance, NumPy's core is written in C, as was elaborated upon in previous essays. The `core-src` and `random-src` components house this C-code. Communication from these components to the rest of NumPy's codebase is done through `core` and `random`. Though `core-src` communicates with `core`, and `random-src` with `random`, it appears that Sigrid was not able to pick up this relationship between Python and C hybrid code when creating the dependency graph.

We kept the `core-src` and `random-src` components visible in this diagram to remain consistent with our previous component definitions. However, to interpret this diagram it is a good idea to regard `core` and `core-src`, and `random` and `random-src` as the same component.

Finally, the `distutil` module is called by all components. This is the case because distutils is a toolbox for distributing packages. Numpy provides extended `distutil` functionality to ease the process of building and installing sub-packages and auto-generating code Moreover, it provides extension modules that use Fortran-compiled libraries.

It is also worth noting that specific, library-like components such as `linalg`, `random`, `fft`, and `polynomial` have the same position in the project, being near-standalone libraries for specific mathematical functions. Therefore, it is to be expected that these components have no coupling between them.

[48] Cataldo, Herbsleb, and Carley. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. ICSE 2008. https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf

[49] Software Improvement Group user manual, retrieved 2020-03-26. https://sigrid-says.com/assets/sigrid_user_manual_20191224.pdf,

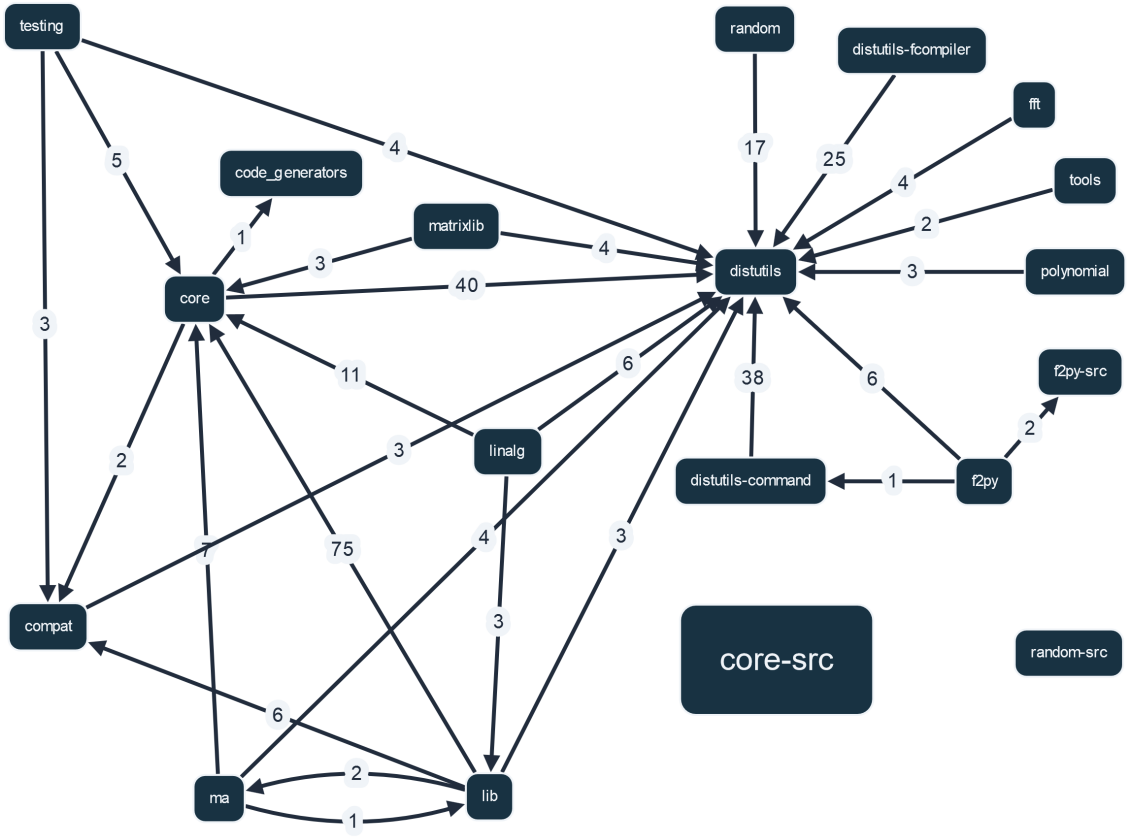


Figure 17.15: Dependency graph. Created by SIG.

17.5.3 Developer Discussion

The other side of Conway's law is the communication between developers[50]. There are multiple commuincation channels for NumPy, there are the mailing lists[51] and of course the GitHub repository itself. In the GitHub repository itself most communication happens in the issues and pull requests.

In order to analyse the communication patterns between developers we wrote a tool that counts the amount of comments of developers on GitHub. These comments are measured using the GitHub API. The tool itself is also open sourced on GitHub[52].

The comments are counted per user and per component using labels. These labels specify the component the issue or pull request is about, for example: `component: numpy.core`. To calculate the overlap score of communication between 2 components for a certain user we use the following formula:

```
1 - ABS(#component A comments - #componentB comments)/(#component A comments + #component B comments)
```

This formula ranges from 0 to 1, when the amount of comments on both component A and B are similar the overlap score will be 1. If the amount of comments differs a lot the score will be closer to 0. We can take the average of these scores for the different users to get a total score for a component pair, which then also ranges from 0 to 1 but will likely be much lower as many users will have a score of 0. Here a high score would mean that users that comment in issues/PRs of component A also comment in issues/PRs of component B, which means that the users in this component pair communicate with eachother a lot.

Some noteworthy results for several component pairs are shown in the table below:

	core-lib	core-linalg	linalg-fft	linalg-polynomial	core-distutils	random-linalg
Total score	0,016687626	0,020836	0,009528	0,009761	0,012406203	0,016775517

As can be seen in the table the scores for `linalg-fft` and `linalg-polynomial` are significantly lower than the scores for `core-lib` and `core-linalg`. This brings us to the results of the analysis in the following section.

17.5.4 Computing congruence and connection to Conway's law

Now that we have discussed the coupling between components and the discussion process of NumPy's developers, we can compare these results and determine if Conway's law appears to hold.

The analysis of dependencies between components told us that `core` and `lib` are tightly coupled and when we ran the github comment analysis tool we obtain a result that conways law predicted indeed. The output of the formula is `0,017163665` which is much higher than the output for `linalg-fft` for example, which was only `0,009279859`.

Furthermore, it seems that communication between specific, library-like components such as `linalg`, `fft`, and `polynomial` is just as limited as the coupling measured in the coupling diagram from a few sections

[50] Conway, M.E. 1968. How do committees invent? Datamation, 14, 5, 28-31. http://www.melconway.com/Home/Committees_Paper.html

[51] NumPy mailing lists overview, retrieved 2020-04-08. https://www.scipy.org/scipylib/mailing-lists.html,

[52] GitHub Counter project page. https://github.com/Jimver/github-comment-counter,

back (here). Namely, these components are not coupled at all, according to the diagram.

This seems to correspond to the scores measured in the previous section; 0,009528 for communication between `linalg` and `fft`, and `0,009761` for communication between `linalg` and `polynomial`. These scores are about 2 times lower than the score for `core` and `lib`.

However, the score for the relation `random-linalg` was `0,016775517` and this is remarklably high. We speculate that, even though it is reasonable to assume that linear algebra functions sometimes use random variables, such as for power iterations, `random` is a relatively separate library. This assumption is reflected in the graph as well.

We can see that in most cases Conway's law seems to hold, there is namely more communication between more tightly coupled components and less communication in more loosely couple components. There are some cases in which Conway's law matches the communication patterns less, but in general, we can conclude that the law applies to the NumPy project.

Chapter 18

Open edX



The Open edX platform provides the massively scalable learning software technology behind edX. The edX platform is the online learning destination co-founded by Harvard and MIT with Open edX being the platform that provides the learner-centric, massively scalable learning technology behind it. It was originally developed for MOOCs, however the Open edX platform has now evolved into one of the leading learning solutions catering to higher education institutions, enterprise and government organizations alike.

edX is a trusted platform for education and learning. Founded by Harvard and MIT, edX is now home to more than 20 million learners, the majority of top-ranked universities in the world and industry-leading companies. As a global nonprofit, edX aims to transform traditional education, removing the barriers of cost, location and access.

The edX platform provides massive open online courses in a wide range of disciplines to a worlwide audience. edX is a nonprofit organization that runs on the free open edX software platform. In addition to offering educational courses, edX also conducts research into distance education trying to better understand how people use the online platform to learn. edX offers certificates of completion and for some courses students are even eligible for receiving credits. The courses will often provide tutorials in the form of videos which are similar to in-class lectures, course notes and online discussion forums where students can ask questions and get help from teaching assistants. Some courses also contain various assignments and online laboratories that students have to complete.

Open edX is the open source software developed by edX which is available to other institutions that wish to provide similar services. The open edX server-side software is mostly based on Python using Django as the web application framework with JavaScript being a very big part of the code base as well.

18.1 Product Vision

In this essay, a deep dive was done into the Open edX project, to ascertain its product *vision*. This vision describes what the system aims to do, and for who. While also keeping the future in mind. For a short introduction into Open edX and what it aims to do, see our intro page.

18.1.1 End-user mental model

When designing software, the end-user's value should always be kept in mind. The main goal of Lean, actually, is to "create a value stream, every one of whose activities adds value to the end user" [1]. Mental models consist of **beliefs**, not **facts**. It is about what they think or expect from a given system, not what actually will happen. We will go over what we believe to be the two main mental models that exist for the Open edX system.

18.1.1.1 Teachers

Teachers are a big part of the end-user base for the Open edX system, as they design, maintain and teach the courses available on it. Their beliefs can be specified as follows:

- The ability to design courses using videos, slides, quizzes and assignments.
- The ability to set deadlines for assignments, including penalties.
- The ability to query important statistics about their course and its participants, as well as past participants.
- The ability of the system to do assessments automatically based on defined rules.
- The ability to send out announcements to all participants, or particular subgroups, of a course.

18.1.1.2 Learners

Learners beliefs can be summarized as follows:

- The ability to enroll for a given course.
- The ability to browse courses systematically (filters, categories).
- The ability to view all enrolled courses, upcoming deadlines and course announcements centrally.
- The ability to approach teachers and teaching assistants with ease.
- The ability to interact with courses on any device.
- The ability to receive credentials for passing a course.

18.1.2 Key capabilities and properties

In this post we provide a short summary of the key capabilities and properties that the Open edX system provides and discuss some of the main features that it has to offer.

18.1.2.1 System goals

The main goal of Open edX is to deliver inspiring learning experiences on any scale. This means delivering online campuses, instructor-led courses, degree programs and self-paced courses using a single coherent platform. In order to achieve this goal, the Open edX platform needs to easily and confidently scale from supporting small learning to thousands of simultaneous learners.

[1] Coplien, J. O., & Bjørnvig, G. (2011). *Lean architecture: for agile software development*. John Wiley & Sons.

18.1.2.2 System capabilities

The key capability of the Open edX platform lies in its ability to empower learners and instructors with the tools they need for either learning or facilitating the learning process. This directly translates in the tools the platform provides with examples such as support for interactive forms and discussion boards, advanced learner and instructor dashboards and live video conferencing capabilities. Moreover, the platform provides cross-device as well as cross-platform capabilities with the ability to seamlessly integrate with other third party tools such as Salesforce. In today's world where most of the online traffic is taking place on mobile devices, offering support for these types of features is therefore very important.

Users of the Open edX platform are also able to customize their learning platform within minutes, having access to a set of tools that offers a rich authoring experience. From interactive content with adaptive video streaming, multimedia animation and simulation to AR and VR support, the Open edX service truly caters for all use cases an instructor might have. The intelligent analytics provided by the platform also offer instructors and researchers key insight and real-time data analysis over their courses and how students enrolled in those courses perform.

18.1.2.3 System features

In the tables below we present the most important features of the Open edX platform.

Accounts	Authentication	Interface options
Add a new user	Active directory/LDAP integration	Block management
Archive users	Custom user login page	Language settings
Browse list of user	Manual accounts	Media embedding settings
Bulk user actions	No login	Multilanguage support
Custom/Mandatory user fields	SAML2/API integration	~~Ready-made themes~~
Upload users	Self-Registration w (or w/o) admin confirmation	~~Calendar settings~~
		~~Location settings~~

Activity grading	Certificate management	Gamification	Learning types	Format	Security
Course History	Certification life-cycle	Badge customization	Asynchronous Instructor-led	Course Discussions	Anti-spam
Gradebook	Manage certification templates	Badges	Asynchronous Self-paced	~~Gamification Format~~	Anti-virus
Gradebook audit trail	Predefined certification templates	~~Customize Gamification mechanics~~	Blended Learning	Learner Upload	IP Blocker
Gradebook comments	Unique Certificate by Course	~~Leaderboards~~	Synchronous Virtual Classroom	LIVE Chat Option	Restrict registration to specific domains

Activity grading	Certificate management	Gamification	Learning types	Format	Security
Manual Grading (“Marking”)	Unique Certification by Curriculum	Levels		LIVE Videoconferencing / Webinar	Strong passwords
Multiple grading scales		~~Points~~		Social Format	
		~~Rewards~~		Topics Format Weekly Format	

Note: crossed out fields are features which are not currently supported by the Open edX platform.

18.1.3 Stakeholder Analysis

To get a clear picture of all the relevant entities that might have an interest, or concerns with the realization of the Open edX platform, a stakeholder analysis was performed. This was done with the classes of stakeholders defined by [2] in mind. Some classes are omitted due to irrelevance in the context of Open edX.

18.1.3.1 Acquirers

Open edX is the open source platform developed by the MOOC provider edX. edX itself was created in May 2012 in a joint venture between MIT and Harvard.

18.1.3.2 Assessors

Open edX is based in the United States, which means that the legal conformity of their platform according to American educational laws would ultimately be assessed by entities such as: the Office for Civil Rights of the Department of Education, the Department of Justice, the Department of Labor and the EEOC [3]. For GDPR laws, national bodies have been selected as assessors.

18.1.3.3 Developers

Since Open edX is open source, developers could be people from all over the world. In practice, however, active frequent contributors are mostly part of the edX team. Other developers include coders from other MOOC platforms which implement Open edX as their backbone. Including educational companies DataQuest and Labster.

See Appendix A for current developers to approach about the Open edX system.

[2] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

[3] National Association of College and University Attorneys (NACUA). (2019, 22 oktober). *Higher Education Compliance Matrix*. Consulted on March 5th 2020, van https://www.higheredcompliance.org/compliance-matrix/

18.1.3.4 Suppliers/System Administrators

To setup your own platform using Open edX, two options are offered: self-managed and fully-managed.

Self-managed solutions include distributions of the Open edX platform with added functionality. Open edX lists 3 suppliers who offer such distributions, namely: IBM, Tutor and Bitnami. When hardware is not available/feasible, cloud providers such as AWS, Google Cloud or Microsoft Azure would be a popular choice.

Fully-managed solutions are provided by other service partners who offer the Open edX platform without any management necessary: EduNext, Learniphi, MOOcit, Appsembler and Edly.

18.1.3.5 Users

Enrollments across all edX courses are over 70 million[4], which is not including rebranded MOOC platforms implementing the Open edX platform.

Besides that, edX lists 57 partnered schools and universities of which 16 are stated to be contributors[5]. Some of these universities include: MIT, Harvard, TU Delft, Boston University and Berkeley University.

18.1.3.6 Competitors

Several other MOOC Providers exist, such as:

- Coursera
- Udacity
- Udemy
- FutureLearn
- Khan Academy
- Canvas
- The Open University
- XuetangX

Of these platforms only Canvas is also open source.

18.1.4 Context View

The context view is used to delimit the system from all its communication partners. To be more precise, in this section the relationships, dependencies and interactions between the Open EdX platform and the external are specified.

A short explanation of some of the components in the context view model:

Developers: As an open source project Open EdX gives the opportunity to individual developers to contribute to the project. Active GitHub users can take part in adding new features and in code maintenance.

Databases: Courses are stored in MongoDB. Per-learner data is stored in MySQL. The analytics that occur from events describing user's behavior are also stored in MySQL.

Documentation: The edX Developer Documentation is created using RST files and Sphinx.

[4] edX. (z.d.). *EdX, Schools and Partners*. Consulted on March 5th 2020, van https://www.edx.org/schools-partners
[5] edX. (z.d.). *EdX, Schools and Partners*. Consulted on March 5th 2020, van https://www.edx.org/schools-partners

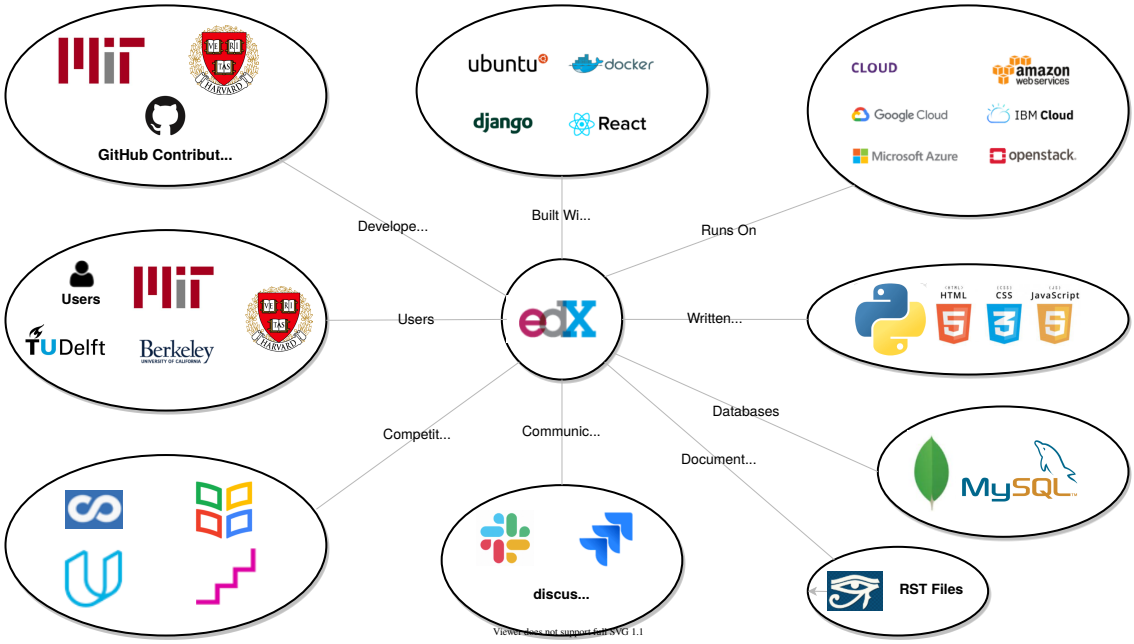


Figure 18.1: Context View

Communication: Technical questions posting and chatting with the community can be achieved at the Open EdX discussion Forum. Moreover, JIRA is used as an issue tracker by the developers. Finally, real-time conversations are available on Slack.

18.1.5 Open edX roadmap

Open edX creates named releases which are different from the daily deployments with a longer release cycle on the order of six months between every release. The named releases are tested by the edX and Open edX community in order to ensure early detection of bugs and fix issues quicker.

18.1.5.1 Future Version: Juniper

The upcoming version is called Juniper[6] and it contains several architectural differences:

18.1.5.1.1 Changes to architecture regarding login credentials: Login and Registration is now solely handled by the LMS component. Studio redirects to the LMS for registration[7]. A new component called learning portal is being introduced.

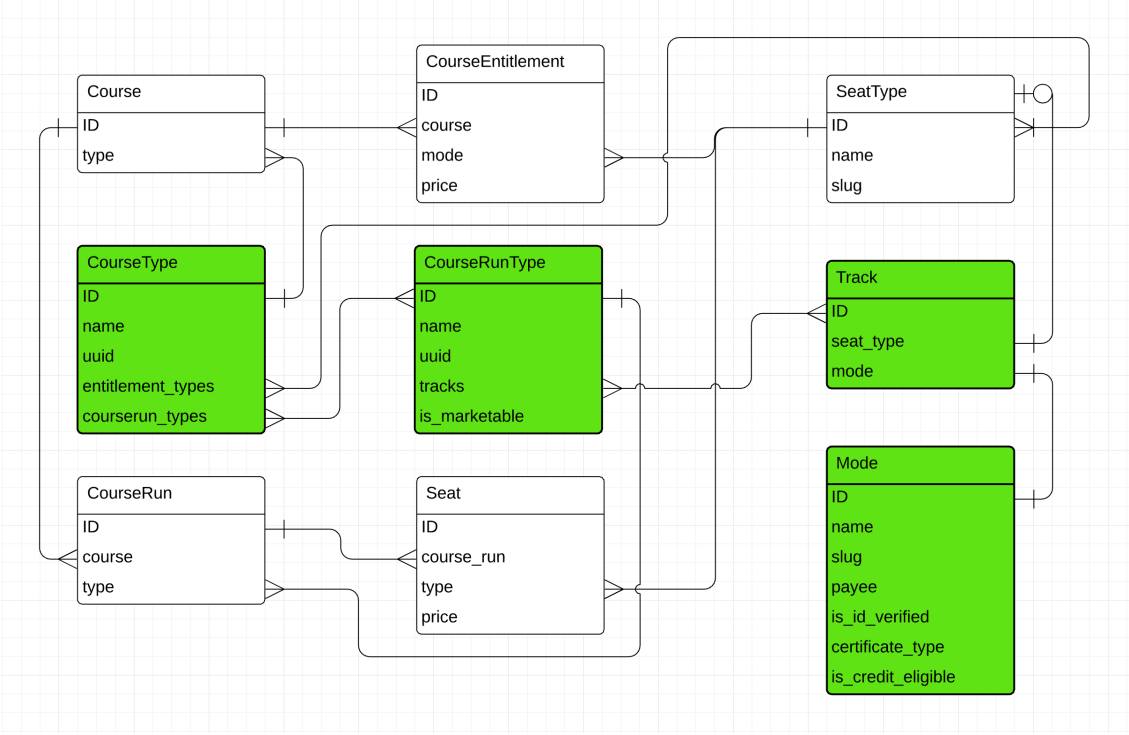
18.1.5.1.2 Documentation necessary for: Gradebook Proctoring ORA-2 Course Dates

[6], https://openedx.atlassian.net/wiki/spaces/COMM/pages/940048716/Juniper
[7], https://github.com/edx/edx-platform/pull/22416

18.1.5.1.3 A new feature component called frontend-app-publisher: Frontend-app-published integrates between course discovery, ecommerce, and Studio services and can be used to create new courses and course runs that will be pushed out to Studio for content authoring [8]. - "there is no standard process for installing Micro frontends, but Publisher is still being provided so the community can become more familiar with it and possibly configure and install it on their own."

18.1.5.1.4 Additions to studio: Reorganizing data structure for course metadata[9]: Each course run now holds a record of the seats in the course, and E-commerce products in the course metaData.

The addition of multiple LMS modes such masters being available now, it has created a divergence between the seats in the course and e-commerce product attached to it. The course run now will hold the ground truth in the meta data. This allows to easily add new LMS modes or products.



[10]

Juniper main references features:

18.1.6 Appendix A

This table lists the most active developers working on Open edX from the edX team.

[8], https://github.com/edx/frontend-app-publisher/
[9],https://github.com/edx/course-discovery/blob/master/docs/decisions/0009-LMS-types-in-course-metadata.rst
[10],https://github.com/edx/course-discovery/blob/master/docs/decisions/0009-LMS-types-in-course-metadata.rst

Name	Github handle
Calen Pennington	[@cpennington](https://github.com/cpennington)
Ned Batchelder	[@nedbat](https://github.com/nedbat)
Diana Huang	[@dianakhuang](https://github.com/dianakhuang)
Nimisha Asthagiri	[@nasthagiri](https://github.com/nasthagiri)
Feanil Patel	[@feanil](https://github.com/feanil)

18.2 From Vision to Architecture

In the previous essay, we focused on the vision of the Open edX system. In this essay, we will focus on the realization of that vision, visible in the architecture of the Open edX system.

This is done by taking a high level view of the architecture of the system. What architectural views are relevant? What main architectural patterns are visible in the system? From there, concrete overviews of the following viewpoints are given:

- Functional
- Information
- Development
- Operational
- Deployment

Lastly, non-functional properties relevant to the architecture are reviewed.

18.2.1 Architectural Style

According to Rozanski & Woods, six core architectural viewpoints exist [11], namely: Functional, Information, Concurrency, Development, Deployment and Operational.

18.2.1.1 Relevant viewpoints

From these the functional, information, development, operational and deployment viewpoints are relevant to Open edX.

The functional view is relevant because it drives the definition of the other architectural views [12] (p. 215). It defines the architectural elements that deliver the system's functionality. As such, its composition ultimately drives the value gained by the end-user.

The information view depicts how information is stored/manipulated/managed/distributed within the system. In a system such as Open edX, which delivers educational content to learners all over the world, this view is quite relevant.

Next, the development view needs to be considered as the Open edX system is open source, and meant to be extended/modified by other parties than edX themselves. This view describes architecture that supports this development process.

[11] Rozanski, N., & Woods, E. (2012). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.

[12] Rozanski, N., & Woods, E. (2012). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.

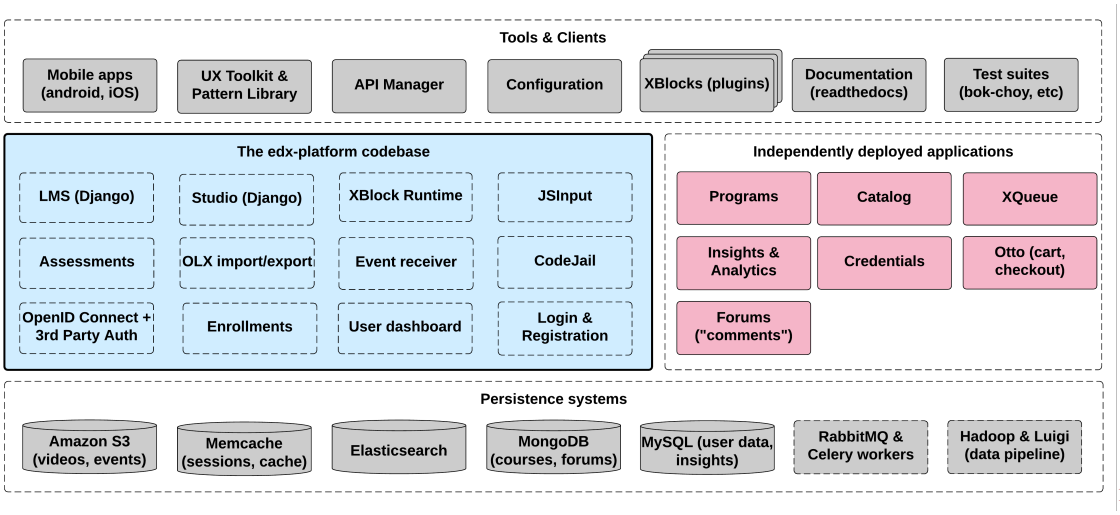
Furthermore, the operational view is considered as the Open edX system will need to be updated/administered, while maintaining consistency and availability for its end users.

Lastly, the deployment view is relevant to Open edX as it is deployed in many different forms and fashions. As discussed in our previous essay, many suppliers exist for Open edX. Each of them provides the Open edX system to schools and universities in different packages and formats. As such, the deployment view is important to consider.

18.2.1.2 Patterns

An overview of the Open edX architecture is given in the image below. From this we can see that the edx-platform codebase captures most of the important functionality of the system. This block is seen as one service, and is supported by the independently deployed applications (IDAs) to the right of it. This means that the edx-platform module can be seen as "a single huge object, with lots of small objects attached to it", or, a "God element" [13]. Which, according to Rozanski & Woods, is stated to be one of the common pitfalls encountered when designing functional viewpoints.

Luckily, the edX team has already acknowledged this and states that "Over time, edX plans to break out more of the existing edx-platform functions into new IDAs" [14]. Which translates back to the principles of cohesion within modules and decoupling between. This speeds up the development process, as relevant responsibilities and functionality are grouped together, minimizing unnecessary coordination between them [15].



16

In all, the goal of edX seems to be this: *To create a modular architecture aimed at reducing complexity, and increasing the ease with which developers can understand and contribute to its IDAs.*

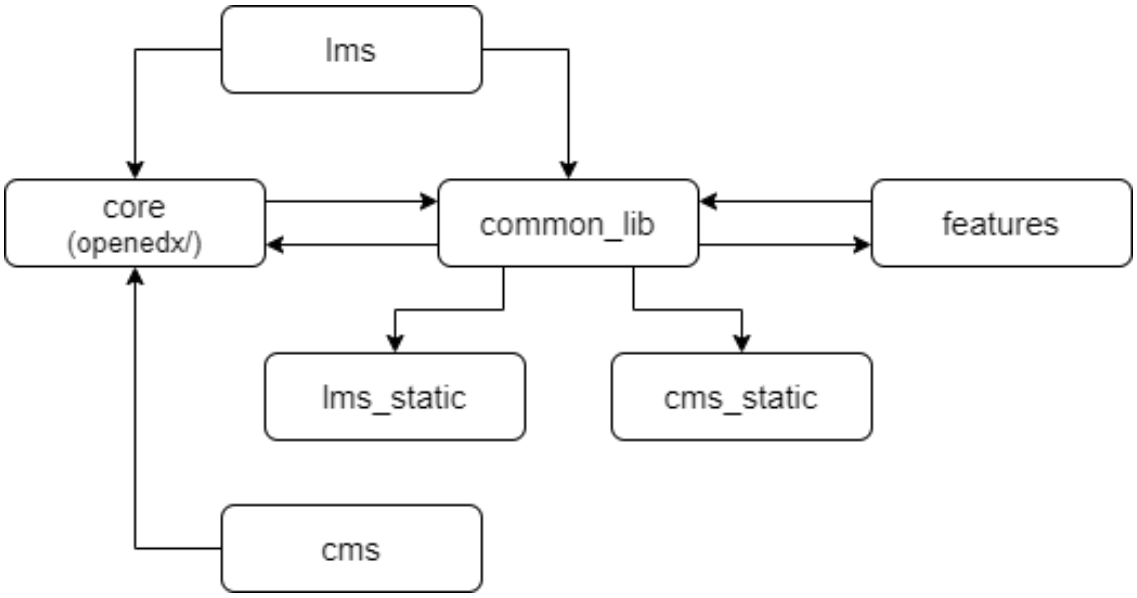


Figure 18.2: Open edX Modules

18.2.2 Development View

18.2.2.1 Module Organization

The core package for Open edX can be found in the `openedx/` folder. The current intention of the developers is that all importable code from Open edX will eventually reside here. This includes the code from the `lms`, `cms` and `common` modules which currently lives in separate directories. The `lms` module houses the functionality of the *learning management system* while the `cms` takes care of the *content management system*.

Since the platform's backend is written in Python, it heavily relies on Django external apps. These reside in the `core/djangoapps` directory. Furthermore, utilities that require Django are placed in the `core/djangolib` directory. Code that does not define Django modules or views of its own is found in the `core/lib` directory. Finally, the `features` module which mainly interacts with the `common` module contains other code which handles various functionality of the edX platform that is not related to either the learning nor the content management systems. On the project's GitHub page there is a note regarding code that is not structured like this being treated as legacy code, which shows the developers are concerned with maintaining high quality standards for the system architecture.

From the top-level decomposition that we performed on the project's codebase we could identify the interactions between the previously mentioned core components. This analysis shows that the `core` and `common` modules are the two most important since these two handle almost all the requests which come

[13] Rozanski, N., & Woods, E. (2012). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.

[14] https://edx.readthedocs.io/projects/edx-developer-guide/en/latest/architecture.html

[15] Coplien, J. O., & Bjørnvig, G. (2011). *Lean architecture: for agile software development*. John Wiley & Sons.

[16] https://edx.readthedocs.io/projects/edx-developer-guide/en/latest/architecture.html

from the other modules namely the `lms` and `cms`.

18.2.2.2 Development Process

Installing and running an Open edX instance is not simple. This is why it is recommended to use the Open edX developer stack which is a Docker-based development environment. The steps for contributing to the Open edX project are clearly described in the contributing document as well as in the documentation pages of Open edx. After setting up the dev stack, one can get in contact with the other developers by joining the Jira server of the project or creating an account on the discuss.openedx.org website. Here, there are multiple threads on which developers can create new discussions and start conversations about the issues they are planning on fixing or features they want to add. On the Jira page, developers can also find an issue tracker with reported problems discovered by other developers and users. Finally, to make a contribution to the project, a developer needs to create a pull request on GitHub with a specific description in which details about the implemented change(s) are given. Once approved, the change gets added to the code base. The code will end up on the edX production servers in the next release, which usually which happens every week.

18.2.2.3 Standardization of Testing

The two main programming languages used in the Open edX platform are Python and JavaScript. A variety of tools are used for checking the codebase for any errors or vulnerabilities as well as enforcing a coding standard and coding style. To this end, developers are provided with a tool for running a check on the overall quality of their code by running the `paver run_quality` command in the root folder of the project. Moreover, a set of different tools is used depending on the programming language.

- **Python:** the pep8 tool is used to follow the PEP-8 guidelines and pylint is used for static analysis and to discover trouble spots in source code
- **JavaScript:** In order to standardize and enforce Open edX's JavaScript coding style across multiple codebases, edX has published an ESLint configuration that provides an enforceable specification. EdX JavaScript style generally follows the Airbnb JavaScript Style Guide, with a few custom rules.

18.2.2.4 Codeline Organization

The Open edX framework does not have a single standard for describing the overall code structure. The framework is mostly written in Python and JavaScript.

18.2.2.4.1 Python Python is used for the platform's backend together with the Django framework. Django is a high level Python web framework that encourages clean and rapid development with a pragmatic design. This framework also takes care of much of the hassle of web development, so the focus is placed on building the application rather than reinventing the wheel. Most of the Python code resides in the `openedx/` folder which contains the core logic of the Open edX platform. Two other important modules are the `lms` (learning management system) and `cms` (content management system) which have separate folders under the root level of the project's repository.

18.2.2.4.2 JavaScript The JavaScript code for this project mostly resides in the `common/js` folder, with multiple sub-directories for the various components than handle user interaction within the platform.

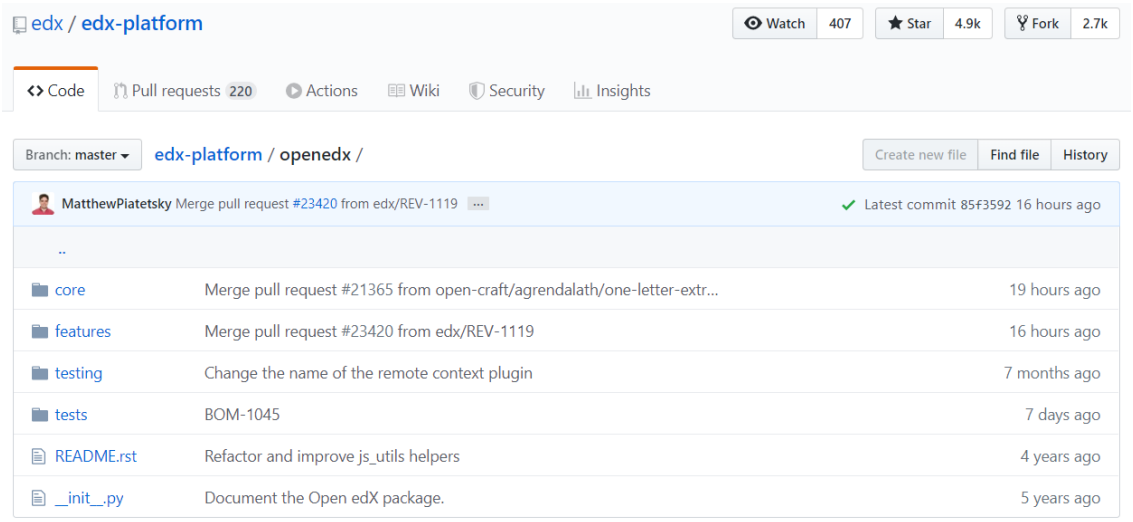


Figure 18.3: Open edX Code

18.2.3 Deployment View

This section provides an overview of the requirements and dependencies needed in order to successfully run the Open edX framework. The requirements for installing the Open edX software (from release Ficus onwards) are detailed on this confluence page. The server requirements are the following:

- **Ubuntu 16.04 amd64** (oraclejdk required). It may seem like other versions of Ubuntu will be fine, but they are not. Only 16.04 is known to work.
- **Minimum 8GB of memory**
- **At least one 2.00GHz CPU or EC2 compute unit**
- **Minimum 25GB of free disk, 50GB recommended for production servers**

A note is also provided for hosting the platform on an Amazon AWS instance. For this, the recommendation is to use a *t2.large* instance with at least a *50Gb* EBS volume for storage.

18.3 Quality and Technical Debt

In the previous essay, we focused on the architecture of the Open edX system. In this essay we will take a look at the quality and technical debt of the Open edX source code. We try to provide examples of coding guidelines and standards that are used by the developers working on the edX project and relate these with the actual code.

We also looked at the overall process of submitting new code to the Open edX platform and the different stages that developers have to go through before having their code accepted and run in the production environment.

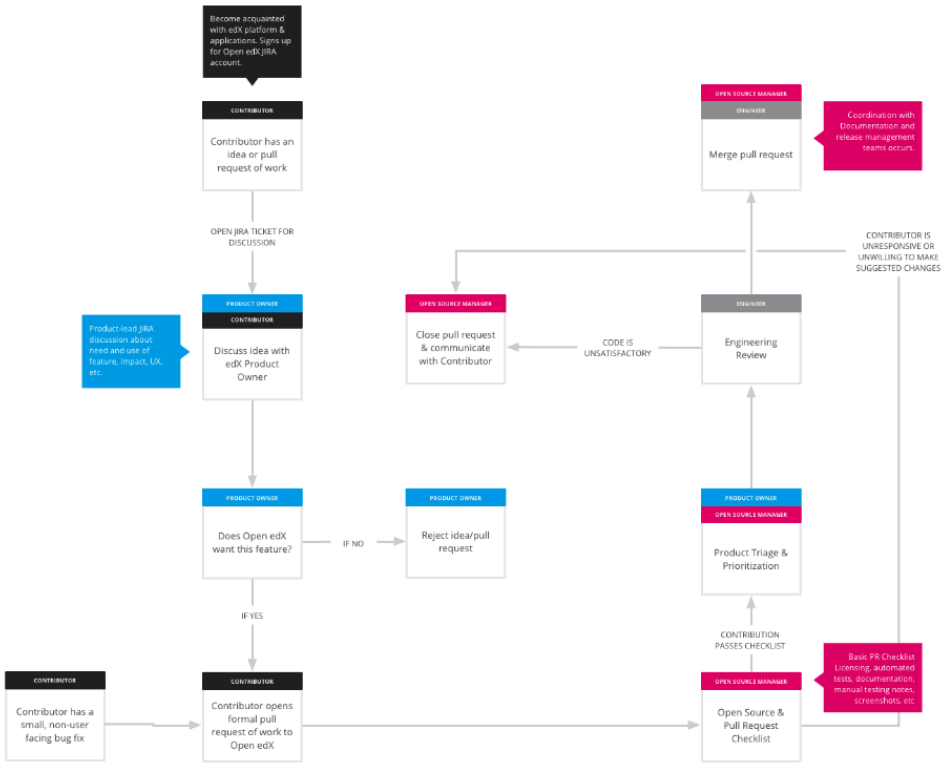
18.3.1 Overview of the Quality Process

The Open edx platform takes several measures to ensure the quality of the contributions made to the software are held to a high standard. An entire developer guide is designated for directing contributors and asserting quality standards for contributions [17].

A process facilitates high quality of contributions and follows a general scheme:

A contributor has to preferably contact Open edX as early as possible during the design cycle of a feature in order to be given guidance and to find out if the feature is already being worked on. Previous contact made and approved will likely have an impact on the pull request acceptance, and time it takes to review.

The figure below shows the intermediary steps in the process of a pull request acceptance.



[18]

There are a number of roles in the code acceptance process: - Core committers: Individuals responsible for accepting a pull request and upholding the quality standards. - Product Owners: Prioritize the work of the core committers, depending on the features needed or requested. - Community managers: Assure healthy

[17] https://edx.readthedocs.io/projects/edx-developer-guide/en/latest/
[18] https://edx.readthedocs.io/projects/edx-developer-guide/en/latest/_images/pr-process.png

development and communication environment. - Contributors: Individual developers wanting to add or improve a feature.

18.3.1.1 The test processes

Once a pull request is made numerous automated tests are run. Testing is something that the Open edX project takes very seriously, a test engineering team is designated to deal with the testing infrastructure. When a new feature is created, two kinds of tests need to be created: general tests that evaluate the feature on the Open edX platform, and tests specific to the new feature. General tests include Django tests as well as acceptance tests, which verify behavior that relies on external systems. Open edX has a Jenkins installation specifically for testing pull requests. Before a pull request can be merged, Jenkins must run all the tests for that pull request: this is known as a "build". If even one test in the build fails, then the entire build is considered a failure. Pull requests cannot be merged until they have a passing build. Code coverage is measured with the use of coverage.py for Python and JSCover for Javascript. The goal is to steadily improve coverage over time, hence a tool was written called diff-cover that will report which lines in a branch are not covered by tests. Using this tool, pull requests have a very high percentage of test coverage – and ideally, test coverage of existing code increases over time. If the code passes the automated tests, it is reviewed based on priority. If the coding standards and functionality are up to par it is accepted by a core committer and added onto the Open edx platform. If it is rejected for any reason, contact is made with the committer, and the reasons are discussed.

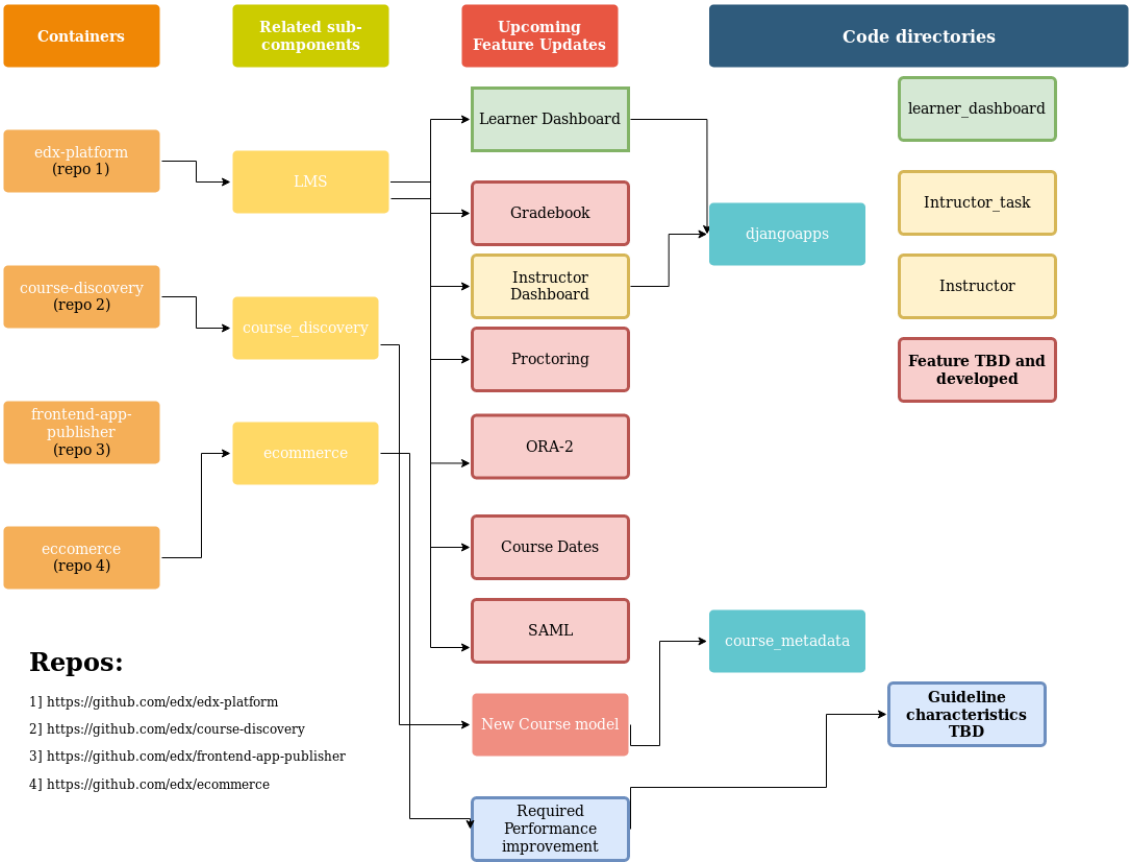
18.3.1.2 Coding hotspots and upcoming features

There are three components that are at the center of attention over the last few months: - the LMS(Learning management system) module - the CMS(Content management system) module - the Common module

These are the three main hotspots for a high coding contribution frequency. Additionally the requirements directory and scripts are updated to support the overall development that occurs on the main components listed above.

The following figure shows the system's roadmap onto the system's components as well as some of the upcoming features that are being developed [19] [20] [21] [22].

[19] https://github.com/edx/edx-platform
[20] https://github.com/edx/course-discovery
[21] https://github.com/edx/frontend-app-publisher
[22] https://github.com/edx/ecommerce



18.3.1.3 SIG assesment

The quality and maintainability of our code were determined by the SIG platform [23] which measures a set of the system's property ratings. These properties are volume, duplication, unit complexity, unit size, unit interfacing, module coupling, component balance and component independence. Open edX scored a pretty low score in duplication (1/ 5), meaning that identical fragments of source code can be found in more than one place in the product. As a result, the unit was also considerably large, thus scoring only 1.6/ 5. It goes without saying that the overall volume of the project was not ideal and only got 3.2/ 5. This negatively impacts the project's analyzability and testability, since the diagnosis of faults or parts to be modified is more difficult or time-consuming. Testability is also involved, since more tests need to be created and maintained for a larger project, increasing the overall effort. The component that achieved the lowest scores in these categories was the Learning Management System (LMS). As it can be observed by taking a look at the roadmap, the architecture team plans to make a lot of changes in this specific model. Hopefully, these changes will reduce the liabilities. Another category where edX failed to score high, was the component entanglement. Component entanglement indicates the percentage of communication between top-level components that are part of commonly recognized architecture anti-patterns. Open edX only scores 1/ 5, and the main reason is the common_lib component. Currently there are no planned feature updates for the specific component. In all the other categories Open edX is above average with the highlight being a 5/ 5

[23] https://sigrid-says.com/softwaremonitor/tudelft-edxplatform/docs/Sigrid_User_Manual_20191224.pdf

score for both module coupling as well as component independence.

18.3.1.4 SIG recommendations

SIG platform offers refactoring suggestions that can improve a project's score in the aforementioned categories. After using this feature we got the following results: 1. Duplication: SIG shows all the parts where code is written more than once. Refactoring candidates are sorted by impact (Lines of duplicated code, times used). As our project achieved a low score for this metric, we can understand that several units are labeled as high-risk. 2. Unit Size: SIG presents for each of the units their lines of code and the risk category for unit size, informing the user which units need to be shorter. As our project achieved a low score for this metric, we can understand that several units are considered high-risk. 3. Unit Complexity: The user is shown the units with the greater McCabe index for the metric. Our project achieved a low score for this metric so we can understand that several units are labeled as high-risk. 4. Unit Interfacing: For this metric the number of parameters is the most critical issue and again we can use SIG to locate which units need an improvement. 5. Module Coupling: The Fan-In is the metric taken into account and for our system only a couple of modules are labeled high-risk. 6. Component Independence: This metric is the one where Open edX achieved the highest score and there are only 16 modules that need to be "isolated". 7. Component Entanglement: SIG suggests that communication lines between specific components should be clearly defined and limited.

18.3.1.5 General coding standards

Moreover, the coding standards that are laid out by Open edx go well beyond merely writing clean code. They are derived from assessing the user potential requirements and enable those requirements. For instance many of Open edx users have some sort of handicap or impairment that require third person software in order to interact with the website. For third person software to function correctly certain requirements have to be directly represented in the code. These requirements can be found in the developer documentation guide: [24]. Further provided coding standards involve support for right-to-left languages and using events and the event API inorder to track analytics. In the pull-request process, a core commiter will review the code to ensure it is up to par. In case of unsatisfactory code contact will be made with the developer. Through the discussions board developers can also be guided to write correct code.

18.3.2 Technical debt

Last but not least, an attempt was made to assess the technical debt [25] in Open edx. There are multiple causes of technical debt, and for that reason Open edx has teams that are devoted to identifying and solving these issues. Causes of technical debt include features that are entangled in one repo but are devoted to be used by several independent components. Such an example is the djangoapps/plugins which Open edx wants to refactor into its own repo. Another technical debt generator is dead code that adds clutter to the software that makes modifications slower. Another technical debt factor that appears in Openedx are different sources of drag, which can include an updated feature causing problems with other features that then need to be sorted out. For this project one the goals our team had was to help reduce the technical debt of the Open edX project. To this end one of our contributions for this project is going to be related to removing deprecated schema models, more specifically the schemas related to the student databse. This will therefore bring value to the overall code quality by reducing the underlying technical debt of the Open edX platform.

[24] https://edx.readthedocs.io/projects/edx-developer-guide/en/latest/conventions/index.html
[25] https://openedx.atlassian.net/wiki/spaces/AC/pages/706183193/Architecture%2BDebt

18.4 Variability Analysis

In the previous essays we focused on analysing the vision of the Open edX project along with some of its architectural patterns and investigated the overall quality of the system. In this essay we will deepen our analysis by looking into the variability modeling, management and implementation mechanisms that are relevant to Open edX. More specifically, we will build a feature model and determine how variability is managed and implemented in the system we analyse.

First we provide some context and explain the underlying principles regarding the notion of variability. If we think about our daily lives and activities, we quickly realise that we encounter variability on a daily basis. An example comes to mind if we think about the numerous cars driving on the road or the vast number of mobile phones being used. This is a perfect example of variability and to be more precise we give a definition for it:

> Software variability is the ability of a software system or artefact to be efficiently extended, changed, customised or configured for use in a particular context.[26]

We now focus on what variability entails in the context of software. When talking about software products, variability can be supported in terms of bundles, command line parameters, plugins, configuration files or even microservices to name a few examples. The problem that quickly arises is how to properly manage and implement variability in a software product. In the following sections we are going to investigate how Open edX does this and to this end we will build two feature models, one for describing the variability of setting up the platform and another one related to variability in terms of configuring the platform.

18.4.1 Feature models

In this section we present the two feature models we built for analysing the variability of the Open edX software. In the first model we look at the variability in the context of installing and setting up the Open edX platform while the second one presents the different services and plugins that can be enabled/disabled in the platform (once this is installed). In the second diagram, one can observe the XBlock features. The XBlocks represent the main point of extensibility for the Open edX platform as we will later discuss in the following sections.

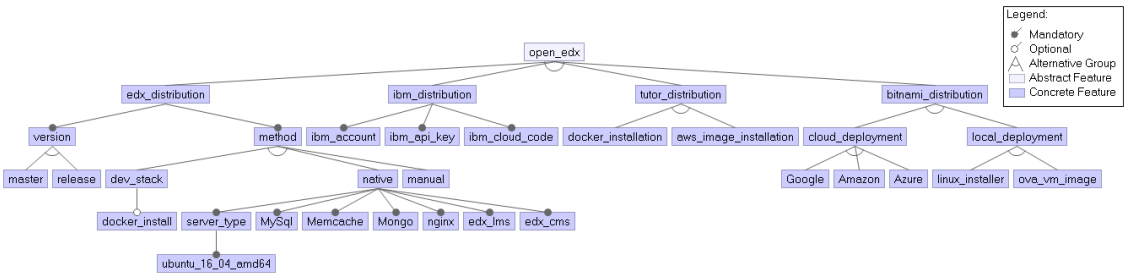


Figure 18.4: Feature model (installation)

[26] Svahnberg, M., et al. (2005) 'A taxonomy of variability realization techniques', Software - Practice and Experience, 35(8), pp. 705–754

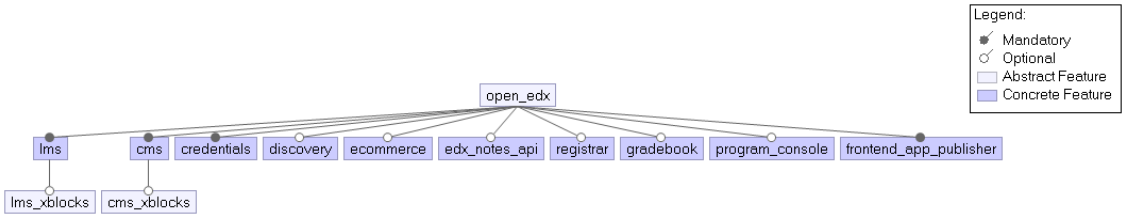


Figure 18.5: Feature model (configuration)

18.4.2 Management of variability mechanisms

The Open edX platform has a number of different distributions each one with its own unique features. As Open edx is used by very different groups of people and for various purposes, the open edX team provides different sources of information for the stakeholders to consult allowing them to choose the right product. An installation should be based on the following questions, how do you wish to use the platform, which version would you like, and which method of installation.

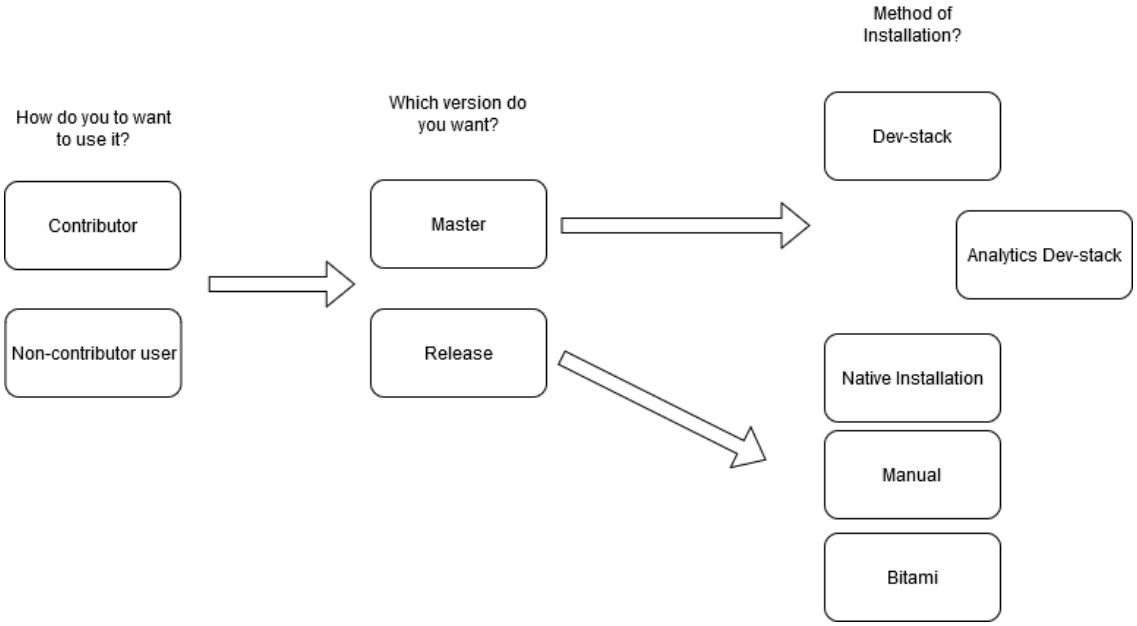


Figure 18.6: Variability Management

Different stakeholders consist also of different types of users and should therefore consider different versions and installation methods of open-edx dependent on their use cases. If the user has no intention of contributing to the software then he should install a release version of open-edx, if the user does intend to contribute then the master should be installed. People with technical knowledge, can get explicit information regarding the different versions and features of the platform from the documentation provided by the Open edX team [27]. A

[27] https://edx.readthedocs.io/projects/edx-installing-configuring-and-running/en/latest/front_matter/index.html

contributor should use the dev-stack installation method which can be found on this repo [28]. If a contributor is also interested in the [different Analytics functionality that is provided] then he could also install the Analytics Destack alongside it. The installations require knowledge of:

- Basic Terminal usage
  - Diagnosing and fixing failures skills
  - Basic knowledge of python web applications regarding build, installation and deployment.
  - Managing a Linux system.
  - Basics of configuration management and automation, using Ansible.
- Software prerequisites
  - make
  - Docker 17.06 CE or later

The dev-stack installation allows for a complete configurable installation, the complete overview can be found here [29]. Potential configurations include site configurations, where urls and api endpoints are set. It is also possible to configure to run multiple sites in an open-edx installation and give each of these sites their own configurations as well. Another configuration is the appearance themes of the site, every front facing component of the open-edx platform has a configurable theme. Created themes need to be referenced from the file system of the component they are coupled with. Tens of other features that can be turned on and off depending on the use cases of the installer.

A non contributor user should install an officially released version of open-edx, a list of the open-edx releases can be found here [30]. These are stable releases that have been tested for wide use. A company or institution that wished to extend a stable release should then still use the dev stack installation. A non contributor user that does not wish to extend open-edx or a user that does not seek to use open edx complete configuration options can install the so called “Native Installation”. This is a simple installation that requires minimal technical skill that can be run on both the cloud or a local linux machine. The instructions for this installation can be found here [31].

There are two other distributions that are worthwhile to mention[], these include the bitnami distribution, which offers an even simpler one click install and can also very easily be run on a cloud (such as Google Cloud , Amazon web services or Azure). Another distribution exists called “tutor”, which is also open source but uses a completely different code base than the open-edx platform. And beside those there are services that offer complete managing of the platform which can be found here [32]

All these different options make it possible for different stakeholders to use the software differently based on different technical resources, and different use cases. A large University or Institution for instance that has the resources could run a more customized and extendable form of the platform while a smaller less resourceful education provider could run a simpler distribution of the platform.

18.4.3 Implementation of variability mechanisms

From the previous sections, we saw that the Open edX system can be set up in various ways, depending on the use case of the end user. We’ve tried to model this variability using feature models. In this section, a

[28] https://github.com/edx/devstack
[29] https://edx.readthedocs.io/projects/edx-installing-configuring-and-running/en/latest/configuration/changing_appearance/theming/index.html
[30] https://edx.readthedocs.io/projects/edx-developer-docs/en/latest/named_releases.html
[31] https://openedx.atlassian.net/wiki/spaces/OpenOPS/pages/146440579/Native+Open+edX+platform+Ubuntu+16.04+64+bit+Installation
[32] https://open.edx.org/get-started/

deeper dive is done into how the variability is managed on the implementation level. For this analysis, 2 particular features were chosen, namely: Credentials and XBlocks.

18.4.3.1 Credentials

In Open edX, credentials are stored in a MySQL database. 3rd party authentication is an option, but is disabled by default. For 3rd party authentication, the following options are available [33]:

- OAuth
- SAML
- LTI

Limited support for additional providers is available, but these "tend to be older and less robustly tested, and have a much more limited feature set" [34].

Enabling 3rd party auth is done by setting a feature flag in a local LMS json file. When this is done similar alterations to this json file are necessary to enable a specific identity provider. Additionally, API keys and secrets need to be fetched from the identity provider themselve [35]. After this configuration is done, the LMS server needs to be restarted. This means that this variability is binded during load-time.

```
 from django.conf import settings

    return configuration_helpers.get_value(
        "ENABLE_THIRD_PARTY_AUTH",
        settings.FEATURES.get("ENABLE_THIRD_PARTY_AUTH")
    )
```

common/djangoapps/third_party_auth/__init__.py

The implementation mechanism used for various 3rd party auth providers is the Template Method design pattern [36]. If your particular identity provider backend is not supported, implementing it is as simple as extending a base class and overriding a few methods. Open edX uses the python-social-auth [37] package for most of its 3rd party authentication handling. Although some slight alterations are made to some backends, for example, the SAML backend [38] has been customized somewhat to fit within the architecture.

```
from social_core.backends.oauth import BaseOAuth2

class GitHubOAuth2(BaseOAuth2):
    """GitHub OAuth authentication backend"""
    name = 'github'
    AUTHORIZATION_URL = 'https://github.com/login/oauth/authorize'
    ACCESS_TOKEN_URL = 'https://github.com/login/oauth/access_token'
    ACCESS_TOKEN_METHOD = 'POST'
```

[33] https://edx.readthedocs.io/projects/edx-installing-configuring-and-running/en/latest/configuration/tpa/tpa_providers.html
[34] https://edx.readthedocs.io/projects/edx-installing-configuring-and-running/en/latest/configuration/tpa/tpa_providers.html
[35] https://edx.readthedocs.io/projects/edx-installing-configuring-and-running/en/latest/configuration/tpa/tpa_integrate_open/tpa_oauth.html#add-the-provider-configuration
[36] https://sourcemaking.com/design_patterns/template_method
[37] https://python-social-auth.readthedocs.io/en/latest/
[38] https://github.com/edx/edx-platform/blob/a33b5e441cd6636b4228d2eba4173d0592a4d771/common/djangoapps/third_party_auth/saml.py

```
    SCOPE_SEPARATOR = ','
    EXTRA_DATA = [
        ('id', 'id'),
        ('expires', 'expires')
    ]

    def get_user_details(self, response):
        """Return user details from GitHub account"""
        return {'username': response.get('login'),
                'email': response.get('email') or '',
                'first_name': response.get('name')}

    def user_data(self, access_token, *args, **kwargs):
        """Loads user data from service"""
        url = 'https://api.github.com/user?' + urlencode({
            'access_token': access_token
        })
        return self.get_json(url)
```

An example of how the Template Method design pattern is implemented in python-social-auth.

18.4.3.2 XBlocks

XBlocks are the extensibility platform for the Open edX system. Adding functionality like in-video quizzes to courses is done by creating a XBlock. The binding of XBlocks is twofold, namely: load-time and run-time. As such, 2 implementation mechanisms are used.

Installing an XBlock involves installing the package itself into 2 containers. One for the LMS and CMS respectively, this is done using the python package manager pip [39]. After this is done, both the LMS and CMS need to be restarted. So the variability in installed XBlocks is binded at load-time. The mechanism used for this is composition. For collections of XBlocks, so called "library XBlocks" are created which can store any amount of XBlocks. These libraries can then be used to give individual courses the access to specific collections of XBlocks.

```
@python_2_unicode_compatible
class LibraryRoot(XBlock):
    """
    The LibraryRoot is the root XBlock of a content library. All other  blocks in
    the library are its children. It contains metadata such as the library's
    display_name.
    """
```

Library XBlocks

During run-time, however, one can choose what XBlocks are available on their platform. So it is possible to install a bunch of XBlocks, and then decide at runtime the availability of them. This is done per course using the frontend of the CMS (often called Studio) [40].

[39] https://edx.readthedocs.io/projects/xblock-tutorial/en/latest/edx_platform/devstack.html#prerequisites

[40] https://edx.readthedocs.io/projects/open-edx-building-and-running-a-course/en/latest/exercises_tools/enable_exercises_tools.html#enable-additional-exercises-and-tools

18.4.3.3 Future proof?

We've seen the ways in which Open edX handles variability for 2 specific features on the implementation level. When handling variability on the implementation level, one has to take in account not only the current context regarding the variability, but also the future context. How is the variability expected to change. Is the implementation mechanism used future proof? If these considerations are taken lightly, developers could introduce significant technical debt into the system.

The 3rd party authentication is binded at load-time, which means that whenever a new identity provider needs to be added, the server needs to restart. This results in downtime, which can be significant for large educational institutions.

For XBlocks, the management and scalability is actually quite good, as XBlocks are independent from each other. The only real downside comes with the fact that XBlocks can't be installed at runtime. This can be partly remedied by installing a bunch of XBlocks beforehand, and letting individual course designers choose for themselves which functionality they want to include.

18.4.4 Conclusions

In this essay, we've tried to analyse the variability in the Open edX system and how it is managed on different levels. We found that there is a lot of variability possible, which is also necessary, as Open edX is a system that is used by universities world wide. Most of the variability is captured in partnered 3rd party distributions, which is good, as it makes it easier for universities to get a particular "flavor" of the base Open edX release, without too much technical hassle. Besides this, variability exists in the base distribution as well. The various ways in which these variabilities are managed were hard to ascertain, and did not seem very uniform. Indicating that there is no universal framework for incorporating variability in place. This could be improved by getting developers on the same page on how to handle variability within the project. Recommendations include:

- Listing common types of variability within the project and what design patterns could be useful to tackle them.
- Giving developers the tools (e.g. FeatureIDE) to create feature models for their own components.

In all, the Open edX platform is a great tool for universities worldwide in creating engaging and tailored educational experiences. Variability is a key component to this, and we hope that Open edX realizes this as well.

Chapter 19

openpilot

openpilot is an open-source driving system developed by comma.ai. As the slogan *'make driving chill'* reflects, the system aims at achieving partial automation (level 2), letting the software control both steering and acceleration, while the human is still responsible for environment monitoring and fallback performance. With the corresponding plug-and-play hardware, consisting of wiring to communicate with the bus of the car and a dashboard unit used for road monitoring, the performance of any of the 63 supported cars should be significantly enhanced and improved compared to factory functionality, for under $1000.

Comma.ai is founded by George Hotz (aka geohot), known for sim-unlocking the first generation iPhone, developing various iOS jailbreaks and hacking the PlayStation 3 to side-load pirated games. In a recent presentation, Hotz states the ambition of openpilot to become the Android of car automation, comparing the Tesla performance to Apple. Comma.ai collects the driving data on a voluntary basis to constantly adjust the model and improve performance. So far, over 22 million kilometres of driving data has been collected since openpilot was made publicly available in 2015. As of January 2020, openpilot has 1500 monthly users. This number is expected to grow since the company recently made its first appearance on the 2020 edition of the Consumer Electronics Show (CES).

The system uses a pre-trained non-deterministic deep neural network to locally make driving decisions, written in Python 3. The Panda subsystem used to communicate with the car hardware is written in C and comes with certain timing guarantees. The system complies with ISO26262 road vehicles safety functionality and furthermore never overrules the factory default safety features.

19.1 Product Vision: Make Driving Chill

19.1.1 Problem Definition

19.1.1.1 Bob's Traveling Problem

The advertising slogan of openpilot is 'make driving chill', indicating that apparently driving is currently not so chill. Imagine Bob who's living in the US, he visits his family on a regular basis and therefore travels between his residence in Austin TX and his family in Houston TX. Public transport is rarely available along this 266 km route, and while taking a flight is the fastest way of traveling, Bob prefers saving both his wallet and the environment by utilizing his car.

He owns a lava-red Toyota Rav4 2016, which was a serious investment for him at the time. Apart from the tough appearance and proven Japanese quality, the convenient lane departure warning (LDW) and adaptive cruise control (ACC) features were decisive in his buying decision. While these features indeed eased his driving experience, the 2,5 hour drive is still a monotone and tiresome task. To better understand Bob's situation, consider his customer empathy map:



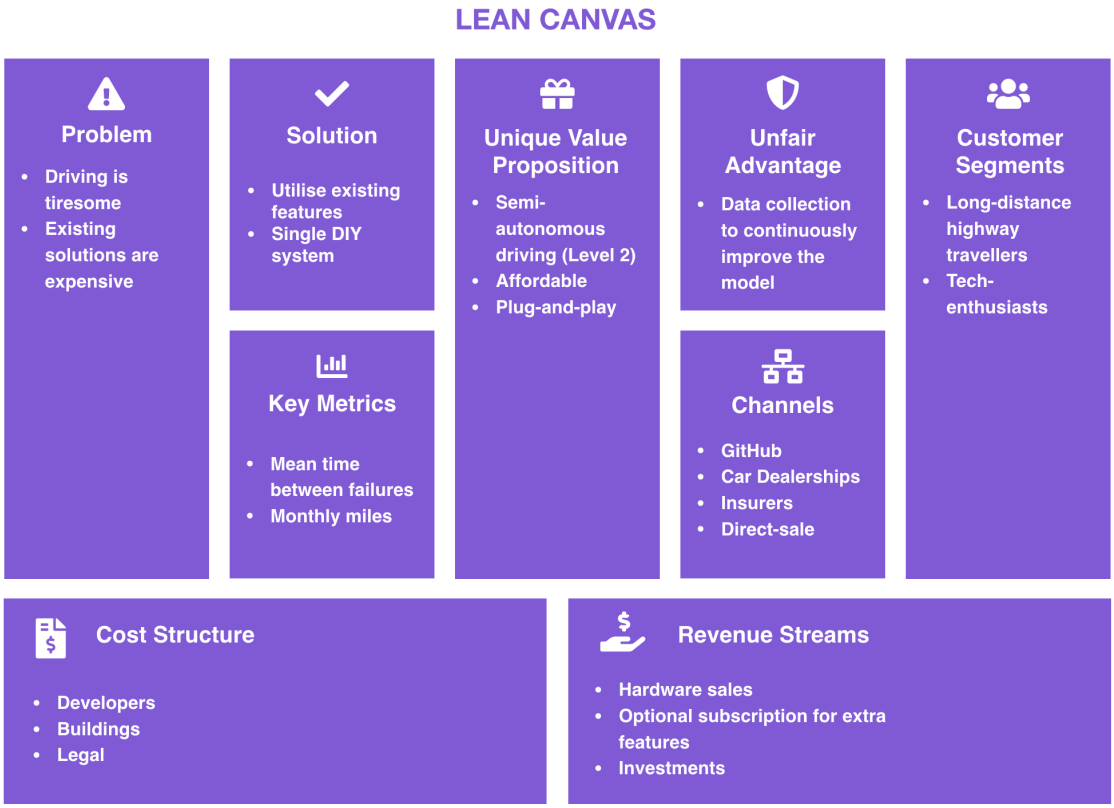
19.1.1.2 George's $1000 DIY Solution

George Hotz, known for sim-unlocking the first generation iPhone and jailbreaking the Playstation 3,

observed Bob's problem in 2015 and founded Comma.ai. He considered the current status of the self-driving car industry:

- Tesla has decent driving assistance, however they are not affordable.
- Existing cars offer LDW and ACC only in isolation.
- Fully autonomous solutions like Waymo require ultra-high definition maps of the environment and multiple expensive lidar sensors.

Considering that fully autonomous car-to-car communication solutions are likely not replacing humans in the near future he proposed a DIY solution. This solution is a $1000 device mounted on the car dashboard, combining the existing features and a camera to take control. The business model consists of selling the hardware, keeping the software open-source. The Lean Canvas provides an overview of openpilot in context.



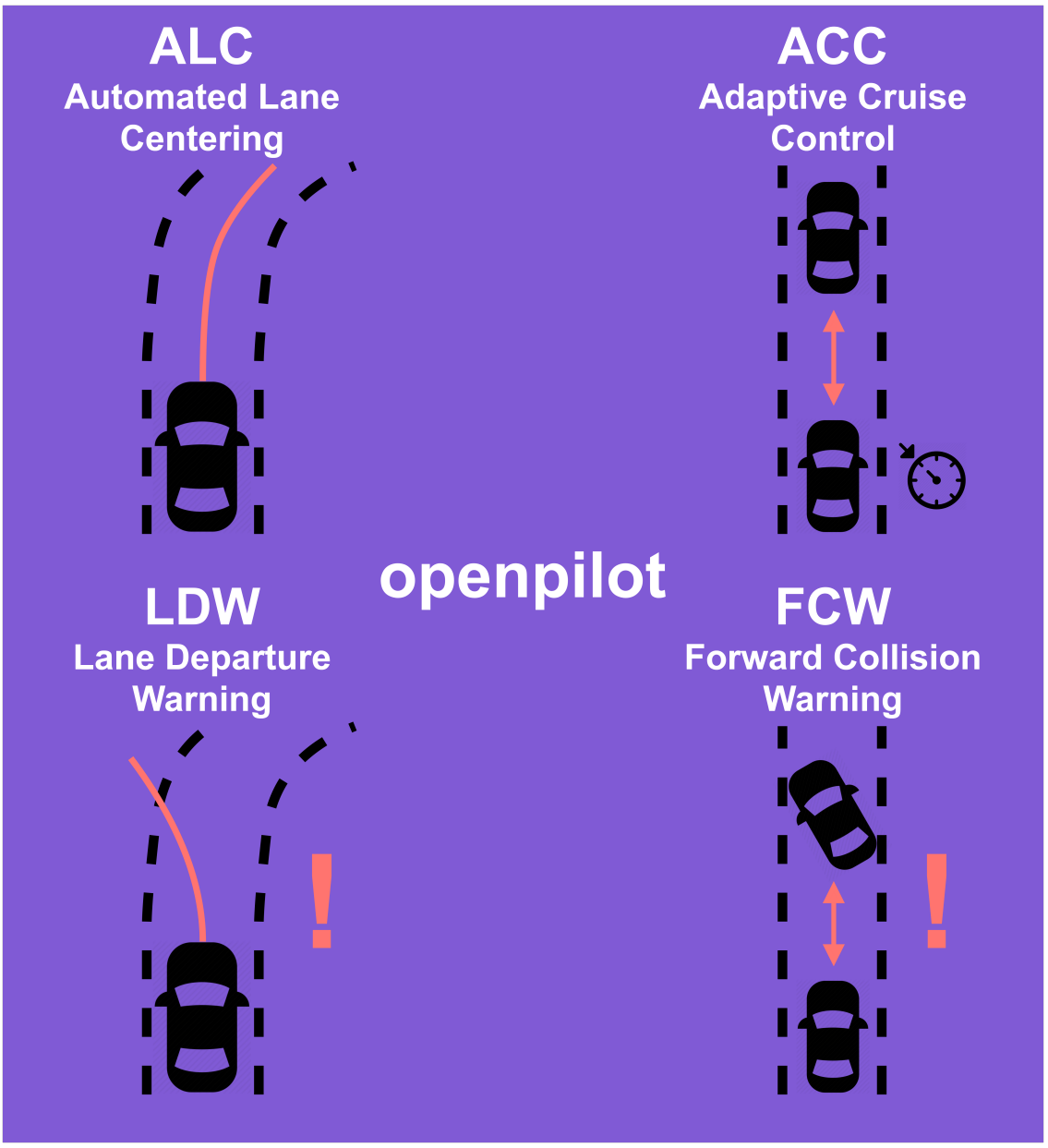
19.1.2 Making Driving Chill: The Requirements

Lean Architecture divides a system into two parts: What the system *is* (the platform) and what the system *does* (the end-user's view of the system). Here, we will focus on the latter.

In the self-driving car world, there exists 6 levels of automation: 1. no automation 2. driver assistance 3. partial automation, 4. conditional automation 5. high automation 6. full automation

Levels 0 to 2 require full drivers attention. From level 3 onward the driver is able to take his eyes off the road.

At the time of writing, openpilot integrates four essential capabilities: ALC, ACC, LDC and FCW. This makes openpilot a level 2 autonomous system (the same level as Tesla's AutoPilot), which means that it requires the driver to pay full attention, at all times.



As can be seen in the Lean Canvas, one of the key metrics for validating the openpilot system is mean time between failure (MTBF). Obtaining this metric is an intrinsic requirement for Comma.ai. Comma.ai's solution to obtaining these statistics is also one of the fundamental capabilities of the openpilot system: crowdsourcing.

Although crowdsourcing does not affect the end-user directly, it is one of the pillars of the openpilot system for it allows Comma.ai to generate useful statistics and improve their machine learning training data. Each openpilot system therefore periodically uploads anonymized driver data to Comma.ai's servers (although, since it is an open-source system, this can be disabled).

19.1.3 Who's Along The Road?

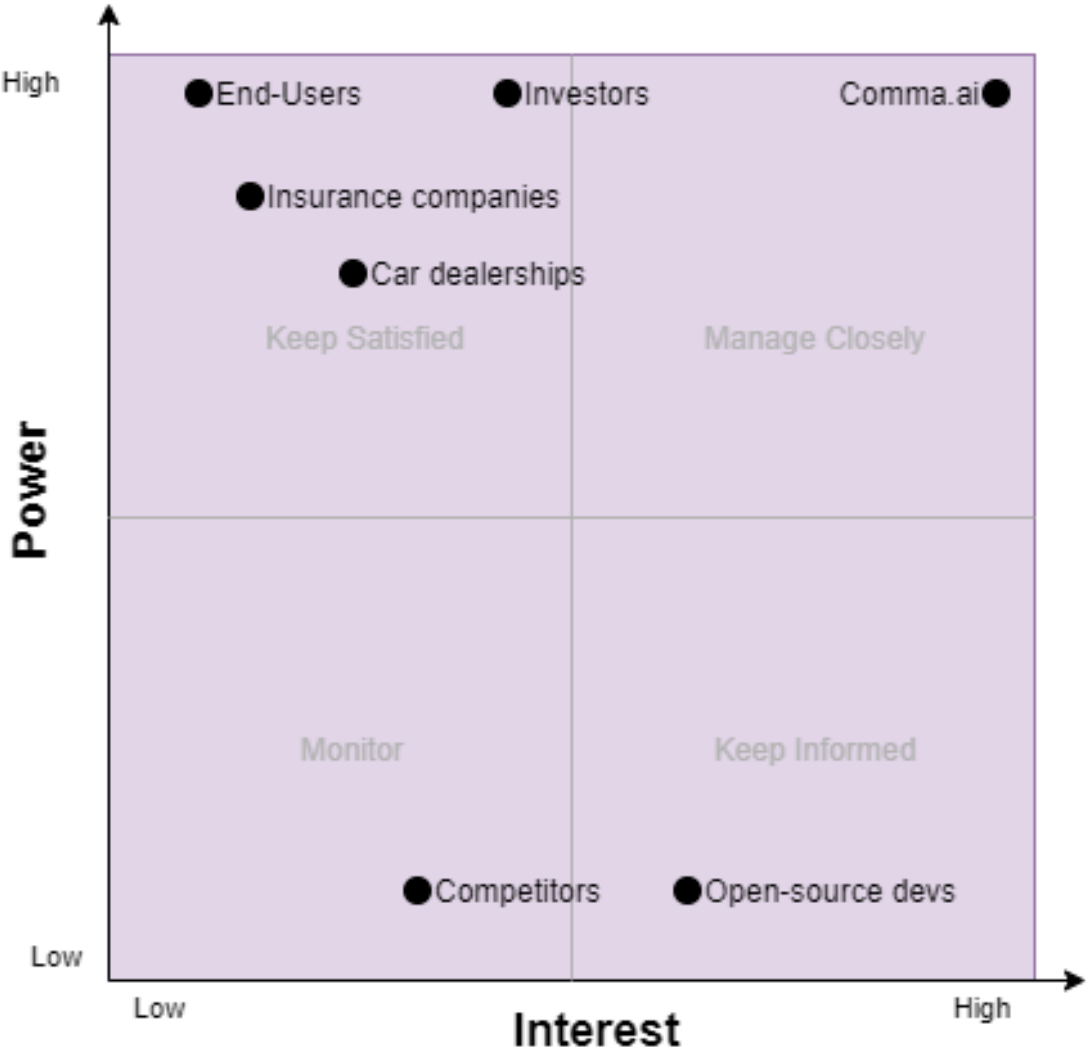
The most important pawn when looking at openpilot is of course the company behind it, Comma.ai. As the main driver behind the product, Comma.ai has a lot of power, as they make the decisions on which car to make compatible, or which feature to integrate into the system next. As the product started out as an idea of the eventual founder of the company, you can imagine how passionate about, and thus interested in, improving the current state of autonomous driving they are.

This brings us back to Bob, or more generally speaking, the end-users of openpilot. Openpilot provides them with a cheap alternative to for example buying a Tesla, in order to create a better and more chill autonomous driving experience. Therefore end-users hold a lot of power, if they do not buy the product, the company will eventually go bankrupt. These end-users are not the only ones wanting to buy the product, car dealerships have also shown some interest in buying and reselling it. They would for example be able to buy the product in bunches of $750 apiece and resell them for $1200 to costumers. Due to the software being open-source, there has also been shown interest from the open-source community to work on the software. This helps them improve as a developer and might gain them some recognition in return.

The product has proven to have a better autonomous driving experience through the integration of LDW and ACC that are provided with the car, therefore making the car safer. This will not only benefit the users, but also the insurance companies, because this will lead to less accidents and thus less insurance to pay out, giving these insurance companies a lot of power. However there has not been a lot of interest from insurance companies, as it is not yet economically viable from their perspective.

Due to the growth of the product, Comma.ai has gotten interest from investors, leading to two investments so far. By buying shares of the company, the investors are able to make profit of the investment when the company's worth grows. In the case of the company growing and an increasing amount of people using openpilot, it can have both a negative effect, reducing sales, as a positive effect, more potential customers due to the increasing market, on competing companies like Tesla and Waymo. Thus these competing companies have an interest in keeping up with the product.

To get an overview of all the stakeholders, consider the Power-Interest matrix below:



19.1.4 A Thing On Journey Versus Destination

At comma.ai, the journey and eventual destination show a high degree of interrelatedness. Contrary to popular belief, the journey matters as much as the destination. As an analogy, take the journey as the current context and the destination as the future context. Using a SOAR analysis, we can differentiate between the internal and external aspects of the journey (Strengths and Opportunities), and between the internal and external aspects of the destination (Aspiration and Results).



19.1.4.1 The Journey

In a business landscape dominated by huge VC investments and marketing campaigns, Comma.ai is not yet taken seriously. This is because their financial numbers and business size do not express reasons to do so. However, progression at Comma.ai is measured in terms of quality. With a very affordable product and increasing functionalities such as real time connection with the network and a driving history, the goals of comma.ai do not include obtianing a billion dollars investment. Instead they care about demonstrating safety and offering this globally. The fact that since it is a profiable company since january 2020, helps them to continue their journey.

19.1.4.2 The Destination

When you have built a system that is too good to deny, business is good. This however is a consequence, the cause is ground-breaking technology. And that technology can only become that good if users teach it how to be good. When public confidence in autonomous driving has reached satisfiable levels, the destination is close. When the MTBF of autonomous driving outperforms that of human driving, driving is chill.

Note: Diagrams created by us. Icons used originate from FontAwesome.

19.2 From Vision To Architecture: How to use openpilot and live

Autonomous driving is a very simple task in theory, but still remains unsolved at large. How does openpilot tackle the problems that arise on the road?

Before we explore openpilot's inner workings, let's do a little thought experiment.

19.2.1 Our Very Own openpilot

Imagine we were designing very simple software for an autonomous vehicle. In order for it to react to the environment (other cars, a lane split), we need to be able to read sensor data, act on that data and update the actuators, such as the steering wheel or throttle.

From this description, we can derive a very simple abstract implementation of the core system functionality:

```
while (true) {
  read sensor data
  compute adjustments using machine-learning model
  apply adjustments to actuators
}
```

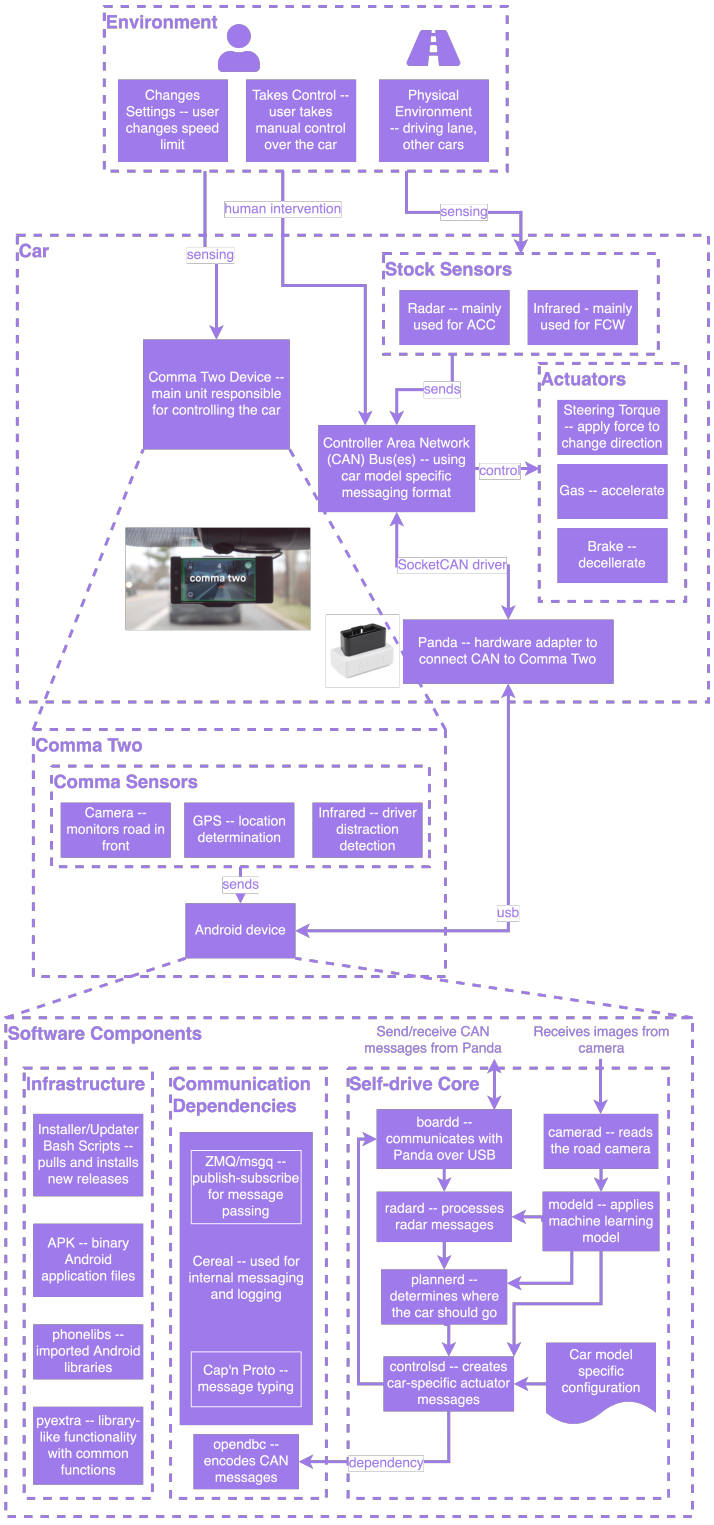
While this may seem overly simplified, in essence this is how openpilot works internally. In reality, openpilot uses over 300 Python files divided into various submodules, dependencies and multiple hardware components to run this system.

This post aims to outline the main architectural components and processes along the path from sensor input to actuator output, their responsibilities, the design patterns applied within or between components and the trade-offs that were encountered in this architecture.

19.2.2 The Architecture

Now that we have a basic idea about how an autonomous system works, let's see how openpilot actually does it.

We identified the main components of openpilot by analysing and running the source code from the GitHub repository and created a neat and tidy diagram from that analysis. Don't worry if this diagram seems overwhelming, in the following sections we will try our best to explain what each component means and does. You, the reader, are encouraged to go back and forth between the text and this diagram to fully grasp openpilot's inner workings.


Environment
Changes Settings -- user changes speed limit
Takes Control -- user takes manual control over the car
Physical Environment -- driving lane, other cars
human intervention
sensing
Car
sensing
Stock Sensors
Radar -- mainly used for ACC
Infrared - mainly used for FCW
Comma Two Device -- main unit responsible for controlling the car
sends
Actuators
Steering Torque -- apply force to change direction
Controller Area Network (CAN) Bus(es) -- using car model specific messaging format
control
Gas -- accelerate
Brake -- decellerate
SocketCAN driver
Panda -- hardware adapter to connect CAN to Comma Two
Comma Two
Comma Sensors
Camera -- monitors road in front
GPS -- location determination
Infrared -- driver distraction detection
sends
usb
Android device
Software Components
Send/receive CAN messages from Panda
Receives images from camera
Infrastructure
Installer/Updater Bash Scripts -- pulls and installs new releases
APK -- binary Android application files
phonelibs -- imported Android libraries
pyextra -- library-like functionality with common functions
Communication Dependencies
ZMQ/msgq -- publish-subscribe for message passing
Cereal -- used for internal messaging and logging
Cap'n Proto -- message typing
opendbc -- encodes CAN messages
Self-drive Core
boardd -- communicates with Panda over USB
camerad -- reads the road camera
radard -- processes radar messages
modeld -- applies machine learning model
plannerd -- determines where the car should go
controlsd -- creates car-specific actuator messages
Car model specific configuration
dependency

The architecture diagram is a combination of run-time and development view, since they are related and best explained together in this system. The logical view is not discussed because the system performs mostly a single task (control your car) and there is not much end-user functionality. The process/deployment view is also not discussed. This is because some parts of comma.ai's deployment procedures and cloud infrastructure are not open source resulting in an incomplete analysis.

Multiple sensors observe the environment in and around the car. The stock sensors included by the manufacturer are a radar and an infrared sensor. The Comma Two delivers a front-facing camera, GPS, and an infrared driver distraction camera. These inputs are unified to deliver the desired functionality.

The diagram conveys another important architectural decision that the openpilot developers made: openpilot is not a single monolithic process but rather a coordinated cluster of processes. Being able to prioritize some processes over others gives openpilot the ability to adhere to the strict timing requirements necessary for autonomous driving.

We're now going to untangle said processes in the bottom part of the diagram by analysing two scenarios:

1. A car is braking in front of us
2. The road is curving

In the following text, we will investigate these scenarios step by step, explaining what the software components are, their responsibilities, and how they communicate.

19.2.2.1 Scenario 1: Lead car brakes/slows down

Now consider a car in front of you that suddenly brakes. Your ACC-equipped car will notice this and its radar will send a message to the CAN bus. The CAN bus is continuously read by a hardware device of comma.ai's making, called the *panda*.

The panda forwards this information to openpilot's comma two. This Android device runs several background processes (~20 in total), that all communicate with each other and coordinate their actions.

All messages to and from the panda are managed by the *boardd* background process. boardd publishes these messages for other processes to use.

radard is listening closely to the messages boardd publishes, and will notice a message from the car's radar system. radard combines the radar CAN message published by boardd with data published by *modeld* (see the next section) and publishes a `radarState` message, which contains information such as the lead car distance.

This `radarState` message is then picked up by *plannerd*. plannerd's responsibility includes, as the name already conveys, planning the path the car needs to take. It publishes this path, which can then be consumed by other processes.

The final process involved in this scenario is *controlsd*. Remember our little pseudo-code loop at the beginning of this post? That while loop is implemented almost verbatim in controlsd. controlsd converts the path published by plannerd into actual CAN messages, and does so in a car model-agnostic way. This is an important detail, because openpilot aims to support as many cars as possible, and having controlsd use interfaces instead of concrete implementations facilitates this vision. Adding support for new car models mainly deals with implementing new concrete implementations of the interfaces that controlsd uses.

Finally, controlsd sends the CAN messages back to boardd, which then writes them to the car's CAN bus via the panda.

19.2.2.2 Scenario 2: Direction changes/road curve

Even highways have curves. How does openpilot sense line markers and act upon curves? Let's investigate.

The starting point for our analysis this time is not boardd, but *camerad*. camerad, on the surface, is a very simple process with the only responsibility of publishing video frames from the back- and front facing cameras. These frames can then be consumed by other processes.

One process that is particularly interested in these frames is *modeld*. modeld transforms the camera frames using its machine learning model and publishes the result.

From this point on, the same processes are involved as in scenario 1. Namely, plannerd watches modeld's output and incorporates it into its path planning. plannerd emits a steering wheel torque change encoded as a CAN message, which is then picked up by boardd and sent to the car's CAN bus.

19.2.2.3 Infrastructure Components

In the above two scenarios we visited arguably openpilot's most important processes. They are, however, not the only components within openpilot's ecosystem.

Other components not directly related to core functionality aren't that interesting, so we won't spend many words on them:

- A collection of bash scripts is used to remotely pull and install new releases.
- Additional APK files to be installed by the user are located in the APK folder.
- Phonelibs contains libraries used on the Comma Two for communication with Android.
- Pyextra contains library-like functionality used by multiple components, including explicit exception messages.

19.2.3 Putting it together with code

Given the architectural outline above, we can look into the process of implementing it. So, put yourself in the shoes of a software architect for an autonomous driving system and think about the requirements that the system must adhere to:

- Atomic (Chapter 6.3) when it comes to signal handling. Because we deal with data that eventually physically moves the car, ambiguous instructions are intolerable.
- Compatible and extensible. Since the vision of comma.ai includes offering support for all popular cars, the code should also be designed for this.
- Universal. When your product is reliant on the open-source community, it helps when common ideas from software design are applied.
- Modular (Chapter 3.5). When changing or improving certain parts in your system, you do not want to touch the whole codebase. Additionally, modularity allows the separate modules to be tested independently and allows the use of multiple programming languages or protocols.

Openpilot possesses these requirements and provides us with nice examples of implemented design patterns, the most noteworthy to shed additional light on is publish and subscribe. This pattern plays a fundamental role in the autonomous driving system and enforces all things listed in the bullet list above. Briefly said, publish and subscribe is a system to orchestrate message passing between different software systems or components. The publishers only send, while the subscriber solely receives messages from the publishers that it is subscribed to.

Pub-Sub is particularly useful when an application asynchronously communicates with other applications that are not necessarily implemented in the same language and executed on the same platform or system. Moreover, within each component it allows for cherry-picking from the available publishers. This ensures that critical information only resides in places where it is of vital importance and introduces extensibility.

19.2.4 Extensibility in general

Next to the publish and subscribe pattern, openpilot has embraced other best practices from software engineering to allow for compatibility and flexibility. Examples include the use of base classes and interfaces. We can relate these phenomena to the product vision that aims to provide a distributed decentralized self driving car platform that is built and maintained by a community. Zooming out a bit and looking from a deployment perspective; cereal, opendbc and panda are becoming standalone projects in 2020. Together with the newly proposed development flow this should make it very interesting for people from the open-source community to contribute and create a system in which quality and community are keywords.

19.2.5 Trade-offs

We have seen some of the functional properties of the system, now we will take a look at some non-functional properties and their trade-offs.

19.2.5.1 Performance vs Privacy

George Hotz mentioned in a talk he gave in June 2019 that he believes that "The definition of driving is what people do when they drive", implying that a driving assistant should be based on data provided by drivers. As mentioned and seen in the architecture diagram, one of the key parts of the system is the driving model, modeld. Because the driving model plays a vital part in the system it needs to be accurate and consistent, in order to prevent potential accidents from happening. The data needed for the model comes directly from drivers that use the openpilot system in their car, which brings up the question of data privacy. Some people might not want their car data to go to comma.ai. However at comma.ai they are very clear about data privacy, if you make use of their system, you give them the consent to use all the data generated while driving, which was also said by George Hotz in the aforementioned talk. Thus, they made a trade-off between model performance and data privacy in which they value the performance and accuracy over the privacy of the drivers.

19.2.5.2 Compatibility vs Maintainability

As mentioned near the end of scenario 1, openpilot aims to support as many cars as possible. Therefore building the system in such a way that it will be compatible with all these different cars and models is very important. As seen in the architecture diagram, and mentioned in scenario 1, openpilot deals with all these different cars and models by creating a new implementation of the interface used by controlsd. However, by creating all these files it also means that they need to be maintained separately. With the ever-growing number of openpilot supported cars, this means that eventually they need to maintain hundreds if not thousands of these interface implementations. Thus meaning that the company chooses to make a trade-off between compatibility and maintainability, by creating a system that has high compatibility, but in return loses some maintainability.

19.2.6 Bringing it all together

What started out as a simple while-loop ended up in a journey across processes, modules and design principles. We saw how pub-sub creates a modular, extensible design. We visited the trade-offs that developers made over the years, and their implications.

openpilot is a uniquely complex project, that aims to tackle a uniquely complex problem.

19.3 When an autonomous car is in the garage

Addressing quality from a software point of view deserves some caution in openpilot's case. Image yourself as an openpilot software engineer and picture a variety of cars driving around seamlessly and without active human efforts because the code you wrote is good. It sure is an awesome feeling. Now imagine a less cheerfull scene with a car collision, beware that this is not your average lightweight javascript view library, peoples lives depend on it. As the company rightly pointed out, quality is represented by the stability of the cars that are driven by their software (and partially hardware). Therefore assessing quality deserves an unusual integral approach in this non-deterministic context that the code runs in. Think about all the slightly different traffic lights at every corner of the street, think about quaint driving lanes with potholes, think about ludacris other drivers, no situation is the same. An autonomous car is not an ordinary car and a comma two is not an oridnary android device.

19.3.1 The quality mark of openpilot's parts

Considering guidelines of maintaiable, future proof code in software engineering, openpilot passes the first inspection. As was highlighted in our previous post, concerns are separated in modules that are loosely coupled through the use of the publish and subscribe pattern. Additionally, it seems like the company is taking small codebases to the next level by advocating externalization. Together with the automated tests (see below) this demonstrates great awareness of building maintainable software. For the remainder of this post, we will look at how these abstract principles are made concrete by comma.ai.

19.3.2 openpilot's CI process

Perhaps the most efficient way to get programmers to write working code is to enforce them in the CI (Continuous Integration) stage, by rejecting any commits that sport the dreaded "Build Failed". openpilot is no different. A variety of unit and integration tests are run against every single commit to ensure no regressions slip through the cracks. The whole test suite takes about 30 minutes to run.

19.3.2.1 Unit Tests

As we learned in our previous post, openpilot is not a monolithic application, but instead consists of many cooperating processes that communicate using a message queue. It seems only logical that the unit tests follow the same pattern. Out of the 20-something total processes that run within openpilot, only a couple have unit tests. Upon investigating specific unit tests, it seems that some tests require a deep understanding of autonomous driving and the mathematics behind it. Many tests contain variable names that are abbreviations of domain-specific jargon.

In the agile world, unit tests are part of the documentation. However, the scarcity of unit tests, cryptic function or variable names, and their complexity makes them impossible to use as documentation.

Most of you will be familiar with the AAA pattern when writing unit tests: Arrange, Act, Assert (although sometimes Given, When, Then is used). Consistently applying this pattern will go a long way in making unit tests small, comprehensible and fast. In openpilot, however, assertions are often mixed with actions. This makes it especially hard to see what is *actually* being tested.

We would even argue that most "unit tests" in openpilot are not actually *unit* tests, and that they are instead much closer to functional tests. They take a long time to run (unit tests should be *fast*), they make many different assertions in one test (assertions and actions should not be intertwined), but above all there should be *more* unit tests.

19.3.2.2 Maneuver Tests? (Functional Tests)

Openpilot, being a autonomous driving system, obviously needs robust functional testing to ensure all of its components work together to create the desired output.

Another step in openpilot's CI pipeline is executing maneuver tests. That is, certain pre-determined maneuvers (scenarios) are presented to openpilot, which are then acted upon by the system. Pass or fail is determined by running a list of checks for each maneuver.

This sounds complex, so let's add an example.

> "fcw: traveling at 20 m/s following a lead that decels from 20m/s to 0 at 5m/s2"

(source)

Translation: check that openpilot gives a Forward Collision Warning when following a lead car that decelerates from 20 m/s to 0 m/s at 5 m/s².

openpilot runs ~25 of these maneuvers. Each takes about 10 seconds to simulate, so that adds up to around 4-5 minutes.

19.3.2.3 Regression Tests (Replays)

Next up in openpilot's CI pipeline: preventing accidental changes in process output. By comparing the output of processes against the output of a reference commit, unintended changes to the output of openpilot can be noticed and mitigated. These tests live in selfdrive/test/process_replay/, along with the reference output. Whenever this test fails, the author can inspect the output and update the reference output if deemed correct.

19.3.2.4 Coverage

There is just one piece missing in the CI pipeline: coverage. Test coverage is definitely a very important tool to find untested parts of your codebase (do, however, keep in mind that test coverage says absolutely *nothing* about the quality of tests). Currently, coverage is not part of openpilot's CI process. It seems that, since openpilot relies mainly on functional tests, the developers deemed it unnecessary to add as part of the development life cycle.

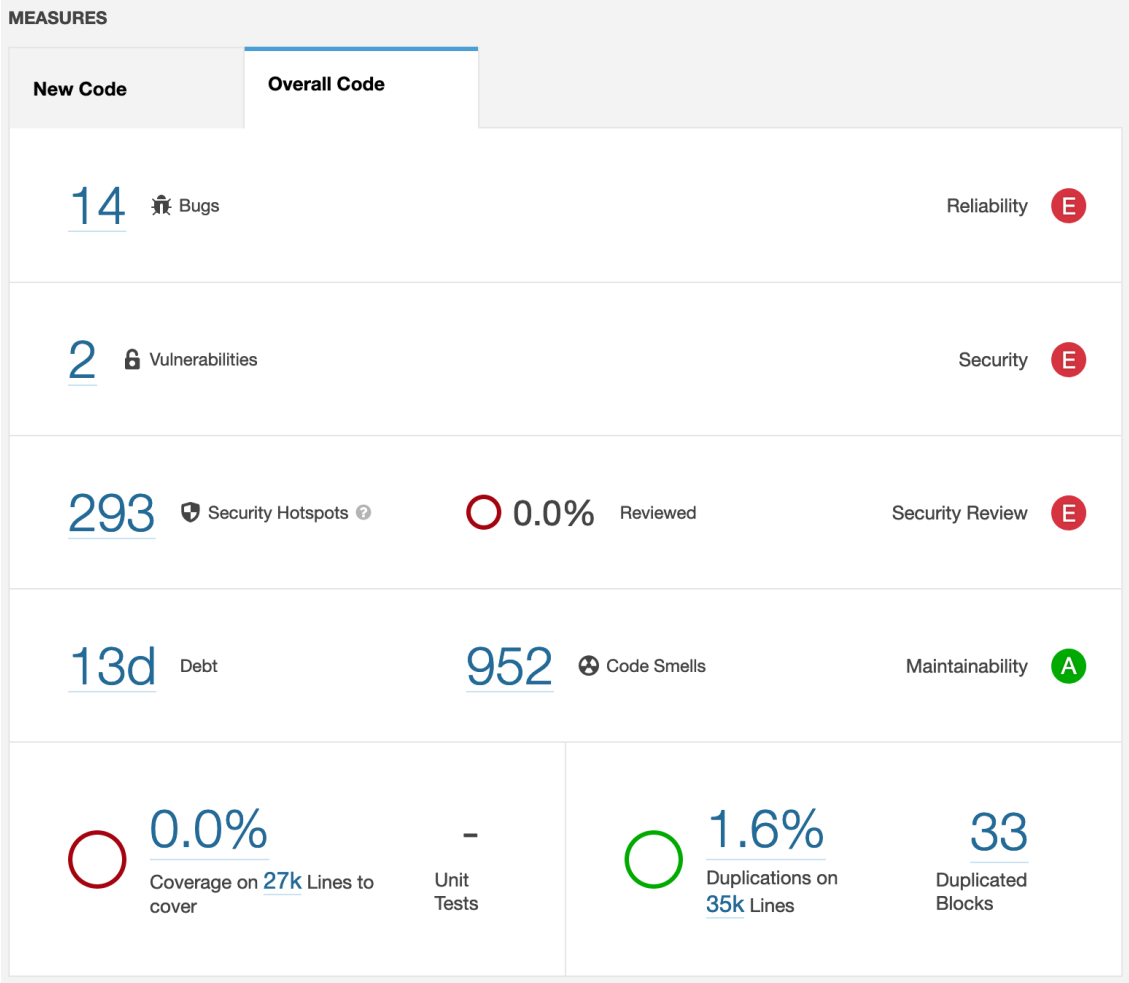
19.3.3 Openpilot's Garage

Now that the context an conditions are clear, it is time to shed some light on the way openpilot is maintained. As mentioned in previous posts, it is maintained by professionals but the decentralized nature allows for

aspiring mechanics to gain experience through open-source contributions, For the occasion of writing this essay, we put ourselves in the shoes of an independent mechanic and ran code analysis tools on openpilot's code.

19.3.3.1 SonarQube Analysis

We ran SonarQube on commit `f21d0f3` and investigated its assessments based on the static code analysis:



Each metric and it's evaluation will be discussed next.

19.3.3.2 Reliability

Because of 14 bugs, openpilot scores an E. The majority of these bugs live in non-critical parts of the codebase like tests and debugging tools. Most bugs relate to an incorrect amount of parameters provided in a function call and useless self-assignments. While these issues should be fixed, we are happy to see that none of them resides in critical core functionality.

19.3.3.3 Security

Because of 2 vulnerabilities openpilot scores an E, one is discussed.

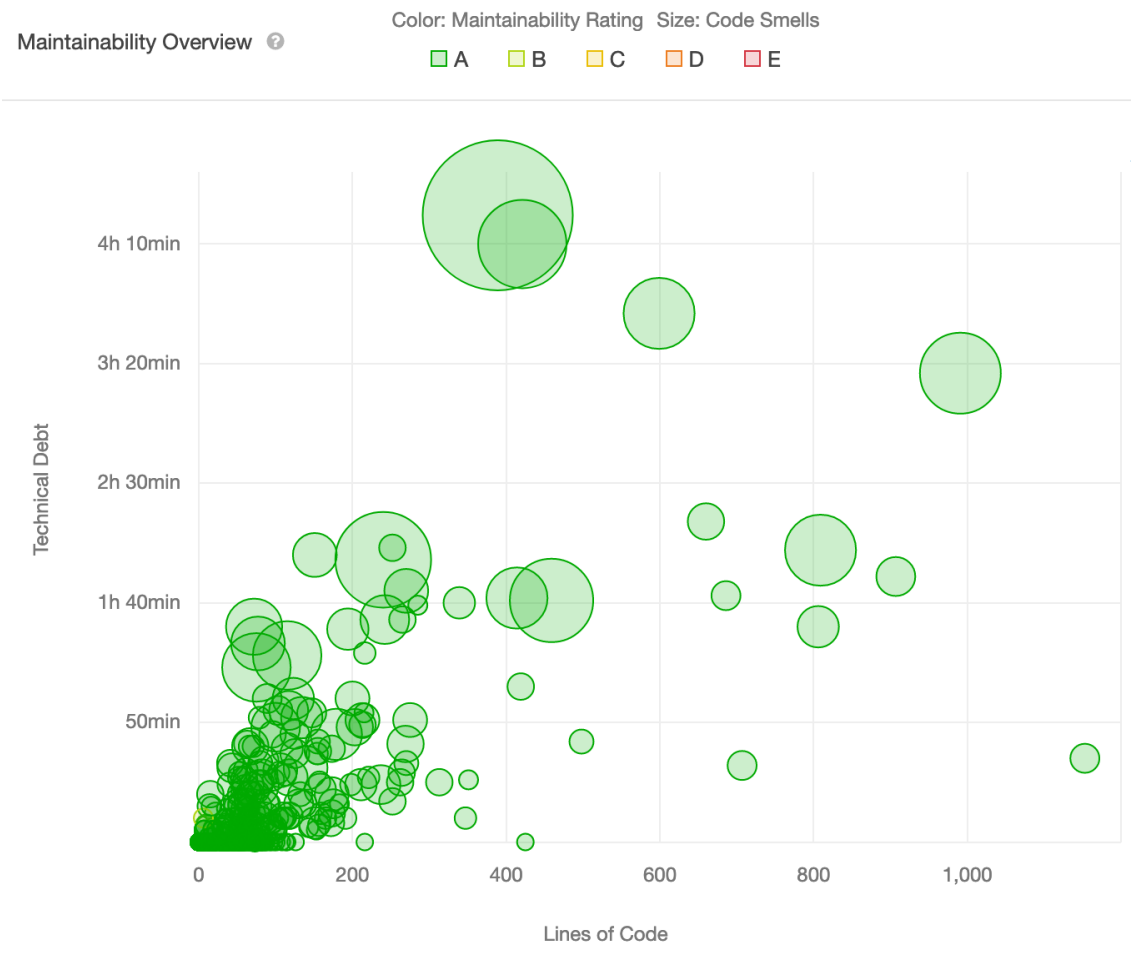
19.3.3.3.1 Weak SSL/TLS protocols should not be used What we see here is a connection with the online logger service, using an insecure TLS protocol version. Exploiting this could possibly expose privacy-sensitive car data.

```
def open_connection(self):
  ...
  ssl_version=getattr(
    ssl,
    'PROTOCOL_TLSv1_2',
    ssl.PROTOCOL_TLSv1
  ),
  ...
```

This should be fixed by upgrading both server and client to the latest TLS (1.3) version.

19.3.3.4 Maintainability

Interestingly, the codebase scores an A on maintainability. We are actually surprised by this result, since we expected it to be a bit lower because of our findings regarding lengthy files and missing comments. Looking further, we see an estimated debt ratio of 0.6% and 13 days of technical debt. While there are 952 code smells in total, they seem very distributed over the 395 analysed files. SonarQube visualizes this in the Maintainability Overview, note that each bubble represents a file.



This highlights refactoring candidates because of their lines of code.

When investigating the code smells, often recurring ones are: - Snake case function and variable naming instead of camel case (the vast majority of all 492 minor issues); - Cyclomatic complexity being (way) higher than the allowed 15; - Duplicated code; - More function parameters than the allowed 7.

These issues can and should be fixed to preserve long-term maintainability. Inconsistencies resulting from missing standards could lead to different coding styles. This makes it hard for new contributors to understand the codebase and decide on which style/standards to adapt.

19.3.4 Other Technical Debt

Most technical debt is very hard to discover or quantify with static analysis tools. This stems from the fact that technical debt most of the time is a *feeling* that a developer has about a codebase, which is quite clearly not objectively quantifiable. Ask a developer to implement a new feature, and he/she will use that feeling to provide a time estimate.

So what do openpilot developers consider technical debt? A look at the current issues page reveals that

project lead George Hotz has identified a number of improvements to be made for openpilot's next release. Some noteworthy issues:

- [issue] Custom VisionIPC should be replaced with the more stable but functionally equivalent msgq. This is an example of technical debt that simplifies a project, and therefore simplifies future contributions.
- [issue] Fingerprinting code needs to be refactored to use a new API. Uniform code is more easily digestible for new and existing contributors.
- [issue] Elimination of code duplication. Less code duplication leads to readable code, which leads to less cognitive overhead, which leads to happy developers.

19.3.5 The Road Ahead

In the first blogpost the roadmap of the system contains two important elements, creating ground-breaking technology by learning from drivers and ensuring that the MTBF of autonomous driving outperforms that of human driving. We will now look at which components of the architecture, described in the second blogpost, will be affected by the roadmap.

19.3.5.1 Learning From Drivers

As mentioned, the goal of the system is to learn everything from drivers. So far the system is already well underway with accomplishing this, but it should eventually become capable of simultaneously performing steering, gas and brake end to end.

If we look at the architecture diagram, displayed in the second blogpost, the components that will be affected by the future changes, are the components within the Self-drive Core part of the architecture. This part of the architecture will be affected as this section receives the driving data in order to return an action. The two most important components to highlight from this section are modeld, it applies the machine learning model on incoming data, and plannerd, it creates a task for the car.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
selfdrive	23,312	6	0	708	196	0.0%	2.1%
athena	557	0	0	14	3	0.0%	0.0%
boardd	365	0	0	5	11	0.0%	0.0%
camerad	174	0	0	1	4	0.0%	0.0%
car	6,822	0	0	205	1	0.0%	0.4%
controls	3,696	1	0	174	1	0.0%	0.0%
debug	2,672	3	0	64	46	0.0%	10.2%
locationd	3,699	0	0	187	17	0.0%	4.3%
loggerd	531	0	0	4	4	0.0%	0.0%
mapd	672	0	0	16	2	0.0%	0.0%
modeld	167	0	0	1	3	0.0%	0.0%
test	2,520	1	0	21	48	0.0%	0.6%
thermald	375	0	0	4	5	0.0%	0.0%

Taking a look at the self-drive core components in the image above, where plannerd, radard and controlsd are located in the controls folder, we see in these sections that there is just one bug, which is within a test in the controls folder, almost no security vulnerabilities, and a couple of code smells, except for the car folder. If we just look at the modeld and plannerd part of the code, we even see that there is just one code smell and three security vulnerabilities. This shows that the overall quality of the self-drive core components is decent, the quality of modeld and plannerd is great, and the maintainability overall is outstanding, as previously seen.

19.3.5.2 Autonomous Over Human Driving

Two important factors to improve the MTBF of the system are having maintainable code and fully understanding bugs/system failures. As the MTBF is related to the whole system, all the components, displayed in the architecture diagram, will be affected. It basically comes down to the reliability and maintainability of the system, of which we have previously seen an overview. There we saw that the reliability of the system got the grade E and for the maintainability of the system, the score was an A. While the bugs found by SonarQube reside in non-critical parts, and the maintainability is great, these bugs and the code smells found by SonarQube should be fixed in order to achieve longterm reliability and maintainability.

We hope we gave you some insights in the perception of quality that openpilot has. In conclusion, we can say that openpilot is a prime example of how much of an interdisciplinary field software architecture is. With dependencies on things like the culture and business, every architectural decision eventually matters. The future is bright for comma.ai and openpilot. Not only is the quality of their software and hence their entire system a key driving factor, more and more people want to enjoy this quality. With the first user willing to monetarily incentivize someone who can help drive his car better in February 2020 and an official bounty

backlog, the urge and urgency for quality is high. Quality that is achieved by writing maintainable code.

19.4 How even a single product can vary

In previous posts, we focused on the business, technical and quality side of openpilot. Today it's time to zoom in on some different aspect that might not be the first thing that pops into your mind when thinking about the architecture of a software product: Variability! Now you might think, variability is everywhere or variability is unavoidable. In the world of software, both are true! This post will be dedicated to defining, identifying and coping with variability from an architectural point of view. We will do this by identifying the aspects of the software that are susceptible for variability and how individual components relate to each other. Subsequently we will look at the meta-side of variability and which effects it has on those concerned. Finally, we will have a look at openpilot's strategy for managing variability.

19.4.1 Good is variable, variable is good

Let us first narrow down our scope and identify three areas that are the potential host of variability and go over the applicability in this context. We can have variability in:

- *Hardware*: In this context, the hardware is the comma two. Since this is an Android device, comma.ai controls and always has controlled the distribution, no variability can be found here that is worth mentioning.
- *Software*: openpilot is a product that in continuous development. As a result, sometimes paradigms shift like when the business advocated externalization. Additionally, in a previous post about the architecture, the modularity of this product was highlighted. Is the architecture well-suited for this kind of variability? Also the fact that openpilot is implemented in both Python and C++ makes it prone for disturbance. Considering the numerous amount of people that contribute to this project, how would you deal with different coding standards, variable naming conventions and formatting settings for linters?
- *Platform*: Although different than our common definition of platform, we will treat the car as the platform for this blog. It goes without saying that the root of a lot of variability in openpilot is due to the large number of cars it supports. As you probably will understand by now, we can not simply ssh into a car to open up a UNIX shell and issue some commands that makes it steer to the right. Each (brand that makes a) car is unique in its own way. Despite the fact that there is some uniformity, there are still fundamental differences.

19.4.2 Should we eliminate it all then?

Important to note here is that in an ideal case, we want have as little variability as possible in either of the domains (software, platform) without limiting functionality. Before addressing how openpilot deals with such a quandary, the picture below depicts a feature diagram. This diagram was made with FeatureIDE and provides a coarse overview of the concrete features and possibilities that openpilot has to offer to it's end-user.

At first sight, this looks quite static since the only (mutual exclusive) choice that developers (product owners in this case) have, concerns the messaging protocol (indicated by the dotted line in the figure). This however is a software variability that is encountered during design time, this can and will be managed by making one default for the future. As mentioned, the platform variability (having to support a variety of cars) is

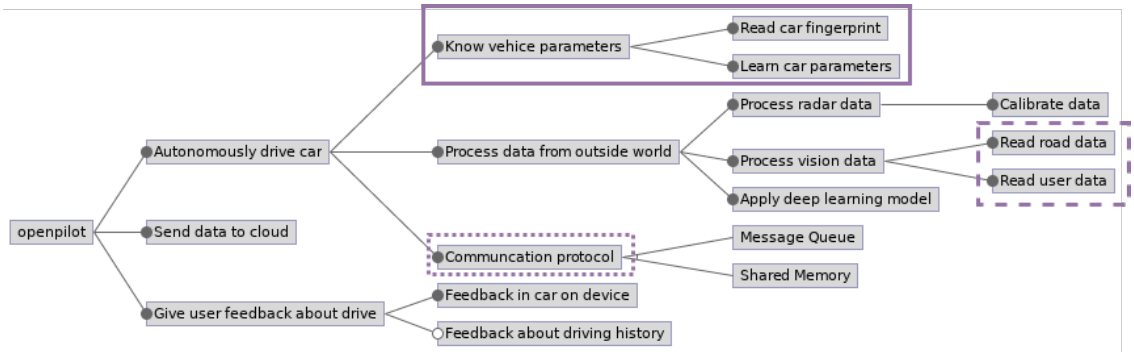


Figure 19.1: Feature diagram, boxes indicate variability

indicated in the figure with the solid line. This variability occurs at load time and you would assume that this can be tested, right? We will address that in further sections. The last box with the dashed border is software variability that occurs during run time. Suppose you place a camera in your car (or another preferred transportation vehicle) and look at the footage you recorded after one day, how many frames are exactly similar? Well, indeed, a fundamental source of variability is the input data of your software. Combine this with the different cars, cloud connection, modular code bases and a messaging protocol and you will end up with an equation that has a lot of parameters. We will discuss how openpilot manages all these parameters in the following sections.

19.4.3 Variability Management

Now that we have seen an overview of the variability within the system, we will take a look at how this variability is managed for different stakeholders and which mechanisms are put in place to help ease this management.

19.4.3.1 Stakeholders

The stakeholders that we will take a look at are the ones that have to do with the product, comma two, namely the end users, open-source developers and comma.ai.

First of all, let's take a look at the end users. The variability that the end users will encounter will be the different car brands and models they use, or the fact that the user switches cars and now wants to use the product in the new car. As long as the car the end user wants to use the product in is supported by openpilot, this variability will all be managed by the product with the use of fingerprinting, which will be explained later on, so even switching cars is no problem. This means that the user just needs to install the comma two in their car, which can be done by following the installation guide on the Comma.ai website, and the product will manage the rest. In terms of variability, the end users are directly linked with the variability in the software that occurs during run (driving) and load (installation) time. This is managed by testing and feeding the data to the neural network to let the model deal with with it.

Secondly, we have the open-source developers, of which most (if not all) are also a part of the end users. As they are primarily concerned with the varibility in the software that occurs at run time and design time, it is key to establish uniform guidelines. The first stop for open-source developers is the GitHub of openpilot. This GitHub, as well as those of the used submodules, contains multiple useful markdown files to read before

you start developing. For open-source developers who would like to make a port for their own unsupported car, there are two medium post, one more general post and an actual port guide, describing how to port a new car. These are posts made by Comma.ai in order to make sure that when an open-source developer adds a new car port, this is done in the same way as the already supported cars. On top of this, there are tests that the developers can locally run in order to check whether or not their implemented changes/additions have the desired outcome.

The last stakeholder we will take a look at is Comma.ai, which mainly consists of developers, thus the points mentioned in the previous section are also in force here. As the owner and supervisor of the project, every type of variability applies to them and they have a decisive role in this. Looking at the figure, we see the (dotted line) possibility to choose between the communcation protocol. This is simply a design consideration and solely depends on their preference as a company. Additionally, the possibile extension of the supported communcation protocols with flexray is something that they have veto authority on. The reason that Comma.ai made the software open-source is because this would create more variability, as people are able to add a port for their own car. In order to manage this variability, Comma.ai decides which cars get fully supported and which not. This is partially based on whether or not the car is used by a significant amount of people. On top of this, Comma.ai will make sure that the code of a new car port meets the safety and code quality standards that they have set for their product.

19.4.3.2 Ease of the Variability Management

In order to make sure that the variability stays manageable, Comma.ai has made certain mechanisms and design choices. One of them is a test to check whether or not the supported cars can be identified correctly, with the use of fingerprinting. On top of that, it is also possible to watch replays/simulations from data of previous driving sessions, which can help identify potential causes of bugs during the process of porting a new car or maintaining a supported car. However adding more cars can make the abstraction within the system weaker, because every car requires its own safety model. This causes duplicate code between dbc files of cars with the same brand but a different model. As the abstraction helps managing the variability, it is important that the architecture of the system keeps the abstraction. One way of keeping the abstraction is with the use of an observer like design pattern, as publish subscribe (explained in a previous post). This pattern fits perfectly for openpilot, as openpilot contains multiple different components that need to communicate with one another. Openpilot also uses Cap'n Proto, which creates common typed specifications of car states, further touched upon in the next section.

19.4.4 Implementation Mechanism and Binding Time

This section discusses how openpilot detects which car it is connected to and how different cars are supported.

19.4.4.1 Fingerprinting

The comma two tries to identify the car it is connected to at engine startup. The series of messages produced on the CAN bus at start is assumed to uniquely identify a car. This series is then matched against a collection of known series, called fingerprints. When openpilot successfully matches against a known fingerprint, it can apply the correct decoding specification to interpret the CAN messages. Later, when sending control messages to the actuators, openpilot is able to take car-specific parameters into account.

19.4.4.2 Implementation Mechanism

The goal of openpilot is to constantly improve the driving quality and safety. To achieve this, the (machine learning) model and other code to compute adjustments is changed and improved frequently. To avoid having to maintain a dozen different versions for each car, a general CarState object is used as an abstraction layer. A CarState object holds information about the runtime state and user settings. Information in the runtime state includes the current estimation of speed and yaw rate. The user settings includes the set speed limit and acceleration. These CarState objects are created by a car specific carstate.py function from car specific CAN messages. Carstate.py uses a car specific .dbc file containing the CAN message encoding/decoding scheme. It furthermore uses the common typed specification of a CarState object created in car.capnp.

The adjustments computed by the model called by the control loop in controlsd.py are returned as CarController objects. These objects are then consumed by a car specific carcontroller.py script. This script in turn creates the car-specific messages for autonomous driving. This procedure is depicted below.

19.4.4.3 Safety Mechanism

One may wonder how it can possibly be safe to use a model operating on a universally modelled abstracted car. Mistakes in the conversions scripts, carstate.py and carcontroller.py, could result in extreme fluctuations in inputs and outputs. To prevent dangerous situations on the road, the Panda, a hardware adapter responsible for the communication between the comma two and the CAN, constantly monitors the actuator messages the comma two wants to apply. It compares the messages against a safety model in the Panda, and if the changes are too abrupt with respect to the current state, it ignores such messages and requests for user engagement.

19.4.5 Variability Galore!

In this post we identified the main aspect of variability with respect to impact on the architecture: the controlled car (platform). We see that increasing the amount of supported cars and easing maintenance is a shared goal of all stakeholders. The architecture realises this goal by operating on a universal car state. Having implemented a car abstraction layer eases the process of adjusting the model for all cars at once. Adding a new car simply boils down to providing the correct car-specific information files and some light tuning. We foresee no problems in adding more and more cars in the future given the current architecture.

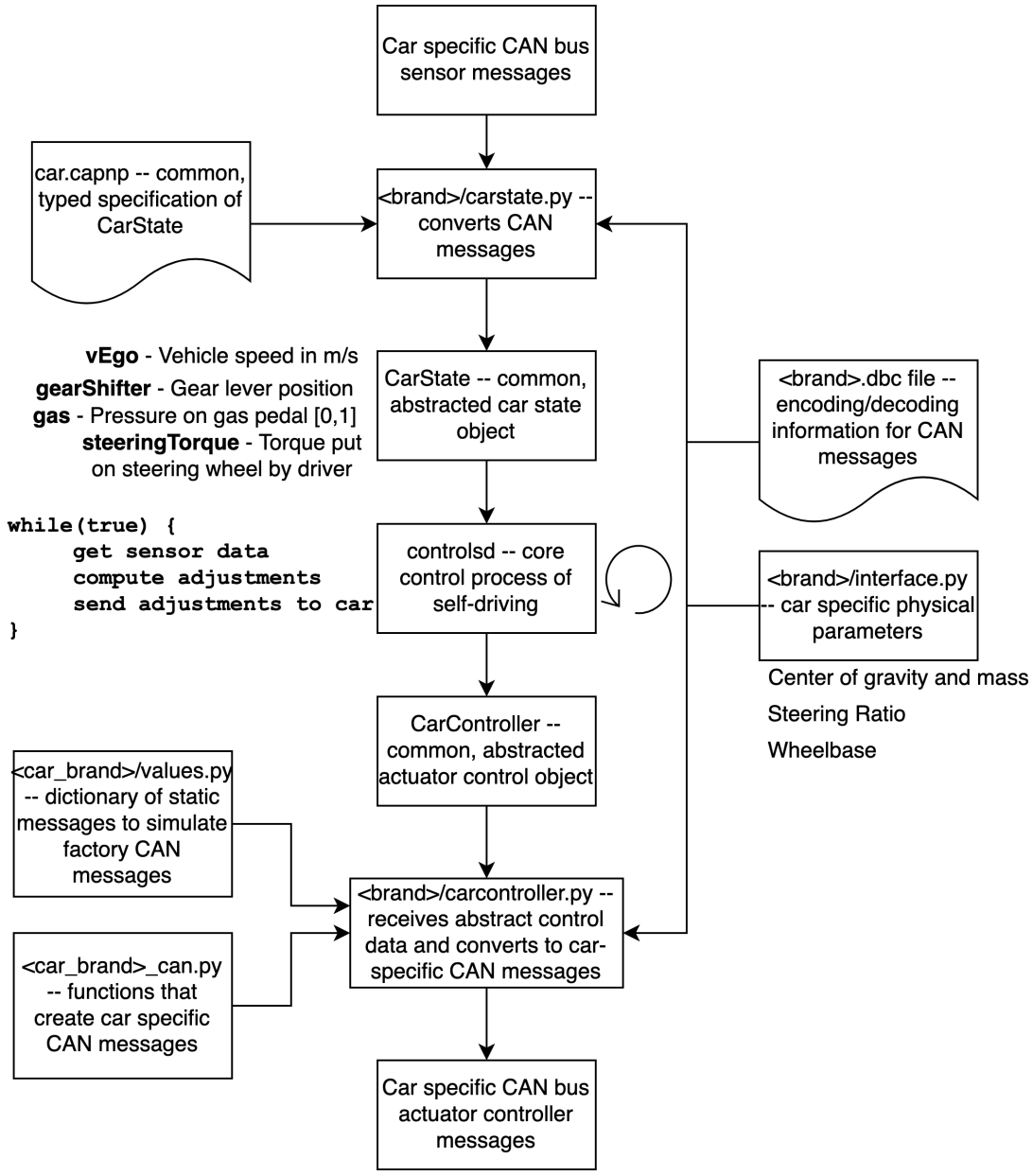


Figure 19.2: Variability Implementation

Chapter 20

OpenRCT2



OpenRCT2 is the open-source implementation of the classic Rollercoaster Tycoon 2 video game released in 1999 by Chris Sawyer. The project aims to recreate the game, add new features to it, and fix bugs that were present in the original.

Rollercoaster Tycoon 2 is a game where the player is supposed to build and manage a theme park. The player can play scenarios or sandbox mode. In scenarios, the player has to build or improve a theme park given certain financial constraints or within a period of time. In sandbox mode, the player is free to build their own theme park unconstrained.

The new features, such as multiplayer, are supposed to be supplemental to the experience of the original game and should not serve as a replacement. A player that has played the game in 1999 should be able to have the same experience with playing OpenRCT2 in 2020.

20.1 Team

- Vanathi Rajasekar
- Jelle Eysbach
- Daan de Heij
- Thijs Versfelt

20.2 OpenRCT2, Porting RollerCoaster Tycoon into 2020

In 1999, Chris Sawyer released the revolutionary and succesful game; *RollerCoaster Tycoon (RCT)*. Many sequels later, the series remains popular to this day, and has inspired fans to develop an open-source re-implementation: OpenRCT2.

The OpenRCT2 project has two goals. The first goal is to bring the original RCT2 experience to modern systems. The second goal is to expand the game. The original game was created in Assembly[1] and only ran on Windows and Xbox.[2] The OpenRCT2 developers reverse-engineered the original game and rewrote it in C++. This allows the game to run on other platforms, like MacOS or Linux. Furthermore, this new code base enables developers to extend or modify the game more easily.

The second goal of OpenRCT2 is to expand upon the original with fewer limitations and new features. Features like multiplayer, auto-saving and 64-bit support were added.[3] Various bugs and issues with the original game were fixed.[4] This was done without changing the original premise.

20.2.1 End-user mental model

After launching the game, the user sees a theme park. The park welcomes guests that have to pay an admission fee to enter the park. Guests can walk around and perform activities, such as entering attractions and buying food and drinks. The guests have different desires and properties, which are communicated to the player through a variety of statistics and emoticons. The player can also read guests' thoughts on the park and attractions.

The player also sees UI components to control the game, such as constructing buildings, pausing the game, or changing settings. Moreover, when playing scenarios, there is a goal to achieve before a certain time (e.g. to reach a number of guests in the park).

The second part of the mental model concerns the way the end-user interacts with the system. To achieve the goal presented at the start of a scenario, the park needs to be managed by the player. The player can build various rides, stalls and sanitary provisions to improve the happiness of the guests. The user can then decide on the ride or store prices and make sure the park runs at a profit. The user can also hire staff, move the people in the park and add cosmetic objects to the park. The user also has access to a menu that sets configurations of the game.

20.2.2 Key capabilities and properties

The key properties and abilities that OpenRCT2 should provide are:

- Has the properties and functionality of the original game
- Has compatibility with the original game
- Runs on modern (operating) systems, in multiple resolutions and graphical modes
- Has online functionality
- Code is open-source and free-licensed

[1] https://www.eurogamer.net/articles/2016-03-03-a-big-interview-with-chris-sawyer-the-creator-of-rollercoaster-tycoon
[2] http://www.chrissawyergames.com/info.htm
[3] https://openrct2.org/features
[4] https://openrct2.org/changelog

Providing the properties and functionalities of the original game was the main goal when the project was founded. After these original functionalities were implemented, key features were added that extended on the original game.

20.2.3 Stakeholders

The stakeholders of OpenRCT2 can be classified in:

- **Players:** People playing OpenRCT2 benefit from the project in numerous ways. Contrary to the original RCT games, OpenRCT2 runs on more (modern) hardware and operating systems. Moreover, it has additional features. In the future, the game is planned to be completely free due to independence from the original game data.
- **Developers:** Open-source development was started by Ted John (IntelOrca), but the project has progressed due to numerous contributors by others. Other current active contributors include *Gymnasiast, duncanspumpkin, tupaschoal, janisozaur, ZehMatt*. Furthermore, it is now much easier and faster for developers to customize or extend the game, thanks to OpenRCT2's open, C++ code-base. In contrary, the original RCT games were closed-source and built using Assembly.
- **Artists**: Artists contribute to the game by providing music, textures and models. OpenRCT2 enables artists to apply their creativity to the game.
- **Translators:** OpenRCT2 has been translated into numerous languages thanks to various translators. This allows the game to be played in more players' native language.
- **Testers:** Testers write tests or play the game to test specific features or behavior to ensure it is correct.
- **Sponsors:** The companies that allow OpenRCT2 to use their tools/resources. The current sponsors for the software are *DigitalOcean*, *JetBrains*, *AppVeyor*, *Travis-CI* and *BackTrace*. These companies benefit through increased exposure from providing these tools/resources.
- **Competitors:** There are similar open-source reimplementation of classic games such as: OpenLoco, OpenTTD, openage, and OpenRA. These can be seen as competitors of OpenRCT2, but may actually also benefit from OpenRCT2 gaining popularity. Furthermore, more recent theme park simulation games can also be seen as OpenRCT2's competitors. These include: Planet Coaster, Parkitect, and RollerCoaster Tycoon 3.
- **Original developer:** The original RCT games were developed by Chris Sawyer. OpenRCT2 might benefit him by providing renewed exposure to his games, and by providing him with a C++ implementation.
- **Original publisher:** RollerCoaster Tycoon 2's original publisher Infogrames might benefit from renowed exposure to the game, and from players purchasing the RCT games in order to obtain the data required by OpenRCT2.

20.2.4 Current and future context

The list below shows the context in which OpenRCT2 currently operates.

- Distribution
  - Laucher (x86/x64) (Windows/Linux/MacOS)
  - Stable builds (x86/x64) (Windows/Linux/MacOS)

  - Development builds (x86/x64) (Windows/Linux/MacOS/Android)
- Compilers
  - msbuild (Windows)
  - Xcode (MacOS)
  - CMake (MacOS/Linux)
- Community
  - Reddit
  - Gitter (dev/non-dev)
  - Forums
  - Twitch
- Programming languages
  - C++
- Integrated Development Environment (IDE)
  - CLion
- Version control
  - Github (Code/Issues/Pull requests/Wiki)
- External libraries
  - Original data files of RCT2
  - OpenGL
- Testing
  - Code tests
  - Play tests
- Continuous integration (CI)
  - AppVeyor (Windows)
  - Travis-CI (Linux/MacOS)
- Hosting
  - DigitalOcean (Domain/Website/Forums)
- License
  - GNU General Public License version 3
- Stakeholders
  - See the *Stakeholders* section.
- Documentation
  - Wiki (GitHub)

20.2.4.1 Distribution

OpenRCT2 currently has stable and development builds for the Windows, Linux and MacOS operating systems. A development build is also available for Android and it can be expected that further development for that platform will allow it to be included as a stable build. There is currently no iOS build (stable or developement) for OpenRCT2 and this might be something that could be added in the future.

A launcher is available for the Windows, Linux and MacOS platforms, which automatically keeps the latest stable build up-to-date.

20.2.4.2 Compilers

OpenRCT2 uses msbuild to build the game for Windows, Xcode for MacOS, and CMake for both MacOS and Linux.

20.2.4.3 Community

OpenRCT2 has an active community of developers and users, which communicate mostly through Gitter. Gitter is a chat and networking platform meant for communication among and between GitHub users and developers.

Aside from Gitter, the official OpenRCT2 forums are also used for user and developer communcations. Other platforms for users to connect on are the OpenRCT2 subreddit and Twitch.

20.2.4.4 Programming languages

OpenRCT2 is written in the C++ programming language, unlike RCT2; which was written in assembly.

20.2.4.5 Integrated Development Environment (IDE)

The developers of OpenRCT2 are provided with a free license of CLion from JetBrains for the development of OpenRCT2.

20.2.4.6 Version control

OpenRCT2 uses only GitHub for storing code, raising and closing issues, releasing builds and writing and updating the wiki.

20.2.4.7 External libraries

OpenRCT2 uses the original data files of RCT2 for the textures and sprites. OpenGL is used to render the graphics of OpenRCT2.

The development team is in the process of developing their own open-source textures and sprites, such that current and future players are not required to use the original data files of RCT2.

20.2.4.8 Testing

Code tests are performed by writing and running unit tests. OpenRCT2 has testers playing the game to test for specific features or behaviour.

20.2.4.9 Continuous integration (CI)

OpenRCT2 uses AppVeyor for continuous integration on Windows and Travis-CI for continuous integration on Linux and macOS.

20.2.4.10 Hosting

DigitalOcean is a hosting company that provides OpenRCT2 with free hosting for the OpenRCT2 domain, website and forums.

20.2.4.11 License

OpenRCT2 is licensed under the GNU General Public License version 3 (GPL v3); a copyleft license that allows anyone to change and distribute the software, as long as the author is credited and the same freedom of changes and distributions are passed on to recipients of the software.

20.2.4.12 Documentation

The OpenRCT2 wiki provides general information about the game, guidelines for contributing to the project, technical information about the game, and information on how to build the project on various platforms.

20.2.5 Product roadmap

The main goals in the future development of OpenRCT2 currently are:

- Raising limitations of the original game
- Creating a new save format
- Creating open-licensed graphics and prerequisites
- Fixing bugs and issues
- Adding features.

RCT2 has various hard limits, such as limits on the number of rides, sprites, or map size. An intended goal in the future development of OpenRCT2 is to increase or remove these limits. Currently, OpenRCT2 uses the save format SV6, the native save format used by RCT2. SV6 imposes many of these limits. Hence, a new save format is being developed. This will allow limits to be removed or increased.

As of now, a copy of RCT is required in order to play OpenRCT2. OpenRCT2 uses the original, copyrighted game data such as graphics and sounds. In the future, free-licensed graphics and sounds will be created in order to remove the dependencies of OpenRCT2 on this original game data. These projects are currently in very early stages of development.

20.3 The architecture of architecting Rollercoasters

This essay delves deeper into the software architecture of the OpenRCT2 project. The architecture is first examined from different views, based on the book *Software systems architecture* by Rozanski and Woods. Then, the software is decomposed and each component's function is explained. Next, the main architectural pattern of the project is explained and finally the trade-offs between the non-functional properties of OpenRCT2 are discussed.

20.3.1 Relevant architectural views



An *architectural view*, according to Rozanski and Woods, is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders.[5]

So, a view should:

[5] Rozanski, Nick, and Eoin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012.

- Enable stakeholders to determine whether their concerns have been met.
- Enable stakeholders to perform their role in the system.

Rozanski and Woods introduce a viewpoint catalog, whose viewpoints can be seen as templates from which views are constructed. These viewpoints apply differently to different architectures. The most relevant viewponts to the OpenRCT2 architecture are discussed here:

- The *Context* viewpoint describes the *"relationships, dependencies, and interactions between the system and its environment"*. This viewpoint addresses concerns of all stakeholders, for example:
  - Players, testers and developers can determine ways of communicating with other players, testers and developers. This enables them to find other people to play together with, to report bugs at the right place, and to organize development.
  - Developers can determine what programming language and version control system is used, enabling them to contribute to the project.
  - Sponsors can determine which of their services are used. This enables them to provide these services correctly.
- The *Functional* viewpoint describes *"the system's run-time functional elements, their responsibilities, interfaces and primary interactions"*. This viewpoint addresses concerns of all stakeholders, for example:
  - Players can determine what kind of gameplay they can expect. It enables them to play the game that they expected to play.
  - Developers can determine the features that they need to implement. It enables them to make valuable contributions.
  - Translators can determine what parts of the system need translation and in how many languages it needs to be translated. It enables them to add translations to the game where it is needed.
- The *Development* viewpoint describes *"the architecture that supports the software development process"*. This viewpoint adresses concerns of developers and testers, for example:
  - Developers can determine what style of coding they should adopt. It enables them to contribute code that is similar to code that other developers contribute.
  - Testers can determine what the behavior of certain components should be. It enables them to write tests that test for component-specific behavior.
- The *Deployment* viewpoint describes *"the environment into which the system will be deployed and the dependencies that the system has on elements of it"*. This viewpoint is addresses concerns of players, developers and testers, for example:
  - Player can determine where to download the latest stable release and what other software they require, such as the original data files of RCT2. It enabled them to actually play the game.
  - Testers can determine where to download the latest development build. It enables them to test the latest commits for bugs.
  - Developers can determine for what kind of platforms the game will be deployed on. It enables them to write code that support platform-specific functionalities, such as different folder structures.

The *information*, *concurrency* and *operational* viewpoints are not very relevant for the OpenRCT2 architecture. OpenRCT2 does not handle big amounts of data, parallelization is limited and the operation is simple because it is an isolated application.

20.3.2 Development view

The software is separated into components. Every component has a set of tasks relating to the in-game entity it corresponds to. For example, the peep component corresponds to the people in the game and handles all their behaviour. The complete list of components and their descriptions can be found in the appendix at the end of this essay.

Individual components have a very large amount of inter-dependencies. Therefore, figure 1 shows the components grouped in a higher level of abstraction. The most important interactions between the groups are displayed as arrows, with the type of interaction as text.

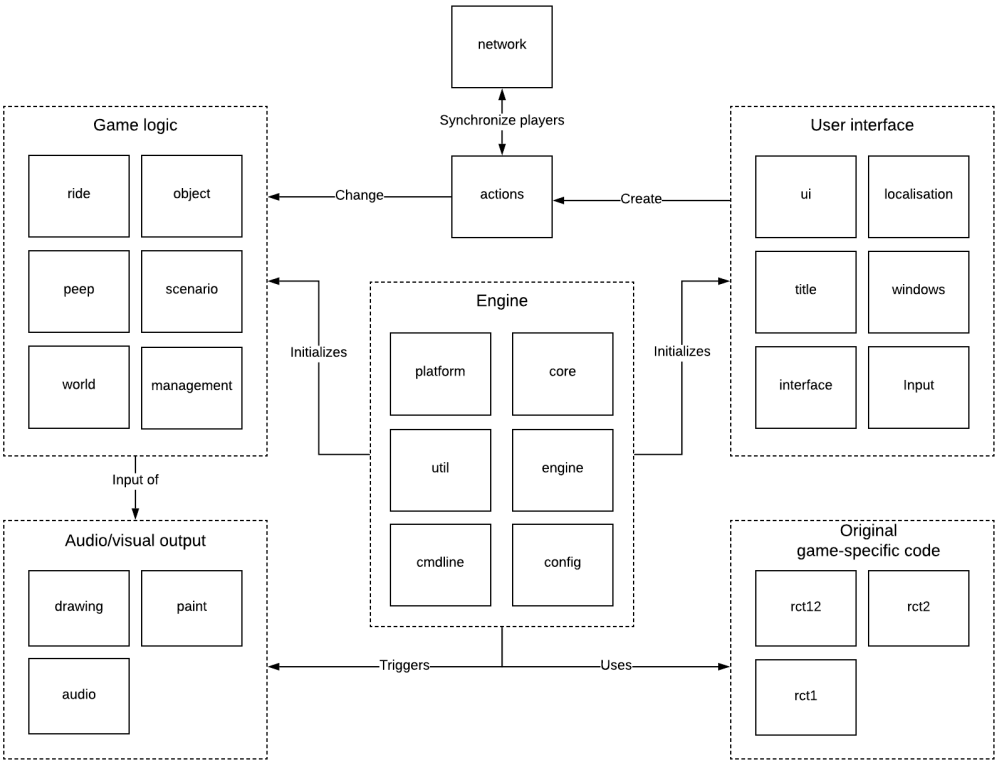


Figure 20.1: System Components

Other aspects of the OpenRCT2 project which make the system more reliable and easier to maintain by the developers include the in-game console. This console can be used to perform actions such as: getting or setting values of variables, listing variables, or manipulating objects. Moreover, the code includes a diagnostic level that can be set before compiling, which changes the level of detail of logging.

The code can be built manually on Windows using `msbuild` or `make` on Linux. The continuous integration system comprises TravisCI and AppVeyor, which automatically build for Linux and MacOS, and Windows respectively. Binaries of builds of the `master` and `development` branches are automatically published to the OpenRCT2 website. Furthermore, a launcher is available, which automatically updates OpenRCT2.

Concerning standardization of design, the OpenRCT2 project has coding style rules that developers have to adhere to. This coding style comprises rules on aspects such as: variable naming, indentation, usage of comments, and maximum line width.

Furthermore, the OpenRCT2 project has a `clang-format` program that is used to check whether code adheres to the coding style, and which can reformat code to adhere to the style if a there is a violation[6]. This `clang-format` program is used by the continuous integration system, ensuring that pull requests do not contain coding style violations. Additionally, a git hook is available, enabling developers to check their code's adherence to the coding style before committing their code. Finally, the project has a specific format for git commit messages. This ensures consistency and precision among commit messages of different developers.

20.3.3 Run-time view

To illustrate how components interact at run-time, we consider the common scenario of performing a start-up of OpenRCT2. In the following scenario, components are written as `component`.

After executing OpenRCT2.exe, the `Context` component from `Engine` performs initialization. First, it tries to locate the RCT2 or RCT1 directory, for the original game data dependencies. Next, internal dependencies such as managers and repositories are instantiated. If the Discord setting is enabled, a service is created for it (`network`). Then the game is set to use the configured language, and defaults to British English if opening the configured language fails. The component warns the user if the game has elevated privileges, as these are not necessary.

Afterwards, a `ui` window is created, and tracks (`track`), scenarios (`scenario`), and objects (`object`) are loaded. The `audio` sub-system is initialized, and sounds are loaded. A `network` environment is created, and a chat is initialized. User files of the original RCT2 game are copied if available. Viewports (`interface`) are initialized and a GameState (`Engine`) is instantiated. Then, the TitleScreen (`title`) is created. Finally, the initialization procedure finishes and the game launch is called.

The launch procedure launches the main menu. In the background a park shows, which is downloaded or loaded from file. If the game is launched as a server, the network fields are populated. Finally, the GameLoop is run.

Now, the main menu is shown to the user, as seen in figure 2. A park is shown in the background, and the user can start or load a game by pressing the buttons on the screen.

The GameLoop calls RunFrame repeatedly until the finished flag is set to true. Each call:

- RunFrame determines the number of ticks that have elapsed since its last run, by calling a function from the `platform` component, as this is a platform dependent operation.
- It calls the `ui` to process user input.
- It updates the time.
- It then calls the update methods of TitleScreen, GameState, UI, console, and chats.
- From the `world` component, it obtains the next position of each sprite.
- It then calls the `drawing` component to draw, and the `paint` component to paint.

[6] https://clang.llvm.org/ Clang: C language front-end for LLVM



Figure 20.2: Main menu

20.3.4 Deployment view

- *Run-time software requirements:* For OpenRCT2 to work, the data files (containing graphics, sounds and models) from the original RollerCoaster Tycoon 2 are needed. The original software has to be bought and installed before playing OpenRCT2. [7]
- *Operating Systems:* The game can run on both 32-bit and 64-bit Windows (Vista/7/8/8.1/10), macOS, Linux, FreeBSD, OpenBSD, and Android operating systems. [8]
- *Hardware Requirements:* OpenRCT2 is able to run on most reasonably modern computers. It requires about 250 MB of disk space, although 1 GB or more is recommended. Furthermore, about 500 MB of free RAM is required. [9]
- *Network requirements:* To use the multiplayer online functionality, a stable internet connection is required. Servers can be created and hosted by any player. There are also dedicated servers run by the community.[10]

20.3.5 Architectural patterns

The primary architectural pattern used in OpenRCT2 is the *model-view-controller* pattern.[11]

The GameState class is a wrapper for all the game logic and represents the model component of the architecture. It initializes the game world and implements an Update functions that runs every frame. This function updates all the game logic components, like guests, rides or stalls.

The Drawing component handles the drawing of sprites and effects. It represents the view component. It uses the state of the game as input and draws the game world.

The controller is represented by the interface component. The interface draws and handles windows and UI elements that the user can see and interact with. Clicking on something in the game opens a window where the user can interact with the object.

[7] https://openrct2.org/quickstart Quickstart guide to run OpenRCT2
[8] https://github.com/OpenRCT2/docs/blob/master/installing/_posts/2018-05-18-system-requirements.md
[9] https://github.com/OpenRCT2/docs/blob/master/installing/_posts/2018-05-18-system-requirements.md
[10] https://github.com/OpenRCT2/docs/blob/master/installing/_posts/2018-05-18-system-requirements.md
[11] Coplien, J. O., & Bjørnvig, G. (2011). *Lean architecture: for agile software development*. Section 8.1. Chichester: Wiley.

20.3.6 Non-functional properties and trade-offs

The OpenRCT2 project has some goals that dictate the most important non-functional requirements of the project, but these requirements also come with trade-offs.

20.3.6.1 Compatibility with modern systems

The game should run on macOS, Linux, Windows and Android. This means the game needs more different build configurations and dependencies to compile for different platforms. Consistently testing all builds is a very time-consuming activity. There are also features of the original game that did not translate naturally to modern systems because of this. For example, text input boxes in the original game depended on windows so in OpenRCT2 a different way had to be developed to allow text input. [12]

20.3.6.2 Graphical interface improvements

The graphical interface of the game was improved to allow features like different resolutions and framerate options. Furthermore, it is possible to choose between different rendering modes such as software rendering and OpenGL.

20.3.6.3 Compatibility with original game files

OpenRCT2 depends on the assets of the original game to run and supports the save files and scenario's of the original. This is very convenient for people that want to load their old save files, but it does mean the limitations of the original game also apply to OpenRCT2. Currently the OpenRCT2 team is working on removing these limitations by implementing a new save format.

20.3.6.4 Resolving trade-offs

The original codebase was developed by IntelOrca[13], he started the project and defined the initial goals and requirements. As the project grew, a core team was introduced that contained 8 other people. Since then, these people discuss trade-offs and decide on solutions with the support of IntelOrca. IntelOrca usually has the final say if the developers are divided. Any other contributors are also free to join in on discussions. The developers usually have the discussions about the design of the project in the issues on github or on the developer chatroom on gitter.

20.3.7 Appendix: Componentisation

- `actions` Handles all player actions and it queues pending actions. For instance, the player can place/remove footpaths, buy/sell land rights, hire staff and load/quit/pause the game.
- `audio` Handles sound for rides, weather and effects. Moreover, it handles music for rides and the game. It also controls volume levels and mixes sound/music.
- `cmdline` Handles and stores command line commands, which specify how the game can be run through the command line, for example, to run the game in headless mode when starting a server.
- `config` Handles the in-game settings, such as the game resolution, enabling multithreading or changing the autosave frequency.

[12] https://mostlyprog.wordpress.com/tag/sdl2-0-text-input/ Developer blog post about text input in OpenRCT2
[13] https://openrct2.org/about About page of OpenRCT2

- `core` Handles reading and writing the console. It also handles encryption and decryption for multiplayer. Lastly, it handles writing, reading and compressing files/data.
- `drawing` Handles drawing of sprites, effects such as rain/light, text (using various fonts) and images. It can do this using OpenGL or the X8 drawing engine.
- `engine` Handles game logic at startup and run-time, which it updates at an interval. It handles how cheats are applied, keeps track of time/date and it enables saving/pausing and quitting. Also, it controls the scenario and track editor.
- `input` Handles player input from the mouse and the keyboard.
- `interface` It controls the in-game windows and their viewports as well as the various UI themes. It is responsible for how the chat is displayed. Stores color and font types. It stores and handles console commands. It handles how screenshots are taken.
- `localization` Handles which language and currency is displayed and stores guest names.
- `management` This handles all park management logic, such as receiving rewards, finances, marketing, news and research. For instance, a park might receive a reward for the "best rides" or there might be news about new research.
- `network` Handles all multiplayer related logic, such as displaying servers, joining/leaving a server, sending andd receiving messages/commands and it synchronises player actions across the network. It also handles the Twitch and Discord integration.
- `object` This controls all park objects behavior (e.g. rides, shops, scenery, paths, etc.).
- `paint` Handles how sprites for all game objects such as rides, shops, scenery should be drawn.
- `peep` Deals with all non player characters (NPC's). It handles their sprite animations, pathfinding, behavior (riding rides, queueing, buying tickets, etc.), thoughts and interactions with player actions.
- `platform` Handles platform specific logic, as specifies directory structures for other OS's and reads and applies the platform language for localization.
- `rct1` Handles importing RollerCoaster Tycoon 1 (RCT1) save and track design files (S4 and T4) and converting them to the RCT2 formats (S6 and T6).
- `rct12` Handles importing RCT1/2 shared files and converting them
- `rct2` Handles importing/exporting RCT2 files (S6 and T6), which are still used by OpenRCT2.
- `ride` Manage all ride (rollercoasters, transport, etc.) and shops behavior. Also controls track design behavior for the rides.
- `scenario` Controls and stores game scenarios.
- `testpaint` Tests rides, such as the ride tracks for specified rides in the game.
- `tests` Tests functionality of components, such as pathfinding in the `peep` component or the ride ratings and encryption/decryption used in multiplayer.
- `title` Loads and updates the title screen sequence.
- `ui` Handles how the the player actions influence the `interace`, such as opening in-game windows. It also specifies how the interface is displayed on other platforms than windows, such as Linux or MacOS.
- `util` Contains a collection of utility functions such as unit conversion, string helper functions, compression and several more.
- `windows` Handles and stores window logic and definitions for all different windows in the game. For instance, it specifies the position of various buttons or a viewport in a window.
- `world` Controls the world objects and mechanics, such as weather, terrain.

20.4 Rollercoaster (tycoon) should not crash

I'm sure if you're reading this, you've most likely at one point as a kid thought that being a game tester would be an awesome job. Playing games and getting paid for it, sounds great. However, why do video game developers rely so greatly on players to test their games?



Writing automated tests is generally a tough job. It requires a software system to be well defined, unchanging and loosely coupled for it to be a reasonable time investment. Video games, unfortunately, tend to be tightly knit systems that can change drastically over the course of their development. [14]

This is one of the reasons why OpenRCT2 does not have a lot of automated tests. It depends on players to find bugs and other issues. When automated testing is not really an option, you have to rely on other ways to maintain the quality of your code and prevent the build-up of technical debt, which is what we'll be covering in this essay.

20.4.1 Software quality process overview

OpenRCT2 uses continuous integration (CI) checks, mandatory code review and bug reports to ensure correct functionality of code changes. The CI builds for Windows, macOS, Linux and Android. The Windows and Linux builds also run the tests. Afterwards, the build artifacts are uploaded to the CI and the website.

Besides code functionality, there are also rules for commit messages [15] and code formatting [16]. The code formatting also has its own CI check. Tracking of bugs is done through the Github issues, where players can submit bugs they find. Windows crash logs are also automatically uploaded here with backtrace.io. [17]

Finally, the in-game replay system can be used for creating tests. It allows for commands to be recorded and saved in a replay file. Then later, a replay can be played to execute the same set of commands.

20.4.2 Importance of code quality, testing, and technical debt

In this section we will look at how OpenRCT2's issues, pull requests, and documentation indicate the importance of code quality, testing, and technical debt in the project.

OpenRCT2 has a page available with planned projects, many of which have the goal of decreasing technical debt, or improving code quality and flexibility. Prime examples hereof are the new save format, or the

[14] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development? *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. doi: 10.1145/2568225.2568226

[15] https://github.com/OpenRCT2/OpenRCT2/wiki/Commit-Messages Commit message rules

[16] https://github.com/OpenRCT2/OpenRCT2/wiki/Coding-Style Clang coding style rules

[17] https://github.com/OpenRCT2/OpenRCT2/pull/8073 Pull request introducing backtrace.io

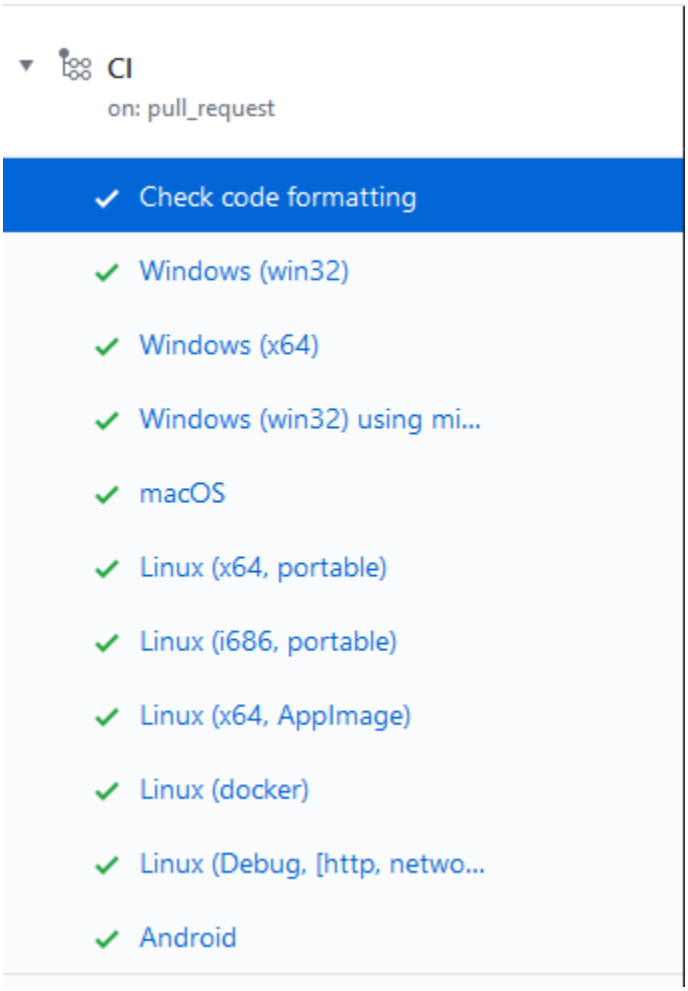
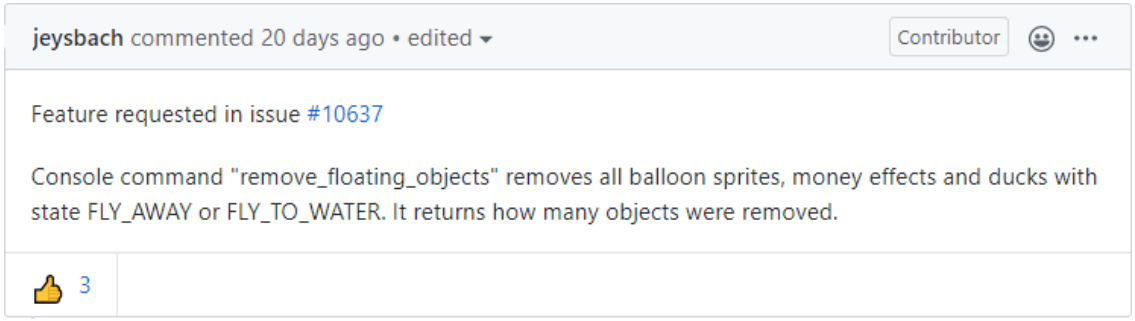


Figure 20.3: CI checks

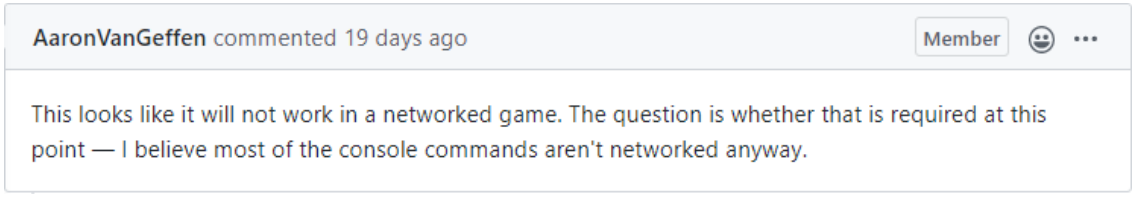
plug-in API.

Another indicator of the importance of code quality in the project is the priority given to refactoring. Currently there are over 30 open issues regarding refactoring, indicating that refactoring is a major concern for the project.

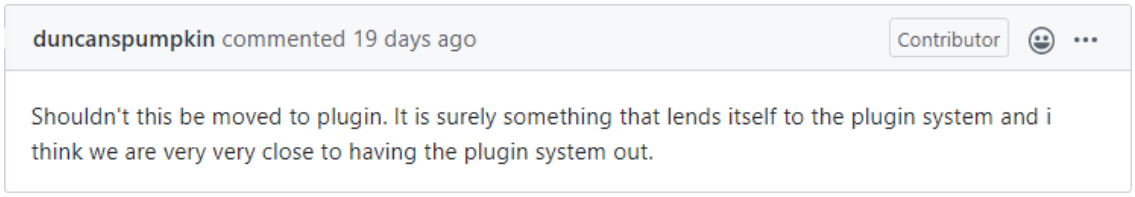
Furthermore, the comment sections of the project's issues and pull requests give additional indication of the importance of code quality and technical debt. For example, let's look at the pull request our team offered to OpenRCT2 earlier this month:



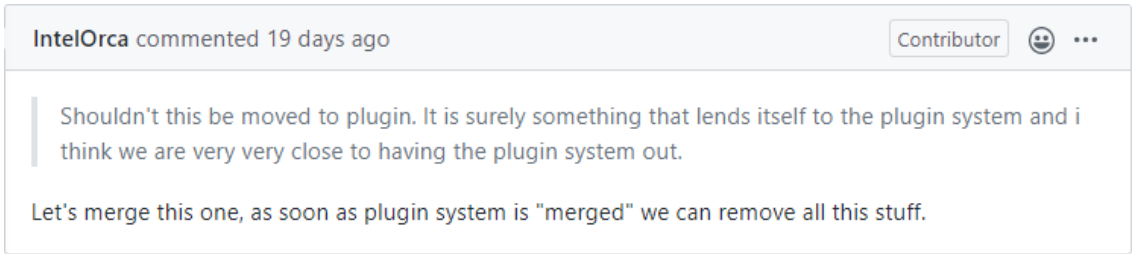
This contribution concerns the addition of a command to the in-game console. In the comments of this pull request, a core developer mentions a potential problem in multiplayer games:



Another core developer comments that this command could perhaps be offered as a plug-in for the new upcoming plug-in system:



Finally, *IntelOrca* (the main developer) responds:



Although this example is no hard evidence of the importance of code quality and technical debt of OpenRCT2, it does offer some insight as to how the core developers value these aspects and discuss them. To illustrate this point even further: as of March 21st 2020, there are `5180` closed pull requests, of which `3066` have at least one comment, and `1238` have more than 5 comments.

20.4.3 Assessment of test process quality



The OpenRCT2 test processes consist of CI checks, unit tests, and manual testing by developers and players. Numerous questions about these test processes arise. Is the quality sufficient? Are the tests adequate?

To start, we will look at the test to code ratio to get a general feel for the adequacy of the unit tests. The number of lines in `.cpp` files in the project's `/src` directory equals `443938`, while this number in the `/test` directory equals `28349`. This yields a test to code ratio of `6.39%`. However, the `ride` component contains many lines of code which should be considered data, and not functional code. This is due to many files in the component containing data on ride properties.

Excluding `ride` results in a test to code ratio of `13.87%`. As `ride` consists partly of data and partly of functional code, this suggests that the real test to code ratio is between these two boundaries.

Should OpenRCT2 spend more resources on testing? It might be wiser to first implement the plug-in system. This would allow continued development of new features and modifications that do not have to be included in the base game. The benefit would be that these functions can be tested seperately, and that this would limit the main functionality of the base game that needs to be tested.

Furthermore, one could argue that small bugs are not a huge problem in a game like OpenRCT2. Large bugs that ruin the end-user's experience, such as the game crashing every hour, should get very high priority. However, once the game is playable and enjoyable by the vast majority of players, occasional small bugs might not have a large negative effect on their experience.

Our recommendation is that the developers of OpenRCT2 keep listening to the community's wishes, and actively request their thoughts and opinions. If many complaints are heard about an aspect of the game, efforts can be concentrated on fixing bugs and improving tests of those components. Additionally, we applaud the goal of developing features such as the plug-in system.

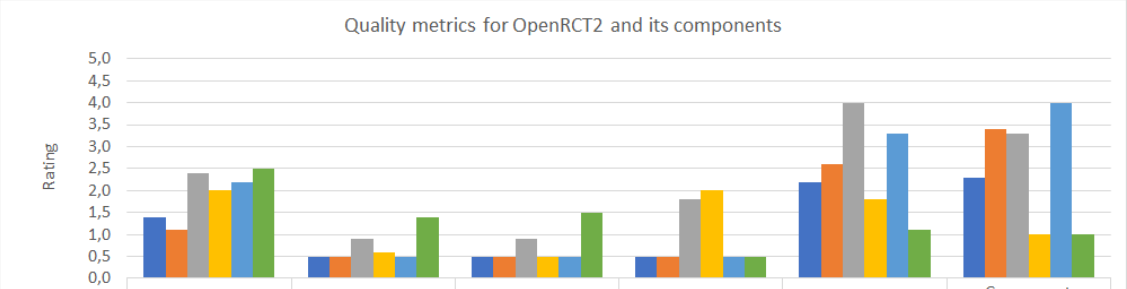
20.4.4 Assessment of code quality and maintainability

Although no single metric can definitely measure code's quality and maintainability, several metrics can give an overall indication.

For clarification, *unit interfacing* refers to the number of unit (method) parameters. Also, a module refers to a file and a component is a collection of modules (files).

The OpenRCT2 project is fairly large, with many architectural components that were discussed in our previous essay. Therefore we have chosen to assess the overall system and the five components that are

most likely going to be affected by future changes based on their volume (lines of code) and the OpenRCT2 roadmap. The corresponding metrics were obtained using Sigrid and visualized in the following diagram.



	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling	Component independence
OpenRCT2	1,4	0,5	0,5	0,5	2,2	2,3
Ride	1,1	0,5	0,5	0,5	2,6	3,4
Windows	2,4	0,9	0,9	1,8	4,0	3,3
Peep	2,0	0,6	0,5	2,0	1,8	1,0
Actions	2,2	0,5	0,5	0,5	3,3	4,0
Drawing	2,5	1,4	1,5	0,5	1,1	1,0

Figure 20.4: Quality Metrics for OpenRCT2 and its components

Based on the results of the measurements, we suggest some refactoring candidates. Using Sigrid, we can see dozens of potential refactoring candidates, but we will only list a couple to support the findings.

It can be seen that the project and its components scores low on *duplication*, so there is a lot of duplicated code present. A good refactoring candidate is the following:

- Remove duplicated code on lines 360-637 in `ImageImporter.cpp` of the `drawing` component and lines 710-987 in `CmdlineSprite.cpp` of the `engine` component.

Even though the score for *duplication* is low, it is important to note that the metric is influenced by the presence of data files, such as the `TrackData.cpp` in the `ride` component. It contains information about which track elements are available for various rides. It contains arrays that are filled with mostly `-1`'s, which are then detected as duplicated code by Sigrid, even though this is not the case.

Furthermore, *unit size* seems to be a problem throughout the entire project, for instance, several methods with hundreds lines of code can be found in the in the `ride` component. The `vehicle_update_track_motion_mini_golf` method in `Vehicle.cpp` contains 661 lines of code, which makes it difficult to understand. It is important to note that OpenRCT2 is a re-implementation of the original RCT2 game, some methods like this one have been added but have not been subject to refactoring.

As can be seen from the diagram, OpenRCT2 and its components score very low on *unit complexity* and *unit interfacing*. This probably explains the lack of tests throughout the project, since it is difficult to write tests for complex methods. We suggest the refactoring of the following two methods:

- The `vehicle_update_track_motion_mini_golf` method in `Vehicle.cpp` of the `ride` component, since it has a McGabe complexity[18] of 111. In contrast, the recommended maximum McGabe

[18] T. J. McCabe, "A Complexity Measure," in IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.

complexity is 15[19].
- The `peep_pathfind_heuristic_search` method in `GuestPathfinding.cpp` of the `peep` component, since it has 10 parameters and a McGabe complexity of 95.

On *module coupling*, OpenRCT2 and its components score a little higher than on other metrics, but the score is still low. Several modules have several hundreds of dependencies. We suggest refacoring those modules, including:

- The `BolligerMabillardTrack.cpp` file in the `ride` component, since it has 359 incoming dependencies from other other modules within `ride`.

Another suggestion is to increase the use of documentation within the code. Very few units and modules contain a description of its concern. This makes it harder, especially for new developers, to make contributions.

A more general suggestion is refactoring components to reduce interdependencies of components. The components of the OpenRCT2 project have a lot of interdependencies. This interdependency has probably been inherited from the original implementation of RCT2, which will briefly be discussed in the last section.

20.4.5 Coding activity

The most frequently changed components in the past two weeks are `ride`, `windows`, `ui`, `actions`, `object`, `network` and `engine`.

Recent coding activity is seen in the following components: `ride`, `windows`, `world`, `actions`, `scenario`, `rct1`, `rct2`, `peep`, `interface`, `ui`, `platform`, `paint`, `object`, `audio`, `drawing`, `management`, `network`, `title`, `localisation`.

Due to active work on a new standard for references to screen space, there have been frequent changes to `windows`, `rct1`, `rct2`, `action` and `ride`. More activity is expected in these components as work on this new standard progresses. Additionally, we expect changes will need to be made to `interface`.

The following is a subset of the system's roadmap and its mapping onto architectural components. It explains the biggest features and their dependency on components.

- *New save format*: currently, OpenRCT2 uses the save format SV6, the native save format used by RCT2. SV6 imposes many limits such as limits on the number of rides, sprites, or map size. Hence, a new save format is being developed to allow limits to be removed or increased. The components involved are `rct1`, `rct2`, `ride`, `action`, `world`, `scenario`, `config`, `windows` and the cheats file.
- *Scripting/plug-in API*: currently, features have to be built directly into the game, which can clutter the source code. API scripting decouples extra features from the core architecture, so it is not influenced by additional features other people implement. Implementation of this feature will require changes to many components, as the feature is so fundamental.

20.4.6 Technical debt

OpenRCT2 is a reverse-engineered version of a game from 2002, a game from an era where developers did not send out day-one patches or monthly updates over the internet. Therefore, OpenRCT2 inherited some

[19] Arthur H. Watson; Thomas J. McCabe (1996). "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric". NIST Special Publication 500-235

```
/**
 * I think this function computes ride upkeep? Though it is weird
 *  rct2: sub_65E621
 * inputs
 * - edi: ride ptr
 */
static uint16_t ride_compute_upkeep(Ride* ride)
{
    // data stored at 0x0057E3A8, incrementing 18 bytes at a time
    uint16_t upkeep = initialUpkeepCosts[ride->type];
```

Figure 20.5: A function in the ride ratings module

technical debt from the original game, in the form of bugs and issues but also code architecture. There still exist some functions that are decompiled versions of assembly code, the functionality of which can also be unclear.

There is also debt in the form of a somewhat limited amount of automated testing. The continuous integration catches code that does not build but errors can still get through that. It would require people to play the game and find a bug to let the developers know it even exists.

The issues page on Github currently contains `1519` open issues, with the oldest being opened the first of September 2014. This can also be considered a form of debt, as the issues are filled with topics that are no longer relevant, or buried so far they will likely never get a resolution. It is important to mention that a large fraction of these issues consist of feature requests and (duplicate) backtrace crash reports. Hence, the number of open issues regarding bugs is much smaller.

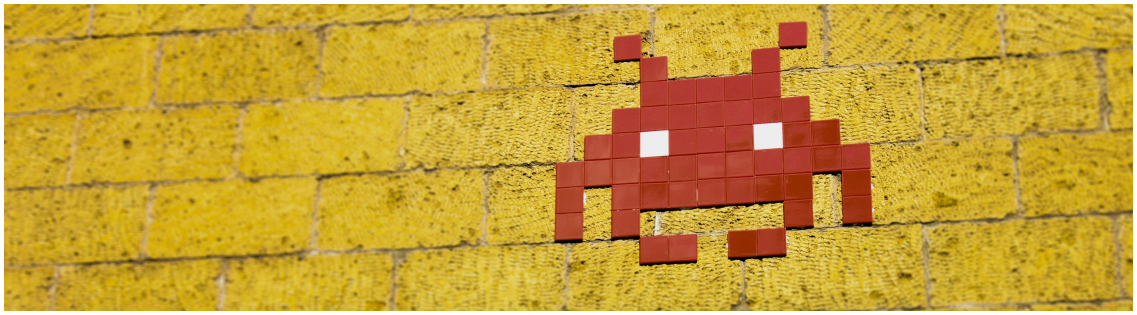
However, after these negative points, the team is very aware of this debt and is working on removing it. Evidence of this can be found in the repository indicators, as mentioned in one of the preceding sections.

In conclusion, there is work to be done in terms of relieving technical debt, but the current efforts going into the new save format and refactoring the references to screen space show that the team is dedicated to making and maintaining a quality architecture.

20.5 Developing fun

A query for *"video game architecture"* in the library of the Institute of Electrical and Electronics Engineers delivers 446 results, whereas the query *"software architecture"* delivers a whopping 83,009 results. Clearly, video game architecture is a less researched aspect of software architecture, even though there are over **2.5 billion** people who play video games around the world. In this essay we delve into the world of video game architecture and try to lift the veil of mystery of designing these popular software systems.

20.5.1 Requirements



The first step in any software project is defining the requirements, or at least it should be. However, defining requirements and designing the software are not so straightforward for video games, compared to other software.

20.5.1.1 Fun-ctional requirements

The quintessential requirement for video games is to be "fun". This requirement alone makes video game development very different from other software. Ask two software engineers to define what is fun and you will get two vastly different answers. Now ask an artist what fun is and they will also explain it in a vastly different way. A design may sound like fun, but it might turn out to be very dull. This is why functional requirements are generally less suitable for video games.[20]

Luckily for OpenRCT2, the formula of the original has already proven itself in terms of being fun. There are also two decades worth of players that can share potential improvements and other feedback with the developers. This means OpenRCT2 is actually able to formulate more functional requirements, like having multiplayer.

20.5.1.2 Non-functional requirements

Video games tend to have stricter requirements for optimization.[21] RCT2's architecture is greatly influenced by the necessity for it to be very efficient. The fact that it is entirely written in assembly should say enough about that. OpenRCT2 does not have strict performance requirements, since most users will have very powerful systems compared to what was required for the original. That being said, there are players having performance issues and the developers do take time to optimize the game performance-wise.[22]

Another requirement that differs from most software is that games can have the requirement to run on multiple platforms. These platforms are not just PC but can also be console or mobile. This also applies to OpenRCT2. The original only ran on Windows and Xbox, but one of the goals of OpenRCT2 was to make it available for macOS and Linux as well. Currently, the project even supports an Android build.

[20] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). *Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?* Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. doi: 10.1145/2568225.2568226

[21] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). *Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?* Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. doi: 10.1145/2568225.2568226

[22] https://github.com/OpenRCT2/OpenRCT2/issues/7908 Performance-related issue on github

Configuration management becomes more important when you support more platforms.[23] The OpenRCT2 continuous integration needs to check whether changes compile and pass tests on multiple platforms.

Lastly, a non-functional requirement for video games is that there is generally more pressure on releasing as quickly as possible.[24] This also has effects on the quality of the documentation and design. OpenRCT2 is open-source, so it does not have this pressure. However, we think most developers would prefer creating new functionality over documentation.

20.5.2 Architectural design



The second step in a software project is to use the defined requirements to come up with a design. There are many ways to design software systems. Some notable designs include the notorious *spaghetti code*[25] and the *big ball of mud*[26], a software system without a perceivable underlying architecture. Both are examples of designs that make systems difficult to change or maintain, so it is important to avoid these kind of designs.

A survey among video game developers showed less time is put into design because of the *fun* requirement. Participants noted there is less time spent thinking about architecture because the system is very likely to change after the initial design. Especially for one-off games, architecture is deemed less important.[27]

For a series, more thought is put into the architecture since components can be re-used in the next game. Rollercoaster Tycoon (RCT) 2 is part of a series and also re-used parts of the first game.[28] Moreover, before Chris Sawyer developed RCT, he developed Transport Tycoon. Since that is also a simulation game, parts of the design could be used in the RCT games as well, so the architecture can be considered quite "mature". OpenRCT2 is easier to develop because of that. For example, RCT2 already contained a *GameAction* system, which queued user commands and executed them from a central component. This made multiplayer gameplay relatively straightforward to implement.[29]

In a previous essay, we showed that the architecture of OpenRCT2 closely resembles the model-view-controller (MVC) architectural pattern. The MVC pattern is frequently used in the design of games and it

[23] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). *Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?* Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. doi: 10.1145/2568225.2568226

[24] Callele, D., Neufeld, E., & Schneider, K. (2005). *Requirements engineering and the creative process in the video game industry.* 13th IEEE International Conference on Requirements Engineering (RE05). doi: 10.1109/re.2005.58

[25] https://en.wikipedia.org/wiki/Spaghetti_code

[26] Foote, Brian; Yoder, Joseph (26 June 1999). *Big Ball of Mud*. laputan.org. Retrieved 2 April 2020

[27] Callele, D., Neufeld, E., & Schneider, K. (2005). *Requirements engineering and the creative process in the video game industry.* 13th IEEE International Conference on Requirements Engineering (RE05). doi: 10.1109/re.2005.58

[28] https://www.gamespot.com/articles/rollercoaster-tycoon-designer-offers-first-details-on-new-title/1100-6092788/ Interview with Chris Sawyer

[29] Interviews with the OpenRCT2 developers

is considered a good base architecture for game development.[30] There are several reasons as to why this architecture is beneficial to the design of video games, including OpenRCT2.

Firstly, it makes it easier to support various input devices.[31] The user input is decoupled from the rest of the system through the *controller* component of MVC. The controller can be seen as an interface that can be implemented for different platforms and a variety of input devices such as joysticks, mouse/keyboard and touchscreens. OpenRCT2 runs on different platforms, including Android, where a touchscreen is the default input device. The decoupling of the user interface (i.e. the *view* compontent) from the rest of the system made it easy to port to Android.

Secondly, MVC makes it easier to support different platforms and hardware. The *view* component of MVC is decoupled from the model and the controller, which allows the view to be adjusted based on the system the software runs on. Some users might have high-end computers, others might run the software on a low-end laptop and some might play the game on a console. All these platforms have different hardware specifications that influence the application's run-time performance. Since the view is independent of the model and the controller, the view can be adjusted accordingly. In OpenRCT2 this is demonstrated through the possibility of using different drawing engines (OpenGL, Software or Hardware based).[32]

Overall, the MVC pattern appears to be a good architectural pattern that can satisfy the unique requirements of video games.

20.5.3 Development process



The third step in a software project is to transform the design into an actual implementation. In this section; we will investigate what causes the development process for game development to be different from non-game development, we will highlight in which ways these differences manifest, and how the development process of OpenRCT2 compares to the rest of the industry.

An obvious example of how game development differs from non-game development is the composition of the development team. As multimedia assets such as audio and graphics are an essential part of a video game, these development teams usually consist of both programmers and artists.[33] This increases the importance of proper management, and also raises the question on how teams are best formed. Combining programmers

[30] T. Ollsson, D. Toll, A. Wingkvist and M. Ericsson, *Evolution and Evaluation of the Model-View-Controller Architecture in Games,* 2015 IEEE/ACM 4th International Workshop on Games and Software Engineering, Florence, 2015, pp. 8-14.

[31] T. Ollsson, D. Toll, A. Wingkvist and M. Ericsson, *Evolution and Evaluation of the Model-View-Controller Architecture in Games,* 2015 IEEE/ACM 4th International Workshop on Games and Software Engineering, Florence, 2015, pp. 8-14.

[32] https://github.com/OpenRCT2/OpenRCT2/wiki/Settings-in-config.ini

[33] C. M. Kanode and H. M. Haddad, *Software Engineering Challenges in Game Development,* 2009 Sixth International Conference on Information Technology: New Generations, Las Vegas, NV, 2009, pp. 260-265.

and artists into a team can offer the benefit of improved communication.[34] On the other hand, not combining them could aid the sharing of specialized knowledge.[35]

Another difference can be seen in the preproduction stage. This stage is made more complicated partially by the less formal requirements of a game. Before the formulation of a (technical) specification, a *Game Design Document (GDD)*[36] is often made. High-level artistic elements such as a game's story, characters, and general game-play are described in this document. A more usual specification is made afterwards. Therefore, there is an extra step compared to the preproduction stage of a typical software project.

Although a thorough preproduction stage is paramount for a succesful game development process,[37] errors in this stage are very common in game development.[38] There are multiple causes of game developers not spending enough time and energy on the preproduction stage: it is common and expected that requirements are discovered when moving from preproduction to production,[39] pressure leads to short-term thinking,[40] and there are indications that in software development culture energy spent on planning and designing architecture is not rewarded much.[41] This could be a major cause of so few projects being completed within the projected budget and time.

Agile processes are common[42] in game development. In contrary, standard waterfall models are not.[43] This can be explained by the need for iterative[44] development: prototypes and iterations are used to 'find' the fun, and to fine-tune game-play.

OpenRCT2's development process is atypical in numerous ways. Due to the goal of recreating the original game, there was no need for much artistic work or preproduction. Moreover, the majority of early development has been single-handedly done by *IntelOrca*. Early development consisted mostly of reverse-engineering, which is a very specific development process. As this recreates the original game, not much thought has to be put into architectural decisions.

The projects that aim to provide free replacement graphics and music for OpenRCT2 are not yet active. Attracting artists to work on these projects would incorporate them in the development process. On the other

[34] C. M. Kanode and H. M. Haddad, *Software Engineering Challenges in Game Development,* 2009 Sixth International Conference on Information Technology: New Generations, Las Vegas, NV, 2009, pp. 260-265.

[35] C. M. Kanode and H. M. Haddad, *Software Engineering Challenges in Game Development,* 2009 Sixth International Conference on Information Technology: New Generations, Las Vegas, NV, 2009, pp. 260-265.

[36] Callele, D., Neufeld, E., & Schneider, K. (2005). *Requirements engineering and the creative process in the video game industry.* 13th IEEE International Conference on Requirements Engineering (RE05). doi: 10.1109/re.2005.58

[37] Callele, D., Neufeld, E., & Schneider, K. (2005). *Requirements engineering and the creative process in the video game industry.* 13th IEEE International Conference on Requirements Engineering (RE05). doi: 10.1109/re.2005.58

[38] Callele, D., Neufeld, E., & Schneider, K. (2005). *Requirements engineering and the creative process in the video game industry.* 13th IEEE International Conference on Requirements Engineering (RE05). doi: 10.1109/re.2005.58

[39] Callele, D., Neufeld, E., & Schneider, K. (2005). *Requirements engineering and the creative process in the video game industry.* 13th IEEE International Conference on Requirements Engineering (RE05). doi: 10.1109/re.2005.58

[40] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). *Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?* Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. doi: 10.1145/2568225.2568226

[41] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). *Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?* Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. doi: 10.1145/2568225.2568226

[42] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). *Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?* Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. doi: 10.1145/2568225.2568226

[43] C. M. Kanode and H. M. Haddad, *Software Engineering Challenges in Game Development,* 2009 Sixth International Conference on Information Technology: New Generations, Las Vegas, NV, 2009, pp. 260-265.

[44] C. M. Kanode and H. M. Haddad, *Software Engineering Challenges in Game Development,* 2009 Sixth International Conference on Information Technology: New Generations, Las Vegas, NV, 2009, pp. 260-265.

hand, translators have been part of the development process relatively early on and are actively maintaining 19 translations.

Arguably, OpenRCT2's development process is more similar to that of other open-source projects. The majority of development is done by a small set of core developers, while users are encouraged to get involved. Communication among and between developers and users occurs online.

20.5.4 Maintenance



After the development process, the final step in a software project is to maintain the implementation. Over the years, the maintenance process of game software has been changing. In the past, conventional video game consoles didn't have any maintenance period. When online services such as Xbox Live for the Xbox came, the developers maintained their software through downloadable patches. In PC development, developers wait for a period of time to receive as many bug reports as possible and then work on a patch.[45] A patch may fix bugs, add features or alter gameplay. In the case of massively multiplayer online (MMO) games, the shipment of the game is the starting phase of maintenance. These are in continuous maintenance as new features are added and the game-world is changed continuously.[46]

Software maintenance is delayed in games more often than in non-game software. One of the reasons for this is that there is a chance that the game may not be a success. If maintenance of a game starts early, all efforts put in may go to waste. Conventional software requires continuous maintenance and competes in the market based on new features, but that's not the case for some games. Hence this can be another reason for lack of maintenance.[47] The cloud changes the way games are maintained. These days, the game maintenance process is starting to look similar to non-game software.[48]

The original RCT game which was played on a console or a Windows PC, had very few patches. Now, the OpenRCT2 software is a little similar to conventional non-game software. Development in OpenRCT2 is quite active. Maintenance of software also goes hand-in-hand. Reports of bugs are actively monitored and fixed along with altering or adding new features to the game. Every now and then, after certain number of fixes and new feature additions, there is a new release of the game.[49] Furthermore, a launcher is available, which automatically updates OpenRCT2 and keeps the latest stable build up-to-date.

[45] https://en.wikipedia.org/wiki/Video_game_development

[46] https://en.wikipedia.org/wiki/Video_game_development

[47] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). *Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?* Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. doi: 10.1145/2568225.2568226

[48] Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). *Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?* Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. doi: 10.1145/2568225.2568226

[49] https://openrct2.org/changelog

20.5.5 Conclusion

Video games have different, less-formal requirements compared to non-game software, such as the need to be fun. This leads to significant differences and specific challenges in choosing architectural designs for games, and for the development processes of games. Due to OpenRCT2 being a reimplementation of a pre-existing game, its architecture was already defined based on the requirements set by RCT's original developers. OpenRCT2's development and maintenance processes are mostly shaped by its reverse-engineering origin and open-source nature.

20.6 On OpenRCT2's Frontlines

In the previous essays we have analyzed OpenRCT2's architecture, but who can give better insights than OpenRCT2's developers themselves? We have asked the OpenRCT2's developers questions relating to OpenRCT2's architecture, the project itself and their experiences while developing for it.

20.6.1 Questions and answers



Q1: Could you tell us about some difficult architectural decisions that had to be made when developing OpenRCT2?

A1 - duncanspumpkin: *"The main difficulties has always been maintaining the compatibility and feel of RCT2. In the early days of the project we also had to maintain memory compatibility. It meant that global variables had to be assigned the exact same memory address as vanilla. For some functions we had to patch the game on the fly which meant we also had to maintain calling convention (Chris Sawyer calling convention is a little unique). Its for these reasons the initial codebase was in C. It was much easier to handle the interfaces between the assembly and the OpenRCT2 code. You can still see that the majority of the codebase is written in almost C like code. After we had implemented the whole game and RCT2 no longer required to be in memory we gained the ability to change memory layout and slowly moved to C++. That is one of the main reasons why the codebase still has many global variables and restrictions. The main architectural differences are with loading/saving; window management; game actions. The loading and saving in RCT2 was pretty much a memcpy from a file into memory. As we want to change memory layout we have to read each individual field, convert it if required to new type, write into memory. In theory this is much safer than RCT2 but it has the trade off of a considerable increase in amount of code. Window management has moved to an intents system to decouple the ui from the base game. This allows the servers to no longer require processing of windows and such. Game actions is our replacement of Game commands. Game commands were very restrictive as to how much data could be stored in them and CS liked to pack his data in many different ways to get around that issue. We are very greatful that CS implemented the game commands like he did as it meant making things multiplayer was not too difficult. The game actions are much easier to understand and they are one of the few areas of the codebase that is actually written as C++. Although admittedly they aren't the best examples of good code."*

A1 - Gymnasiast: *"I haven't been involved in architectural decisions much. I think that's something that IntelOrca, Duncanspumpkin, janisozaur and ZehMatt can tell you more about. What I can say is that initially, OpenRCT2 didn't really have much of an architecture. It simply implemented the RCT2 subroutines one by one as separate C functions, creating some huge files of mostly related functions in the process. But since then, there has been a lot of refactoring, including splitting the engine from the UI and slowly moving code to object-oriented C++ rather than just C functions."*

Q2: In what ways does OpenRCT2 deviate from the original RCT2 architecture and why?

A2 - Gymnasiast: *"The above-mentioned split between engine and UI and the object-oriented approach are two important ones, I think. The original RCT2, being written in assembly, was mostly procedural and monolithic. We have done our changes to make the code more robust and easier to maintain and extend, and also to allow using the engine for other purposes than simply playing the game, like setting up a headless server with a minimum of overhead for example. It's also possible to use the engine in other projects now."*

A2 - janisozaur: *"The original architecture of RCT2, or rather the implementation detail thereof, code written in x86 asm turned out to be a blessing when porting the project to Linux.*[50] *As there's just a handful of calls to system or any external libraries from the game, I managed to create a process to host some memory at expected layouts and then load original binary there. From inside the asm code, there is no difference what the operating system is, as long as the memory layout is what the game expects. This attracted some more developers and was used for creating OS X builds in similar fashion.*

Inside RCT2 virtually all the actions affecting game logic were sent to a single dispatch mechanism which then executed relevant handlers. This was used by other team members to create a multiplayer, akin to one found in OpenTTD or CS's Locomotion, usually referred to as "lockstep simulation". Initial implementation used to have permissions which had to be granted to each new client upon joining. With frequent crashes, disconnects and inherently long play sessions of multiplayer, this meant server hosters were left to either spend lots of time assigning permissions manually, default to allow new clients to build (and remove) things in park which invited vandalism, or password-protect their servers. I set out to create a system[51] *that would address all those issues at once and offer persistent permissions based on PKI. This was well-received and lives on in the code to this day.*

One major improvement OpenRCT2 has over the tight coupling of RCT2 is how we split it to some sub-components.[52][53] *That work basically splits the project into 'OpenRCT2 client' and 'libopenrct2', the former provides user-facing part, latter has all the game logic in it. This allowed us to create lots of utilities inside the project relatively easy. Based on this work we were able to add Google Test-based test suite, benchmarks, format converters, headless clients..."*

Q3: What are the main trade-offs that needed to be considered during development of OpenRCT2?

A3 - Gymnasiast: *"Mostly ease of coding versus performance. We are lucky to have people like janisozaur and ZehMatt on the team, who are good at squeezing out more performance."*

---

[50] https://github.com/OpenRCT2/OpenRCT2/pull/1956
[51] https://github.com/OpenRCT2/OpenRCT2/pull/3699
[52] https://github.com/OpenRCT2/OpenRCT2/pull/5336
[53] https://github.com/OpenRCT2/OpenRCT2/pull/5458

Q4: Are there any parts of the architecture you would have designed differently in the original game?

A4 - Gymnasiast: *"Chris Sawyer had good reasons to develop RCT and RCT2 in assembly: he was used to it and it made the game perform extremely well, even on computers that were very old for 90s standards. I have personally run RCT1 on a 1995 computer and RCT2 with half the minimum specified RAM. That's no mean feat, and I think his decisions paid off. So I'd say he made the right call."*

Q5: In what way did your (work/study) background help with developing this project?

A5 - Gymnasiast: *"Apart from general programming knowledge, not a whole lot. I never touched any C or C++ before this project, coming from a PHP background, and suddenly having to deal with memory management, string handling and other stuff that higher-level languages such as PHP and Java so kindly abstract away for you, that was quite a steep learning curve initially."*

A5 - janisozaur: *"During my studies I majored in 'Games and Computer Simulation', but never really wanted to get into gamedev. I did follow it as my passion, though, as I often inspected new releases of emulators, game engines (such as VCMI, DesMuMe...) and was always into ways of improving their performance. These days I find a lot of inspiration in OpenRCT2 that helps me a lot in my day job. I can freely test out new concepts, coding techniques, tools on OpenRCT2 code and I can spend as long as I want or need on any given task."*

Q5: Do you have any tips for (new) developers wanting to contribute to open-source projects like OpenRCT2?

A5 - Gymnasiast: *"Read the available documentation and join the chat, which almost all projects have. Ask any questions you have there. I like to think that we are a very friendly open source project and don't mind explaining everything, but some are a little bit more strict and expect people to carefully read the documentation and rules in advance.*

When picking out your first contribution to make, take a simple one. Some projects (like ours) have labelled some issues 'good first issue' or something similar. In other cases, you can ask on their development chat. My first contributions were simply bug reports.

My first code contribution was simply making two hardcoded strings translatable. It grew from there. I also got acquainted with the code base by just browsing through and sometimes modifying code, just to see what happened."

Chapter 21

RIOT



Figure 21.1: RIOT: The friendly Operating System for the Internet of Things

Thanks to the IoT revolution of the last few years, the number of embedded systems is growing rapidly. The ever-increasing demand for functionality has provided a challenge in the development of these systems. The code is no longer simple and using a single task, instead it is complex and often consisting of multiple tasks which need to be scheduled. Besides the complexity of the code itself, these systems require a (constant) Internet connection and very low power consumption. To allow efficient development for this kind of applications, a specifically targeted operating system could be of great value.

RIOT is such a real-time embedded operating system, which provides in a lot of these requirements for IoT applications. The roots of RIOT date back to 2008 and the project has been subject to active, open-source development since 2013. RIOT aims to provide *the full set of features expected from an OS, ranging from hardware abstraction, kernel capabilities, system libraries, to tooling.*[1] The significance of this hardware

[1] E. Baccelli et al., "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," in IEEE Internet of

abstraction is proven by the multitude of supported hardware architectures such as AVR, MSP430, ARM Cortex-M and ESP32. Besides supported embedded hardware, there is also a native port of RIOT, which allows development and testing without even having the actual hardware of a target platform at hand.

We will be looking further into the software architecture, vision, design choices and overall structure of RIOT during the course and its accompanying assignments. Based on the findings we aim to provide useful insight in and contributions to the development of RIOT.

21.1 About us

We are four master students from Delft University of Technology, with backgrounds in Computer Science and Embedded Systems.

- Koos Habte
- Yuxiang Liu
- Millen van Osch
- Michael Treffers

21.2 RIOT: The future of IoT

The number of IoT devices powering our daily lives becomes larger every day. On top of that the applications running on these devices become more and more complicated. To keep up with these developments, the developers of such systems need proper tools. An Operating System is an essential part, and provides a lot of basic building blocks. RIOT is such an OS, it's feature-rich, open-source, adopted by academics and under active development.

The Figure[2] below shows a timeline of relevant developments in relation to RIOT. It shows Linux, which the RIOT community considers as an example for open-source development. A few competing OSes and relevant technical developments are listed to indicate why and how RIOT came about.

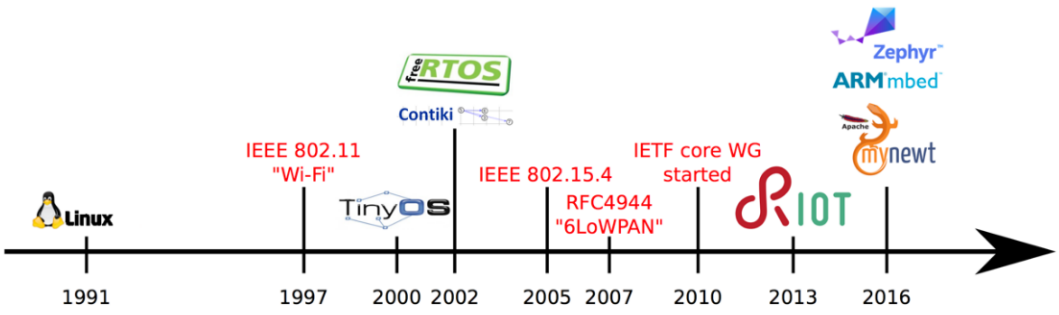


Figure 21.2: Timeline of relevant developments in relation to RIOT

This post will explore what RIOT is, what it tries to be and who it is for.

Things Journal, vol. 5, no. 6, pp. 4428-4440, Dec. 2018.

[2] https://riot-os.github.io/riot-course/slides/01-introduction/#9

21.2.1 What is RIOT?

RIOT is a real-time embedded operating system aimed at the ease of development and portability of IoT applications. RIOT can be compared to other operating systems aimed at embedded devices like Zephyr, mbedOS, Contiki-NG and FreeRTOS.

While an operating system can provide a lot of useful features for the development of IoT applications compared to bare metal programming, it has to consider the limited resources of IoT devices. Thus, one of the most fundamental features of an embedded OS is its resource management. Another important feature is support for various networking technologies and protocols which would be applied in IoT applications.

RIOT provides support for an extensive list of networking technologies and protocols which are widely used like Bluetooth, LoRa, 6LoWPAN and MQTT and many more. The full list can be found in RIOT's documentation[3].

The Figure[4] below shows a high-level overview of the structure of RIOT. Important to note is the separation between hardware-*independent* and hardware-*dependent*. Development of generic functionality only has to be done once and will work for all supported hardware platforms. This results in a small amount of hardware-specific code, which eases the adoption of RIOT on new hardware platforms.

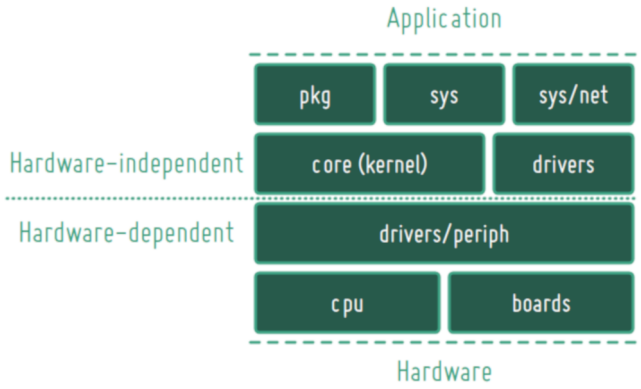


Figure 21.3: Structural elements of RIOT

Another result of this hardware abstraction is that it provides the opportunity to optimize the configuration of RIOT for individual devices. Since each cpu and board have their own configurations, a developer can disable certain unsupported or unused features of the OS. This reduces compile time and more importantly reduces the footprint of the OS on the device's memory.

Overall, the currently implemented features show that RIOT tries to provide a full-fledged embedded operating system, or as defined in a paper co-authored by some of the main developers of RIOT: "*RIOT provides the full set of features expected from an OS, ranging from hardware abstraction, kernel capabilities, system libraries, to tooling.*"[5]

[3] https://doc.riot-os.org/group__net.html
[4] https://doc.riot-os.org/index.html#structure
[5] E. Baccelli et al., "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," in IEEE Internet of Things Journal, vol. 5, no. 6, pp. 4428-4440, Dec. 2018.

21.2.2 End-user mental model

The end-user mental model of a software product is a model to describe the end-users' cognition when using this software product.[6] In this section, we would briefly analyze the end-users' behaviour of RIOT from an architectural perspective.

RIOT is designed to be deployed on mainstream IoT devices. Its target end-users are mainly engineers from IoT or related fields and technology enthusiasts. In other words, we would expect the end-users to have a certain level of professional knowledge of IoT and embedded software. Our end-user mental model is described as below:

- The end-users should understand how to identify the type of board they are working with and they would be satisfied if they could easily locate the compatible version of RIOT with their boards.
- The end-users usually work on low-level hardware programming and they would be familiar with C.
- They would be looking for a short manual when they are trying to set up and give a test run of the RIOT system.

- They would expect RIOT to support wireless communication protocols that are heavily used in IoT development.

- The user would expect basic real-time OS features as IoT devices are often involved in real-time applications.
- The user would need to test multiple instances on a single physical machine and in this case virtualization support may be required.
- Since the end-users also have the knowledge of software development, they may want to contribute to the RIOT project. A guide on how to contribute and the format of contributions may be necessary.

21.2.3 Stakeholders

According to chapter 3 of Lean Architecture[7], there are 5 major stakeholders: the end users, the business, the customers, the domain experts and the developers. Since RIOT is open-source software, there is much overlap in these areas and some do not really apply at all.

The customer stakeholder area is not really applicable here since Lean Architecture defines the customer more as a middle man who sees the software as a product that passes through their systems. They might repackage it but in the end it is still the same product. Since RIOT is open source it goes from source directly to the end users.

The business stakeholder area is not really applicable here since there is no single organisation working on this product. It is developed by an international community of developers, there are no managers, no board of directors or weekly meetings. A business cares about serving customers and increasing revenue, but since there aren't really customers or revenue, this stakeholder is not really applicable.

In the case of RIOT, the end user, developer and domain expert stakeholder areas have a lot of overlap. Because RIOT is open source any end user could also be a developer or domain expert or both, and the same for developers and domain experts. Anyone who wants to use RIOT in their embedded platform can be seen as an end user, and the benefits of this system is that RIOT is a low memory use operating system. But in contrast to other low memory use operating systems RIOT allows for application programming with

[6] Coplien, James O., and Gertrud Bjørnvig., "Lean architecture: for agile software development", John Wiley & Sons, 2011.
[7] Coplien, James O., and Gertrud Bjørnvig., "Lean architecture: for agile software development", John Wiley & Sons, 2011.

the programming languages C and C++, and offers full multithreading and real-time abilities. Since most developers are likely also end users they get the same benefits out of this system on top of the experience of contributing to an open-source system.

According to Lean Architecture, the domain experts are the folks in an organization who know stuff, but because of RIOT's open-source nature they fall more under developers than a separate stakeholder area. They do however get the added benefit of being more likely to steer where the product is going or how it is developed because of their expertise.

Another stakeholder area which is not mentioned in Lean Architecture but we find worth mentioning are academic institutes. A lot of RIOT's early and main developers are from FU Berlin and INRIA, and academic institutes often have a lot a research in IoT devices, sensor networks and other embedded systems. The benefit of RIOT is that it is a nice lightweight open-source platform that might fit the needs of their research.

21.2.4 Roadmap

The development of RIOT started when IoT started to become more and more popular. According to RIOT's vision document[8], at that moment the RIOT founders thought that software platforms for IoT applications would go through a similar evolution as smart handheld devices. Expected were multiple closed-source, slow progress, proprietary solutions at the beginning, after which a couple of big players would create new standards, such as openness and interoperability.

At the moment there are multiple software platforms powering IoT devices. Some of those are closed source and for example made by Android, Windows and Huawei. Next to that there are also multiple open-source software platforms for IoT, for example Zephyr, mbedOS, Contiki-NG and FreeRTOS.

This development is in line with the predictions of the RIOT founders, as stated in their vision. Unfortunately the vision hasn't been updated since 2014, so to give an estimate of the future development, we looked into some task forces[9], open issues[10] and the roadmap[11].

The biggest enhancements RIOT is currently working on is changing the way it is configured by the user, adding more available boards is a high priority and increasing the responsiveness by rewriting some blocking functions.

Furthermore the roadmap provides some specific improvements split in the following subjects:

- Network Stack High layers: Mostly clean up, develop long-term plans and extend CoAP support with blockwise transfer
- Network Stack Low layers: Fully open source support 6LoWPAN over BLE, re-integrate 802.15.4 TSCH, retransmissions by MAC, Point-to-Point Protocol
- Power Modes: integrate generic power management functions in device driver API's, more advanced LPM concepts
- Peripheral drivers: implement i2c_slave and spi_slave
- Software Updates: no unfinished milestones at this moment
- Documentation: write and publish more RDM's, revamp RIOT website
- Low-level Hardware Support: improve MIPS support, radio support for multiple components, ST Nucleo support

[8] https://github.com/RIOT-OS/RIOT/wiki/RIOT-Vision
[9] https://github.com/RIOT-OS/RIOT/wiki/Task-Forces
[10] https://github.com/RIOT-OS/RIOT/issues
[11] https://github.com/RIOT-OS/RIOT/wiki/Roadmap

- Testing: automated network functionality tests, on-board CI testing
- Security: default secure RIOT configuration, 802.15.4 link layer security

21.3 RIOT: From Vision to Architecture

In the previous post we discussed what RIOT is, what it tries to be and who it is for. With this in mind we will focus in a more theoretical manner on some Software Architectural aspects of RIOT.

21.3.1 Architectural views

We've taken the architectural views of Rozanski and Wood. They describe 7 of them in their book Software Systems Architecture[12]. The 7 views are: context, functional, information, concurrency, development, deployment and operational.

`1. Context view`

The context view is relevant to this system, it describes the relations, dependencies and other interactions between its components. Because RIOT is developed by an international community, a view like this can bring new contributors up to speed more quickly. It doesn't explain everything in detail however, but a developer can get the gist of how components in the software are related.

`2. Functional view`

In order to know what components do exactly, a functional view is needed. This view is also relevant to this system for the same reason as the context view. It is very beneficial to get new contributors up to speed but also for existing developers to refresh their knowledge and take design decisions.

`3. Information view`

The information view describes how information is stored, manipulated, managed and distributed. Although memory management is crucial for RIOT to keep the memory footprint as small as possible, this view is more about the management of information in the sense of data. Memory management to keep the memory footprint small is more part of the functional view, and managing data for installing and upgrading falls more under the operational view. Therefore the information view is not as relevant to this system.

`4. Concurrency view`

RIOT offers full multithreading and interprocess communication, so a concurrency view is somewhat relevant to the system. In order to prevent deadlock or corruption of information, having this view to describe how threads are coordinated and how variables are shared is useful.

`5. Development view`

The development view is definitely relevant to the system, it might even be the most relevant view. Since RIOT is developed by an international community, one where anyone can join, it is worthwhile to describe how the development process works. If not, everyone does it the way they think is best which can result in messy and unclear code.

`6. Deployment view`

[12] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

When anyone can download and build the software it is very practical to describe for which environments the software was intended. If someone wants to run RIOT, it is useful to know which hardware you need and what third-party software to install, but this view is also valuable for developers. For them it is helpful to know what third-party software is available for them to work with and for what platforms they should develop. So for RIOT, having a deployment view is relevant.

`7. Operational view`

Because RIOT is under active development, upgrades are bound to happen. RIOT's common use case is to be run on a multitude of different IoT devices, all of which need installation and software updates. And since many devices need over the air updates, describing how the software is supposed to be installed and upgraded will ease this process for the user. Upgrading is more complex than just installing because existing data and settings need to be considered. Considering these challenges, an operational view is valuable to have.

21.3.2 Architectural style

The main architectural style of RIOT is described as a microkernel architecture [13]. Such an architecture consists of two parts, namely the core and plug-in modules. In RIOT's case the core consists of the `core` module and the plug-in modules are `cpu`, `boards`, `drivers`, `pkg` and `sys`. This style is very applicable to RIOT, because it requires the core to be the near-minimum amount of software to implement an operating system. Next to this it has multiple (almost) independent plug-in components to expand the functionality.

21.3.3 Development view

According to Chapter 20 of Software Systems Architecture[14], the Development Viewpoint *describes the architecture that supports the software development process*. This concerns aspects like organizing the code in functional modules, standardization of design and test procedures and control over the build, test and release process.

The Development Viewpoint can be described using six *concerns*:

1. `Module Organization`: organizing the code in functional modules
2. `Common Processing`: isolating a common process for reusability and configurability
3. `Standardization of Design`: using standardized design methods to assure maintainability and reliability
4. `Standardization of Testing`: implementing testing infrastructure and means to automate tests
5. `Instrumentation`: capability to provide logging information
6. `Codeline Organization`: Ensuring overall maintainability and deployability by means of the previous concerns

While all six concerns apply to RIOT, two interesting ones will be highlighted.

The `Module Organization` is described in RIOT's documentation[15], which shows that the code base is structured into five groups. These groups and the actual modules in the code base, are listed below:

- The kernel (`core`)

[13] https://github.com/RIOT-OS/RIOT/blob/master/README.md

[14] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[15] https://doc.riot-os.org/index.html#structure

- Platform-specific code (`cpu`, `boards`)
- Device drivers (`drivers`)
- Libraries and network code (`sys`, `pkg`)
- Applications for demonstrating features and for testing (`examples`, `tests`)

The modules and their high-level dependencies are illustrated in the Figure[16] below. An application will communicate with the hardware-independent modules, which in it turn communicate with the hardware-dependent modules. Finally, the hardware-dependent modules communicate with the actual hardware, thus creating a nice separation of dependencies irrespective of the hardware.

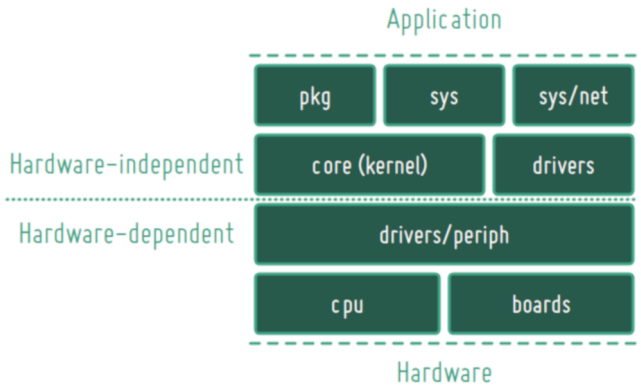


Figure 21.4: Structural elements of RIOT

An example of the `Instrumentation` concern can be found in `core/include/debug.h`. This globally available implementation can be used to print debug messages by setting `#define ENABLE_DEBUG (1)` and using `DEBUG("myMessage")`. Note that a define is used to enable this functionality. If the code used to provide debug information is contained within `#ifdef ENABLE_DEBUG` and `#endif`, it will not be compiled if `ENABLE_DEBUG` is not defined. This provides a flexible way to include debug output for development, but minimize the binary size of production code.

21.3.4 Run time view

The run time view will describe the interaction between different components of RIOT, the user and RIOT, the boot order and the dependencies.

Looking at the schematic below[17], most of the modules defined before can be traced back. Starting at the bottom, the hardware corresponds to the `cpu` and `board` modules. The hardware interacts with 3 different parts, namely the core, network device drivers and the sensor and actuator drivers, via respectively CPU abstractions or peripheral APIs. The core interacts via core APIs with potential libraries and the developed applications. The network stacks are dependent on libraries and interact with the network device drivers and provide a socket to the applications. Finally, the sensor and actuator drivers interact via a sensor/actuator abstraction layer (called SAUL) with the applications.

[16] https://doc.riot-os.org/index.html#structure
[17] https://riot-os.github.io/riot-course/slides/03-riot-basics/#2

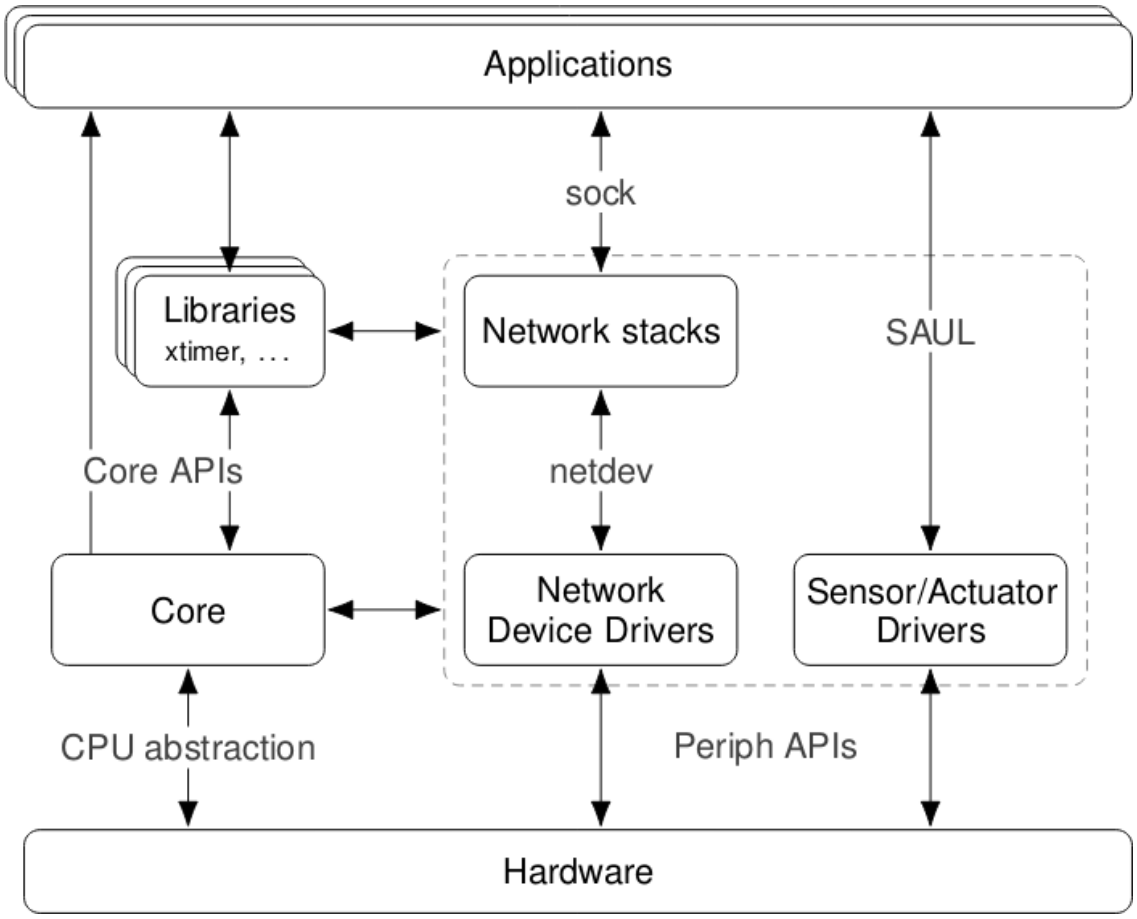


Figure 21.5: RIOT API'S and their relations to each other

Since RIOT is an OS, in almost all cases the interaction with the user is via a user-defined application, that either starts automatically at boot or is called via the shell, or standard RIOT shell commands.

When developing an application, dependencies such as drivers or libraries can easily be included by adding them to the `USEMODULE` or `USEPKG` variable of the application's Makefile. The standard board can be specified in the Makefile, but might also be set in the make command.

The Figure[18] below shows the boot procedure of RIOT. First the board is initialized (cpu, clock, peripherals and optionally on-board peripherals), after which the OS (idle thread and main thread) can be initialized. The main thread might also initiate more threads and add extra commands to the RIOT shell.

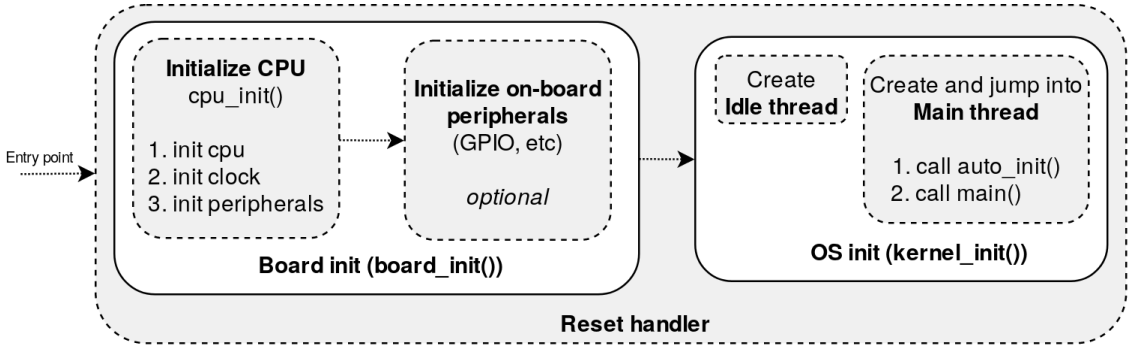


Figure 21.6: Boot sequence of RIOT

21.3.5 Deployment view

Based on Chapter 21 of Software System Architecture[19], the Deployment Viewpoint is a description of the *physical environment* that this piece of software is designed to work in. To be more specific, the *physical environment* includes:

- the hardware or hosting environment
- the technical environment requirements each type of processing nodes has
- the runtime environment of the software

21.3.5.1 Concerns

In the following discussion, we will first provide a general idea of RIOT from a deployment viewpoint while considering these 7 concerns:

`1. Runtime Platform Required`

RIOT is an OS designed for IoT applications, where a computational node capable of running real-time application and wireless communication is required. It could be hosted on three types of computational nodes: IoT devices, open-access testbed hardware and the PC. The physical IoT devices act as its working environment. The open-access testbed hardware and the PC are mainly used as development and testing platforms for the IoT applications build with RIOT.

[18] https://riot-os.github.io/riot-course/slides/03-riot-basics/#7

[19] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

`2. Specification and Quantity of Hardware or Hosting Required`

RIOT supports a variety of typical IoT devices (8-bit, 16-bit and 32-bit micro controllers)[20]. The open-access testbed hardware is mainly referring to the service provided by IoT-LAB. The PC host should be a Linux/FreeBSD/OSX machine in order to run RIOT virtually as a process.

`3. Third-Party Software Requirements`

RIOT has a couple of software requirements to be able to build, flash and debug[21]:. * Build-essentials: make, gcc(depending on board used) * Native dependencies if running in a virtual environment: host dependent[22] * OpenOCD, for debugging purposes

Furthermore developers can use supported libraries while developing an application. These libraries will automaticly be downloaded (and patched) at compile time.

`4. Technology Compatibility`

RIOT is compatible with the following hardware platforms[23]:

- ARM
- ATmega
- ESP
- MSP430
- MIPS
- native
- SmartFusion2
- RISC-V

`5. Network Requirements`

Considering RIOT is aimed at IoT devices it obviously requires networking capabilities. Depending on the type of application, an Internet connection or other networking technologies might be required like Bluetooth or LoRa. To accomodate this, the hardware should provide the necessary networking capabilities.

`6. Network Capacity Required`

RIOT is running as the OS and that implies it actually helps to coordinate network communications.

`7. Physical Constraints`

RIOT is mainly deployed on Embedded platforms and it is light-weighted due to limited storage. Also, power consumption is an important part of IoT applications.

21.3.5.2 Models

`Runtime Platform Models`

This is the core part of the deployment view. The following UML diagram illustrates the runtime platform models for RIOT.

[20] https://doc.riot-os.org/group__boards.html
[21] https://github.com/RIOT-OS/Tutorials
[22] https://github.com/RIOT-OS/RIOT/wiki/Family:-native#dependencies
[23] https://github.com/RIOT-OS/RIOT/wiki/RIOT-Platforms

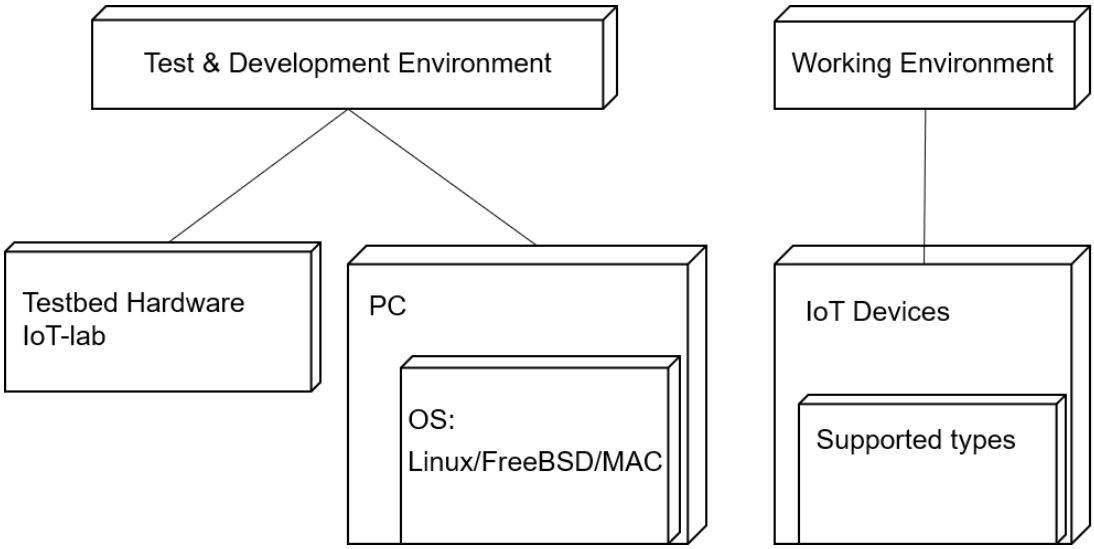


Figure 21.7: Runtime Platform Model

21.3.6 Non-functional properties

According to Chapter 12 of Software Systems Architecture[24] a non-functional property *is a constraint on the manner in which the system implements and delivers its functionality*.

An interesting non-functional property is related to the use of third-party software. To make sure that all required tools for hardware-independent development and testing with RIOT are available, a lot are included in RIOT's code base in `dist/tools`. This ensures that developers have all required tools at hand, and prevents the cumbersome process of finding, compiling, and installing them. However, since the provided tools are snapshots and not updated regularly there might be issues with outdated or buggy versions of a tool. Since RIOT is open-source and everyone can contribute, this can easily be solved by opening a Pull Request with the updated code of a tool.

21.4 RIOT: Quality and Technical Debt

After covering the architectural basics of RIOT in the previous post, we will now see this affects the development quality. This posts covers the way how testing and assessing code quality is currently covered by RIOT and will also address our view on the code quality and technical debt.

21.4.1 Software quality process

In order to maintain code quality, RIOT uses quite a few tools:

`Vagrant`

[24]Taylor, Richard N., Nenad Medvidovic, and Eric Dashofy. Software architecture: foundations, theory, and practice. John Wiley & Sons, 2009.

Vagrant is a tool used to build and manage virtual machine environments. RIOT has a Vagrant file to download a pre-configured Linux virtual machine that contains all the necessary toolchains and dependencies.

`cppcheck`

Cppcheck is a static analysis tool for C and C++. It can be used to detect bugs but has a focus on detecting dangerous coding constructs and undefined behaviour like integer overflows and out of bounds checking.

`coccinelle`

Coccinelle is a utility tool used to match and transform source code. Coccinelle was initially used to perform collateral evolutions, that is when a library API changes the client code needs to change with it, but it can also be used to find defective programming patterns.

`RIOT native`

RIOT native is a hardware visualizer that can be used to emulate hardware at the API level. This helps if developers don't have a supported hardware platform, or want to test an application without the limitations and requirements of actual hardware.[25]

`embUnit`

Sourceforge's Embedded Unit is a unit testing framework whose design is copied from JUnit and CUnit, and adapted for Embedded C systems.

`Valgrind`

Valgrind is a programming tool used for memory debugging, memory leak detection and profiling. Valgrind's tool suite allows developers to automatically detect memory management and threading bugs, but it can also be used to perform profiling.

`GCC stack smashing protection`

Stack smashing protection is a compiler feature in the GNU compiler collection used to detect stack buffer overrun.

`DES-Virt`

DES-virt is a virtualization framework for DES-Testbed, it allows for the configurations in topologies of virtual networks.

`Wireshark`

Wireshark is a packet analyzer. It is used to analyse and troubleshoot networks, help develop software and communication protocols and for education.

`gdb`

GDB, or the GNU project Debugger, is a debugger that allows developers to see what is going on inside a program while it executes or what it was doing when it crashed.

`gprof`

Gprof, or GNU profiler, is a performance analysis tool for Unix applications. It can be used to identify bottlenecks in programs.

[25] RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT

`cachegrind`

Cachegrind is a cache profiler tool in Valgrind's tool suite.
It pinpoints sources of cache misses, identifies the amount of cache misses, memory references and instructions executed for each line of code.

`OpenOCD`

OpenOCD is a on-chip debugging, in-system programming and boundary-scan testing tool which can also be used as a flasher.

`edbg`

Edbg, or now known as CMSIS-DAP is a simple command line utility tool for programming ARM-based MCUs through CMSIS-DAP SWD interface.

`RIOT shell`

RIOT provides a command-line interpreter similar to a shell in Linux, it facilitates debugging and run-time configuration while testing.[26]

In order to use these tools appropriately, the maintainers at RIOT wrote a workflow on how and when to use them:
The first step is an optional step, it is more aimed at new developers for RIOT. In this step you set up a pre configured VM using Vagrant.
The next step is the first real step; check your code using static analysis. For this step they use the tools cppcheck and coccinella.
In step 2 the code should be run on a RIOT native instance, then it's time for dynamic analysis.
This means running Unit tests using embUnit, integration tests, and using Valgrind as well as GCC stack smashing protection to detect invalid memory access.
Step 3 is in case of networked applications or protocols. To test this run several instances of native communicating via a virtual network, and then use Wireshark to analyze the network traffic.
In step 4 they analyze the system state for semantic errors on RIOT native using gbd.
Step 5 is for when there is a suspected bottleneck. Here they use performance profilers like gprof and Cachegrind to detect and identify these bottlenecks.
The next step is for when the code is bug free on RIOT native, then a developer can move on to actual hardware. This means flashing the binaries to the target IoT hardware using a flasher like OpenOCD or edbg.
Furthermore one should use RIOT shell running on the target IoT device for easier debugging.

21.4.2 CI Pipeline

RIOT makes use of tools for their software quality management, and their CI is no different. RIOT's CI consists of the following tools:

`Murdock`

Murdock is a simple CI server written in Python, its job is to act as a bridge between github and RIOT's running scripts.

`dwq`

[26] RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT

Dwq, or Disque Work Que is a tool used to distribute Jobs on git repositories across multiple machines.

`disque`

Disque is a distributed, in-memory message broker. In RIOT's CI it is the backend of dwq.

`Docker`

Docker is a service used to build standardized containers. These containers can be used by developers to isolate their application from its environment.

`SSH`

Ssh is a software package that allows for secure system administration and file transfers over insecure networks. It can also be used to authenticate a client to a server.

`Other`

Aside from the previous tools, the CI also contains some other things such as:
Scripts bridging Murdock build jobs and using dwq to build them on worker slaves.
Some HTML files to nicely present Murdock's state. And a web server proxying HTTPS to Murdock.

Before a pull-request is merged in the master branch, it is first tested via Murdock. Murdock automates the testing of the static tests, unit tests and compile tests for over 15000 build configurations. Furthermore it automates functional tests on selected platforms.[27] By using a Webserver to display Murdock's state, it is easy to see if a build succeeded or if and why it failed. ## Testing RIOT utilizes two ways of testing, namely unittests and functional, interactive tests.

The main goal of the unittests is to ensure proper functionality of main parts of the RIOT code base. These tests are also incorporated in the CI pipeline, to ensure they are never broken after code updates.

The other way of testing in RIOT takes the hardware into account. A lot of RIOT's functionality can't be assessed by simple unittests, but has to be visually checked. An example is an LCD display. To ensure the output is correct, the test, as shown left in the Figure below, will display a few different messages. The user will have to verify the functionality by checking the output on the LCD display. As seen on the right in the Figure below, the output of the test only indicates that the code has been executed, but doesn't provide insight if it actually works.

21.4.3 Active development

Using the tool CodeScene we have run an analysis to gain insight in what parts of RIOT are under active development.

As expected the `cpu` module has a lot of activity. All different kinds of CPUs are often updated and new ones are added, as described in a previous post this is an important step in the roadmap of RIOT.

Interesting to see that there is a lot of development in the `net` folder of the `sys` module. This `net` folder contains all networking libraries. From the commit history it shows almost daily commits. This hotspot makes sense both from the roadmap, but also in general considering the requirements for IoT applications.

A nice thing to notice is that the `tests` module seems to be a hotspot as well. By keeping the tests up to date, technical debt can be kept to minimum.

[27] RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT

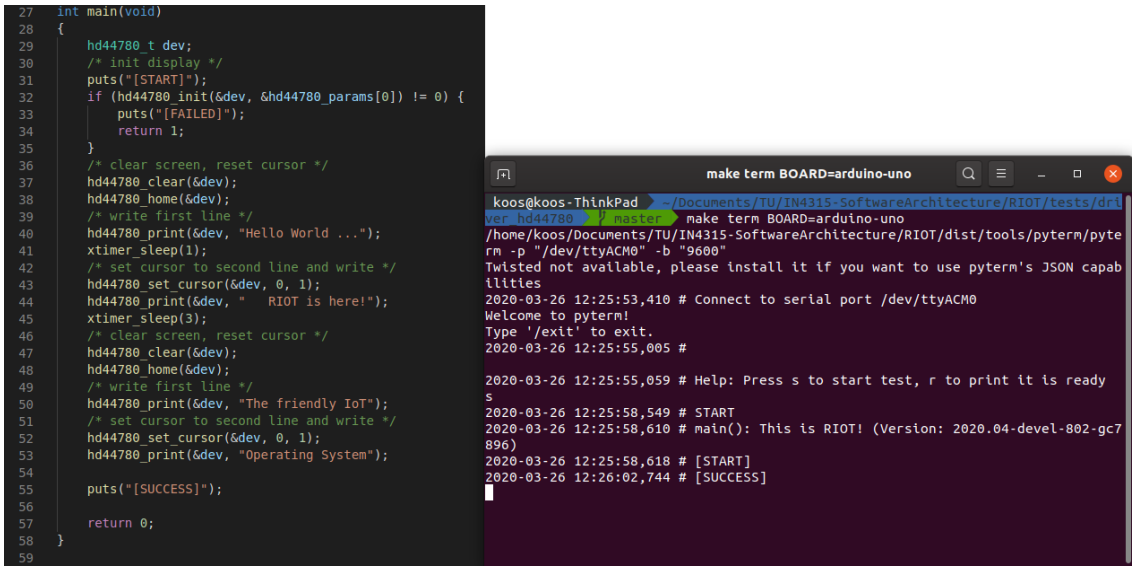


Figure 21.8: Interactive test for an LCD display

21.4.4 Code quality

To measure code quality, multiple metrics can be used. An interesting one is code duplication. The reason why code duplication has an impact on the code quality, is the effect it has on maintainability. The result is that a single change in functionality might require modifications in multiple locations. In most cases, code duplication is a sign of lazy programming and can be solved by making the code (more) modular.

While evaluating an analysis of RIOT's codebase provided by SIG, it appeared that it contains quite a lot of code duplications. However, to conclude that this is a result of low development quality is not correct. As shown in the Figure below, the main sources of duplications are the modules `boards`, `cpu` and `tests`.

The duplications in `boards` and `cpu` can be easily explained, since those components are not 'real' software. These folders contain hardware-specific implementations and configurations. Since each board and cpu needs its own configuration files, a lot of duplication is introduced because many devices are very similar. While the approach of having a configuration folder for every single device might have seemed practical in the early development of RIOT, by now it is getting unmanageable. To solve this problem, developers are working on creating common configurations for similar hardware, this effort can already be seen in the `cpu` and `boards` folders.

21.4.5 Why code quality?

The analysis of the importance of code quality, testing and technical debt is divided in three parts, corresponding with different stages of the coding process.

First we take a look at the documentation, specifically at the contributions.md. This document gives some general tips on how to make changes, such as keeping it small, modular and simple, this is in order to simplify the reviewing and speed up the merging process. It describes the requirements for contributing to any changes. The most important section is about how a pull request should be organised, it is mandatory to

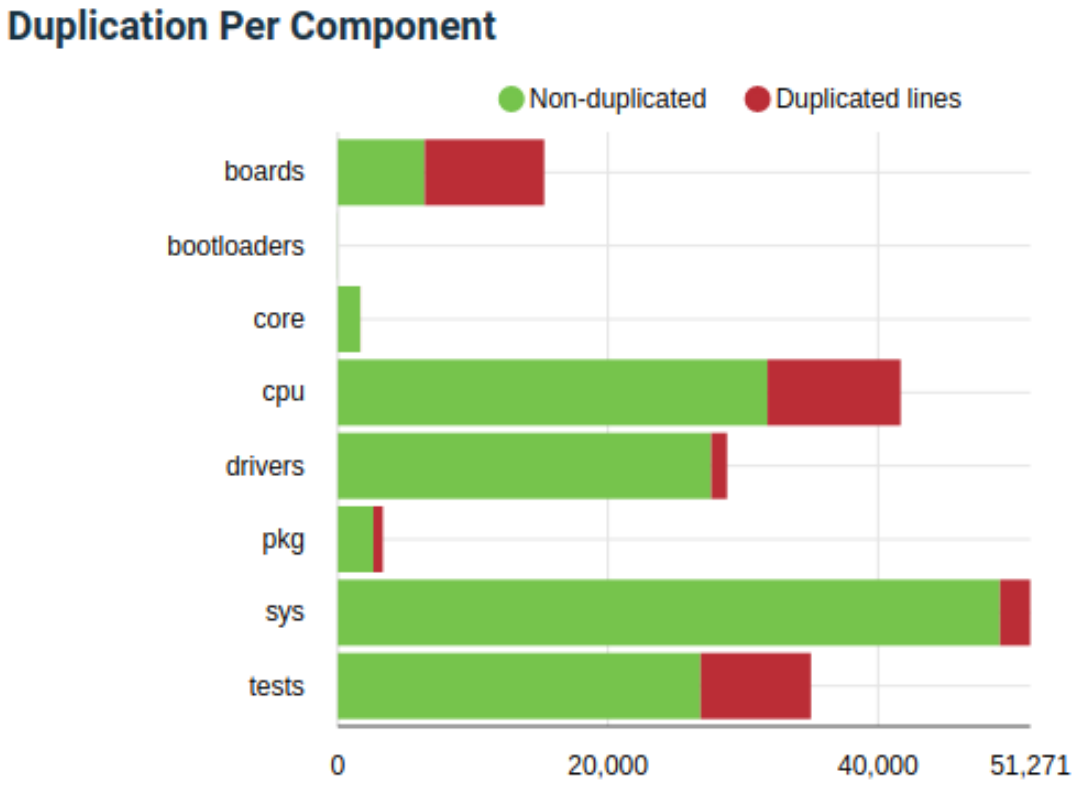


Figure 21.9: Code duplication per component

give an extensive description of the change, which should be as small as possible while still be runnable on their own. Next to this a clear testing method should be defined so reviewers can also check the work.

Next we take a look at some open issues of new features to see if there is any discussion about code quality, testing or technical debt. The amount of comments for each new feature issue range from 2 to 34, the range between 4 and 15 is nicely distributed. Most of the issues start with a comment from one of the members telling if it is a good idea and how to continue on this issue and where to find more information. The more commented issues have a longer discussion from different members or contributors about the implementation or there might be a discussion about the usefulness, as is the case with issue #5825. Issue #13469 is an open discussion about how to improve the stdio implementation to support layered stdio functionality, so there are also discussions initiated by the RIOT members.

Finally a look into the pull requests, the first noticeable thing is that the the pull requests have significantly more comments, on average around 40-50 with peaks up to 180 comments per pull request. Looking at pull request #12877 for example, has implications on all ports and drivers, which will increase the technical debt. So this comment suggests different ways to proceed to change all the ports and drivers, either all at ones or with some backward compatibility in smaller PR's. Also almost all pull requests regarding new features request changes in the code, either because of style mismatches or failed tests by reviewers.

Overall the RIOT community does a good job at discussing the code quality and testing in the different parts of the contributing process.

21.4.6 Technical Debt

To analyse the technical debt of RIOT, the online code quality tool Sonarcloud is used to provide some overview. The following graph shows the technical debt of the different files within RIOT with respect to the lines of code per file. Most of the files have the highest possible ranking (A) and so the technical debt ratio is only 0.6%. However this is still an overall estimated technical debt of 646 days. When looking at the most files with the most technical debt, sonarcloud shows that most of these files are located in the cpu module. The biggest issue here is duplicated code, this is due to the modularity of RIOT and their conscious choice to specify each board and cpu separately. The debt of the CPU module is around 600 days. Furthermore the modules boards, sys and tests have a technical debt of around 15 days. The remaining modules have a technical debt of minutes or hours.

On a side note, the evaluation of sonarcloud is not always the best. For example it counts examples in the doxygen comments as commented code, which gives an unfair picture on the finale score.

To conclude, the technical debt of RIOT is quite good. Especially when taking into consideration that the technical debt caused in the cpu module, by separating all types of cpu's to keep it a very modulair system, is a design choice. By removing that part of the debt, only a couple of days are left, which will result in a debt ratio of less than 0.1%.

21.5 RIOT: The power of variability

After we covered the quality and technical debt of RIOT's code in our previous post, we will now take a deeper look into RIOT's variability.
Since RIOT is an OS that is supposed to work with quite a lot of different hardware, and applications using RIOT with all kinds of different modules, there are a lot of options and different features to choose from.
What it comes down to is that RIOT offers variability, and a lot of it. But how it offers this and how it

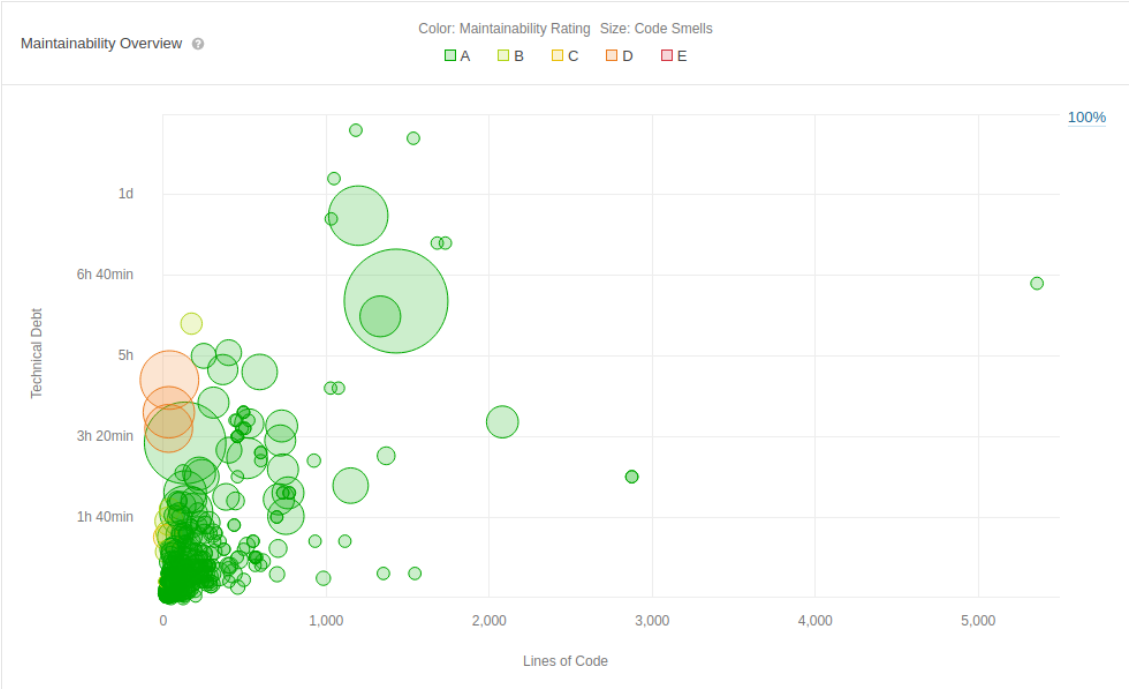


Figure 21.10: Technical debt per file

implements this is this post's topic.
In this post we will look at RIOT's main features, its variability management and its implementation.

21.5.1 Variability modeling

The variability of RIOT can be categorised in mandatory choices and optional choices. First the mandatory choice for RIOT is the target board on which the software will be flashed. At the moment of writing, developers can choose up to 144 different boards. Furthermore it is optional to use different drivers and network libraries to implement the desired functionality. RIOT categorises these drivers into Actuator Device Drivers, Display Device Drivers, Miscellaneous Drivers, Network Device Drivers, Peripheral Driver Interface, Power Supply Drivers, Sensor Device Drivers, Soft Peripheral Driver Interface and Storage Device Drivers. Furthermore it supports different network libraries, such as IPv4 and IPv6 protocols, MQTT, LoRa, 6LoWPAN, Bluetooth, CAN and many more.

Naturally there are also incompatibilities and requirements between the different features. For example, a standard Arduino nano does not support ethernet communication by default. To accomodate this functionality an ENC28J60 can be used as an extension board. Furthermore some libraries are only implemented for certain boards, for example the ws281x library is at the moment of writing only implemented for Atmega and ESP32 boards as well as the native environment.[28]

[28] https://doc.riot-os.org/group__drivers__ws281x.html

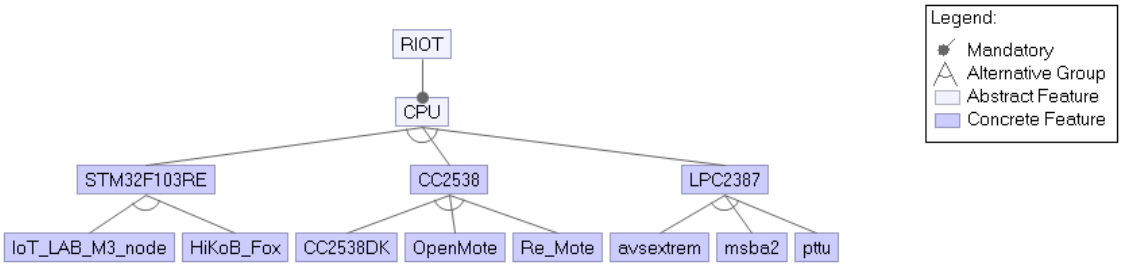


Figure 21.11: Feature model of the mandatory CPU choice. For readability only a selection of the possible cpu's and boards is shown

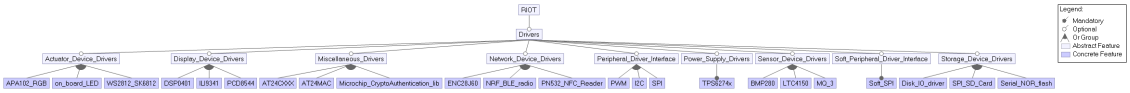


Figure 21.12: Feature model of the optional drivers. For readability only a small selection of the possible drivers is shown.

21.5.2 Variability management

In our first post we identified 5 major stakeholders: the end users, the business, the customers, the domain experts and the developers and a minor stakeholder: academic institutes. In our post we determined that the business and customer stakeholders weren't really applicable, so they will not be discussed here. We also saw that the end users, the domain experts, and the developer stakeholder have a lot of overlap. Furthermore we wanted to mention academic institutes, but in this section they would not add anything that the end users, the domain experts, and the developer already do so they also will not be discussed here.

Variability is managed the same for end users, the domain experts, and the developers. They might use it toward different ends, but it is managed the same nonetheless. The main way variability is managed is through Makefiles. Every application using RIOT has a Makefile, in the Makefile one would specify which board they would like to use, which modules and dependencies they would like to be included, as well as which flags they would like to be set or unset. The modules included in the Makefile can then be used by the C or C++ file containing the main function positioned in the same directory. The Makefiles only specify what is to be used, to actually provide the right environment is still the responsibility of the end users, the domain experts, or the developers using it.
Luckily, the maintainers at RIOT provided some general documentation and some documentation to help out with running apps.
As for documentation for updating RIOT and applications, there isn't a one size fits all solution. To update RIOT itself, the easiest way is to pull the latest version of the repository. The same is valid for applications running RIOT.

In RIOT's coding workflow they already recommend developing on their own emulator first, and if it is bug-free, develop further on the target hardware. Furthermore, their Makefiles are set up with the macros `BOARD_BLACKLIST` and `BOARD_WHITELIST`. When a board is not on the whitelist, it essentially means that errors are expected when an application is built for that board. The same behaviour results from any board on the blacklist. These macros are especially useful for the CI pipeline. If a board is on the blacklist or not on the whitelist then the application for that board is not built at all. But this also means that a developer can

test an application for multiple boards at once.
This variability does have an impact on testing though. The time required to create and test a module becomes longer with every board it is to be used on. Not just because there is more to develop, but also because debugging becomes longer and harder. Bug fixes for one board need to make sure they don't break the code for other boards.

21.5.3 Variability implementation

The use of Makefile already hinted a bit at the variability implementation mechanism and what binding time the developers at RIOT use. It is clear they use compile-time variability binding as they use Makefiles to configure the settings they wish to use for their application before it is compiled. Furthermore many modules in RIOT offer configuration options that are considered during compile-time. This also makes it clear RIOT makes use of conditional compiling, as only the environments and modules the developers wish to use are compiled and built. They also make a little use of run-time binding time. For example some aspects of their own RIOT native instances can be configured at runtime. The implementation mechanism they use for this is a design pattern, in particular the decorator pattern.

As mentioned in section Variability modeling the first mandatory variable feature is choosing a target board. This feature is implemented using Makefiles; in the Makefile the developer chooses which boards to use and during compile-time the application will only be built against the boards specified in the Makefile. Since an application is not compiled for all boards, but only those which satisfy the condition of being specified in the Makefile, one could conclude that the implementation mechanism used here is conditional compiling.

21.5.3.1 Hardware variability implementation

IoT applications are naturally involved with different types of devices. It is reasonable for the design of an IoT operating system to configure for the variability of running on various platforms. From common x86 architecture to popular STM32 family, RIOT supports a wide range of IoT platforms.[29] If we take a closer look into RIOT's implementation, we can find that all the platform related code is organized in separate folders named `cpu` and `boards`. As mentioned before, different Makefiles are contained in these files and are used to configure RIOT to run on the specific platform.

Under the folder `cpu`, files related to certain cpu/mpu specific functions and the mapping of the peripherals of the cpu/mpu are included. Most files are collected into the subfolder named as the platform that those files are related to, for instance, `lpc1768`. One thing to notice is that certain common files shared among different types of cpu/mpu of the same base architecture are put into separate folders (like `arm7_common`).

In `boards`, locate the boards related initialization files. Each subfolder contains the drivers (I2C, SPI, UART), pinouts information and other board specific files. Like what the maintenance team did for the cpu part, they also extracted common files of a family of boards and organize them into a single subfolder `boards/common`.

Another crucial feature for an IoT operating system is to ensure the connection between all kinds of devices wirely or wirelessly. Also different types of connection protocol is used under different scenarios for IoT applications. RIOT handles this variability by including different `driver` files to support different communication protocols.

Apart from communication other optional features are also handled by RIOT through the `driver` files. A wide range of sensors , ADC/DACs and other components (servos for example) used for IoT development

[29] https://github.com/RIOT-OS/RIOT/wiki/RIOT-Platforms

are included in this folder as well. This way of organization allows the maintainer to keep adding new functionality and features of RIOT, which increases the flexibility and variability in features of RIOT.

21.5.3.2 Variability, done right?!

This post has discussed the different forms of variability available in RIOT. Since the way of implementing the variability is a design choice, it is interesting to see if this choice is a good one. The variability for cpus and boards is well implemented, and offers the required flexibility of adding new boards. However with the increasing number of cpus and boards, it would be nice to organize the `cpu` and `boards` into folders for similar devices.

Regarding the variability implementation of the drivers, there might arise a scalability issue in the `drivers/Makefile.dep` file. This file contains information of supported architectures and required features for the use of a specific driver. This file already has 822 lines, and is not easily readable.

Concluding, the compile-time variability implementation with Makefiles seems like a useful option for RIOT, however there is room for improving the implementation for scalability.

Chapter 22

Ripple



Ripple is a real-time gross settlement system, currency exchange, and remittance network created by Ripple Labs Inc. It is built upon a distributed open-source protocol. It purports to enable secure, instantly, and nearly free global financial transactions of any size with no chargebacks. XRP is Ripple's coin, and Ripple has a definitive number of XRP in rotation – 100 billion tokens to be precise. There is no mining involved in the way XRP works; all tokens are pre-mined, and therefore they can be traded with the lowest possible transaction costs.

In this chapter, we study Ripple's Framework. In order to do this, we first understand the Vision behind Ripple - the context in which it exists, the project's roadmap as well as undergo a detailed analysis of its stakeholders. Following, we look into the architectural decisions made at Ripple, quality control and

assessment, and also try to contribute to their Open Source Project.

22.1 About Us

We are a group of four Masters Students at the Delft University of Technology.

- Deniz Danaie
- Yanzhuo Zhou
- Ravisankar A V
- Weikang Weng

22.2 Surfing through Ripple: A Grand Tour

The following post examines the Vision and Mission behind the Ripple Team. The essay is carried out by studying the Ripple environment, the stakeholders, and the team's accomplishments and possible future plans.

22.2.1 An Introduction to Ripple

Ripple is a technology company that provides the most efficient solutions to send money globally using the power of blockchain technology. Ripple's vision is to enable the world to move value like information moves today—the Internet of Value. Ripple is the only enterprise blockchain company today with products in commercial use. Ripple's global payments network, RippleNet, includes over 300 financial institutions across 40 countries and six continents. Ripple works with regulators, governments, and central banks—not against them—to improve the way the world moves money.[1]

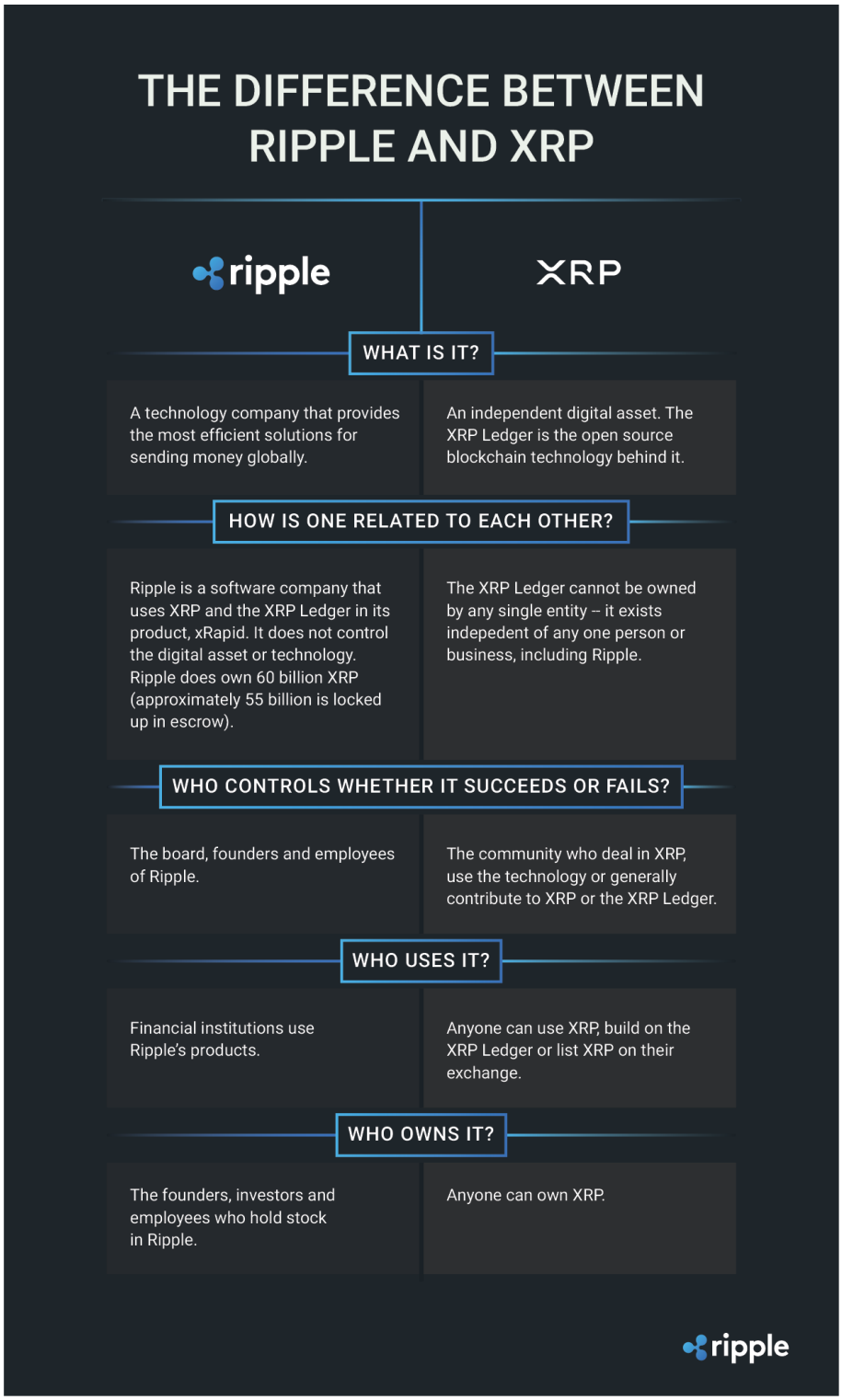
This study targets the open-source server software built upon the XRP Ledger, `rippled`. To begin, first we briefly describe the three notions that are repeated frequently in the survey.

The **XRP Ledger** is a decentralized cryptographic ledger powered by a network of peer-to-peer servers. The XRP Ledger is the home of XRP, a digital asset designed to bridge the many different currencies in use worldwide. [2]

rippled, the open-source server software that powers the XRP Ledger is called rippled and is available under the permissive ISC open-source license. The rippled server is written primarily in C++ and runs on a variety of platforms. [3]

and finally, **RippleNet**, as Ripple puts it, is the *" The world's most accessible global payments network*[4]*."* Ripple solves the needs of individuals and businesses sending cross-border payments through RippleNet, a network of banks, payment providers, and others. RippleNet aims to provide real-time, low-cost, and fully trackable payments on a global scale.[5]

[1] https://ripple.com/faq
[2] https://xrpl.org/xrp-ledger-overview.html
[3] https://github.com/ripple/rippled
[4] https://ripple.com/ripplenet
[5] https://ripple.com/files/ripplenet_brochure.pdf


THE DIFFERENCE BETWEEN RIPPLE AND XRP
ripple
XRP
WHAT IS IT?
A technology company that provides the most efficient solutions for sending money globally.
An independent digital asset. The XRP Ledger is the open source blockchain technology behind it.
HOW IS ONE RELATED TO EACH OTHER?
Ripple is a software company that uses XRP and the XRP Ledger in its product, xRapid. It does not control the digital asset or technology. Ripple does own 60 billion XRP (approximately 55 billion is locked up in escrow).
The XRP Ledger cannot be owned by any single entity -- it exists indepedent of any one person or business, including Ripple.
WHO CONTROLS WHETHER IT SUCCEEDS OR FAILS?
The board, founders and employees of Ripple.
The community who deal in XRP, use the technology or generally contribute to XRP or the XRP Ledger.
WHO USES IT?
Financial institutions use Ripple's products.
Anyone can use XRP, build on the XRP Ledger or list XRP on their exchange.
WHO OWNS IT?
The founders, investors and employees who hold stock in Ripple.
Anyone can own XRP.
ripple

"To help clarify how Ripple, the technology company, and XRP, the independent digital asset, are distinctly different, we've outlined in a simple infographic the most frequently asked questions related to the two"[6]

22.2.1.1 Key Features of the XRP Ledger

TODO intro for this part, summerize

- **Censorship-Resistant Transaction Processing:** No single party decides which transactions succeed or fail, and no one can "roll back" a transaction after it completes. As long as those who choose to participate in the network keep it healthy, they can settle transactions in seconds.
- **Fast, Efficient Consensus Algorithm:** The XRP Ledger's consensus algorithm settles transactions in 4 to 5 seconds, processing at a throughput of up to 1500 transactions per second. These properties put XRP at least an order of magnitude ahead of other top digital assets.
- **Finite XRP Supply:** When the XRP Ledger began, 100 billion XRP were created, and no more XRP will ever be created. The available supply of XRP decreases slowly over time as small amounts are destroyed to pay transaction costs.
- **Responsible Software Governance:** A team of full-time, world-class developers at Ripple maintain and continually improve the XRP Ledger's underlying software with contributions from the open-source community. Ripple acts as a steward for the technology and an advocate for its interests, and builds constructive relationships with governments and financial institutions worldwide.
- **Secure, Adaptable Cryptography:** The XRP Ledger relies on industry standard digital signature systems like ECDSA (the same scheme used by Bitcoin) but also supports modern, efficient algorithms like Ed25519. The extensible nature of the XRP Ledger's software makes it possible to add and disable algorithms as the state of the art in cryptography advances.
- **Modern Features for Smart Contracts:** Features like Escrow, Checks, and Payment Channels support cutting-edge financial applications including the Interledger Protocol. This toolbox of advanced features comes with safety features like a process for amending the network and separate checks against invariant constraints.
- **On-Ledger Decentralized Exchange:** In addition to all the features that make XRP useful on its own, the XRP Ledger also has a fully-functional accounting system for tracking and trading obligations denominated in any way users want, and an exchange built into the protocol. The XRP Ledger can settle long, cross-currency payment paths and exchanges of multiple currencies in atomic transactions, bridging gaps of trust with XRP.

22.2.2 Stakeholder Analysis

People have different interests in the Ripple Project. In this study, Rozanski & Woods' stakeholder types [7] are used as a guideline to identify the various stakeholders. In the table below, a brief definition of stakeholder types can be found.

Type	Brief description
Acquirers	Oversee the strategy of the product and seek for commercial profits
Assessors	Investigate the entire company for compliance with legal matters or industry standards

[6] https://ripple.com/insights/difference-ripple-xrp/?
[7] https://www.viewpoints-and-perspectives.info/home/stakeholders/

Type	Brief description
Communicators	Provide information and details of the system for other stakeholders
Support Staff	Provide users with technical support and answer potential questions
Developers	Develop the software system based on development needs
Maintainers	Manage the evolution of the system once it is deployed and operational
Testers	Operate the program under specified conditions to find program errors and measure software quality
Investors	Legal persons or companies who invest cash to purchase certain assets to obtain benefits or profits
Users	Individuals and companies who use Ripple for blockchain business
Competitors	Compete for a commercial gain of the same product or in the same industry

22.2.2.1 Acquirers

From the analysis of investments, MoneyGram International invests PIPE-IV round with $20M, and SBI Investment and Hinge Capital invest Series B with $55M. It is easy to conclude that the management members of these companies are the acquirers of Ripple. For fintech companies, they sponsor Ripple and need the real-time money transfer and estimation services provided by Ripple. For capital companies, they invest Ripple and gain dividends from stocks.

22.2.2.2 Assessors

The Financial Crimes Enforcement Network ("FinCEN") has the authority to investigate money services businesses. This organization assesses legal matters related to Ripple and checks whether Ripple is compliant with the standards for the regulation of a commercial blockchain company. Besides, CryptoNewsZ is a company that makes assessments for crypto markets. Ripple (XRP) is one of the leading bitcoin cases, and there are yearly assessments with its stock price, future trend, and market conditions made by CryptoNewsZ.

22.2.2.3 Communicators

The Ripple communication system is **RippleNet**. It enables access to new markets, expands users' services, and delivers the best customer experience in global payments. Customers can access the blockchain technology for global payments, payout capabilities, On-Demand Liquidity as an alternative to pre-funding, and operational consistency through a common rulebook.

22.2.2.4 Support Staff

Ripple offers three types of support on the official website. A form to contact the Sales Team, a form to contact the Press Team for press requests and finally, FAQs answering questions about Ripple company, XRP Token, XRP Tech, and Merchandising.

For more technical support, there are other resources, such as the XRP Ledger platform, which is an open-source technology. One can integrate and contribute to the platform using the tools and information provided.

22.2.2.5 Developers

The most active repositories of the Ripple project are xrpl-dev-portal, XRP Ledger developing documentation, rippled, The source code and, ripple-lib, an API development for interacting with the XRP Ledger. On average, these repositories have more than 70 Contributions. Below some of the leading developers of the project have listed.

Active Repository	Github ID	Affiliation	Contribution	Role
rippled	@JoelKatz	David Schwartz	4,251 commits	Ripple CTO and network architect
	@vinniefalco	Vinnie Falco	2,633 commits	Former Ripple Engineering Manager, President at C++ Alliance
	@ahbritto	Arthur Britto	1,384 commits	
	@nbougalis	Nik Bougalis	589 commits	Ripple Software Engineer
	@mellery451	Michael Ellery	160 commits	Ripple software generalist
	@justmoon	Stefan Thomas	194 commits	Open-source developer
xrpl-dev-portal	@mDuo13	Rome Reginelli	1,806 commits	Ripple Labs Employee
	@jhaaaa	Rome Reginelli	100 commits	
ripple-lib	@wltsmrz		528 commits	
	@geertweening	Geert Weening	262 commits	Ripple Employee
	@intelliot	Elliot Lee	205 commits	Ripple Employee

David Schwartz is the CTO at Ripple and is one of the original architects of the Ripple consensus network. Vinnie Falco is also one of the leading developers, and he took over David's work since 2013. The early development of Ripple was from 2012 to 2014, and the subsequent jobs are maintaining and updating.

22.2.2.6 Maintainers

Depending on the timeline of Ripple's commitment, different engineers and open-source developers are responsible for Ripple's maintenance. Since there is no clear description of the division of labor during the

program development and operation and maintenance, all the developers above can be seen as maintainers.

22.2.2.7 Testers

The developers themselves look into the testing procedures. They undergo unit testing on each component of the repository.

22.2.2.8 Investors

The key investors of Ripple are players such as Andreessen Horowitz, SBI Investment, and more. In angel rounds, the company attracts corporate supporters and famous crypto bulls, including Tim Kendall, the former President of Pinterest, Jesse Powell, Co-founder, and CEO of Kraken exchange and others. In Venture Capital series rounds, specific capital companies and venture companies like SBI Investment and Accenture fund Ripple for commercial usage.

22.2.2.9 Users

There are hundreds of financial institutions around the world, relying on RippleNet to process their customers' payments anywhere in the world. Some of the important users are mentioned below.[8]

- American Express
- MoneyGram
- Santander
- SCB
- SBI Remit
- NIUM
- Banco Rendimento

22.2.2.9.1 Case Studies

- Santander

  Customers who were not doing international transfers are now using the service. Customers who were using international transfers are now doing it more. Furthermore, the customers who had gone to use fintech competition have come back because of the One Pay offering.

- NIUM

  Through NIUM's use of Ripple in the Philippines and Mexico corridors, they have been able to eliminate pre-funding requirements. They offer faster remittances at a lower cost.

- SBI Remit

  They must continuously search for superior technological solutions to deliver ever improved remittance flows. Ripple helps open up new revenue potential for business and a better overall experience for customers.

[8] https://ripple.com/customers

22.2.2.10 Competitors

As part of the hype surrounding cryptocurrencies, there is a huge number of competitors for Ripple, including Bitcoin, Dogecoin, Litecoin, Ethereum. Among them, Bitcoin is the most widely used.

Litecoin is a fork version of the bitcoin system that has more features such as:

- a decreased block generation time (2.5 minutes)
- increased maximum number of coins
- different hashing algorithm (script, instead of SHA-256)
- slightly modified GUI

Dogecoin is a further fork of the Litecoin system, which used primarily as a tipping system on Reddit and Twitter to reward the creation or sharing of quality content.

One can see that Ripple faces competitors in high quantity as well as of great quality. Hence, it is ever so important to leverage help from the open-source community and create the best software possible.

22.2.3 Product Context

In this section, the Ripple environment, its connection to external entities, people, and systems are portrayed. The context view diagram represents an overview of the different entities currently associated with Ripple:

The most popular users of Ripple are large financial institutions like American Express, MoneyGram, and SBI Remit since Ripple is one of the pioneers in Cryptocurrency. It is also backed by companies such as Standard Chartered and a16z. However, Ripple is in stiff competition against heavy-weight leaders such as Bitcoin and Etherium. They are also in the careful inspection of The FinCEN. RippleNet, a project of Ripple, provides banks and businesses different solutions to transfer money globally. The codebase maintained on GitHub is written in C++ and has support in Linux, macOS, and Windows.

22.2.4 Roadmap

Roadmap offers us a strategic plan that defines a goal and includes the milestones needed to reach it. It also presents what problems Ripple has overcome by reviewing the development history.

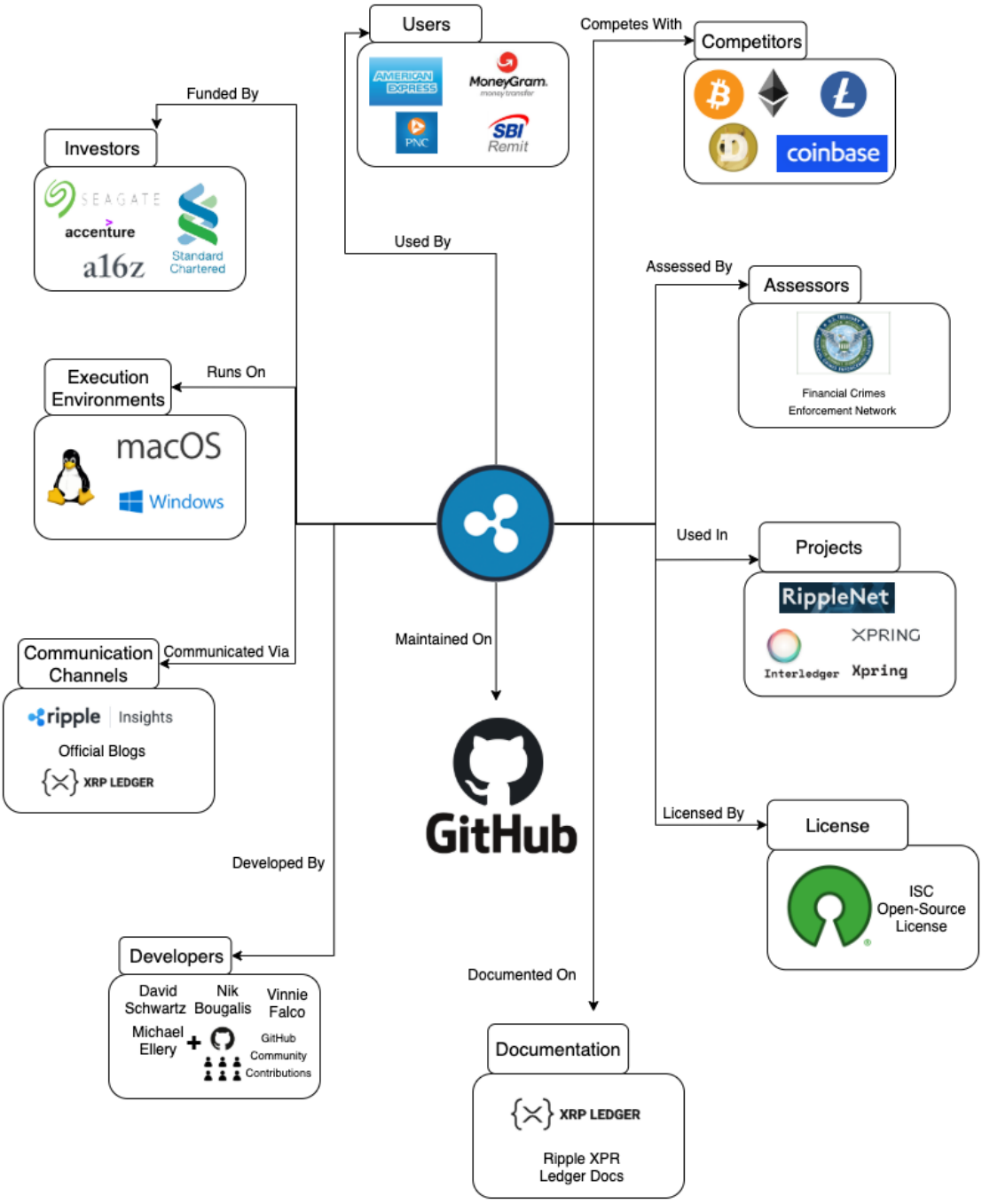


Figure 22.1: Product context



[9]

The figure above is the company's commercial road map since founded. Although the protocol as a working prototype was created away back in 2004, the company started in 2012 when Jed McCaleb secured investment for Ripple Labs.[10] The chart below shows the additional functions the developers introduced to the ripple core.

22.3 From Words to Actions: How Ripple Gets it done

The following post examines the underlying architecture of Ripple. It is carried out by studying how Ripple realizes the different properties mentioned in the previous essay and the principles of its design and evolution. We will look into the architectural views relevant to Ripple, such as development, run time, and deployment views. Furthermore, we will also discuss the non-functional properties of the system.

[9] https://ripple.com/company
[10] https://capital.com/ripple-xrp-price-prediction-2020

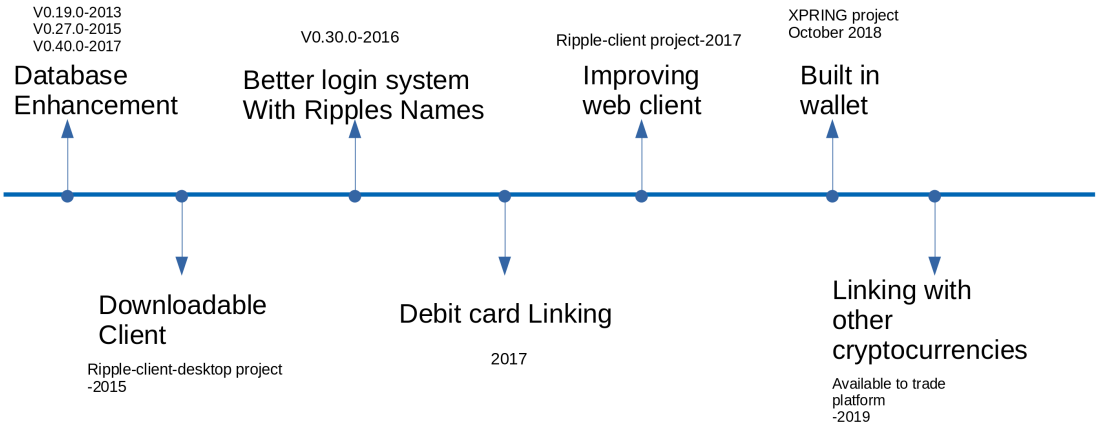


Figure 22.2: Software Roadmap

22.3.1 Architectural View

To put it plainly, building a suitable architecture for a software project is hard, and you can imagine that it doesn't come easy for a product as large as Ripple. One look at the Rippled GitHub repository gives the idea of how big the ecosystem is. From RippleNet codebase that manages user platform to the XRP codebase that takes care of their ledger, there is a lot of software development going on. So how does one even try to begin to analyze this massive architecture? This post answers that question by leveraging the main ideas of Architectural View, as presented in Rozanski and Woods[11].

To understand the architecture of Ripple, rather than looking at it as a single overloaded model, we will partition it into different components and *"view"* them separately. Partitioning into such views helps us to focus on that particular component, and collectively, in the end, we will get a solid picture of the system as a whole. Now the question is, "What establishes a *"view"* and how can we use them?"

Rozanski and Woods[12] defines Architectural View as

- "A way to portray those aspects or elements of the architecture that are relevant to the concerns the view intends to address—and, by implication, the stakeholders for whom those concerns are important."*

The different stakeholders in considerations are already discussed in the previous post. Hence in this essay, we partition the architecture by viewing it from the perspective of stakeholders such as developers, users, and maintainers. Rather than reinventing the wheel on deciding what should go into each view, we borrow the concept of Viewpoints. Architecture Viewpoint is a collection of patterns, templates, and conventions for constructing one type of view[13]. Hence the viewpoints we will be concerned about are Development View, Run Time, and Deployment View.

[11] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[12] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[13] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

22.3.1.1 Architecture Styles and Patterns

Rather than working on the project as a whole, Ripple implements a Component-based architecture style that divides the project into different sections and works on it individually. More on how they manage it is covered in the Development View section.

22.3.2 Development View

Kruchten[14] defines Development View as the view that focuses on the organization of the actual software modules in the software development environment. The software is packaged in small chunks, program libraries, or subsystems that can be developed by one or more developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it.

Thanks to the excellent management of the developing team, the complexity of Ripple is divided into different segments allowing one to analyze the various components such as front-end design, configurations, software functions, auxiliary functions, testing modules, documents, etc. separately. Here we only select the software function modules in the system decomposition and list them in the form below:

Folder	Descriptions
app	Application of the ripple client
basics	Utility functions and classes. ripple/basic should contain no dependencies on other modules.[15]
beast	A HTTP and WebSocket library Beast [16]
conditions	Ripple configure requirements and runtime requirements
consensus	Implementation of a generic consensus algorithm. [17]
core	Core of the ripple program [18]
crypto	Crypto providing the crypto algorithm
json	Third-party library to do JSON operations. [19]
ledger	Data structure for ripple transaction ledger
net	network components for communications
nodestore	Providing an interface that stores, in a persistent database, a collection of node objects that rippled uses as its primary representation of ledger entries. [20]
overlay	Each connection is represented by a *Peer* object. The Overlay Manager establishes, receives, and maintains links to peers. Protocol messages are exchanged between peers and serialized using [21].
peerfinder	Maintaining and storing addresses for different endpoints and establishing and managing connections inside peers to peers network
proto	Protocol buffers source code. The protocol tool saves the output of the *.proto* files in the build directory.
resource	Working on identifying load balancing between each endpoint. Sharing load information in a cluster, warn and/or disconnect endpoints for imposing load. [22]
rpc	Allowing suspension with continuation. And a default continuation to reschedule the job on the job queue.

[14] P. B. Kruchten, "The 4+1 View Model of architecture," in IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995.

Folder	Descriptions
server	Ripple's server configurations. It configures Ripple's port and executes Internet sessions. [23]
shamap	A given SHAMap is a Merkle tree or a radix tree storing Transactions and account states.
unity	Providing links and references to other modules.

Since these modules are created independently, we divide them into the following layers according to their functions:

- Application layer
- Core
- Network layer including server, RPC, overlay, proto, beast, and net.
- Architecture layer, including nodestore, peerfinder, resources, conditions, and JSON.
- Blockchain layer, including consensus, Crypto, shamap, and ledger.

As is shown in the graph below, some of the observed dependency insights are:

- App is the entrance of Ripple's system.
- Core is the key component that schedules other layers.
- Unity is called by core; in fact, it includes all the global library inferences of other modules.
- In the Architecture layer, JSON is also globally used since it serves as a data provider for other components. Resources connect extra functions and libraries outside the software architecture. The data is saved in the nodestore.
- In the Network layer, net, server, RPC are executing net communications. Their online data should meet the protocol defined in the overlay.
- In the Blockchain layer, confidential data hashed by the Crypto is stored in shamap and finally represented by the ledger.

22.3.3 Run Time View

The figure below shows the Ripple's flow chart during its run time. The whole run time process can be divided into 4 phases:

- The application will pick up the users' actions and put them inside the Core.
- The architecture layer parses data, test application metrics, and stores data.
- The network layer configures the servers to ensure end-to-end remote communication.
- Finally, the blockchain layer encrypts and handles the Bitcoin business.

[15] https://github.com/ripple/rippled/tree/develop/src/ripple
[16] https://github.com/vinniefalco?tab=overview&org=ripple
[17] https://github.com/ripple/rippled/tree/develop/src/ripple
[18] https://github.com/ripple/rippled/tree/develop/src/ripple
[19] https://github.com/ripple/rippled/tree/develop/src/ripple
[20] https://github.com/ripple/rippled/tree/develop/src/ripple
[21] https://developers.google.com/protocol-buffers/
[22] https://github.com/ripple/rippled/tree/develop/src/ripple
[23] https://github.com/ripple/rippled/tree/develop/src/ripple

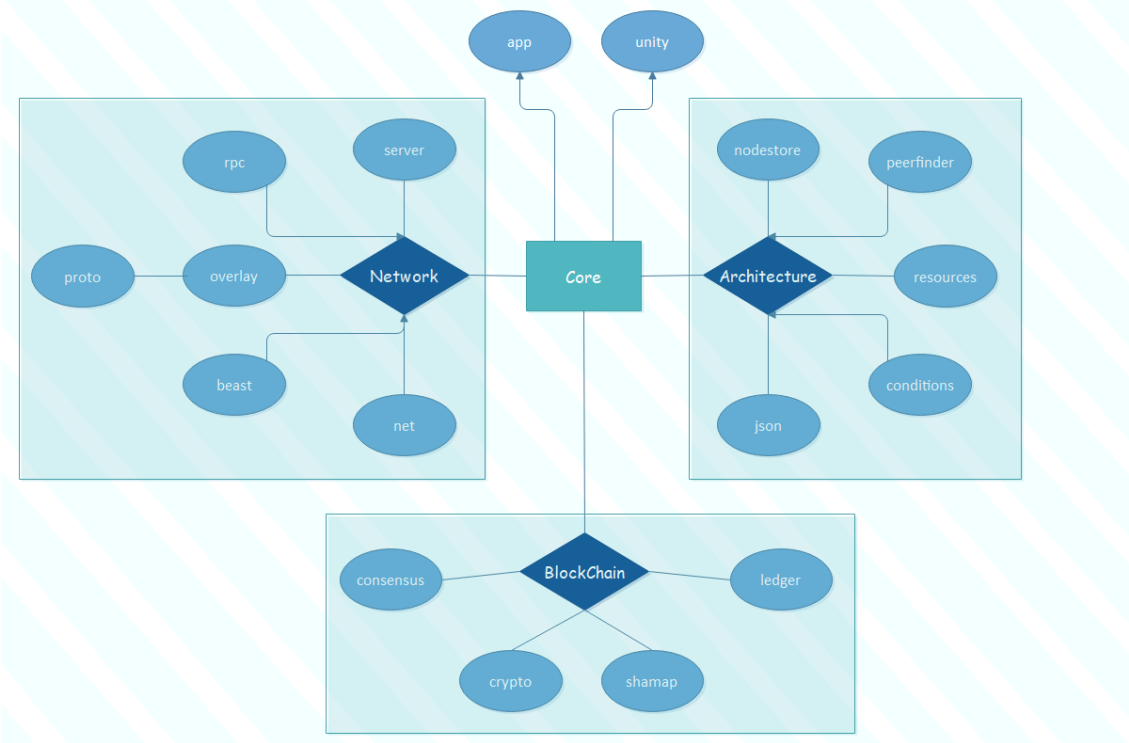


Figure 22.3: Module Dependensies

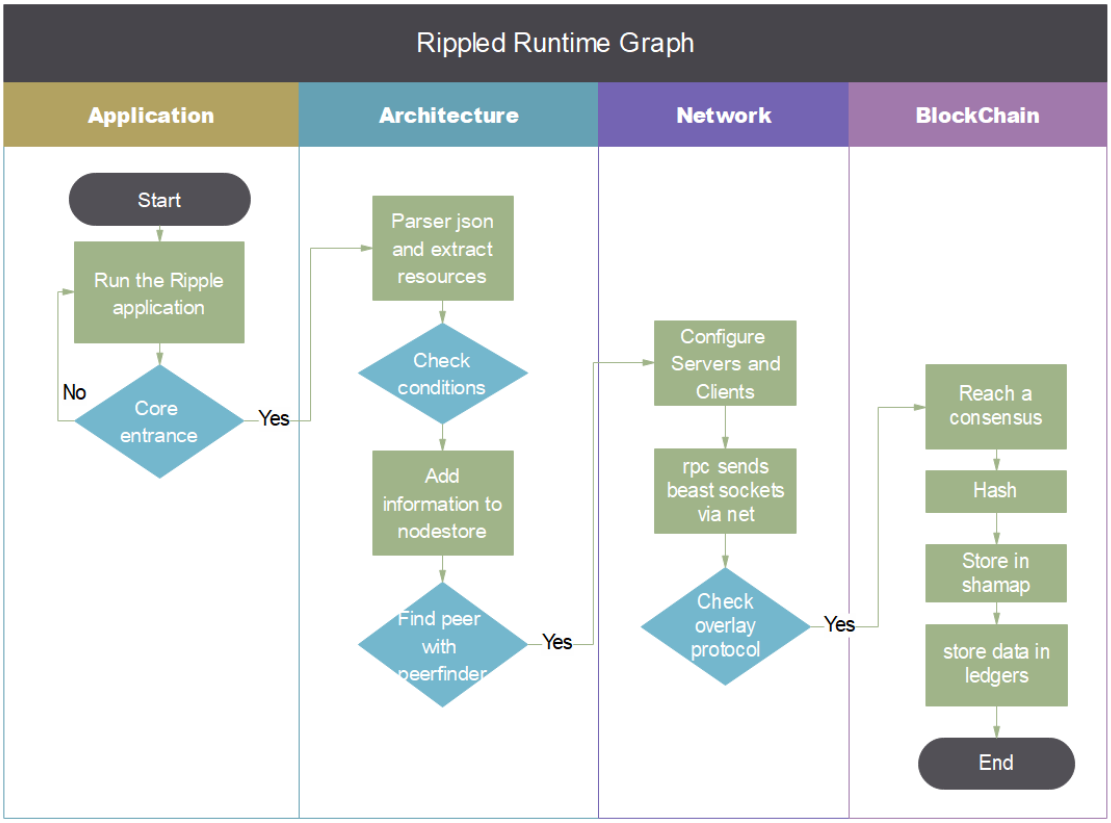


Figure 22.4: Ripple Run time

22.3.4 Deployment View

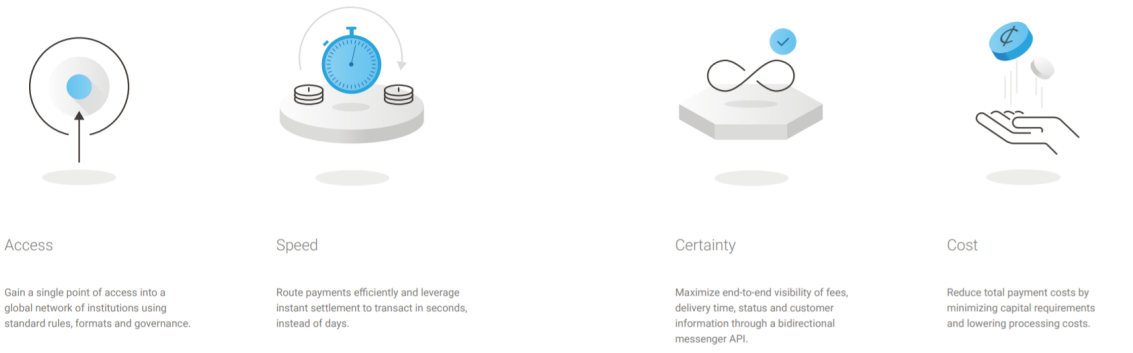
22.3.4.1 Where is Ripple deployed

As a commercial product, Ripple's system is deployed on its own trading network RippleNet. It's a network of banks, payment providers, and others. It can be seen as a cloud data community. Employing Ripple's solutions and a standardized ruleset allows for those connected on RippleNet to send and receive payments around the world efficiently.

22.3.4.2 How Ripple is deployed

The deployment of Ripple can be divided into different components. The main components deployed in RippleNet are as follows:

- Access entrances connected into a global network of institutions using standard rules, formats, and governance.
- The run time checks according to Ripple's rulebook, which is a legal framework about the rights, obligations, and business rules of network participants.[24]
- Servers who perform pathfinding capability and then ensure that payments are routed from the originator to the beneficiary in the most efficient way possible.
- Messenger API, which provides payment certainty with instant bidirectional messaging.
- The domestic digital assets converting the originating currency to XRP.



[25]

22.3.5 Non-functional Properties

One common goal followed by software engineers is to deliver a product that satisfies the requirements of different stakeholders. Software requirements are generally categorized into *functional* and *Non-Functional* Requirements (NFRs). [26] Broadly, functional requirements define *what* a system is supposed to do, and non-functional requirements define *how* a system is supposed to be. While NFRs may not be the main focus in developing some applications, there are systems and domains where satisfying NFRs is even critical and one of the main factors which can determine the success or failure of the delivered product. Hence, in this

[24] https://ripple.com/files/ripplenet_brochure.pdf
[25] https://ripple.com/files/ripplenet_brochure.pdf
[26] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements

section, we will look into the NFRs of Ripple. One functional requirement could be *creating a new block*; meanwhile, an NFR would be *specifying the hash function and cryptographic protocols it will use* to satisfy the functional requirement.

22.3.5.1 Why NFR is a challenging task

Addressing NFRs in the development of a software product is a challenging task. Often NFRs are expressed informally and abstractly. they can be considered as a specification of global constraints on the software product, such as security, performance, availability, and so on, which can crosscut different parts of a system. [27]

Another problem which is mostly observed in large organizations is that different teams may have different interpretations of an NFR, or refer to one NFR using different terms. Therefore, a coherent way of representing and defining NFRs can help mitigate such problems. [28]

22.3.5.2 Ripple NFR

In the case of Ripple, main NFRs that are mentioned in a more formal approach in the rippled repository are as follows:[29] [30]

- **Secure, Adaptable Cryptography:** The XRP Ledger relies on industry-standard digital signature systems like ECDSA (the same scheme used by Bitcoin) but also supports modern, efficient algorithms like Ed25519. The extensible nature of the XRP Ledger's software makes it possible to add and disable algorithms as state of the art in cryptography advances.
- **Responsible Software Governance:** As an entity that is obligated to hold large amounts of XRP for the long term, Ripple has a strong incentive to ensure that XRP is widely used in ways that are legal, sustainable, and constructive. Ripple provides technical support to businesses whose goals align with Ripple's ideal of an Internet of Value. Ripple also cooperates with legislators and regulators worldwide to guide the Implementation of sensible laws governing digital assets and associated businesses.
- **Censorship-Resistant Transaction Processing:** No single party decides which transactions succeed or fail, and no one can "rollback" a transaction after it completes.
- **Fast, Efficient Consensus Algorithm:** The XRP Ledger's most significant difference from most cryptocurrencies is that it uses a unique consensus algorithm that does not require the time and energy of "mining." The XRP Ledger's consensus algorithm settles transactions in 4 to 5 seconds, processing at a throughput of up to 1500 transactions per second.
- **Finite XRP Supply:** The rules of the XRP Ledger provide a simple solution to hyperinflation: the total supply of XRP is finite. Without a mechanism to create more, it becomes much less likely that XRP could suffer hyperinflation.

22.3.5.3 Addressing NFR and its Trade-Offs

Considering that NFRs are usually specified informally and abstractly, providing a more formal approach which enables to raise the abstraction level can help with the treatment of NFRs. Moreover, an explicit

[27] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements
[28] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements
[29] https://github.com/ripple/rippled/blob/develop/README.md
[30] https://xrpl.org/xrp-ledger-overview.html#xrp-ledger-overview

treatment of NFRs facilitates the predictability of the system in terms of the quality properties of the final product more reliably and reasonably. [31]. In satisfying NFRs, the dependencies among them should not be neglected, as meeting one NFR can affect and impair the satisfaction of other NFRs in the system. Therefore, performing a trade-off analysis to establish a balance among NFRs and identify such mutual impacts is necessary. [32]

Let's take a look at the first two NFRs mentioned above: Secure Cryptography and Responsible Software Governance. Although implementing secure cryptographic protocols that have a large key length will make the system close to tamper-proof, it has a glaring disadvantage: criminals could use the system to partake in illegal activities, and it will be harder to regulate. Of course, one might argue that the whole purpose of Distributed Ledger Systems is to free us off from these regulations. Still, in a perfect world, Ripple should decide on how to compromise between regulation, responsible governance, and the strength of these cryptosystems.

22.4 Assessing Ripple

In this essay, we will focus on the means to safeguard the quality and architectural integrity of the underlying system. First, We will overview the software quality characteristics. Then we will look into Ripple's GitHub repository to see if Ripple is applying any tests to achieve better software quality. Finally, we will discuss the analysis of Ripple's source code using SIG.

22.4.1 What do we mean by software quality

With key aspects of Ripple's architecture described previously, it is important to understand its software quality - specifically, what constitutes a good code and what are the different ways we can judge it. Even though quality is somewhat a subjective attribute that may be understood differently by different people, Hruschka and Starke[33] describe a quality tree checklist with which we can derive pointers to assess the quality of the project.

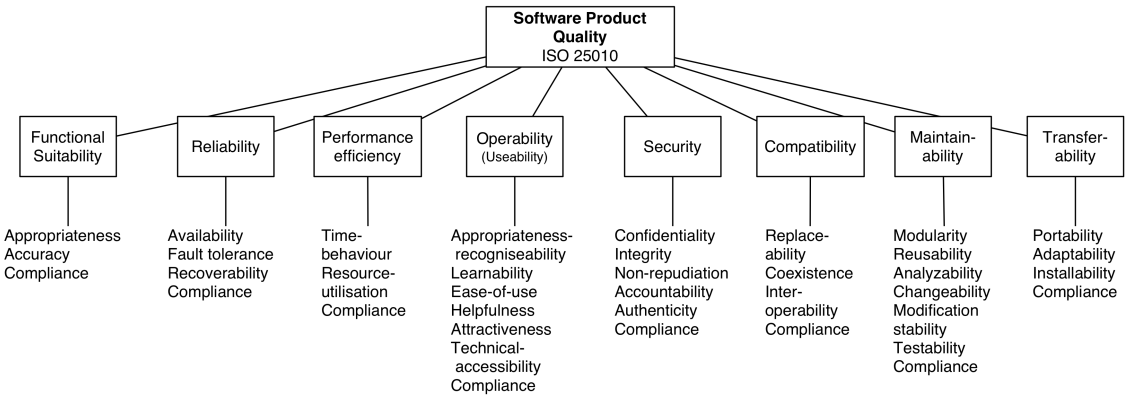


Figure 22.5: Software Product Quality

[31] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements
[32] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements
[33] https://docs.arc42.org/home/

Ripple does not have explicit documentation on how they measure the quality of their product. However, from their GitHub contributors Guidelines[34], test scripts and modules[35][36], and their travis-ci page[37], we were able to add the missing pieces together. Whenever new code is pushed into Ripple or its sibling repositories, the request triggers automated unit and integration tests, flow type checking, ESLint checks, automated testing of the documentation, as well as code reviews by several developers.

22.4.1.1 Main Checkpoints and Test Analysis

If we take a look at their test script file[38], we can find that it calls 14 manually written tests. Some of the tests that are compatible with the tree checklist mentioned above are discussed here—the `ripple.consensus.ByzantineFailureSim` script undergoes a fault tolerance test on the ledger. Every time the code belongs to the ledger is changed, this script invokes the Byzantine Failure Simulation and makes sure that the system will be reliable in production. Similarly, the `beast.unit_test.print` assess the functional accuracies of each component, and the integration test addresses the interoperability between the components. Since Ripple works on three main OS, the transferability of the product must be assessed. In Travis CI, we can find tests that assess the reproducibility of Ripple in Linux, macOS, and Windows systems. Whether or not Ripple assesses the other checklists mentioned in the diagram above is not clear because such information is not publicly available.

Rather than manually running all these tests, Ripple leverages the power of Travis Continous Integration service. Whenever a new push/pull request is submitted, Travis CI will check out the relevant branch and run the commands specified in the `.travis.yml` file, which usually builds the software and runs any automated tests. Ripple adopts a five-stage CI check that has 33 jobs in total. An infographic of the result of their CI test for March is shown below.

22.4.1.2 How much do they care about tests

From the infographic above, we find that most of the builds are either passing or failing. By comparing these builds to the several pull requests at GitHub, we find that failed or even error builds are not merged to the branch. This implies that the Ripple dev team takes these tests very seriously and will only merge if the CI tests pass.

From the Codecov site[39], we see that Ripple has 70% code coverage. Even though they measure it, we haven't found any discussion between developers where they address it. Even though there is a coverage bot that displays the code coverage for every pull request, the participants don't address it. One reason why it might be just 70% is that the Ripple repo contains code for all three major OS, and maybe some tests cases don't apply to a specific platform.

22.4.1.3 A mapping of recent coding activity on key architectural components

As we discussed in our architectural components before, we decomposed the system into three layers with several components. In this section, we look into how the Ripple team has brought them alive through code.

[34] https://github.com/ripple/xrpl-dev-portal/blob/master/CONTRIBUTING.md
[35] https://github.com/ripple/ripple-lib/tree/develop/test
[36] https://github.com/ripple/rippled/blob/develop/bin/ci/test.sh
[37] https://travis-ci.com/github/ripple/rippled
[38] https://github.com/ripple/rippled/blob/develop/bin/ci/test.sh
[39] https://codecov.io/gh/ripple/rippled

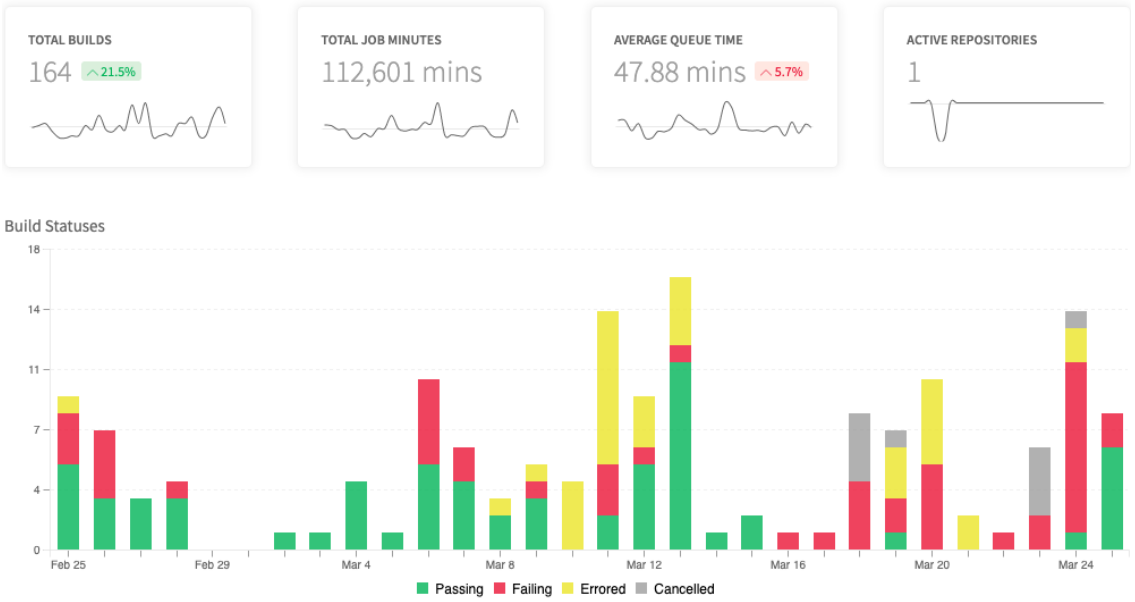


Figure 22.6: travisbuild

22.4.1.3.1 Blockchain layer

- 2020-01-13: Ripple reorganizes its SQL databases to remove some unused data, which provides better data management for the **ledger**.
- 2016-07-19: The TrustSetAuth is enabled in the **ledger** to authorize other addresses to hold their issued currencies and build connections to the **shamap**.
- 2014-06-26: A new signature scheme is implemented based on DSA in the **crypto** to ensure data security for the **ledger**.

22.4.1.3.2 Network layer

- 2019-10-02: The fixMasterKeyAsRegularKey is implemented in the **proto** to set regular key pair to master key pair.
- 2017-03-16: Gateway Bulletin is implemented to ensure a trust line quality and is realized in the **RPC, server, client** components.

22.4.1.3.3 Architecture layer

- 2018-11-07: Node store is upgraded to shard store in the **node store**, providing reliable paths toward ledger history across the XRP Ledger Network. So it is connected to the **ledger**.
- 2014-04-28: Ripple's Naming system is used to standardize the configuration. It is checked in the **conditions**.

Most of our representative code activities are serving the **blockchain** layer, especially the **ledger** component, performing better currency transactions and encryption. The XRP Ledger is the focus of programmer development, and other layers are providing technical support for XRP in the blockchain layer.

22.4.1.4 Mapping of the system's roadmap onto architectural components

In this section, we extract key events in our roadmap analysis and map it according to architectural layers in the graph below.

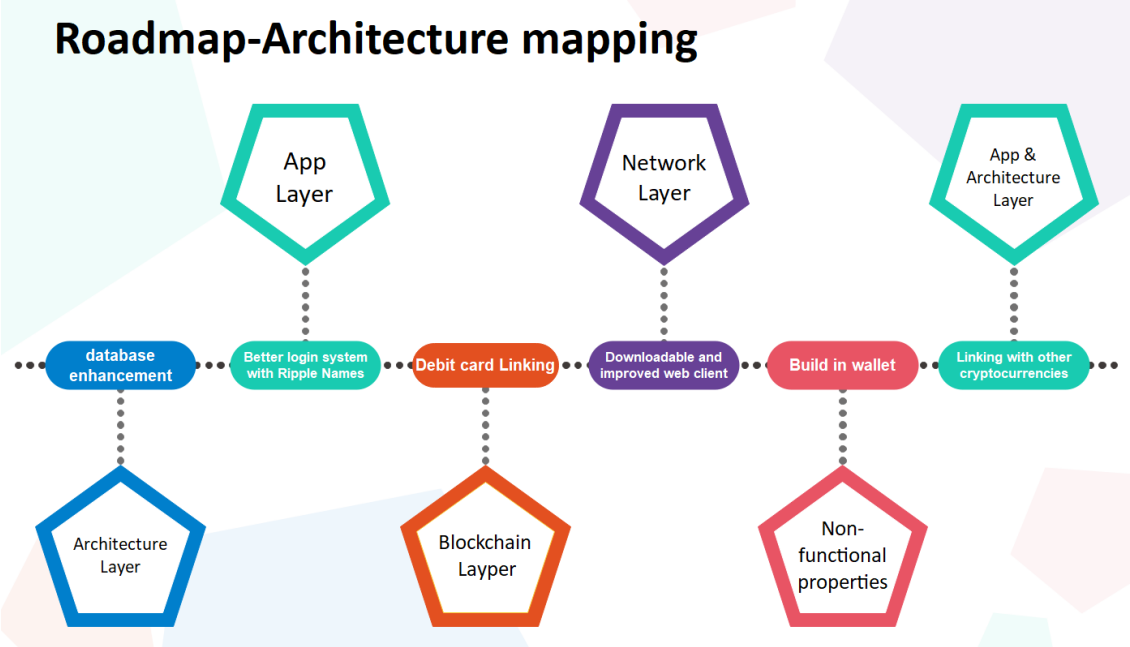


Figure 22.7: Roadmapping

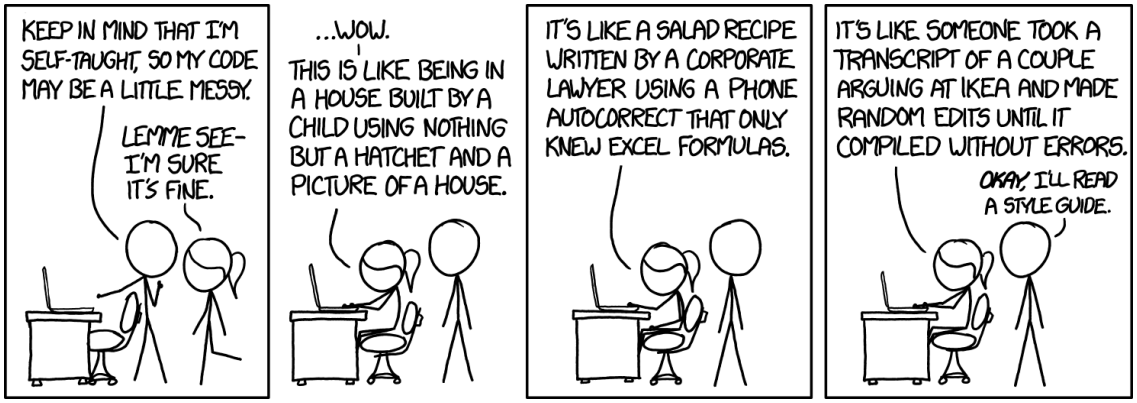
1. Database enhancement:
   - Since the database is the basis of the Architectural layer, a robust database ensures smooth operation of other architecture layers.
   - It is mainly implemented in the **nodestore** and **resources**.
2. Better login system with Ripple Names:
   - As an entrance, the login system belongs to the App layer, providing better interactions for Ripple users.
   - The naming system is more of a written specification.
3. Debit card Linking:
   - Payment handling is a key topic of the blockchain layer, whose job is mainly to oversee the operation and payment of digital currencies.
   - As for codes, **all the components** in the Blockchain layer is responsible for payments.
4. Downloadable and improved web client:
   - Ripple introduced several WebSocket amendments and gateway services APIs, providing straightforward calls that clients can use to route payments appropriately. This reliable client serves for the Network layer.
   - **All the components** in the Network layer work for this client and WebSockets.
5. Build-in wallet:
   - This is realized by Ripple's XRP project. This project is a goal of Ripple NFR, which is a key component of non-functional properties.

- The wallet is mainly implemented in the **ledger**.

6. Linking with other cryptocurrencies:
   - This improvement is beneficial to both the upper-level Blockchain transaction architecture and App layer.
   - Codes are integrated into **app**.

22.4.2 Code Quality and Maintainability

In this section, first, we will discuss the code quality and maintainability and then analyze the results from Sigrid to see how much Ripple is applying the theory in practice.



Source[40]

Code quality can have a significant impact on software quality, on the productivity of software teams, and their ability to collaborate. But how can one measure code quality?[41]

Here are some of the main attributes that one can use to determine it:[42]

Attributes	Describtion
Clarity	Easy to read and oversee for anyone who isn't the creator of the code.
Maintainablity	A high-quality code isn't overly complicated. Anyone working with the code has to understand the whole context of the code if they want to make any changes.
Documentation	The best thing is when the code is self-explaining, but it's always recommended to add comments to the code to explain its role and functions. It makes it much easier for anyone who didn't take part in writing the code to understand and maintain it.
Well-tested	The fewer bugs the code has, the higher its quality. Thorough testing filters out critical bugs ensuring that the software works the way it's intended.

[40] https://xkcd.com/1513/
[41] https://www.sealights.io/code-quality/code-quality-metrics-is-your-code-any-good/
[42] https://codingsans.com/blog/code-quality

Attributes	Describtion
Efficiency	High-quality code doesn't use unnecessary resources to perform a desired action.

22.4.2.1 Refactoring Suggestions

To Analyze the code quality and maintainability of Ripple, we used a behavioral code analysis tool, namely Sigrid.

The distribution of code volume over language are as follows:

Ripple achieve a good score in the SIG assessment system due to its high unit interfacing(4.6). But it falls heavily in the unit size(0.5) and unit complexity(0.5) sections. The duplication score (3.7) can also be improved further.[43]

Some of the maintainability violations found are listed in the table below.

Violation type	Instances in Ripple
Unit size	632
Unit complexity	239
Unit interfacing	12
Duplication	934

The unit size and unit complexity are all belonged to the unit test level.[44]

22.4.2.1.1 Duplication Duplication inside the codes means how many codes are repeated. It can be checked easily through some existing tools. To improve it, duplicated codes can be stored in separate functions or files to be reused. But it will also add some dependencies inside the program.

22.4.2.1.2 Unit size Unit size is a parameter that belongs to testability to the software's maintainability[45]. During the unit test of software, individual functions and procedures need to be tested separately to ensure each unit works properly[ref]. The unit size means how much codes inside each size. According to the SIG results[46], the unit size of the ripple projects is seen as large and complex. During testing, these large units may lead to an insufficient unit test.

To improve this part, some large functions may split into multiple simpler functions. By doing this splitting, some functions can be reused to enhance the duplication. But this split will bring more functions inside the program and make it harder to read.

Ripple's unit size keeps increasing during its further developments. They are adding more features, more services, and making the software more secure and robust to meet the requirements of markets. To reduce the unit size, developers need to review their design and do some split to different functions. Splitting some functions to become separate units.

[43] https://sigrid-says.com/maintainability/tudelft/rippled/refactoring-candidates
[44] http://softwaretestingfundamentals.com/unit-testing/
[45] http://softwaretestingfundamentals.com/unit-testing/
[46] https://sigrid-says.com/maintainability/tudelft/rippled/refactoring-candidates

Language	Files	Code	Comment	Blank
C++	568	150,470	22,987	27,844
C	652	80,042	26,436	19,536
CMake	40	5,855	0	398
Markdown	44	5,714	0	2,491
Shell Script	28	1,283	143	194
JavaScript	10	898	122	200
Assembly	1	748	87	84
YAML	3	521	150	69
Java	4	438	187	143
Python	1	324	22	59
Text	5	245	0	34
PHP	1	114	0	20
Spec	1	91	1	21
JSON	1	45	0	0

Figure 22.8: codelines

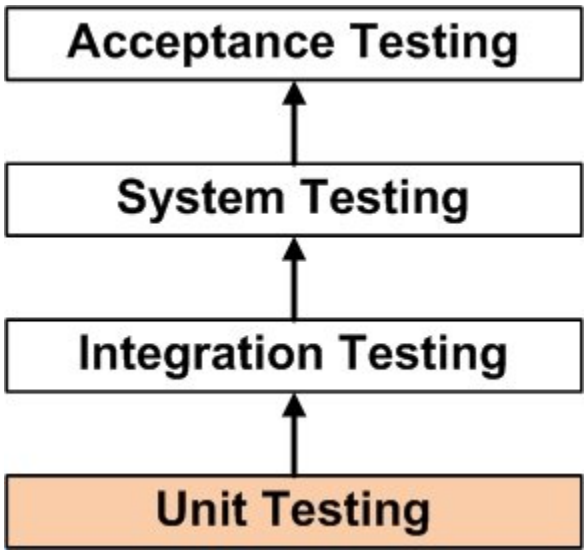


Figure 22.9: Test layer

22.4.2.1.3 Unit complexity The complexity of each unit has some relation to the unit size but contains more than just size. It also shows the complexity of control logic and code complexity. Cyclomatic complexity invented in 1976 by Thomas McCabe Snr is the most widely accepted metric for evaluating the code complexity. By measuring and reducing the code complexity, it brings several advantages like better tests, reduced risks, lower costs, and greater predictability. It can also help the developer to improve their programming skills.[47]

To improve this part, use smaller methods and reducing the if/else statements will help.[48] This may need a lot of effort to read through all the existing codes.

To achieve security and stability in Ripple's working scenario, methods, functions, and control logic are designed to be robust and complicated. Those codes are growing more and more complex during development. It will be useful to revisit their codes and design to see whether there are redundancy codes or meaningless control logic. This review can be handy in reducing unit complexity.

22.4.2.1.4 Module coupling and Component independence Scores for both Module coupling and Component independence on Sigrid are 5.5 and (N/A), respectively. One can easily argue that something must be wrong with these scores since zero interdependencies between components of the software as huge as Ripple seems impossible. As soon as we can use other code behavioral analysis tools, we will analyze these factors as well.

[47] https://blog.codacy.com/an-in-depth-explanation-of-code-complexity/
[48] https://www.axelerant.com/resources/team-blog/reducing-cyclomatic-complexity-and-npath-complexity-steps-for-refactoring

22.4.2.2 An assessment of technical debt

Ripple, as blockchain technology variations, aims to solve some real-world problems. [49] Ripple wants to build a global system of payments, settlements ad exchange. XRP is the cryptocurrency they use to solve the problem.

22.4.2.2.1 Debt about the blockchain technology The blockchain technology is a hot debt topic these years, mainly because of its decentralized design, anonymous system, and cryptocurrency.

22.4.2.2.2 Debt about the ripple approach (DLT and XRP ledger) Ripple as one of the blockchain variations, they chose to go their technical solutions. The DLT (Distributed Ledger Technology) they use is an open-source protocol hosting a shared and public ledger, using a consensus mechanism to secure security[50]. The DLT provides faster transaction speed and fewer transactions fee. The consensus is achieved by those validators, which is not anonymous and need to be proved by the ripple lab(The company who made Ripple). This idea against the decentralized idea inside the blockchain. In a word, Ripple is trading their decentralized for speed and fee.

22.4.2.2.3 Debt about the consensus algorithm The consensus algorithm ripple use is their RPCA algorithm. To achieve a fast speed of consensus, they using a UNL(Unique Node List) design among all servers. In this design, it requests several requirements, including the fault nodes is decreased to less than 20%; the network topology needs to avoid a single connection between two blocks.

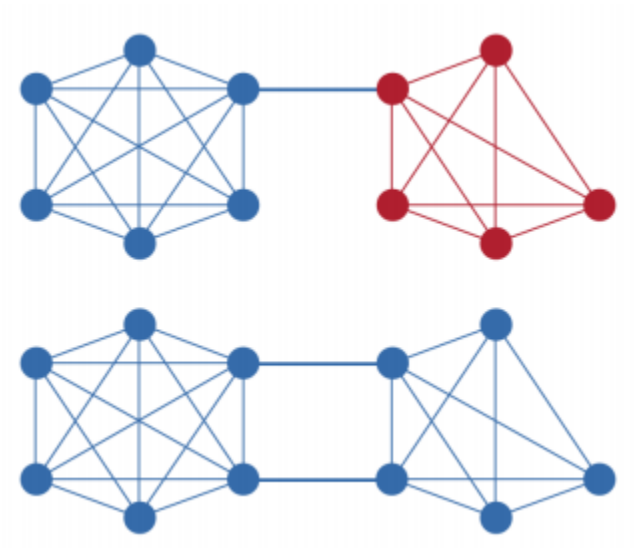


Figure 22.10: Network

For all those algorithms, the failure of the physical layer is hard to avoid. The Ripple approach requires monitoring of the global network structure to ensure some boundary conditions.[51]

[49] https://medium.com/@siddharth.sitpure/a-closer-look-at-ripples-blockchain-technology-and-xrp-9e036e1bf019
[50] https://www.bitdegree.org/tutorials/ripple-coin/
[51] https://ripple.com/files/ripple_consensus_whitepaper.pdf

22.5 Here To Stay : How Ripple Achieves Sustainable Development

The following essay details different approaches taken by Ripple to attain progress that not only satisfies the stakeholders but is also sustainable. The article examines Ripple's strategies that make sure that they achieve environmental, economic, societal, and technical development without compromising the ability of future generations to meet their needs.

22.5.1 Understanding Software Sustainability

Sustainability has emerged as a broad concern for society. Many engineering disciplines have been grappling with challenges in how we sustain technical, social, and ecological systems. Misperceptions among practitioners and research communities persist, rooted in a lack of coherent understanding of sustainability and how it relates to software systems research and practice. [52] Designers of software technology are responsible for the sustainability and long-term consequences of their designs. [53]

But what exactly is sustainability? There is a narrow perception of sustainability that frames it as protecting the environment or being able to maintain business activity. Whereas sustainability is at its heart a systemic concept, it does not confine only to the designed system, but also to the environmental, economic, individual, technical and social contexts of that system, and the relationships between them: [54]

- **Environmental**: concerned with the long term effects of human activities on natural systems. This dimension includes ecosystems, raw resources, climate change, food production, water, pollution, waste, etc.
- **Social**: concerned with societal communities (groups of people, organizations) and the factors that erode trust in society. This dimension includes social equity, justice, employment, democracy, etc.
- **Economic**: focused on assets, capital, and added value. This includes wealth creation, prosperity, profitability, capital investment, income, etc.
- **Technical**: refers to the longevity of information, systems, and infrastructure and their adequate evolution with changing surrounding conditions. It includes maintenance, innovation, obsolescence, data integrity, etc.
- **Individual**: refers to the well-being of humans as individuals. This includes mental and physical well-being, education, self-respect, skills, mobility, etc.

These dimensions are interdependent. Cumulative effects from the environmental dimension will bleed into the individual, social, economic dimensions. Yet, these dimensions provide a useful tool for dis-aggregating and analyzing relevant issues. [55]

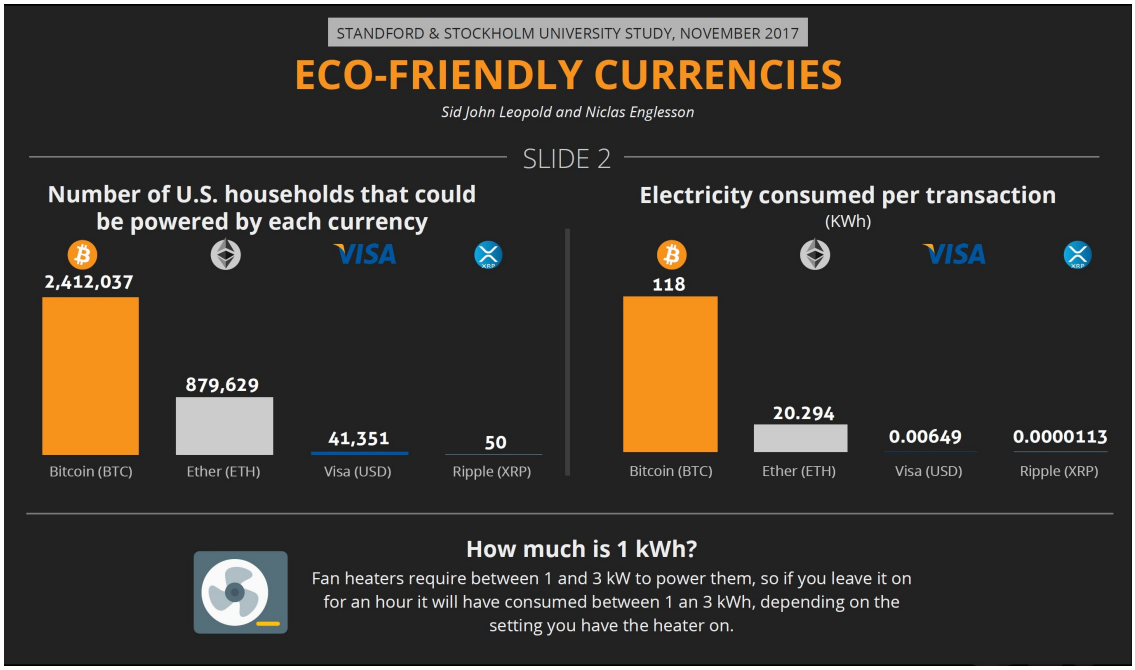
22.5.2 Ripple (XRP) Is The Most Energy Sustainable Currency

Softwares can contribute to environmental sustainability in various ways. With the crypto world overgrowing in size, many efforts are being made to optimize various processes such as cryptocurrency mining. Although there are different methods of mining for cryptocurrencies, the most common is PoW or Proof of Work, which is the mining method used by crypto giants like Bitcoin. This method has long been viewed as an expensive and time-consuming task and has a reputation for hogging energy.[56]

[52] https://elevenews.com/2020/03/01/98-percent-of-bitcoin-btc-mining-rigs-to-become-obsolete/
[53] https://elevenews.com/2020/03/01/98-percent-of-bitcoin-btc-mining-rigs-to-become-obsolete/
[54] https://elevenews.com/2020/03/01/98-percent-of-bitcoin-btc-mining-rigs-to-become-obsolete/
[55] https://elevenews.com/2020/03/01/98-percent-of-bitcoin-btc-mining-rigs-to-become-obsolete/
[56] https://www.cryptohopper.com/blog/199-green-cryptocurrencies

In 2017, a scientific study was done by Standford & Stockholm University to determine the most eco-friendly currency. In this study, which was conducted amongst Bitcoin (BTC), Ethereum (ETH), Ripple(XRP), and Visa, it was revealed that in terms of energy consumption rate, Ripple (XRP) stands as the best sustainable and most competent currency. [57]

Examining the annual electricity consumption rate of the four samples, Bitcoin consumed the most, topping with 26.05TWh, Ethereum 9.68TWh, VISA 0.54TWh. In contrast, Ripple (XRP) consumed the least energy with 0.0005361TWh. According to the report, Bitcoin consumes more electricity yearly than Scotland and Nigeria. After a careful analysis of the entire study, it was concluded that Ripple (XRP), amongst the other currencies, costs less and is most efficient in terms of electricity and CO2 emissions. [58]



A comparison study between BTC, ETH, VISA and XRP (Standford & Stockholm University Study, November 2017)[59]

So why Ripple is so power-efficient? Here we give two main reasons explaining how Ripple reduces energy consumption.

1. Improvements in the blockchain consensus mechanism skyrocketed energy efficiency.

   Since the creation of Bitcoin, proof-of-work (PoW) has been the predominant design of peer-to-peer cryptocurrency. It requires enormous amounts of energy, with miners needing to sell their coins to foot the bill ultimately.

   In contrast to Bitcoin and Ethereum, Ripple uses proof-of-stake (PoS), which doesn't use that much energy. Its concept states that a person can mine or validate block transactions according to how many coins he or she holds. It gives mining power based on the percentage of coins held by a miner.

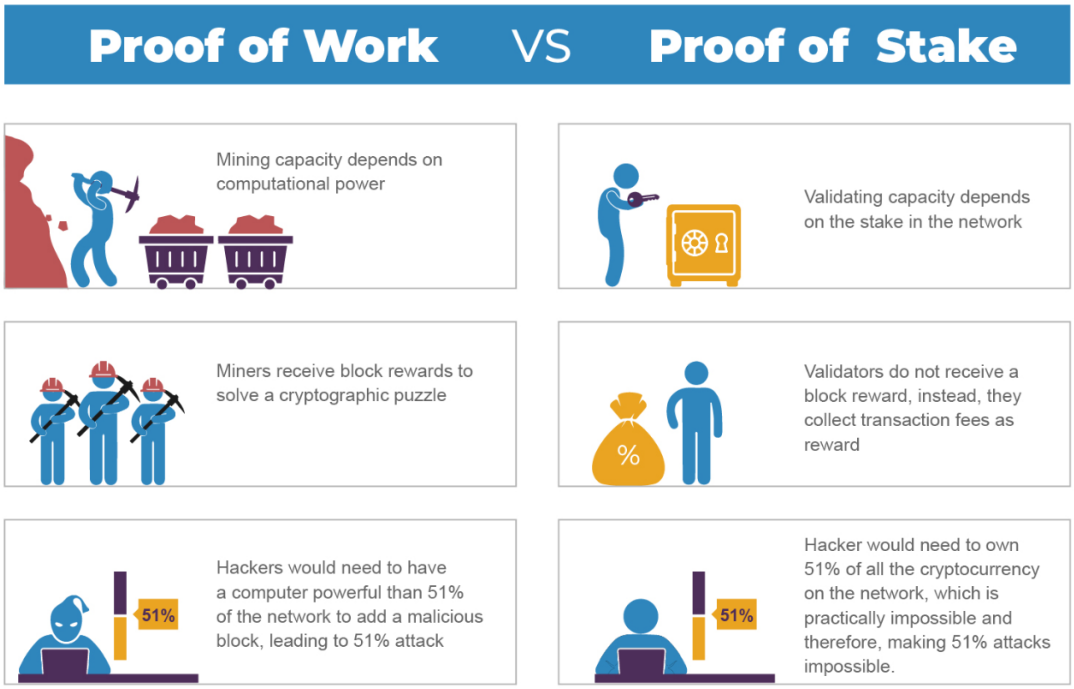
[57] https://newslogical.com/university-research-ripple-xrp-remains-the-most-energy-sustainable-currency/
[58] https://newslogical.com/university-research-ripple-xrp-remains-the-most-energy-sustainable-currency/
[59] https://www.stedas.hr/infographic-xrp-the-most-eco-friendly-currency.html

Another example is EoS[60], which applies the delegated proof-of-stake (DPoS) consensus mechanism. Similar to how basic PoS protocols work, DPoS does not depend on the energy-intensive mining process for validating blocks on its network. Compared with EOS, Ripple Labs' XRP-powered platform consumes only about 0.0005361 TWh per year, which is half as much as what the EOS network requires annually.

2. Adapting to power-efficient smart contracts.

   Most of the popular blockchain networks are not energy efficient. On the other hand, the primary resources of producing electricity are by burning fossil fuels[61], which, to some extent, adds up to the environmental challenges that the world is facing today. However, at the same time, it holds enormous opportunities when it comes to the energy grid and green energy financing.

Contrarily, Ripple uses certain functionalities such as Smart Contracts to open opportunities for lean, efficient, inclusive, and global green energy funding for both enterprises and end consumers. Recently, Ripple also revealed new partnerships[62] to better support Smart Contract Capability for XRP Ledger, which furthers the company's mission to execute eco-friendly payments.



[63]

Proof of Work v/s Proof of Stake

Even though Ripple is more eco-friendly and consumes less energy compared to other cryptocurrencies, there is still room for improvements. To ensure digital asset management, Ripple has made some trade-offs:

[60] https://www.cryptoglobe.com/latest/2018/10/eos-ripple-s-networks-consume-far-less-energy-than-bitcoin-and-ethereum-s-blockchain/

[61] https://www.express.co.uk/finance/city/904999/Bitcoin-price-live-ripple-green-issues-bring-down-crypto-giants-ethereum

[62] https://dailyhodl.com/2019/11/06/ripple-reveals-new-partnership-to-enable-smart-contract-capability-for-xrp-ledger/

[63] https://dzone.com/articles/the-proof-of-work-vs-proof-of-stake-an-in-depth-di

it's not just the cryptocurrency industry that consumes large amounts of energy, but also the network of banks, and payments firms that act as financial middlemen around the world. Moreover, even though Ripple XRP digital asset is far greener than Bitcoin, there are concerns on how truly decentralized it is, considering Ripple's control of more than 50 billion coins. The total supply of XRP is just shy of 100 billion. Thus, in order to 'green-wash' things, Ripple refuses to waste energy on a much more secure and decentralized network. Otherwise, they would have to sacrifice a part of their business relations.

22.5.3 Internet of Value Powers Economic and Social Sustainability

Aside from environmental footprints, a software can also have an impact on the economy. Economic sustainability can refer either to the continued success of an economy over time or to the way an economy operates sustainably, protecting social and environmental elements.

Ripple's vision is for value to be exchanged as quickly as information. Although information moves around the world instantly, a single payment from one country to another is slow, expensive, and unreliable. In the US, a typical international payment takes 3-5 days to settle, has an error rate of at least 5%, and an average cost of $42. Worldwide, there is $180 trillion worth of cross-border payments made every year, with a combined cost of more than $1.7 trillion a year. [64]

With the Internet of Value, a value transaction such as a foreign currency payment can happen instantly, and it's not just money. The Internet of Value will enable the exchange of any asset that is of value to someone, including stocks, votes, intellectual property, and more.

At the moment, there exists a multitude of competing blockchains that do not necessarily connect, so assets cannot be exchanged like information just yet. For the internet of value to become a reality, industry standards must be adopted to homogenize the world's different financial systems. [65]

This is why Ripple, along with a growing community of financial institutions and payment providers, support Interledger Protocol (ILP), which standardizes how to settle transactions across different ledgers and networks instantly. ILP can be thought of much like the protocol HTTP.

The most significant benefit of the internet of value will be for payments. Making cross-border payments faster, cheaper, and reliable will bring considerable benefits to consumers, businesses, banks, and governments. While also introducing a standard protocol for how every institution and individual connects across various networks to exchange data. Doing this will: [66]

- Connect billions of people around the world to transact
- Give rise to entirely new businesses and industries
- Increase financial inclusion for millions of under-banked consumers.

22.5.4 Achieving Technical Sustainability

Ripple manages to attain technical sustainability through different methods. With security in mind, they focus on adopting **Secure, Adaptable Cryptography.** The XRP Ledger relies on industry standard digital signature systems like ECDSA (the same scheme used by Bitcoin) but also supports modern, efficient algorithms like Ed25519. The extensible nature of the XRP Ledger's software makes it possible to add and disable algorithms as the state of the art in cryptography advances. Ripple also implements **On-Ledger**

[64] https://ripple.com/insights/the-internet-of-value-what-it-means-and-how-it-benefits-everyone/
[65] https://ripple.com/insights/the-internet-of-value-what-it-means-and-how-it-benefits-everyone/
[66] https://ripple.com/insights/the-internet-of-value-what-it-means-and-how-it-benefits-everyone/

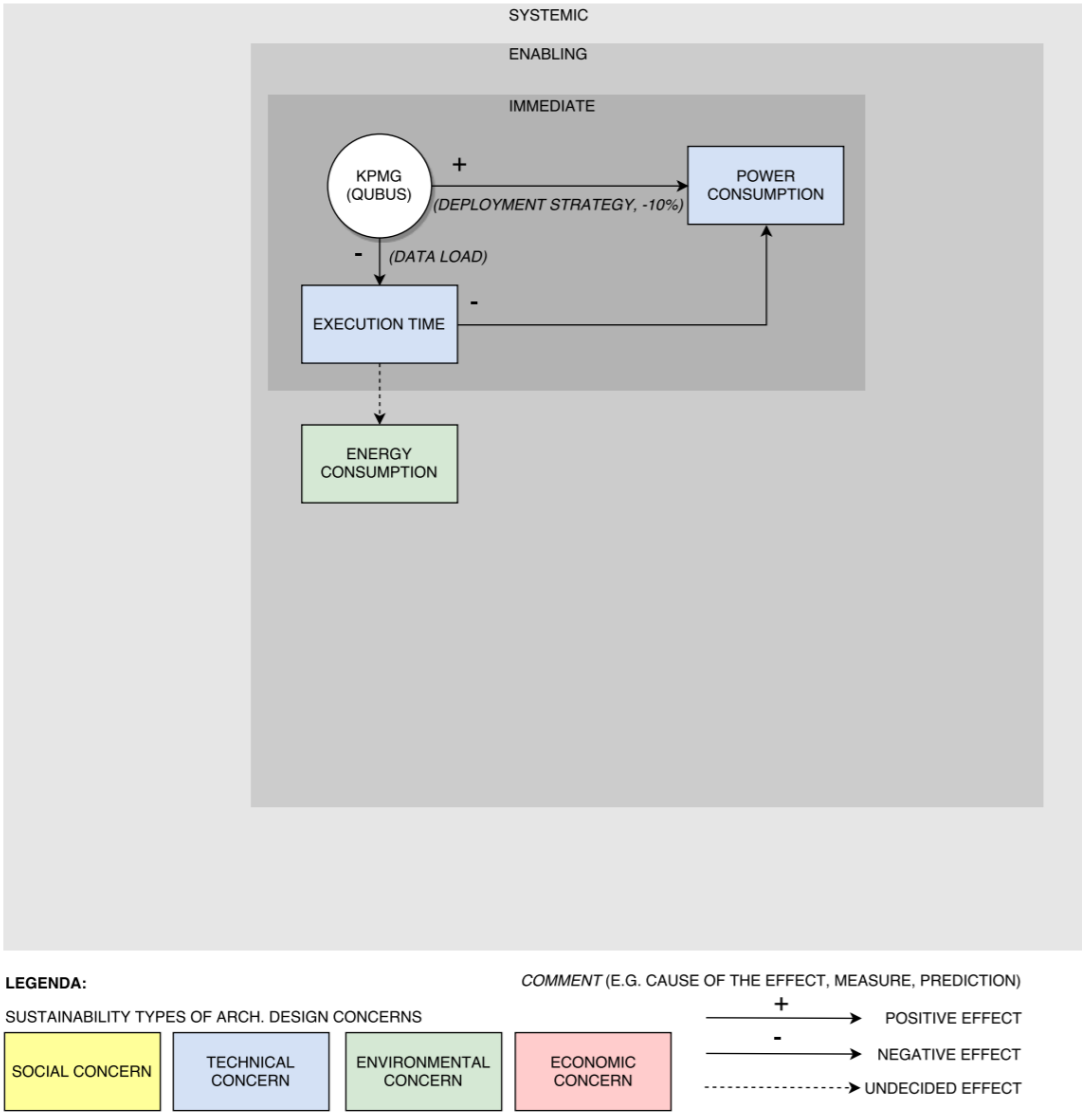
Decentralized Exchange. In addition to all the features that make XRP useful on its own, the XRP Ledger also has a fully-functional accounting system for tracking and trading obligations denominated in any way users want, and an exchange built into the protocol. The XRP Ledger can settle long, cross-currency payment paths and exchanges of multiple currencies in atomic transactions, bridging gaps of trust with XRP.

22.5.4.1 How Architecture Impacts Sustainability

As the vast energy consumption caused by mining bitcoin became a growing concern among peoples, more and more researches have come out to measure the energy consumptions and try to invest method to reduce it. Among those researches, software architecture is one of those approaches to make the software more sustainable.

In the paper [67], Patricia Lago introduced several different sustainability, including environmental sustainability, which is learned with the KPMG Qubus platform project. Here, the author observed that both deployment strategies and software release significantly influence the energy consumption of the hardware infrastructure. The highlighting result from this project is that the main feature responsible for negative effects on execution time is data load. i.e., the exchange of data between client- and server-side necessary for executing the user services.

[67] https://research.vu.nl/ws/portalfiles/portal/90224889/plago_seis_2019_submitted_version.pdf



[68]

KMPG

As mentioned in the second essay, Ripple uses a component-based software architecture and a client-server structure, which is widely used in distributed and embedded systems. Thus, compared to other (non-blockchain based) distributed systems, Ripple has a significant data usage. However, comparing to permissionless blockchain systems, a permissioned blockchain system like Ripple has better performance. In the data load perspective, the user of the Unique Node List(UNL) limits its consensus algorithm to achieve consensus among its set of servers[69]. Comparing to Bitcoin's consensus among all nodes, this is way less data load.

[68] https://research.vu.nl/en/publications/estimating-energy-impact-of-software-releases-and-deployment-stra
[69] https://cryptoguide.ch/cryptocurrency/ripple/whitepaper.pdf

22.5.4.2 Automated Tools To Predict Energy Cost

Chiyoung Seo, Sam Malek, and Nenad Medvidovic suggested a framework to estimate the energy consumption of a distributed software system at the level of its components and computational energy cost model for software components.[70] In this paper, they conducted further researches on Client-Server style systems, which is implemented by Ripple. An essential contribution of this work is its platform-independent characterization of the energy consumption of architectural styles.[71] Since we couldn't find any automated tools to check energy consumption for a system like Ripple, their work could be used in building such tools.

22.5.5 Conclusion

In conclusion, the essay discusses the various methods taken by Ripple to sustain their development progress. When compared to its counterparts in digital payments and blockchain systems, Ripple apparently fairs very well in tackling the environmental problems and helps to bridge the gap in modern payment systems. However, there are trade-offs that needs to addressed, like the case of amount of ledger usage and carbon emissions. We propose that since Ripple has an exemplar model eco-friendliness when compared with other crypto-currencies, they could compensate some of it to better their mining activities.

[70] http://seal.ics.uci.edu/publications/2008CBSE.pdf
[71] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.207&rep=rep1&type=pdf

Chapter 23

Scikit-learn



Scikit-learn (formerly known as scikits.learn) is an opensource machine learning library for Python. It features various preprocessing, classification, regression and clustering algorithms. What's most interesting about this library is that almost all these algorithms are implemented using the same API. This API has become so familiar that even other machine learning projects such as Keras have to chosen to use the same API design. This chapter contains a collection of views and perspectives on Scikit-learn's architecture.

The project was created by David Cournapeau as a Google Summer of Code project. The project has the SciKit, short for SciPy Toolkit because it builds on top of the SciPy library. In 2010 Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort and Vincent Michel took over leadership of the project and made the first public release in that same year. Currently, the project has become one the of the most well known general-purpose open-source machine learning libraries.

23.1 About us

We are four master students from Delft University of Technology, with backgrounds in Computer Science. - Toon de Boer - Thomas Bos - Jordi Smit - Daniël van Gelder

23.2 Scikit-learn, what does it want to be?

tags: `Scikit-learn Python Software Architecture Machine Learning Developers Roadmap Stakeholders`

In this first blog post, we will examine the following aspects of scikit-learn. First, we will describe what scikit-learn is and what it is capable of doing. Then we will describe what stakeholders are involved in the project and finally, we will lay out a roadmap of future development of scikit-learn. This essay thus gives the necessary context for anyone how wants to study the architecture behind scikit-learn.

23.2.1 Scikit-learn: All Machine learning models with a single API

Let's start by discussing: what is the problem scikit-learn wants to solve? Scikit-learn is a machine learning (ML) library containing a lot of ML models and techniques. Scikit-learn has been designed to be simple and efficient, accessible to non-experts, and reusable in various contexts.

Scikit-learn is very cautious in its selection of techniques. For example in their selection of new techniques they only consider techniques whereby it is at least 3 years since its publication, it has 200+ citations and it fits in the API of scikit-learn.

All algorithms, both learning and pre-processing, in scikit-learn have been implemented with the same `fit`, `predict` and `transform` API. As soon as you have learned this API you can use any algorithms without knowing the exact details of how it works. It also makes algorithms, for the same learning problem, interchangeable in the code. The API also hides all the complex optimization choices that have to be made. You can control these by changing the hyper-parameters of the `estimator`. The effects of these choices have been well documented in the API documentation and the provided tutorials of scikit-learn. However, in most cases, this won't be necessary as one of scikit-learn's core design principles is to provide sensible defaults. Making scikit-learn efficient and easy to use for non-experts.

23.2.2 Key Capabilities and Properties

Knowing the problem scikit-learn tries to solve, we will provide the key capabilities and properties of the library and discuss how far the library is able to solve ML problems. The scikit-learn library includes a lot of ML algorithms, both supervised and unsupervised[1]. Moreover, scikit-learn provides algorithms to tune hyper-parameters automatically.

For supervised learning, the problem can either be a classification or regression problem [2]. Scikit-learn provides lots of algorithms for these types of problems such as linear models, Bayesian Regression, k nearest neighbours, the decision tree, etc.

Scikit-learn also provides algorithms to solve unsupervised learning problems[3], such as clustering, density estimation, etc. The choice of algorithms per problem is also quite diverse as is shown in the image below.

The image clearly shows how different algorithms perform on different input data. All clustering algorithms require unlabeled data and a function so that it will return an array of labeled training data.

[1] https://scikit-learn.org/stable/tutorial/basic/tutorial.html#machine-learning-the-problem-setting
[2] https://scikit-learn.org/stable/supervised_learning.html#supervised-learning
[3] https://scikit-learn.org/stable/unsupervised_learning.html#unsupervised-learning

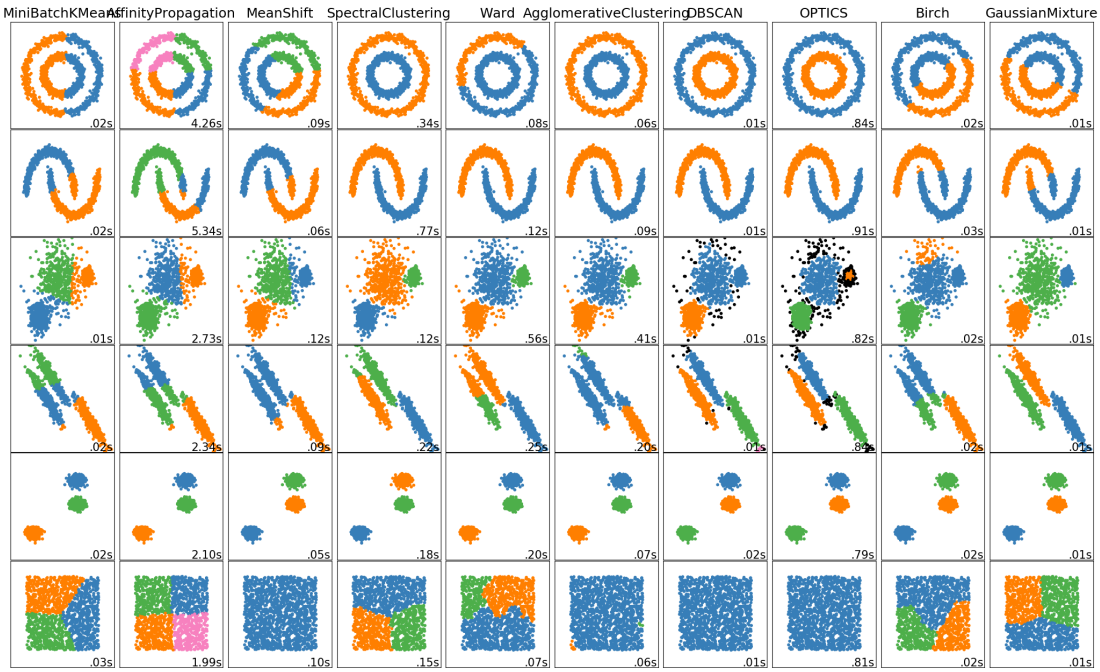


Figure 23.1: Different types of clustering algorithms

Tuning hyper-parameters can be a lot of work, since it is not always clear what the impact of different values for those parameters are. Luckily Scikit-learn provides a tool[4] to automatically find the best set of hyper parameters via cross validation.

23.2.3 Presenting the solution to the users

So that is the problem scikit-learn wants to solve. Now let's discuss how scikit-learn presents its solution to its users. Let's discuss this using the end-user mental model from the book: Lean Architecture [5]. This model consists of two parts. First, let's discuss the "What is the system?" part. For the users, scikit-learn is an API that consists of 3 types of core components:

Component	Description
Estimators	Every algorithm, both learning and data-processing, in scikit-learn are implemented as an *estimator*. This creates a universal way of initializing algorithms and retrieving both the hyper and learned parameters.
Predictors	These are *estimators* that implement the predict & score related methods. These are for example algorithms that do classification, regression, clustering, etc.

[4] https://scikit-learn.org/stable/getting_started.html#automatic-parameter-searches
[5] Lean Architecture: for Agile Software Development by James O. Coplien, Gertrud Bjørnvig

Component	Description
Transformers	These are *estimators* that implement the transform related methods. They are mostly used to modify or filter data as a pre-processing step before feeding the data to Predictors. For example, both PCA and one hot encoding are transformers.

Scikit-learn also has 3 more components that are based on the composition of estimators:

Component	Description
Meta estimators	These are *estimators* that combine one or more *estimators* into a single estimator. This allows us to transform a single classification *estimator* into a multi-classification *estimator* or to reuse other *predictors* in an ensemble method.
Pipelines	*Pipelines* combine multiple *transformers* and their final *predictor* into a single estimator. This means that all fit, transform and predict methods are combined into a single method. Making a *pipeline* less error-prone for future predictions.
Model selector	These are *meta-estimators*, that will train the *estimator* multiple times with different values for the hyper-parameters when the fit method is called. It will then expose the best_score_, best_params_ and best_estimator_ as attributes.

Python does **not** have interfaces. That is why scikit-learn implements these components via duck typing and estimator tags.

Now, let's discussed the "what the user does with the system" part using these components. Scikit-learn provides its users with the ability to train and use ML models. Which means that a typical scikit-learn user

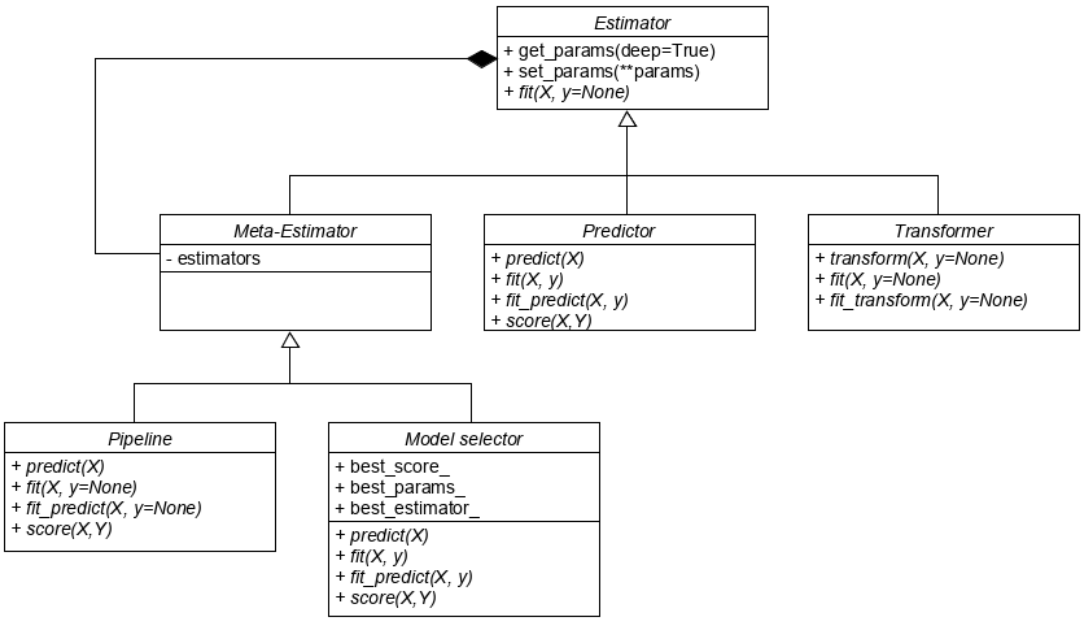


Figure 23.2: UML of the scikit-learn components

will encounter 2 types of use-cases, whereby the user encounters all the previously defined components.

First, let's look at the model training use-case. Hereby the user selects a *predictor* and one or more *transformers* to pre-process the data. He then combines these into a single *pipeline*. The user then searches for the best set of hyper-parameters using *model selection*. And finally, the resulting model is tested and if deemed sufficiently accurate is stored using pickle to await usage in production.

The second use case is using the *predictor* in production. In this use-case the user loads the pre-trained model from disk. This restores the entire pipeline the user has created in the first use-case. Allowing the user to start making predictions without setting up a new pre-processing pipeline as this is already part of the loaded model.

This is scikit-learn solution and how it presents it to its users. However one question still remains.

23.2.4 Who is involved?

For any project it is crucial to identify the stakeholders, as they determine how to project evolves. More importantly, if a project fails to identify stakeholders, it might abandon its audience which would make the project irrelevant. An extensive stakeholder analysis has been performed previously in the 2017 edition of the course [6]. The current analysis builds heavily upon that edition. However, the project has evolved since then and it is important to clarify the changes. We propose a similar categorization of stakeholders to the previous analysis. That is, a categorization as follows:

[6] https://delftswa.gitbooks.io/desosa-2017/content/scikit-learn/chapter.html

- **Contributors**: Anyone contributing to the *code base* in some form.
- **Users**: Organizations or individuals that utilize scikit-learn.
- **Funders**: Entities that support scikit-learn in some form, either material (e.g. providing servers) or monetary.
- **Competitors**: Tools/libraries/frameworks that are similar to scikit-learn and might target the same userbase.

Stakeholder Category	Stakeholder
Contributors	Core contributors (Jérémie Du Boisberranger, Joris Van den Bossche, ...), Community Contributors (> 1600)
Funders	Members of the Scikit-Learn Consortium, Columbia University, Alfred P. Sloan Foundation, The University of Sydney, Anaconda Inc
Users	J.P. Morgan, Spotify, Inria, betaworks, Evernote, Booking.com, and more
Competitors	TensorFlow, PyTorch, MLlib (Spark), Dask

The table shows an overview of the relevant stakeholders according to this categorization at the current state of the project. Note that there are several differences compared to 2017. This probably follows from the changes in context that the system operates in as well as general project evolution. The relevant stakeholders were derived from the project's website [7]. Note that in particular, the list of competitors has changed compared to 2017.

Having described the role of the stakeholders in scikit-learn, we now look forward to see its role in the world in the future.

23.2.5 What can we expect from scikit-learn in the future?

At the time that scikit-learn was developed and released, none of the now equally popular ML frameworks (e.g. PyTorch or TensorFlow) existed. Therefore, scikit-learn was the pioneer in providing a framework for ML. The first public release was in 2010 and consecutive development benefitted greatly from a large international community. PhD students in ML formed a significant part of the contributor community. However, currently these students are more likely to contribute to one of the other popular ML frameworks (as acknowledged by the authors of scikit-learn). Scikit-learn currently operates in a time where ML is experiencing unprecedented popularity. So how will this library continue to grow in such a different context?

Although scikit-learn has been a prominent framework within the ML world, it has also suffered from the paradigm shift within the field. People tend to move to deep learning oriented frameworks like PyTorch and TensorFlow.

The graph above [8] depicts the code frequency of the scikit-learn github repository. The blue arrow indicates the launch date (November 2015) of Tensorflow. The release of the, deep learning capable, framework seems to be a likely cause for the decline in attention for scikit-learn.

Even though the decline of contributions is acknowledged by the creators of scikit-learn [9], they are not

[7] https://scikit-learn.org/stable/about.html
[8] scikit-learn_code_frequency: https://github.com/scikit-learn/scikit-learn/graphs/code-frequency
[9] scikit-learn_roadmap: https://scikit-learn.org/stable/roadmap.html



Figure 23.3: Code frequency of the scikit-learn github repo

less encouraged to provide a high-quality, fully maintained and well-documented collection of ML and data processing tools. The methods used to realize this objective have improved with more advanced computational tools and better high level python libraries like Cython and Pandas [10].

The main goals then, in this era of the project, as stated by the scikit-learn team are to [11]:

- maintain a high-quality, well-documented set of ML and data processing tools within the scope of interest of the scikit-learn project (predicting targets with simple structure).
- make it easier for users to develop and publish external components.
- improve integration with modern data science tools and infrastructures.

We think that keeping the toolkit powerful, yet easy to access, keeps it a gateway into the world of ML, but that making sure that the supported frameworks are up to date will help it to stay relevant in this era.

23.3 From Vision to Architecture

tags: `Scikit-learn` `Python` `Software Architecture` `Machine Learning` `Developers` `Roadmap` `Stakeholders`

Scikit-learn's main goal is to make machine learning as simple to use for non-experts while remaining as efficient as possible. To do this scikit-learn has to hide all the complexities and variations between the different machine learning algorithms. In this blog post, we will explore how these requirements have resulted in the current architectural style and which trade-offs have been made to achieve it.

23.3.1 Architectural design choices

Scikit-learn has made 3 key architectural design choices to implement its goals. Firstly, scikit-learn truly embodies the design idiom: "program an interface, not an implementation". From a users perspective, all estimators have the same API.
Even though they are doing completely different things behind the scenes [12]. They make the usage of different machine learning algorithms as easy as learning the very small and well-defined API, while still allowing developers to change the behaviour behind the scenes through hyper-parameters. Secondly, scikit-learn is designed as a library, not a framework. There is no need to inherit from a specific class. Users only need to import the required modules and instantiate the object with the desired functionality. This design property makes scikit-learn easier to use. It also helps the developers to create a more modular structure in the code. Thirdly, the creation of an estimator has been decoupled from the learning process. Hyper-parameters are provided at the creation of the estimator. When the `fit` method is called with the training data the hyper-parameters are considered frozen. This design choice is similar to the idea of currying. This choice allows users to combine multiple estimators into a single pipeline estimator with the same API. These design choices also put some additional constraints on the non-functional requirements. Which meant that some trade-offs had to be made.

23.3.2 Non-functional trade-offs

To achieve its goals scikit-learn had to make 3 architectural design trade-offs:

[10] scikit-learn_roadmap: https://scikit-learn.org/stable/roadmap.html
[11] scikit-learn_roadmap: https://scikit-learn.org/stable/roadmap.html
[12] # Scikit-learn, what does it want to be? https://desosa2020.netlify.com/projects/scikit-learn/2020/03/06/scikit-learn-what-does-it-want-to-be.html

- **Readability vs Performance:** Readability is, in general, deemed more important as this helps to improve understanding and ease of maintainability. When implementing algorithms it is prefered to implement them in Python using Numpy and SciPy. To make use of their readability and vectorized performance properties. However, in some cases, it is not possible to efficiently implement algorithms using only vectorized code. Only then is it allowed to implement the bottleneck parts using Cython to improve performance [13].
- **Performance vs API design:** Scikit-learn wants to gain as much performance from vectorization as possible [14]. Vectorized algorithms achieve higher performance on batched inputs rather than single sample inputs. That is why scikit-learn's API has been designed to prefer batched inputs over single sample inputs [15]. This is also the reason why data is encoded as NumPy arrays instead of objects.
- **Usability vs Decoupling:** In its design, scikit-learn has decided to combine the model factory (estimators) and the trained model produced by the factory (models) into a single object. This increases useability since there is no need for a second class. It also increases understandability as there are no parallel class hierarchies. Due to this choice users have to deal with more complex software dependencies in some specific cases, such as exporting fitted models [16]. However, since these there are fewer of these cases, scikit-learn has chosen usability over decoupling.

We now know the key architectural choices that have been made to implement scikit-learn. Now let's look at how they affected the development, runtime and the deployment view.

23.3.3 Development view

The development view "describes the architecture that supports the software development process"[17]. Rozanski et al. specify six main concerns and we will cover them in this section.

23.3.3.1 Module Organisation

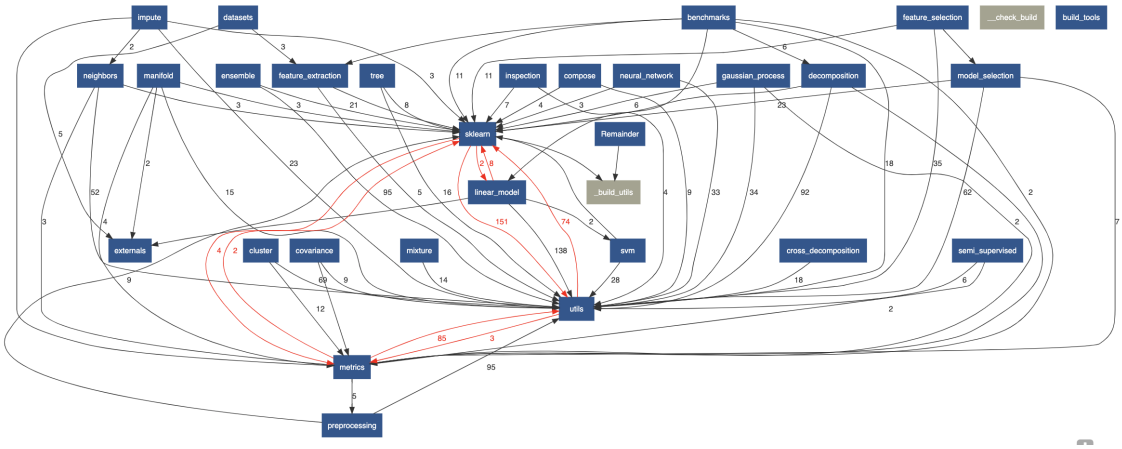
Our colleagues from 2017 [18] have already discussed the module organisation aspects. They have, for example, already discussed the grouping of related code in the code base[19]. The structure of scikit-learn has not changed a lot over time, so we will extend on their work by adding other viewpoints and other common practices within scikit-learn development with help of the other five criteria.

23.3.3.2 Common Processing

It is important to identify and isolate common processing into seperate code units[20] and in Figure 1, which contains the dependency graph of the scikit-learn code base, we can see that scikit-learn has achieved this

[13] How to optimize for speed, https://scikit-learn.org/dev/developers/performance.html
[14] How to optimize for speed, https://scikit-learn.org/dev/developers/performance.html
[15] API design for machine learning software: experiences from the scikit-learn project, https://arxiv.org/pdf/1309.0238.pdf?source=post_elevate_sequence_page
[16] API design for machine learning software: experiences from the scikit-learn project, https://arxiv.org/pdf/1309.0238.pdf?source=post_elevate_sequence_page
[17] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[18] DESOSA 2017, https://pure.tudelft.nl/portal/files/37061591/desosa_2017.pdf
[19] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[20] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

with three packages: `sklearn`, which is the base package, `utils`, and `metrics`, which contains code for evaluating prediction performance[21]. `utils` contains utilities specified by a list of common development practices which means tools in several categories such as linear algebra & array operations, matrix operations, random sampling and input validation tools. Furthermore, it is encouraged to let the NumPy and SciPy libraries handle as much of the processing as possible[22]. As stated earlier, scikit-learn has also added support for Cython which allows c-like performance within Python[23]. The use of Cython is encouraged when "an algorithm cannot be expressed efficiently in simple vectorized NumPy code"[24].



The dependency graph of the scikit-learn code base.

23.3.3.3 Standardisation of Testing

Testing in machine learning frameworks is very important as bugs do not necessarily have to be apparent on the surface. The code can still run without errors or slow-downs and networks will still train and losses will still drop [25]. Scikit-learn is no exception. Thus scikit-learn expects all code to have a test coverage of at least 90%. All testing is done using the pytest[26] package and each package in de code base has a dedicated test folder.

23.3.3.4 Instrumentation

Rozanksi et al. define instrumentation as "the practice of inserting special code for logging information about step execution, system state, resource usage, and so on that are used to aid monitoring and debugging" [27]. However, there are no best practices specified on instrumentation in the scikit-learn guide, but there are practical guidelines for optimization using IPython[28] which will highlight issues in the code regarding memory usage and processing bottlenecks[29].

[21] Model Evaluation, https://scikit-learn.org/stable/modules/model_evaluation.html#model-evaluation
[22] How to optimize for speed, https://scikit-learn.org/dev/developers/performance.html
[23] Cython, https://cython.readthedocs.io/en/latest/
[24] How to optimize for speed, https://scikit-learn.org/dev/developers/performance.html
[25] How to unit test machine learning code, https://medium.com/@keeper6928/how-to-unit-test-machine-learning-code-57cf6fd81765
[26] pytest: helps you write better programs, https://docs.pytest.org/en/latest/
[27] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[28] IPython Documentation, https://iPython.readthedocs.io/en/stable/
[29] How to optimize for speed, https://scikit-learn.org/dev/developers/performance.html

23.3.3.5 Codeline Organisation

All code resides on GitHub and multiple continuous integration (CI) systems are used. Azure CI is used for platform specific (e.g. Windows/Linux/macOS) testing, CircleCI is used to build documentation for viewing and some Linux specific testing[30]. Travis CI is used for development builds where also non-stable libraries are used. These CI jobs run in set intervals as opposed to after each pull request because their failure does not necessarily have to do with errors in pull requests[31]. Furthermore, scikit-learn specifies a lot of guidelines regarding contributing code and for the pull request reviewing process in order to guarentee stability[32]. Backward compatibility is delivered by supporting deprecated methods for two releases and annotating them as deprecated in that time[33].

23.3.3.6 Standardisation of Design

The main design pattern used in the scikit-learn codebase is the fit-predict-transform-estimator API we discussed in our first post "Scikit-learn, what does it want to be?" so we recommend reading that if you have not already.

Now that we considered the design from a development perspective, we focus on the concerns a user might have at runtime.

23.3.4 Runtime view

A runtime view defines how components interact at runtime to realize key scenarios [34]. In the case of scikit-learn, this involves the entire pipeline as described in the previous section. First, let's describe the (runtime) dependencies that scikit-learn contains:

- Python, version: `3.5+`
- Numpy, version: `1.14.0+`
- Scipy, version: `1.1.0+`
- Matplotlib, version: `1.5.1+` (only for plotting capabilities)
- scikit-image, version: `0.12.3+` (only for some examples on website)
- pandas, version: `0.18.0+` (only for some examples on website)

Having established runtime dependencies, let's investigate how the components described previously interact at runtime.

One of the great aspects of scikit-learn from an architectural perspective is that the architectural design allows for a very modular specification of the model. It is therefore easy for a user to adapt his/her model to changing needs. The only interaction with each model is through the functions: `fit`, `predict` and `transform`. Therefore, at runtime, the dependencies are only through the `estimator`s and their nested calls. Scikit-learn has, as mentioned in the previous essay, two main use-cases: Selecting a model and processing the data to train the model and applying the model in production.

In the first use case, the user selects a model to use for fitting the (labelled or unlabelled) data. Data can also be preprocessed in some way (separating labels from features, etc). Data must be provided in the form of a `NumPy` array. Once the user has selected a model and has formatted the data correctly, he can begin to

[30] Contributing, https://scikit-learn.org/dev/developers/contributing.html
[31] Maintainer / core-developer information, https://scikit-learn.org/dev/developers/maintainer.html
[32] Contributing, https://scikit-learn.org/dev/developers/contributing.html
[33] Contributing, https://scikit-learn.org/dev/developers/contributing.html
[34] https://www.viewpoints-and-perspectives.info/home/viewpoints/operational/

train the model by calling the `fit` function on the data. This returns the model with trained parameters. Thus the same model is returned, but its internal state has changed. This reveals the clever architectural design discussed previously. While the user is able to interact with all the models in a similar fashion, the complexity that runs behind that interface is hidden away from the user. Also, as described earlier, the user needs not implement any classes in order to use scikit-learn to train a model.

The second use case involves using the model trained previously and applying it to an (unlabelled) data set. Data can be classified using the `predict` function of the model. This returns labels which can consequently be used for plotting, checking, etc. Again, the architectural design that we have discussed so far hides away the complexity of applying the model to the data. In addition, the interaction during end-to-end execution is only with the `scikit-learn` library. This hides away all dependencies that might occur if it were implemented with inverted control.

There are many more involved use cases, but the previously highlighted scenarios illustrate the general execution of the project in a runtime fashion. More examples of use-cases can be found on the website.

23.3.5 Deployment view

Now that the runtime view has been established, we will consider the deployment view. This describes how the application is deployed such that the end-user, developers who want to use scikit-learn, can use it. We will illustrate how the system can be deployed on a computing system and what (runtime) requirements the system imposes on that system [35].

Scikit-learn can be deployed through Python package managers such as pip and conda. The installation is relatively straightforward, the user simply has to type the installation command for each respective package manager: `pip install scikit-learn` (pip) or `conda install scikit-learn` (conda).

However, to deploy scikit-learn, there are some minimum system requirements. The theoretical minimum requirements are any minimal computing machine that can run Python and has enough disk space to install all the dependencies. Python does not officially have any minimum hardware requirements, but one can assume that anything that can perform basic computing tasks can run Python (Python is known to being able to run on small devices like a Raspberry Pi).

In practice, however, one would require a much more sophisticated computing system to run scikit-learn. Since machine learning problems often involve repeated intense calculation on big data sets, one would expect a system running scikit-learn to have above-average hardware specifications.

23.3.6 Conclusion

In this essay, we have explored how the previously outlined vision of scikit-learn has been translated into its architecture. One of the advantages of scikit-learn's API design is that it allows for a very modular use where one model can be easily replaced by another for changing requirements. This essay also considered all components and provided an analysis of the six concerns outlined by Rozanski et al. [36]. Furthermore, we considered other views of the architecture: the runtime, deployment and non-functional view.

[35] https://www.viewpoints-and-perspectives.info/home/viewpoints/deployment/

[36] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

23.4 Scikit-learn's plan to safeguard its quality

tags: `Scikit-learn` `Python` `Software Architecture` `Machine Learning` `Developers` `Code Quality` `Testing` `Coverage` `Contributions`

In our previous blog posts [37] [38], we examined the vision and goals behind scikit-learn and discussed how these elements have been combined into the underlying software architecture. We considered several views and perspectives in which the software product operates and the trade-offs that were made to balance functional requirements with non-functional requirements.

In this blog post, we will evaluate how scikit-learn safeguards its architectural, code and testing quality. Ensuring quality is essential for any open source project since contributors might have a wide variety of quality standards or use different conventions for programming. Therefore a project should specify clear guidelines that people can use. In addition, we will relate the latest code changes to the roadmap specified by the authors.

The first section discusses the concrete guidelines that are in place for contributing. Then we will investigate how these guidelines are used in practice, followed by an analysis of how effective the guidelines are. Finally, we investigate the current focus of the project and how it relates to the quality guidelines as well as the previously specified roadmap.

23.4.1 The plan

Before evaluating the state of code quality in the project, we will consider how the project managers encourage contributions to the project. Proper guidelines are crucial for contributors to be able to make a valuable contribution.

The repository of scikit-learn on Github links to several documents providing guidelines for contributors on the website [39] [40]. In addition, the repository contains a `CONTRIBUTING.md` file providing a summary of the information mentioned in the larger documents on the website.

The extend of the guidelines indicates that the authors of scikit-learn appreciate community contributions to the project and are certainly welcoming to changes and contributions to the project. Let's investigate what the specific guidelines specify.

23.4.1.1 PR/Issue guidelines

Formulating clear issues and pull requests are important to the project. There are specific guidelines for bug reports in particular. Bug reports need to contain extensive information regarding the issue and the surrounding environment in which the problem occurred. (stacktrace, code snippets, OS version, etc.).

There are templates for both issues and pull requests. Issues fall into five different categories: Bug report, Documentation improvement, Feature request, Usage questions and "Other". Each type of issue has its associated guideline. Pull requests are not divided into sub-categories as they all follow the same template.

[37] Scikit-learn, what does it want to be?, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/06/ scikit-learn-what-does-it-want-to-be.html

[38] From Vision to Architecture, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/15/ from-vision-to-architecture-20893.html

[39] Scikit-learn's developers guide page, https://scikit-learn.org/stable/developers/index.html

[40] Contributing guidelines for developers of scikit-learn, https://scikit-learn.org/stable/developers/contributing.html

However, contributors must follow a checklist [41] before they are allowed to submit their pull request. During code review, the scikit-learn maintainers will take a set of code review guidelines [42] into account.

23.4.1.2 Coding guidelines

Using guidelines scikit-learn is able to maintain a standardised format over the whole codebase, with over 1600 contributors [43]. There is a special chapter dedicated to contributing code [44] which specifies how you can contribute. This process is a bit more involved then branching from the main repository and changing the code. The authors recommend users to *fork* the repository on Github and build scikit-learn locally.

There are no specific guidelines on coding conventions/style but the review guidelines [45] specify to what quality standards the code must comply. Code needs to be consistent with the scikit-learn API as described in our previous essay [46], as well as have necessary unit tests associated with each component. Readability and documentation are also important for a good contribution. Finally, and more relevant to the specific case of scikit-learn, the efficiency of the code is also taken into account during the review. Some guidelines on coding style and conventions can be found spread across the guidelines for contributing page [47].

23.4.1.3 Test guidelines

Scikit-learn has a very high test coverage, as mentioned in the previous post [48]. This implies that testing is considered very important for developers of scikit-learn. The guidelines [49] specify that all public function/class should be tested with a reasonable set of parameters, values, types and combinations between the former. In addition: the guidelines refer to an article explaining conventions for writing tests in python in general [50].

23.4.1.4 List of CI Tools

Scikit-learn uses a variety of tools for Continuous Integration (CI). Every contribution must be guaranteed to not cause faults in different parts of the system. Depending on the part of the system that is affected, a commit might have up to 19 CI checks!

Three tools are used to run CI [51]: - Azure Pipelines is used to test scikit-learn on different Operating Systems using different dependencies and settings; - CircleCI is used to building the documentation, linting the code and testing with PyPy on Linux; - CodeCov is used to measure code coverage of tests across the system.

---

[41] Scikit-learn pull request checklist, https://scikit-learn.org/dev/developers/contributing.html#pull-request-checklist

[42] Code review guidelines for scikit-learn, https://scikit-learn.org/dev/developers/contributing.html#code-review

[43] List of contributors of scikit-learn, https://github.com/scikit-learn/scikit-learn/graphs/contributors

[44] Code review guidelines for scikit-learn, https://scikit-learn.org/dev/developers/contributing.html#code-review

[45] Code review guidelines for scikit-learn, https://scikit-learn.org/dev/developers/contributing.html#code-review

[46] From Vision to Architecture, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/15/ from-vision-to-architecture-20893.html

[47] Contributing guidelines for developers of scikit-learn, https://scikit-learn.org/stable/developers/contributing.html

[48] From Vision to Architecture, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/15/ from-vision-to-architecture-20893.html

[49] Code review guidelines for scikit-learn, https://scikit-learn.org/dev/developers/contributing.html#code-review

[50] Jeff Knupp, Improve Your Python: Understanding Unit Testing, Dec 9, 2013, https://jeffknupp.com/blog/2013/12/09/improve-your-python-understanding-unit-testing/

[51] Contrinous Integration specification for scikit-learn: https://scikit-learn.org/stable/developers/contributing.html#continuous-integration-ci

23.4.2 Execution

All these guidelines and CI tools are only useful if they are actually effective and are being consulted by developers. In this section, we explore the impact of these guidelines and CI tools.

23.4.2.1 Code coverage overview

The pull request reviewing guidelines state all code added in a pull request should aim to be 100%. Untested lines are only accepted with good exceptions[ˆscikit-learn-contributions]. With this guideline, scikit-learn aims for total code coverage of at least 90%[52]. When running the code coverage tools on version 0.23.dev0 we found a branch coverage of 96%. This shows that the goals on code coverage set by the scikit-learn guidelines are realised.

23.4.2.2 Effect of failing tools in Pull Requests

Scikit-learn developers aim to maintain a high level of code quality in the master branch of the scikit-learn repository. To illustrate this, we added a selection of comments from pull requests. In these images, it is clear to see that even small issues like code quality warnings are addressed.

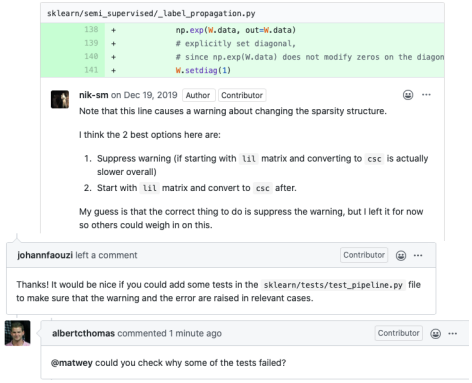


Figure 23.4: Comments from pull requests 15922, 16714, and 16721 respectively.

Furthermore, investigation of the list of pull requests reveals that new features are almost never merged when the pipeline is failing.

23.4.3 Analysis of the effectiveness

In this section, we will first discuss the SIG results and how they are related to the code quality plans. Then we will discuss how these results are related to the technical debt in scikit-learn.

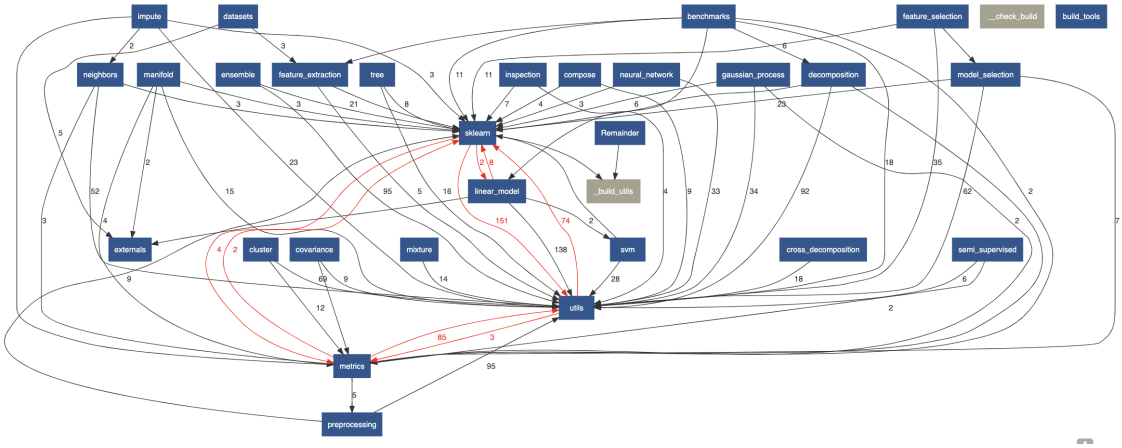
23.4.3.1 The relationships between the SIG results and the code quality plan

SIG has run a maintainability analysis on a snapshot of the master repo from 20-02-2020.

[52]Contributing guidelines for developers of scikit-learn, https://scikit-learn.org/stable/developers/contributing.html

Type	Rating
Test code ratio	120.2 %
Volume	4.3 / 5
Duplication	4.0 / 5
Unit size	1.7 / 5
Unit complexity	1.6 / 5
Unit interfacing	0.5 / 5
Module coupling	3.6 / 5
Component balance	4.5 / 5
Component independence	2.3 / 5
Component entanglement	2.1 / 5

There are four important things to note in this table. Firstly, there are more lines of test code than lines of production code. Which means that the test guidelines have paid off. Secondly, the duplication and the module coupling metrics have also scored very high. This is an indication that the strict review guidelines also have paid-off. Thirdly, the unit related metrics score very poorly. This is mostly caused by the relatively high amount of arguments used in scikit-learn functions. Scikit-learn prefers to improve useability and readability by using keyword arguments instead of encapsulating these parameters in an object [53]. All the keyword arguments also have a default value to improve usability, which increases the McCabe complexity [54] as a side effect.



Finally, the component independence and entanglement are a bit on the low side. This is caused by the cyclic dependencies between the `sklearn`, `metrics` & `utils` modules as can be seen in figure. Which is interesting because there are no specific guidelines or plans to prevent these kinds of issues.

23.4.3.2 Technical debt and possible refactor options.

SIG has also provided a list of possible refactoring candidates that will help to reduce technical debt. We will discuss the three refactor categories we deem the most useful for scikit-learn.

[53] Scikit-learn, what does it want to be?, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/06/ scikit-learn-what-does-it-want-to-be.html

[54] Cyclomatic complexity on Wikipedia, https://en.wikipedia.org/wiki/Cyclomatic_complexity

Firstly, the McCabe complexity in some functions is very high. This complexity is mostly created by very long functions or by branches that are responsible for checking the pre-conditions on different (optional) parameters. Either way, we recommend that scikit-learn tackles this problem by splitting these functions into smaller ones. This will not only decrease the complexity but also increase the readability and maintainability of the code [55].

Name	Lines of code	McCabe complexity	Number of parameters	Component	Technology	Status
_lobpcg.py:lobpcg(A,X,B,M,Y,tol,maxiter,largest,verbosityLevel,retLambdaHistory,retResidualNor	301	62	11	sklearn/externals	python	Open
_least_angle.py:_lars_path_solver(X,y,Xy,Gram,n_samples,max_iter,alpha_min,method,copy_X,eps	244	65	16	sklearn/linear_model	python	Open
LogisticRegressionCV.fit(X,y,sample_weight)	194	40	3	sklearn/linear_model	python	Open
BaseDecisionTree.fit(X,y,sample_weight,check_input,X_idx_sorted)	189	51	5	sklearn/tree	python	Open
BaseHistGradientBoosting.fit(X,y)	180	33	2	sklearn/ensemble	python	Open
RANSACRegressor.fit(X,y,sample_weight)	158	35	3	sklearn/linear_model	python	Open
_locally_linear.py:locally_linear_embedding(X,n_neighbors,n_components,reg,eigen_solver,tol,m	150	27	12	sklearn/manifold	python	Open
_logistic.py:_logistic_regression_path(X,y,pos_class,Cs,fit_intercept,max_iter,tol,verbose,solver,c	148	36	20	sklearn/linear_model	python	Open
_openml.py:fetch_openml(name,version,data_id,data_home,target_column,cache,return_X_y,as_	133	42	8	sklearn/datasets	python	Open
_agglomerative.py:linkage_tree(X,connectivity,n_clusters,linkage,affinity,return_distance)	126	32	6	sklearn/cluster	python	Open
LinearModelCV.fit(X,y)	121	35	2	sklearn/linear_model	python	Open
NeighborsBase._fit(X)	111	32	1	sklearn/neighbors	python	Open
validation.py:check_array(array,accept_sparse,accept_large_sparse,dtype,order,copy,force_all_fi	110	42	12	sklearn/utils	python	Open
LogisticRegression.fit(X,y,sample_weight)	110	29	3	sklearn/linear_model	python	Open
_ridge.py:_ridge_regression(X,y,alpha,sample_weight,solver,max_iter,tol,verbose,random_state,r	104	33	14	sklearn/linear_model	python	Open
TSNE._fit(X,skip_num_points)	99	30	2	sklearn/manifold	python	Open
estimator_checks.py:check_classifiers_train(name,classifier_orig,readonly_memmap,X_dtype)	99	26	4	sklearn/utils	python	Open
_pprint.py:_safe_repr(object,context,maxlevels,level,changed_only)	97	38	5	sklearn/utils	python	Open
partial_dependence.py:plot_partial_dependence(estimator,X,features,feature_names,target,resp	97	35	16	sklearn/inspection	python	Open

Figure 23.5: A subsection of the recommended unit complexity refactors.

Secondly, there are cyclic dependencies between the `sklearn`, `metrics` & `utils` modules that should be solved. Cyclic dependencies are dangerous because they can cause unexpected bugs that are very hard to resolve. They also limit the flexibility in developing new features as developers need to work around these cyclic dependencies. Scikit-learn realized this and is currently adressing it [56].

Thirdly, the number of function arguments in scikit-learn is relatively high as indicated by the unit interfacing metric. This is a design choice so we won't recommend them to change this [57] [58]. However, what has become a problem is that scikit-learn accepts both positional & keyword arguments. Making it easy to introduce swapped argument bugs. The contributors have also realized this and they are planning to solve this by only allowing keyword arguments as stated in SLEP009. Which is a statement that updates their future API roadmap.

[55] Clean Code by Robert Cecil Martin

[56] https://github.com/scikit-learn/scikit-learn/issues/15123

[57] Scikit-learn, what does it want to be?, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/06/ scikit-learn-what-does-it-want-to-be.html

[58] From Vision to Architecture, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/15/ from-vision-to-architecture-20893.html

23.4.4 Coding Activity

The analysis by SIG are a good indication of how scikit-learn is doing and where it can improve, but does this really represent the current focus of the library? In this section we will discuss the hotspots and the mapping of the system's roadmap onto the architectural components and its underlying code.

23.4.4.1 Scikit-learn's hotspots

Using the git history, were are able to discover the hot spots of the scikit-learn repository. We have mapped the 20 most changed files overall and of the past 6 months in a bar graph and we can see in the figure below that the main focus in both cases is in the documentation.

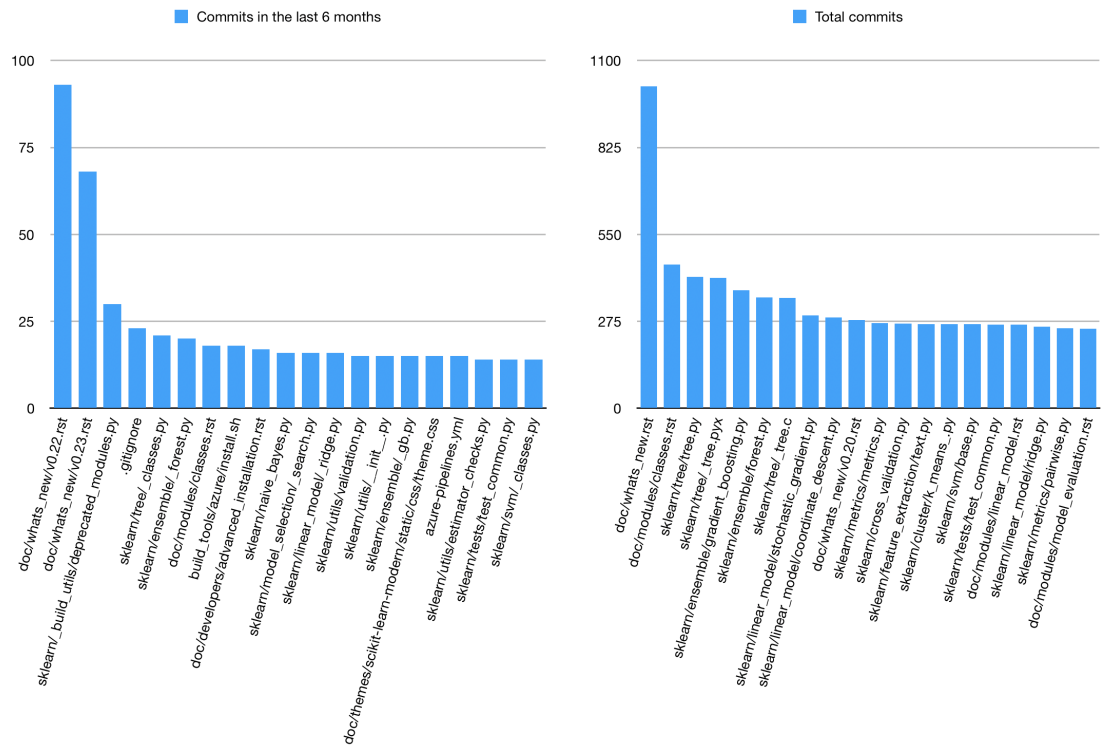


Figure 23.6: Amount of commits per file in the scikit-learn repository from the past 6 months and in total untill 25 March 2020 of the 20 most changed files.

The most recent coding activity is located in utility files. Those files contain tools, which are meant to be used internally in scikit-learn to help with development[59]. For the total amount of commits, most of the work is in the different machine learning models.

[59] Utilites scikit-learn, https://scikit-learn.org/stable/developers/utilities.html#utilities-for-developers

23.4.4.2 Relation to the roadmap

As described in an earlier blog post[60], the roadmap of scikit-learn is focussed on maintaining high quality and well-documented models, to make it easier for developers to develop external components and to improve the integration with modern data science tools. We can see in the coding activity of the recent months that scikit-learn is most active in the documentation and utility files. The improvement in the documentation makes it easier for people to understand scikit-learn. The improvements and additions of utility files make scikit-learn easier to use for developers. So both area's of improvement reflects scikit-learn's roadmap.

23.4.5 Conclusion

In this essay, we tried to answer the question: how does scikit-learn seek to safeguard its quality and architectural integrity? We have provided an overview of the guidelines and tools that scikit-learn uses to promote its quality and integrity. We have also analyzed the effectiveness of these measures. Finally, we mapped recent coding activity to the roadmap that was defined previously by the authors.

We found that scikit-learn's measures have been very effective in promoting quality: the test coverage is very high, there is a strict code review and inadequate code is not merged into the repository. The measures have also been effective with regards to ensuring architectural integrity: most flaws in the code are a result of design choices, not inadequate contributions. However, we found that there are still some refactoring improvements possible. Finally, it seems that recent coding changes reflect the desired changes laid out by the roadmap. These are mostly focussed on utilities and documentation.

That being said, a project always keeps evolving and it's a continuous struggle to maintain a project. However, we are confident that scikit-learn's measures will ensure a healthy project evolution in the coming years.

23.5 How does scikit-learn balance usability with variability?

tags: `Scikit-learn` `Python` `Software Architecture` `Machine Learning` `Developers` `Configurability` `Variability` `Usability` `Features`

So far we have covered many aspects of scikit-learn from a software architectural perspective. In our first essay [61], we described the vision behind scikit-learn. In the second essay [62], we described how this vision translates to an architecture. Then, in our last essay [63], we investigated how scikit-learn safeguards its quality as an open-source system.

In this essay, we will consider another specific architectural aspect of scikit-learn. That aspect is configurability (or variability) and the tradeoff that needs to be made between configurability and usability. It can be hard for systems to balance these two aspects as allowing users to have more freedom in configuring the system also introduces extra complexity. We have found in our investigation of the project, that scikit-learn has found some clever ways to optimize this balance.

[60] Scikit-learn, what does it want to be?, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/06/ scikit-learn-what-does-it-want-to-be.html

[61] Scikit-learn, what does it want to be?, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/06/ scikit-learn-what-does-it-want-to-be.html

[62] From Vision to Architecture, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/15/ from-vision-to-architecture-20893.html

[63] Scikit-learn's plan to safeguard its quality, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/26/scikit-learns-plans-to-safeguard-its-quality-31705.html

First, let us investigate what kind of configurability is desired by the relevant stakeholders. Here, we consider a configuration of scikit-learn to be an ML pipeline that uses scikit-learn to solve an ML problem.

23.5.1 The need for Configurability

As we described in our first essay [64], scikit-learn is designed to be simple, efficient, reusable and maybe most importantly, accessible to non-experts. So let's find out how these design goals translate to variability requirements.

In our first essay [65] we defined four groups of stakeholders: competitors, funders, users and contributors. Each of these stakeholder groups have varying desires of configurability. Due to the lack of involvement in the project, we will leave the group of competitors out of scope for our analysis now. We will, however, describe the configurability needs of the other three groups of stakeholders. The developers of scikit-learn need to take this into account during system design.

Funders want the configuration abilities that suit their (business) needs. In scikit-learn's case, the funders are both educational institutions (universities) and businesses. Universities may want the product to be highly configurable for research purposes but also accessible for non-experts (students). Business may want the product to be configurable for their business needs to deploy it in their service line.

Scikit-learns user base is broad: the website [66] lists several notable businesses, but it is also the go-to library for anyone who wants to play around with machine learning. Therefore, configuration needs to be open to a very wide user base.

Contributors are concerned with the implementation of the application in various forms. Therefore, they seek to have a very smooth development process.

Contributors thus desire that configurability does not add additional complexity to the development process, but still meets all requirements. In the next section, we will see how this balance is managed by scikit-learn.

23.5.2 Maximizing configurability and usability

In machine learning, your chosen implementation and hyper-parameter configurations have a significant impact on both the accuracy and the performance of your model. Which means that there is a huge need for configurability. However, if configurability is implemented incorrectly your product will be hard to understand and use. So there must be a natural balance between configurability and usability. This balance is implemented directly into scikit-learn's estimator design. Every estimators `fit` method is highly configurable. In this section, we will first discuss how configurability works from a users perspective. Then we will discuss how this balance has been implemented. Finally, we discuss the trade-offs that had to be made.

23.5.2.1 User perspective

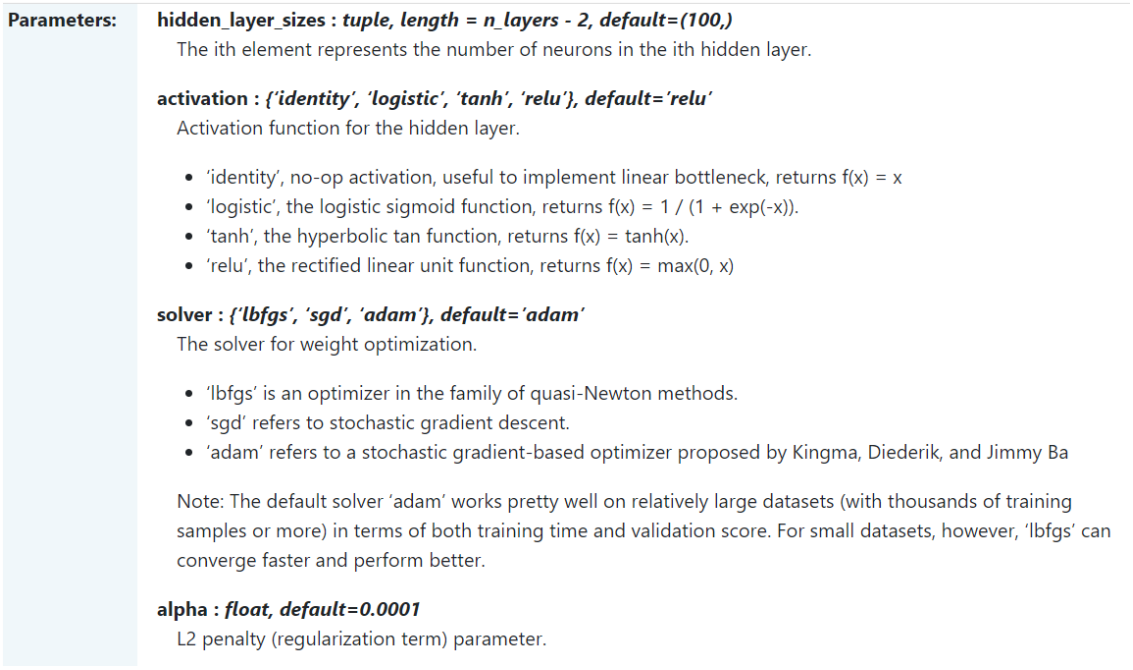
From a users perspective, configuration happens in the following way. First, the user selects the model he wants to configure. Then the user looks at the API documentation on scikit-learn's website. As can be seen

[64] Scikit-learn, what does it want to be?, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/06/ scikit-learn-what-does-it-want-to-be.html

[65] Scikit-learn, what does it want to be?, T. de Boer, T. Bos, J. Smit and D. van Gelder, https://desosa2020.netlify.com/projects/scikit-learn/2020/03/06/ scikit-learn-what-does-it-want-to-be.html

[66] Scikit-learn's users/testimonials, https://scikit-learn.org/stable/testimonials/testimonials.html

in figure 1[67], this documentation provides a clear overview of the configuration options and their effects.



Parameters:

hidden_layer_sizes : *tuple, length = n_layers - 2, default=(100,)*
The ith element represents the number of neurons in the ith hidden layer.

activation : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'*
Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver : *{'lbfgs', 'sgd', 'adam'}, default='adam'*
The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

alpha : *float, default=0.0001*
L2 penalty (regularization term) parameter.

Figure 23.7: 1: A small part of the configuration documentation for an MLPClassifier.

The user then constructs the selected model and provides the configuration through the parameters. All the missing parameters will keep their default configuration. The configuration will only be validated once the user calls the `fit` method. This method will raise a `ValueError` if the validation is invalid. If the configuration is valid the `fit` method will act as described in the configuration.

The configuration is only considered frozen during the execution of the `fit` method. The user is free to change the configuration dynamically using the `set_param` method at any time. So from a users perspective, the binding flow is rather easy and well documented. However, from a configurability perspective, the configuration is limited to a fixed set of options.

23.5.2.2 Implementation

The constructor and the `set_param` method only store the configuration. They don't validate them. While the `fit` method both validates and executes the configuration. Which means that this method can quickly become rather complex. However, scikit-learn does it best to hide this complexity from the user using a Façade design pattern [68].

All this complexity is also hard for the developers to keep in mind. That is why they divided this large function into 3 major parts. First, the `fit` method will always call the `_validate_hyperparameters`

[67] Documentation from the `MLPClassifier`, https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn-neural-network-mlpclassifier

[68] Façade design pattern, https://en.wikipedia.org/wiki/Facade_pattern

method. This method will raise a `ValueError` if any of the parameters don't adhere to the value constraints as stated in the documentation (figure 2 [69]).

```
379     def _validate_hyperparameters(self):
380         if not isinstance(self.shuffle, bool):
381             raise ValueError("shuffle must be either True or False, got %s." %
382                              self.shuffle)
383         if self.max_iter <= 0:
384             raise ValueError("max_iter must be > 0, got %s." % self.max_iter)
385         if self.max_fun <= 0:
386             raise ValueError("max_fun must be > 0, got %s." % self.max_fun)
387         if self.alpha < 0.0:
388             raise ValueError("alpha must be >= 0, got %s." % self.alpha)
389         if (self.learning_rate in ["constant", "invscaling", "adaptive"] and
390                 self.learning_rate_init <= 0.0):
```

Figure 23.8: 2: An example of how a _validate_hyperparameters method in an MLP

Secondly, after validating the parameters, the `fit` method will update the parameters that are incompatible with each other to values that are. Automatically correcting the incompatible configurations. This code can usually be found in the main body of the `fit` method. For example figure 3[70] shows how the `batch_size` configuration parameter is updated based on the solver type.

```
# lbfgs does not support mini-batches
if self.solver == 'lbfgs':
    batch_size = n_samples
elif self.batch_size == 'auto':
    batch_size = min(200, n_samples)
else:
    if self.batch_size < 1 or self.batch_size > n_samples:
        warnings.warn("Got `batch_size` less than 1 or larger than "
                      "sample size. It is going to be clipped")
    batch_size = np.clip(self.batch_size, 1, n_samples)
```

Figure 23.9: 3: An example of how incompatible parameters are handled in an MLP

Finally, the configured method is implemented as a separate method. This method will be called at the very

[69] The source code in the example of how imcompatbile parameters are handled, https://github.com/scikit-learn/scikit-learn/blob/eaf0a044fdc084ebeeb9bbfbcf42e6df2b1491bb/sklearn/neural_network/_multilayer_perceptron.py#L347

[70] The source code in the example of how imcompatbile parameters are handled, https://github.com/scikit-learn/scikit-learn/blob/eaf0a044fdc084ebeeb9bbfbcf42e6df2b1491bb/sklearn/neural_network/_multilayer_perceptron.py#L347

end of the `fit` method with the configured parameters. For example, figure 4 [71] show the configured method for the MLP is finally called, whereby the implementation truly depends on the configuration.

```
# Run the Stochastic optimization solver
if self.solver in _STOCHASTIC_SOLVERS:
    self._fit_stochastic(X, y, activations, deltas, coef_grads,
                         intercept_grads, layer_units, incremental)

# Run the LBFGS solver
elif self.solver == 'lbfgs':
    self._fit_lbfgs(X, y, activations, deltas, coef_grads,
                    intercept_grads, layer_units)
return self
```

Figure 23.10: 4: An example of how the configured fit method is called in an MLP

23.5.2.3 Trade-offs

Based on the previous section, it is obvious that scikit-learn has chosen to increase configurability and usability at the expense of development complexity. This also becomes clear when we want to add a new configuration parameter to an estimator. This requires a rigorous amount of testing due to all the branches and edge cases it introduces. Secondly, usability is still a bit more important than configurability. This is why users are not able to extend the configuration with their own code, as is possible in some other frameworks.

23.5.3 Configurability in scikit-learn's features

Now we know why it is important for scikit-learn to be configurable and how scikit-learn balances its configurability with usability, how can these aspects be found in the architecture of scikit-learn? To answer this question we will look into the different functionalities provided by scikit-learn and how they compliment each other, then we will go more in depth into one feature and see how configurability is handled on a lower level. In both cases we will model the configurability with the help of a feature diagram.

23.5.3.1 scikit-learn's feature sets

To model the features of scikit-learn we first need to look back at the functionality that scikit-learn offers. We have of course discussed this in our previous essays, but we will summarize the most important functionalities which are relevant to create our feature diagram.

23.5.3.1.1 Estimation Most of scikit-learn's functionality is based around estimation. "An estimator is any object that learns from data; it may be a classification, regression or clustering algorithm or a transformer

[71] The source code in the example of how the final configured method is called, https://github.com/scikit-learn/scikit-learn/blob/eaf0a044fdc084ebeeb9bbfbcf42e6df2b1491bb/sklearn/neural_network/_multilayer_perceptron.py#L369

that extracts/filters useful features from raw data."[72].

23.5.3.1.2 Model Evaluation After estimation comes evaluation in which the performance of a model is measured. The main components of model evaluation are the different metrics offered by scikit-learn. These metrics can be used on separate predicted/actual value pairs or in combination with a scorer which uses the given metric to train a model[73].

23.5.3.1.3 Inspection Summarising module performance using metrics is, however, often not enough as then the assumption is made that the metric and test dataset perfectly reflect the target domain, which is often not true. Scikit-learn, therefore, offers a module which offers tools for model inspection such as partial dependence plots and the permutation feature importance technique[74]. Scikit-learn also offers an API for creating quick visualisations without recalculation[75].

23.5.3.1.4 Datasets Then, when users want to test their models, scikit-learn offers a datasets module. This module contains a dataset loader, which can be used to load small toy datasets and a dataset fetcher which can fetch larger datasets which contain 'real-world' data commonly used by the machine learning community to benchmark algorithms[76].

23.5.3.1.5 Feature diagram Figure 5 views the most general features, but to get a sense of the amount of variability within scikit-learn we need to go deeper. When picking scikit-learn as a machine learning library the most important variabilities are within the different estimators that are available, therefore, we will go more in depth into how that can be seen in the scikit-learn architecture in the next section.

23.5.3.2 We need to go deeper

In order to demonstrate the variability with estimators we will give two examples. For the first example, we will take a look at the linear models. Scikit-learn has divided the linear models into 7 categories. Each category contains several linear models that are implemented in scikit-learn as an estimator. And each estimator can have a lot of parameters that can be configured. In figure 6 the feature model is given for linear models with most categories and models collapsed for a more clear overview.

Most of the parameters in the estimators share their names and functionality. A list of the most common parameters can be found in the scikit-learn's glossary[77]. We can see that the model `LinearRegression` only has 4 parameters, `fit_intercept`, `normalize`, `copy_X` and `n_jobs`. Parameters can be any data type and are all optional because they contain a default value. This estimator does not have a lot of parameters compared to the 19 parameters of `SGDRegressor`. This gives already a lot of options for `Classical Linear Regressors` because there is a choice of model and a lot of parameters per model. The other six categories also contain a lot of models, for example, there are 11 estimators in `Regressors with Variable Selection` with each estimator having its own parameters which can be configured. This gives scikit-learn already a lot of configurability for linear models.

[72] Statistical learning: the setting and the estimator object in scikit-learn, https://scikit-learn.org/stable/tutorial/statistical_inference/settings.html
[73] Metrics and scoring: quantifying the quality of predictions, https://scikit-learn.org/stable/modules/model_evaluation.html
[74] Inspection, https://scikit-learn.org/stable/inspection.html
[75] Visualization, https://scikit-learn.org/stable/visualizations.html
[76] Dataset loading utilities, https://scikit-learn.org/stable/datasets/index.html
[77] Scikit-learn's glossary of the parameters, https://scikit-learn.org/stable/glossary.html#parameters

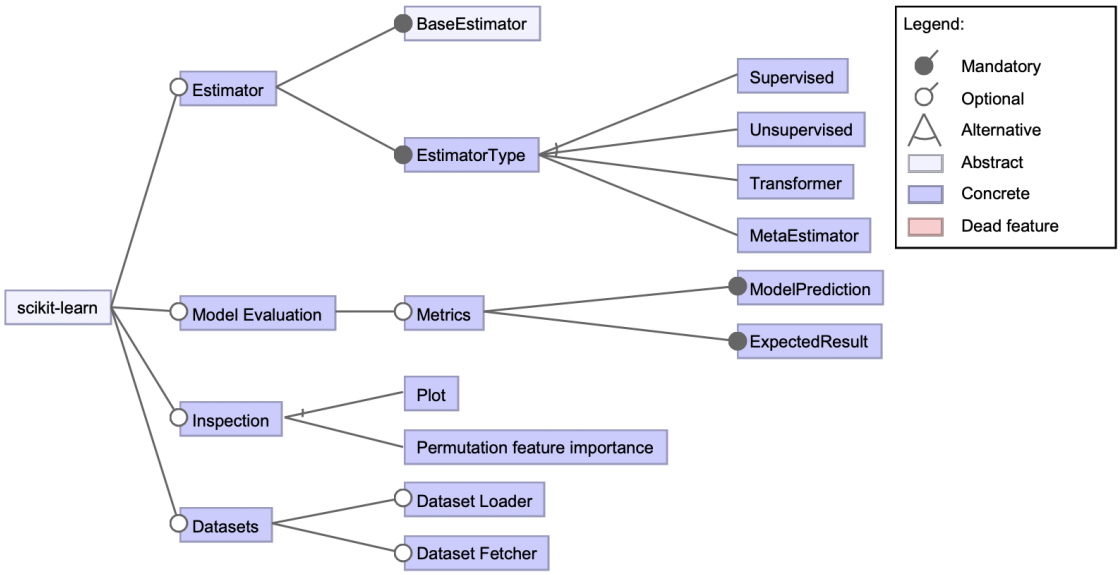


Figure 23.11: 5: A feature diagram viewing the most general features of scikit-learn.

If we then look at the LinearClassifiers we find the `SGDClassifier`, the `Perceptron` and the `PassiveAggressiveClassifier` which are three variants of models using the Stochastic Gradient Descent algorithm. These are all variants of the `SGDBaseClassifier`. This relation is modelled in Figure 7. The `BaseSGDClassifier` contains the option for 9 different loss functions, however, when choosing the `Perceptron` variant of the `BaseSGDClassifier` the perceptron loss is the only loss applicable. Also, the learning rate parameter of the `BaseSGDClassifier` has four values: 'constant', 'optimal' , 'invscaling' and 'adaptive'. For the `SGDClassifier` and `PassiveAggressiveClassifier`, all options are open, but for the `Perceptron` the 'constant' value is set.

Also, the learning rate parameter of the `BaseSGDClassifier` has four values: 'constant', 'optimal' , 'invscaling' and 'adaptive'.

23.5.4 Conclusion

In this essay we have taken a look at scikit-learn from a variability perspective. That is, in what ways can scikit-learn be configured for varying requirements? We have seen that scikit-learn offers a lot of configuration options for its users and that it tries to balance configurability and usability. While scikit-learn focuses more on the usability at the expense of complexity and configurability, it offers an enormous amount of variability in the estimators. The reason for this is to make scikit-learn easily accessible for non-experts, which is one of the core design goals of the project. Overall, we conclude that scikit-learn has been succesful in offering variability for its users while not sacrificing usability.

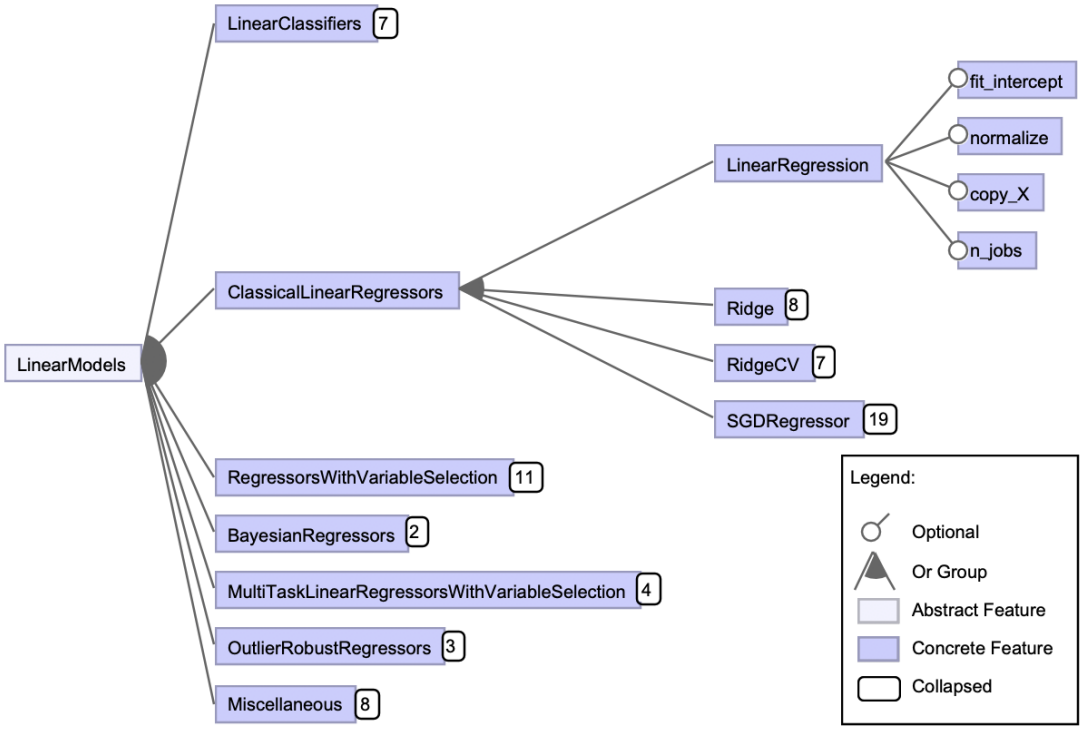


Figure 23.12: 6: Feature model of linear models in scikit-learn.

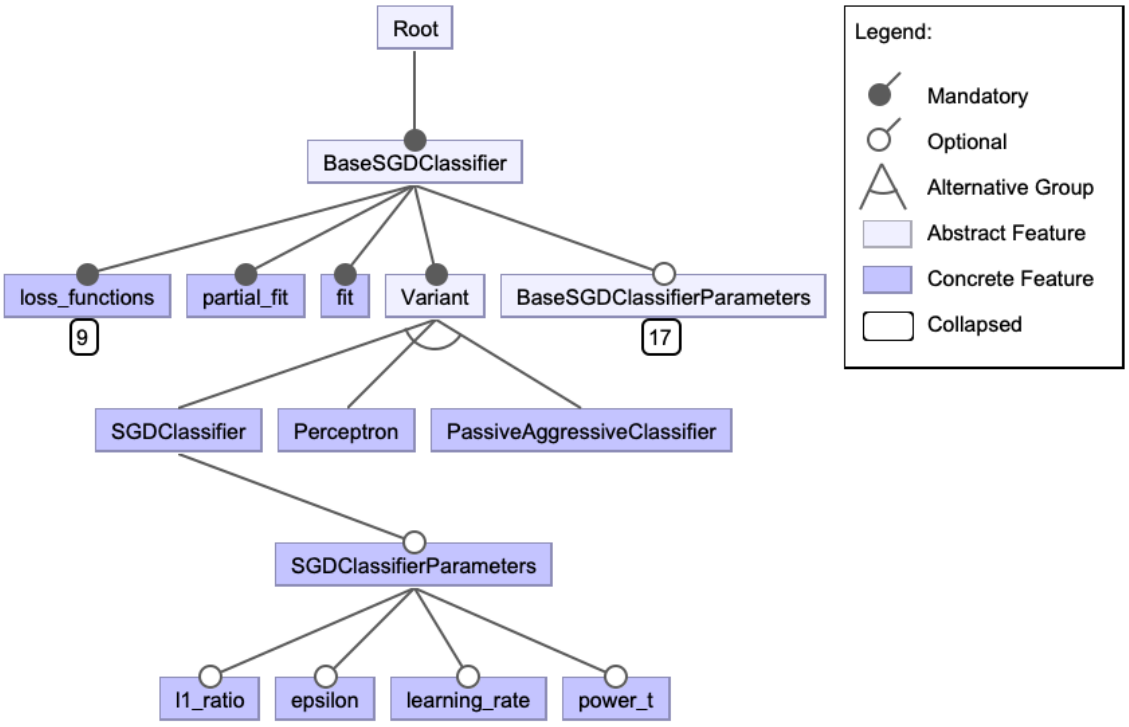


Figure 23.13: 7: Feature model of the SGD variants.

Chapter 24

Sentry



By Nick Belzer, Martijn Comans, Philippe Lammerts and Daan Oudejans

---

A simple error can cause a lot of headaches, chaos and even financial loss. Sentry helps mitigating these problems by enabling developers to quickly triage, analyze and resolve errors and bugs. Sentry works for nearly every stack, and has a lot of useful features like distributed tracing across different stacks, extensive context descriptions and breadcrumbs. Every day Sentry processes billions of exceptions for more than 50,000 customers, including companies like Microsoft, Uber, Airbnb and PayPal. Thanks to Sentry, developers can create better and more reliable software.

This chapter gives an overview of the work that we have done towards analyzing the software architecture of Sentry. First, we will describe Sentry in a broader context. Then, a deeper analysis will be discussed, which takes a look at Sentry's implementation and underlying architecture.

24.1 Putting Sentry into Context

Imagine you created a nice application for other people to use. Your product is gaining traction, new users start flowing in and you start earning some money. Life is great and you decide to roll out a big update you have been working on. But after a few days of the release, engagement rates and sales seem to drop and

you are not sure why: all of your automated tests pass and you have extensively tried your app on your machine. A little panicked, you start investigating. You are still very lost when an email comes in: a kind user took the effort to submit a bug report! The bug report is very vague, but at least you have a sense of what is happening. You finally find the culprit after another day of work: three lines of code in the backend that caused unexpected behavior in the frontend. Almost a week later, you fixed the issue and users are slowly returning to your app.

Now let's look at the same scenario if you would have used Sentry. Again you've rolled out a big update and life is great until you get a notification from Sentry saying that a fatal issue is occurring for multiple users. Using Sentry's cross-stack tracing, you trace the error to the backend where the breadcrumbs and stack trace immediately show what is wrong. Two hours after the first report, you have deployed an update.

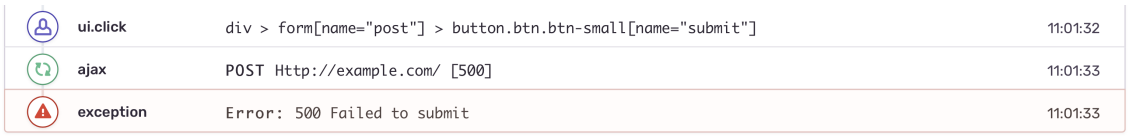


Figure 24.1: Screenshot of the breadcrumbs feature for an example issue

Sentry automatically reports exceptions, giving developers insights into bugs with as much detail as possible by providing session information like events leading up to the error or user device information which can even include the state of the processor. It supports error monitoring for a wide variety of languages and frameworks and it keeps track of errors related to different versions of your code and gives you one dashboard to look through, filter and analyze reported errors. Sentry removes the guesswork and reproducing issues becomes easy as developers can use the reported information to understand where and how the error occurred. In order to fully understand Sentry as an open source project, we will first put the software into context in this essay.

24.1.1 End-user mental model

The end-user mental model consists of two parts. The first part, the what-the-system-is, is about what the end-users see when using Sentry and it describes the form of Sentry[1]. Once the end-user, a developer, installs Sentry with the appropriate SDKs, Sentry starts to monitor all exceptions automatically in real-time that are occurring in the application of the end-user. It detects and groups individual errors into single larger issues. An overview of this is shown to the user in the Sentry dashboard. Sentry provides a separate overview for each error with information about for example its impact, the stack trace and the context of the states of the system[2].

The second part, the what-the-system-does, is about how end-users use Sentry when they are interacting with it and, therefore, it describes the functionality[3]. The end-users can automatically create new issues for errors in their code repository, such as GitHub, using the Sentry dashboard. Sentry analyzes and assigns the most suitable engineer or team for each issue that needs to be solved based on the integration with the code repository. Users can also trace specific errors through their whole codebase as Sentry supports a combined analysis of the front-end and the back-end code. Finally, developers can analyze the history

[1] Coplien, J. O., & Bjørnvig, G. (2011). Lean architecture: for agile software development. John Wiley & Sons.

[2] Sentry. (2020). Error Monitoring Software & Crash Reporting Tools for Apps. Retrieved February 26, 2020, from https://sentry.io/features/

[3] Coplien, J. O., & Bjørnvig, G. (2011). Lean architecture: for agile software development. John Wiley & Sons.

of the monitoring data using different metrics such as the number of affected users, the number of events or the number of errors for each release[4]. An overview of the what-system-is and the what-system-does is presented in the image below, showing a visual version of the end-user mental model as described by Coplien and Bjørnvig[5].

24.1.2 Key capabilities and properties

From the characterization and mental model above, we can derive a list of key capabilities and properties of Sentry:

- Open-source
- Real-time error and application monitoring
- Support for all major languages and frameworks for a wide appeal
- Integration with other applications and services (GitHub, Slack, and more)
- Error tracing with breadcrumbs and state information
- Security and privacy, using industry-standard technologies[6]
- Easy-to-use

24.1.3 Stakeholder analysis

In order to get an overview of the parties with an interest in Sentry, we conducted a stakeholder analysis. We categorize these stakeholders in the categories introduced by Rozanski and Woods[7]. Since it seems to be the case for Sentry that most of their developers are also maintainers and testers, we combine these stakeholders in the developers category. We also left out the *production engineers* and *system administrators* classes since Sentry is an open-source project. However, the Sentry team working on the SaaS version likely has these stakeholders. One could also consider competitors to be stakeholders[8], thus we include them for sake of being thorough.

Category	Stakeholder analysis
Acquirers	Sentry's senior management. This category also includes Sentry's investors: Accel and NEA and angel investors like Nat Friedman, Ilya Sukhar, and Greg Brockman.
Assessors	Assessors ensure that the system satisfies legal requirements and standards[9]. We were not able to identify stakeholders with this role.
Communicators	Sentry's communication happens through various channels, including their blog, Twitter, Forum and GitHub. Since Sentry is a company, most if not all of their employees can be considered communicators (i.e. the core team).
Developers	The main Sentry repository on GitHub has 430 contributors at the time of writing. Most top contributors are Sentry employees, and we consider these people to belong to the core team.

[4]Sentry. (2020). Error Monitoring Software & Crash Reporting Tools for Apps. Retrieved February 26, 2020, from https://sentry.io/features/

[5]Coplien, J. O., & Bjørnvig, G. (2011). Lean architecture: for agile software development. John Wiley & Sons.

[6]Sentry. (2020). Security & Compliance. Retrieved March 3, 2020, https://sentry.io/security/

[7]Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

[8]Archer, G. R. (2006). Six Good Reasons to include competitors as stakeholders. Available at SSRN 1437117.

Category	Stakeholder analysis
Suppliers	Like many other large software projects, Sentry has numerous dependencies on which they build their platform and thus can be considered suppliers. Examples include: Django, Docker, PostgreSQL, etcetera.
Support Staff	If you just want to use the open-source version of Sentry, support is relatively limited (e.g. through their forum). However, if you make use of their SaaS solution, you have access to Sentry's support team.
Users	Sentry has roughly two types of users: those who use the open-source version to host Sentry on-premise, and those who pay Sentry for their SaaS version. Sentry has many big users like Microsoft, Uber, Airbnb, and PayPal[10].
Competitors	Sentry operates in roughly two markets: error monitoring and application monitoring. There are various competitors in these fields such as Rollbar, Bugsnag and Raygun.

Now that we have identified the stakeholders, we can prioritize them using a Power-Interest Grid in order to gain more understanding of the relationships with these stakeholders[11].

24.1.4 The current and future context

The context of a system describes the relationships and dependencies between the system itself and its environment[12]. The context of Sentry is visualized in the Context Model below.

[9] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

[10] Sentry. (2020). About Sentry. Retrieved March 3, 2020, from https://sentry.io/about/

[11] MindTools. (n.d.). Stakeholder Analysis. Retrieved March 3, 2020, from https://www.mindtools.com/pages/article/newPPM_07.htm

[12] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

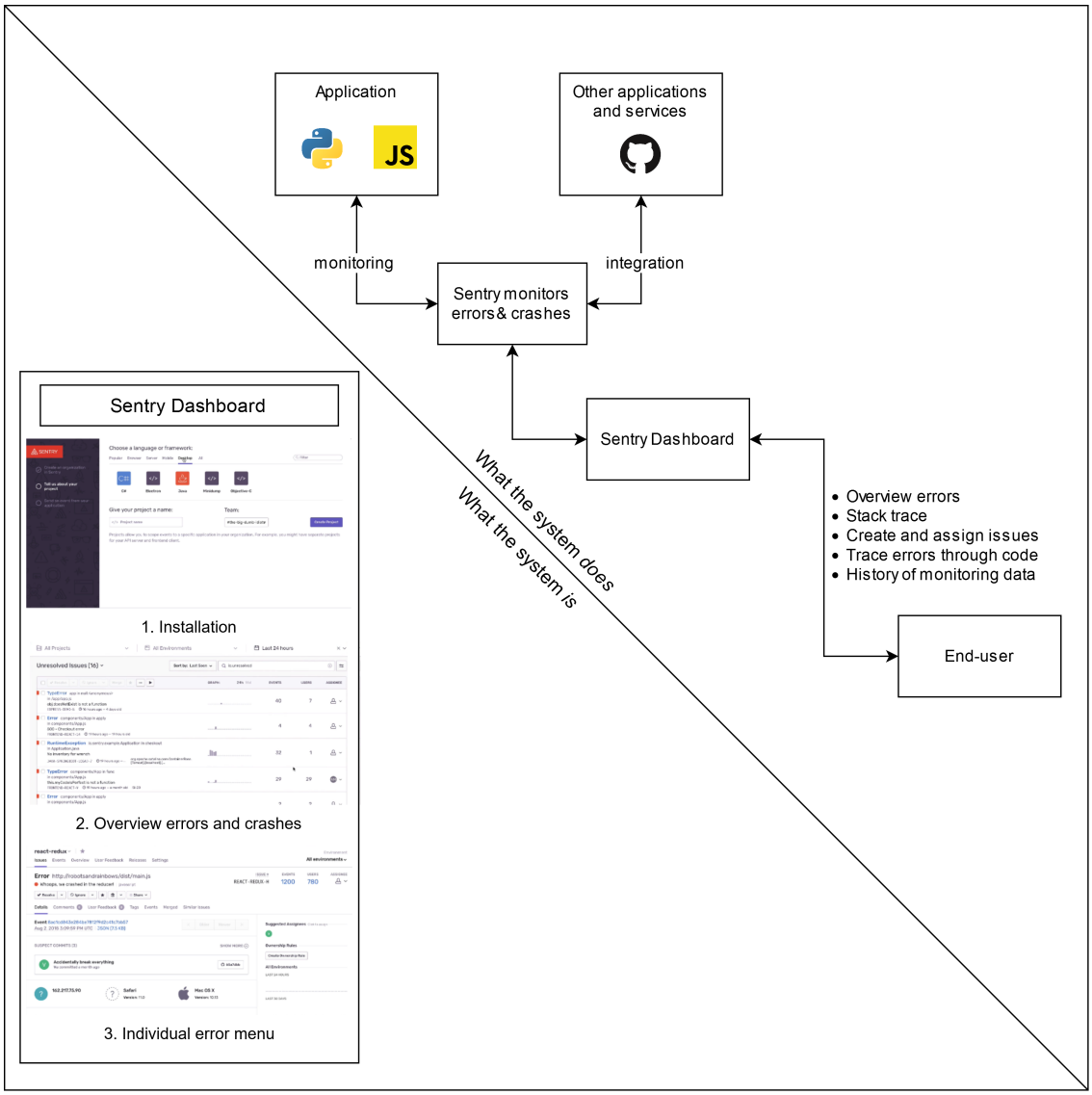


Figure 24.2: End-user mental model

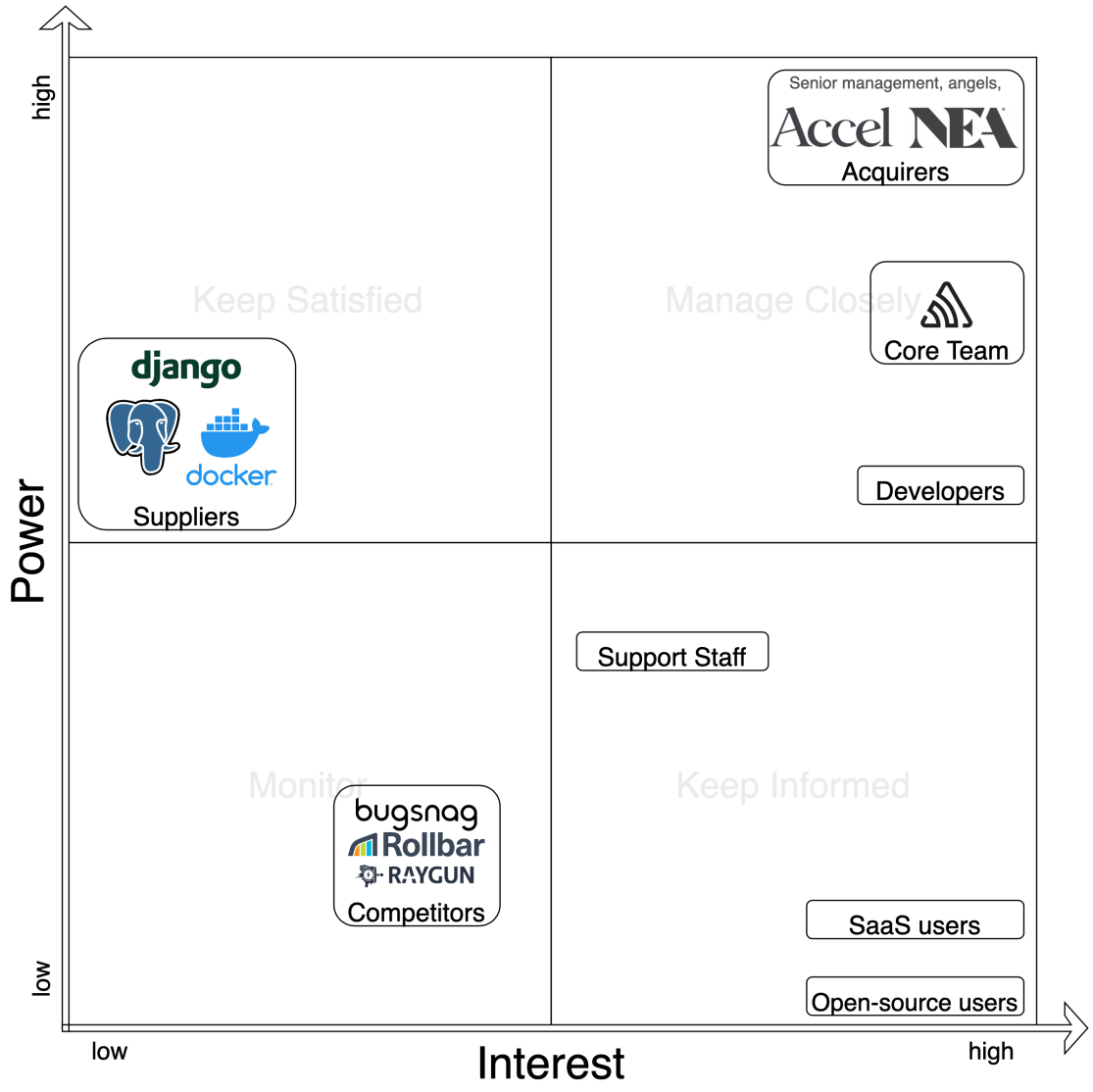
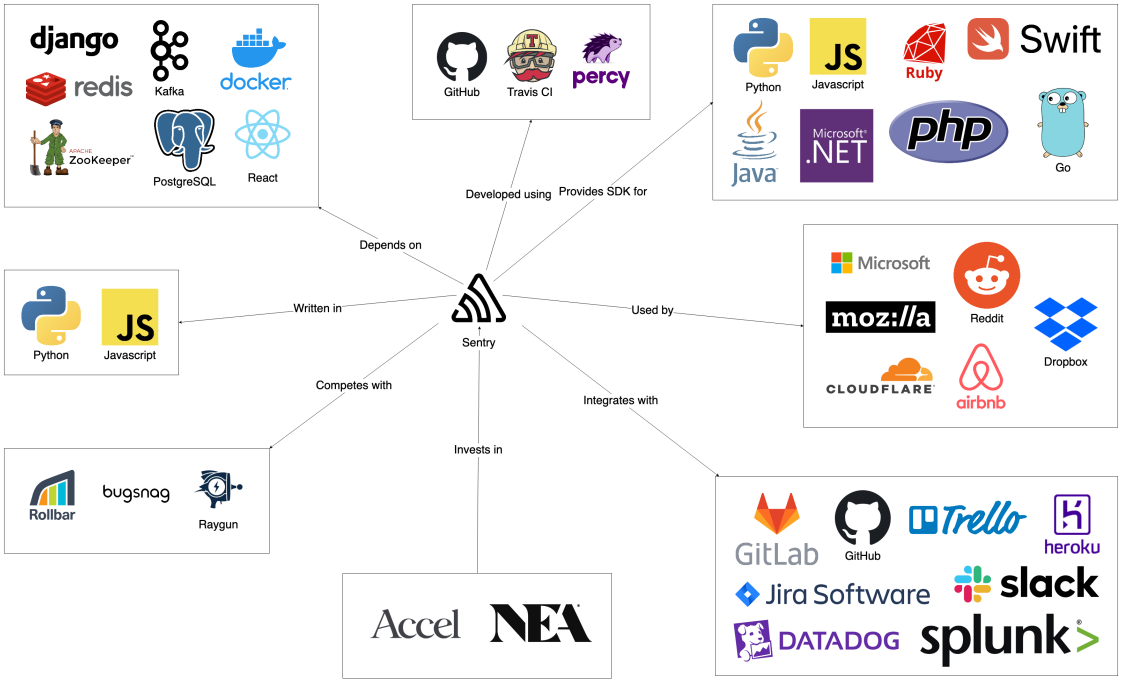


Figure 24.3: Power-Interest Grid



This context model is not exhaustive since there are for example more languages for which Sentry provides an SDK. The following relationships can be seen in the model:

- **Dependencies:** Like stated before in the stakeholder analysis, Sentry has numerous dependencies which include libraries and frameworks like Django and React, database systems like PostgreSQL, and other tools like Docker.
- **Development:** Sentry is developed using GitHub for version control hosting, Travis CI for Continuous Integration and Deployment, and Percy for visual testing. The backend of Sentry is written in Python while the frontend is written in JavaScript and TypeScript.
- **Integration:** Sentry provides client SDKs that send errors to Sentry for almost all frequently used programming languages and frameworks, a selection of which are shown in the Context Model. Furthermore, Sentry features integrations that allow users to track errors and issues across multiple platforms, like Jira, Slack, and Datadog.
- **Users:** Sentry has many users, including some large companies like Microsoft, Cloudflare, and Airbnb.
- **Competition:** Some other companies provide error tracking as a service, like Rollbar, Bugsnag, and Raygun. These services are however not open-source.
- **Investors:** As already mentioned in the stakeholder analysis, Sentry has investors including Accel and NEA.

24.1.5 Product roadmap

At the time of writing there does not seem to be a roadmap available for the Sentry project, neither on Sentry.io, the GitHub project nor on the Sentry forums. The latest version of the on-premise version (10.0.0) was released on the 7th of January 2020.

Based on an inquiry with David Cramer, who is the founder and CTO of Sentry, we found that the Sentry team operates in the short term roadmaps with some long term goals in mind. The primary goals for Sentry at this point being to support every runtime and to move into the broader space of application monitoring (also APM). In the shorter term, they are therefore working on improving the experience for developers for platforms where you are not in control of the device itself like mobile, browser and desktop. Next to that, they are working on the functionality required to move into application monitoring like expanding upon distributed-tracing to track events across different elements of your application like frontend and backend.

24.2 The Architecture Powering Sentry

In the last essay we gave an overview of Sentry and its context. We discussed topics like Sentry's key capabilities, stakeholders and relationships with its environment. But that essay also serves as a stepping stone to the topic of this essay: the architecture behind Sentry. Architecture of software can be viewed from multiple perspectives, as has for example been described by Rozanski and Woods[13]. Guess what? The last essay already covered two architectural viewpoints, namely the contextual and functional viewpoints! In this essay, we will take a Snuba dive into some more technical architectural viewpoints: development, deployment and runtime.

These different viewpoints by Rozanski and Woods have been listed in the table below along with some explanation and prioritization[14]. Like we mentioned, this essay will cover two important new viewpoints from this table: the development and deployment views. This essay also adds a third viewpoint, the runtime view, which has been inspired by the arc42 architecture documentation template[15]. Whereas the development view discusses what software components Sentry is built from, the runtime view will discuss how these components interact in practice. Then, the deployment view will highlight how this software is being deployed as a system.

Viewpoint	Importance	Explanation
Context	medium	The context of Sentry has already been analyzed in our product vision. Here we explained the relationship between Sentry and its environment using context diagrams and a description of the end-user mental model.
Functional	high	This is an important viewpoint as it provides a general description of how the core functionality of Sentry is coupled in its architecture without going into too much detail.
Information	medium	The system does save monitoring data for its history feature. However, saving static data and guaranteeing information quality and consistency are not of high priority in Sentry's functionality.
Concurrency	medium	This viewpoint is not highly important since the system does not need to be optimized in performance and scalability using threads or distributed systems.

[13] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

[14] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

[15] Starke, G. (n.d.). arc42 Documentation. Retrieved March 18, 2020, from https://docs.arc42.org/home/.

Viewpoint	Importance	Explanation
Development	high	It is an open-source system so the software architecture needs to be understandable for outsiders. They implemented many unit and end-to-end tests so that other people can easily contribute as well without breaking the product.
Deployment	high	It needs to be easy to integrate Sentry into the many different development environments of the customers.
Operational	medium	This viewpoint depends on the version of Sentry. For the on-premise edition, the customer is responsible for the operationality. For the SaaS edition, both the Sentry team and the customer are responsible for operating the Sentry system.

24.2.1 The main architectural style

Due to the scale of Sentry we can find several patterns throughout the code. We will focus on the main patterns that are encouraged by the frameworks used. As discussed in our first essay Sentry has been built up from two parts: a front-end (written in JavaScript, uses React) and back-end application (written in Python, uses Django). The front-end is served to the client through the back-end. Together they are bundled in to a single docker image for deployment.

In general this means that due to the separation of data presentation in the front-end and the data storage and logic in the back-end one could argue that from this high level application overview a generic model-view controller has been applied with the front-end as view and the back-end as model. However digging deeper both the front-end and back-end in itself provide similar abstractions.

The back-end of Sentry is written using the Django web framework. This framework follows an MTV (model, template, view) architecture as for every response one defines the template (how the data is shown), the view (which data is shown) and the model (the application logic)[16].

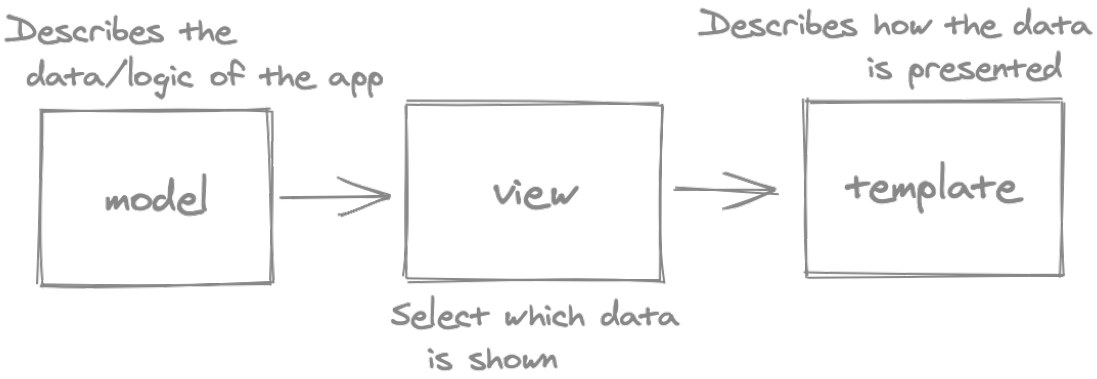


Figure 24.4: Model Template View Architecture, visual representation.

[16] Django Docs - FAQ MVT

This is closely related to the MVC model. This MTV model is followed closely for the authentication part of Sentry, which is the part that is directly served by the back-end (likely for security reasons). The rest of the front-end (once logged in) is rendered by a React application.

The front-end is built with React and makes use of the flux/store pattern to control the data flow and manage state throughout the application.

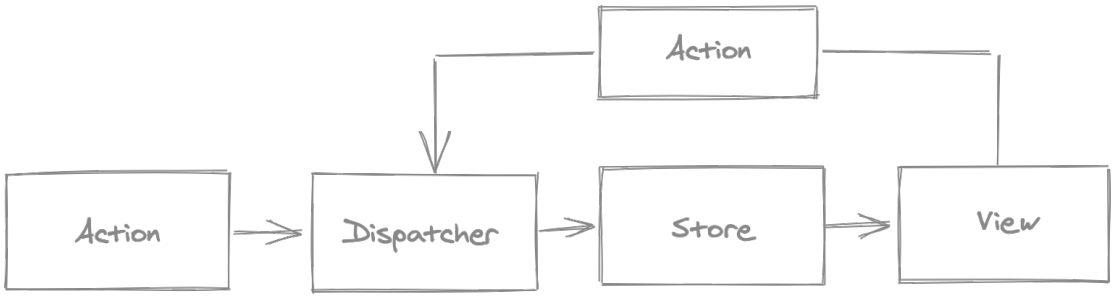


Figure 24.5: Flux Pattern Action->Dispatch->Store->View.

This pattern provides a clear definition on how state is stored. As shown in the diagram above it allows updates to the state using predefined actions and then updates this state across all components that are subscribed it, therefore updating the view. In the view new actions can be triggered by user behavior.

The Sentry front-end application uses these stores to manage different parts of the application like global configuration to projects.

24.2.2 Decomposing Sentry

Sentry is a very large system. According to Sigrid, a static code analysis tool by SIG, the Sentry codebase contains around 25 man-years worth of code. Furthermore, we have used SonarQube for analysis, which reported about 437k lines of code. Due to this size, it is hard to gain an immediate understanding of how Sentry works as a system. In order to get more insight into how components interact in a large system like Sentry, we will need to decompose its structure. Since we have already discussed the System Context of Sentry in the last essay, we will start with decomposing the system according to Level 2 of the C4 Model for visualizing software architecture[17]. The result of this can be seen in the figure below.

Although this diagram contains some explanation, we will clarify a few details. The Django and Python back-end is the most important component in the system. Not only does it serve the React front-end to the user, it also receives and processes events from the Sentry SDK in the software it monitors and stores it for later use. The events are stored in a ClickHouse column-oriented database. For reading and writing to this database, the makers of Sentry made a tool that acts as an abstraction layer for ClickHouse called Snuba in order to hide underlying complexity[18]. Snuba works together with Kafka (which in turn depends on Zookeeper) in order to stream any new events and insert them in batches in the database. Besides also using Snuba for querying, Sentry uses a Redis instance for caching query results in order to remove some load from the ClickHouse cluster. The Sentry back-end uses a PostgreSQL database for storing 'regular' non-event

[17] Brown, S. (n.d.). The C4 model for visualising software architecture. Retrieved March 14, 2020, from https://c4model.com/

[18] Sentry. (2019, May 16). Introducing Snuba: Sentry's New Search Infrastructure. Retrieved March 14, 2020, from https://blog.sentry.io/2019/05/16/introducing-snuba-sentrys-new-search-infrastructure/

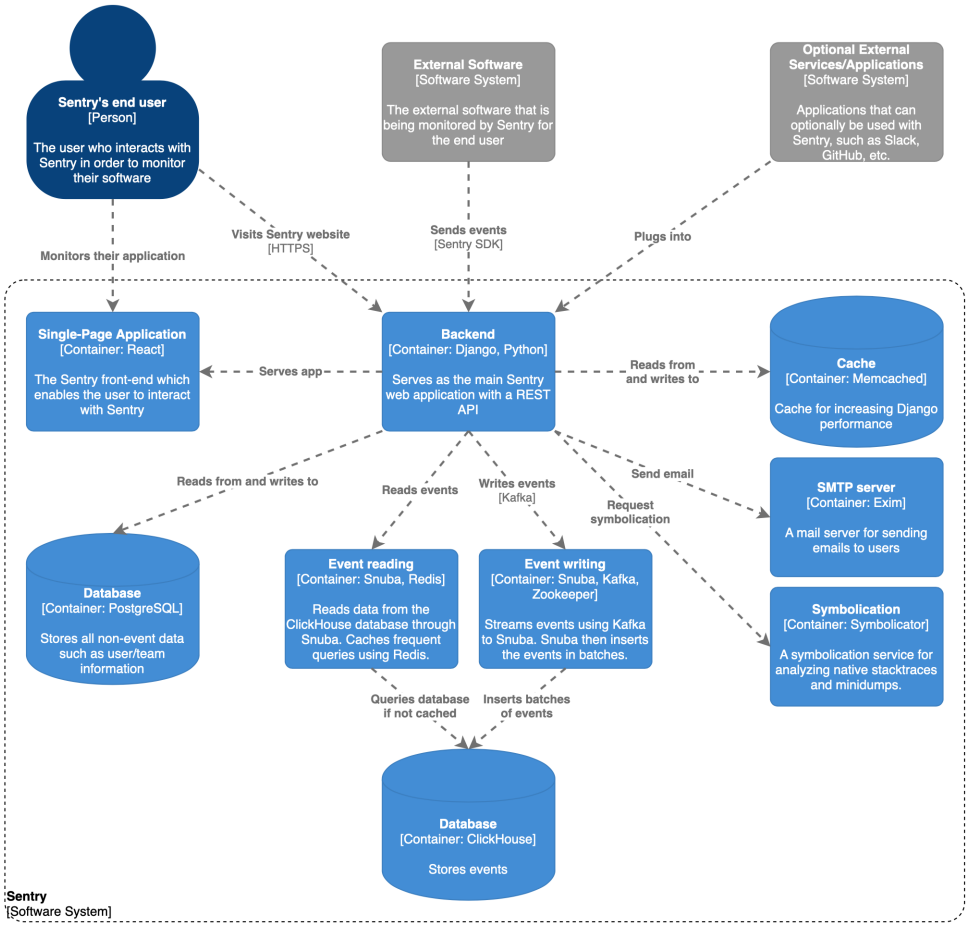


Figure 24.6: C4 Model for Sentry, Level 2 decomposition

data and it uses Memcached which enables Django to cache expensive calculations. Sentry has also built their own Symbolication service called Symbolicator, which analyzes native stacktraces and minidumps[19]. Furthermore, the back-end can use Exim for sending emails and lastly, Sentry supports various integrations and plugins (which are modeled as an external component) such as Slack for notifications or GitHub for linking events to commits and issues[20][21].

In order to understand the source code structure, from the Level 2 diagram we can now derive a Level 3 diagram, by decomposing the back-end container[22]. The result of this can be seen in the figure below.

Due to Sentry's size, Sentry has a lot of interconnected modules. For the sake of simplicity and readability, we have left out a number of modules that in our opinion are less important to the overall representation of the source code structure. This includes various smaller modules, as well as modules that contain setup and configuration code, and plugins and database migrations. As can be seen, the whole React front-end is contained in the `static` module of which the files are included in server-side rendered templates in the `templates` module, which are in turn served by code in the `web` module. The two modules that have API endpoints also make use of the `auth` module for authentication. The `web` module has API endpoints for pushing events to Sentry, which in turn spawns tasks for processing them using the `tasks` module. This module contains tasks that can be used to write and query events, but also for example for requesting symbolication. Lastly, the model also shows some separate modules listed as *highly-connected modules*, `utils` and `models`, which are connected with most if not all of the modules in the diagram and thus shown without any arrows for clarity.

The complexity of Sentry really shows in the dependency graph generated by Sigrid, even when only selecting the modules used in the figure above. Sigrid clearly does not even recognize all of the dependencies and connections between modules. That is why we made the simplified version. The dependency graph generated by Sigrid (only with the modules used in the figure above) can be seen below.

24.2.3 A run time view

A basic understanding of the run time view can be derived from the C4 figures above, as key interactions between the components are already shown in these diagrams. To further highlight how these components interact, we will analyze how one of the most important processes of Sentry is realized at runtime.

24.2.3.1 Event processing and storage

The main feature of Sentry is the capturing, processing and storage of error events. In the figure below, the flow for this scenario is illustrated. This figure is simplified, since there are many more interactions between more modules than shown here.

In the monitored application, the error event is captured by the Sentry SDK and sent to the back-end using the `/api/{project_id}/store` endpoint which is handled by the **web** module. From this endpoint tasks are started in the **tasks** module that process the event. One of these processing steps is analyzing the native stacktraces in the error event, which is done through the **lang** module by **Symbolicator**, an external service which is also developed by Sentry. After processing, the events are stored through the **event_manager**. Release and environment data linked to the event are stored in the **PostgreSQL** database, and the event

[19] Sentry. (2019, June 13). Building Sentry: Symbolicator. Retrieved March 14, 2020, from https://blog.sentry.io/2019/06/13/building-a-sentry-symbolicator

[20] Sentry. (n.d.) Slack + Sentry Integration. Retrieved March 15, 2020, from https://sentry.io/integrations/slack/

[21] Sentry. (n.d.) GitHub + Sentry Integration. Retrieved March 15, 2020, from https://sentry.io/integrations/github/

[22] Brown, S. (n.d.). The C4 model for visualising software architecture. Retrieved March 14, 2020, from https://c4model.com/

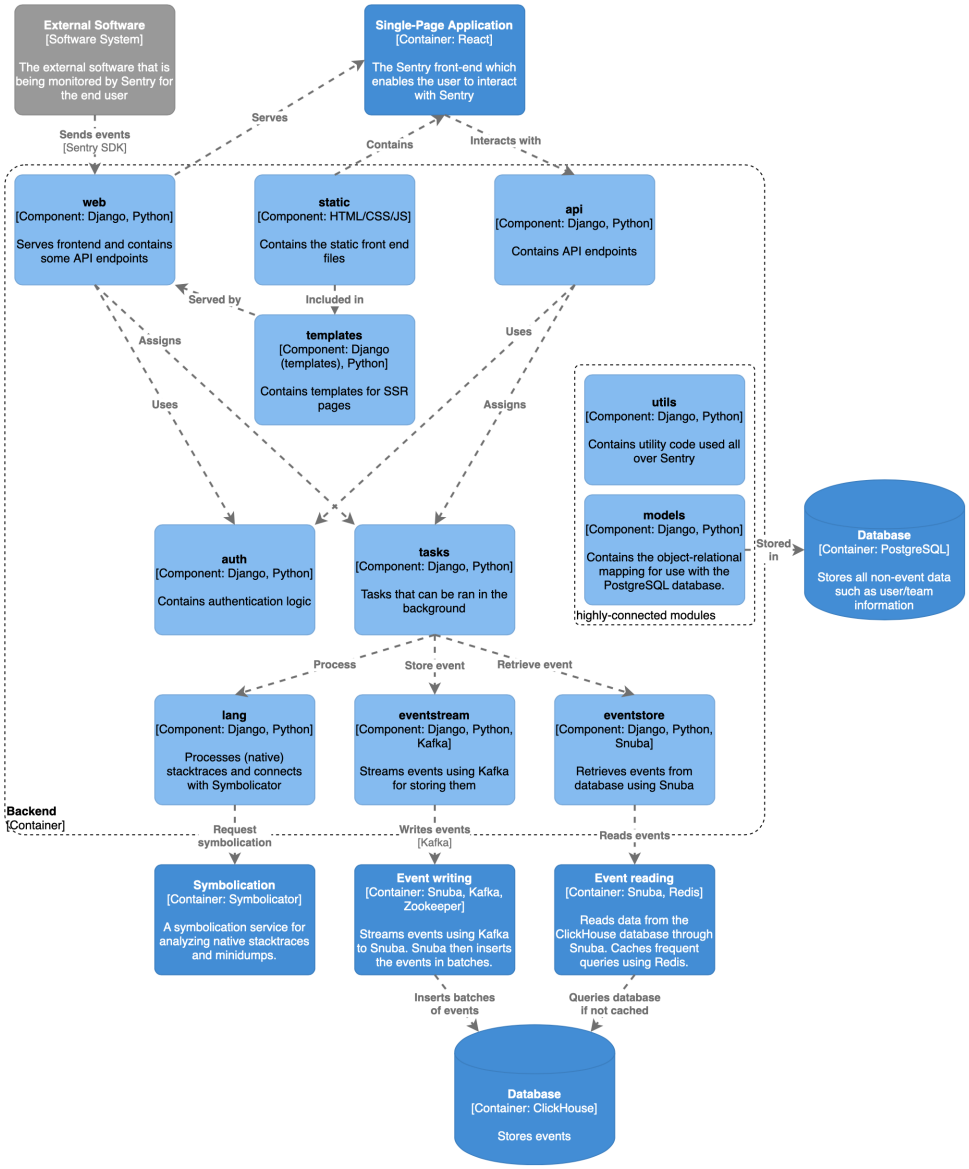


Figure 24.7: C4 Model for Sentry, Level 3 decomposition of back-end (simplified)

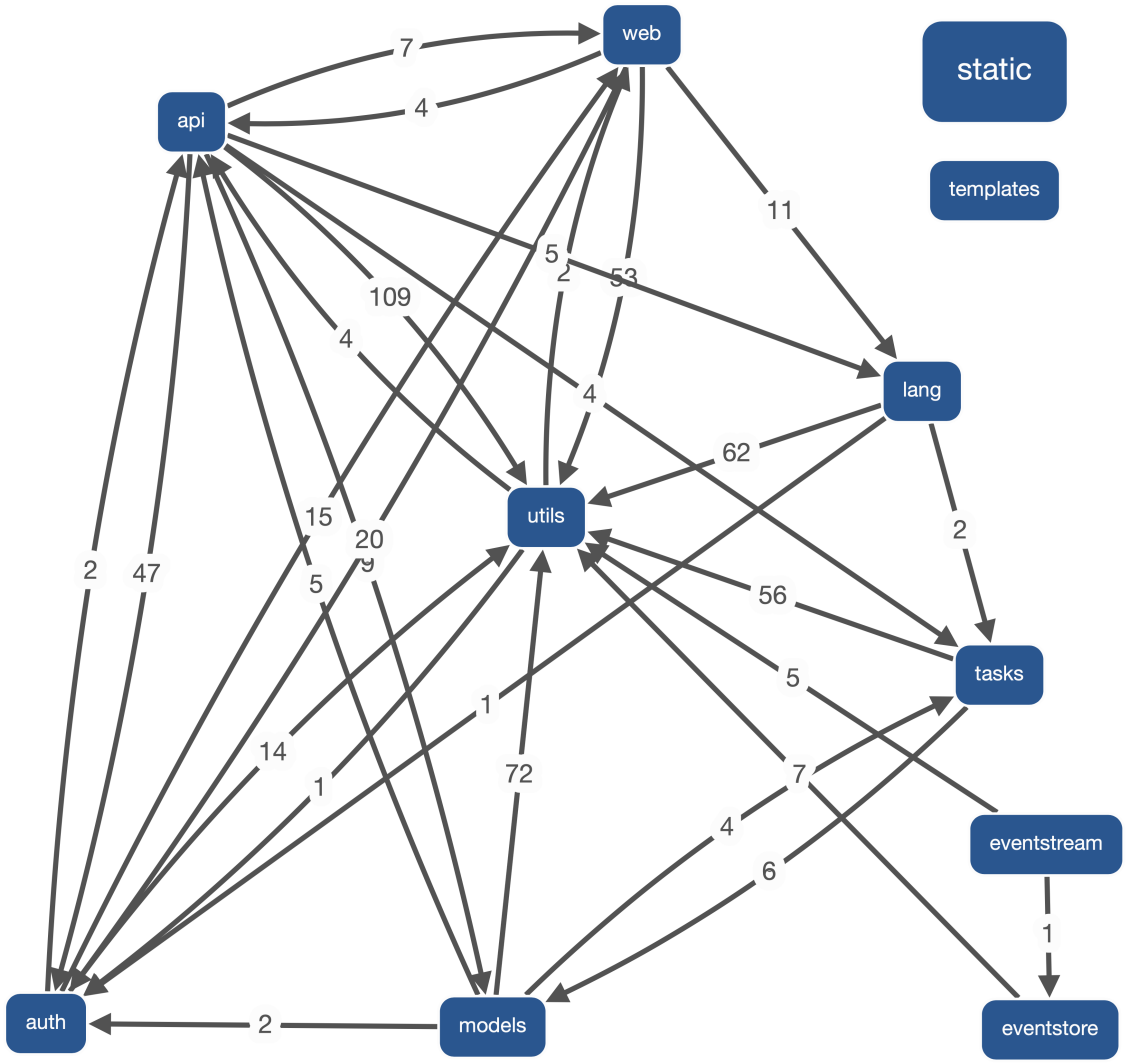


Figure 24.8: Dependency graph for Sentry generated by Sigrid (with only modules selected from figure above)

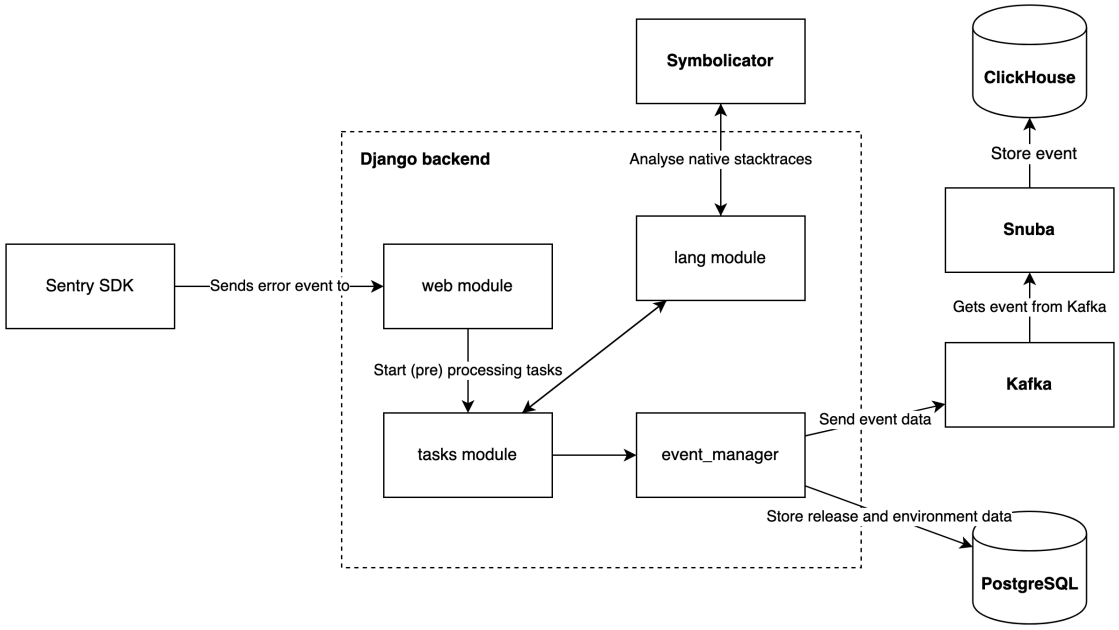


Figure 24.9: Event processing at runtime

itself is sent to the **Kafka** stream as a job. This event is picked up by **Snuba** and stored into the **ClickHouse** database.

24.2.4 A deployment view

Sentry is being deployed in three ways: on-premise, SaaS and front-end preview. The on-premise version is orchestrated by Docker Compose and is available on Sentry's onpremise repository. This starts containers for Sentry itself and all the services that Sentry relies on[23]. These services can be seen in the Level 2 decomposition above, and include:

- PostgreSQL
- Redis
- Memcached
- ClickHouse
- Snuba
- Symbolicator
- Kafka
- Zookeeper
- Exim

Deployment of Sentry itself is continuously done by a CI/CD pipeline task on the master branch, which builds and pushes an image of Sentry to Docker Hub. This image is then used in the on-premise setup.

[23] Sentry. (n.d.). Self-Hosted Installation. Retrieved March 15, 2020, from https://docs.sentry.io/server/installation/

Sentry also provides a paid SaaS version, but the deployment details for this version are not publicly visible. Lastly, the Sentry front-end is deployed to ZEIT Now for previewing purposes only (i.e. for pull requests).

24.2.5 Non-functional properties

- **Security**: is one of the important nonfunctional aspects of Sentry as it is monitoring the errors and sometimes events made by the tools of the customer. These can include private tools (which deal with private information) or public tools which deal with customer data. Making sure the communication between these SDKs and Sentry itself is secure is therefore important. They use different industry-standard technologies and services to secure their customers' data and provide mechanisms to filter out sensitive data in the SDKs[24]. As Sentry itself is open-source, it could be more vulnerable to attackers since adversaries can analyse the source code directly in order to find exploits. Hence, they urge developers to disclose vulnerabilities to the Sentry team directly[25].
- **Own tools vs Existing tools**: The Sentry team has opted to develop their own tools for event processing (Snuba) and a stack tracer parser (Symbolicator). For many years Sentry used existing tools to handle the storage of events using Search, Tagstore and TSDB[26]. However, as explained on their blog post from March 2019, this infrastructure was causing problems in scaling as well as for development[27]. This lead to an eventual choice to develop their own event storage solution based on the columnar store Scuba by Facebook (which is closed source). Initially, they used KSCrash for iOS platforms and Google Breakpad other platforms to catch crashes and compute stack traces. However, these tools were not targeted to support automated crash processing. Therefore, Sentry decided to create a standalone native symbolication service, Symbolicator[28].

24.3 Monitoring Sentry's Software Quality

Thousands of customers use Sentry for creating better software, but how is the software quality of Sentry itself? In this essay, we will analyze how the quality of Sentry's underlying code and architecture is safe-guarded. The architecture powering Sentry has already been discussed in the last essay ("The Architecture Powering Sentry"), so we recommend you to read that first. Having an understanding of Sentry's core software architecture, we can take a look at the architecture from yet another perspective by addressing topics like software quality assessment, code quality, and technical debt. This perspective is inspired by the *Quality Scenarios* and *Risks & Technical Debt* aspects from the arc42 framework[29].

24.3.1 Sentry's Software Quality Process

We will start this analysis by discussing the overall software quality process that the Sentry project uses. Sentry does not have a lot of explicit documentation on its overall quality assessment process. Hence,

[24] Sentry. (n.d.). Security & Compliance. Retrieved March 18, 2020, from https://sentry.io/security/#data-into-system

[25] Sentry. (2018, August 15). Sentry Scouts: Security - A Recap. Retrieved March 18, 2020, from https://blog.sentry.io/2018/08/15/sentry-scouts-security-recap

[26] Sentry. (2019, May 16). Introducing Snuba: Sentry's New Search Infrastructure. Retrieved March 14, 2020, from https://blog.sentry.io/2019/05/16/introducing-snuba-sentrys-new-search-infrastructure/

[27] Sentry. (2019, May 16). Introducing Snuba: Sentry's New Search Infrastructure. Retrieved March 14, 2020, from https://blog.sentry.io/2019/05/16/introducing-snuba-sentrys-new-search-infrastructure/

[28] Sentry. (2019, June 13). Building Sentry: Symbolicator. Retrieved March 14, 2020, from https://blog.sentry.io/2019/06/13/building-a-sentry-symbolicator

[29] Starke, G. (n.d.). arc42 Documentation. Retrieved March 24, 2020, from https://docs.arc42.org/home/.

Sentry's overall software quality process and the importance of it will be discussed in three parts by looking at Continuous Integration pipelines, automated checks, and GitHub usage.

24.3.1.1 Continuous Integration

Both Continuous Integration (CI) and Continuous Deployment (CD) are used in their development process[30]. They use short release cycles to maintain high code and product quality, to make sure that release issues are resolved quickly and to be able to resolve customer complaints faster[31]. Whenever a pull request is made, the following checks and pipelines are being run[32] [33] [34]:

Check	Description
Travis CI	Travis runs jobs for various tasks such as linting and various test suites for the frontend, backend, integrations, etcetera. They also test parts of their code against Python 3 in order to proactively support the porting process since Sentry still uses Python 2.7. The complete list of jobs can be seen in the screenshot below.
Percy	Visual changes in the UI are automatically monitored using Percy, which are then put up for review.
Codecov	The Travis CI pipeline sends test coverage reports to Codecov, which visualizes and reports the code coverage of the system and whether it has been negatively affected.
Snyk	Sentry uses Snyk to detect vulnerabilities in packages that they are using and to check for license issues.
ZEIT Now	ZEIT Now is being used to deploy a preview version of the front end for a pull request.
Google Cloud Build	Cloud Build is used to run the complete Sentry system using the on-premise repository and to perform some smoke tests. Then, for commits on master, it pushes a new Sentry Docker image to Docker Hub as a release.

24.3.1.2 Automated Testing

We found by analyzing the codebase that Sentry has a broad range of tests, including unit tests, regression tests, integration tests, and acceptance tests. They emphasize in their blogs on the importance of automated

[30] Sentry. (2018, August 6). Minimize Risk with Continuous Integration (CI) and Deployment (CD). Retrieved March 24, 2020, from https://blog.sentry.io/2018/08/06/minimize-risk-continuous-shipping-integration-deployment

[31] Sentry. (2018, July 23). Modernizing Development with Continuous Shipping. Retrieved March 24, 2020, from https://blog.sentry.io/2018/07/23/modernizing-development-continuous-shipping

[32] Sentry. (2018, August 6). Minimize Risk with Continuous Integration (CI) and Deployment (CD). Retrieved March 24, 2020, from https://blog.sentry.io/2018/08/06/minimize-risk-continuous-shipping-integration-deployment

[33] Sentry. (2018, July 23). Modernizing Development with Continuous Shipping. Retrieved March 24, 2020, from https://blog.sentry.io/2018/07/23/modernizing-development-continuous-shipping

[34] Sentry. (2018, October 11). Shipping Clean Code at Sentry with Linters, Travis CI, Percy, & More. Retrieved March 24, 2020, from https://blog.sentry.io/2018/10/11/shipping-clean-code-linters-travis-ci-percy

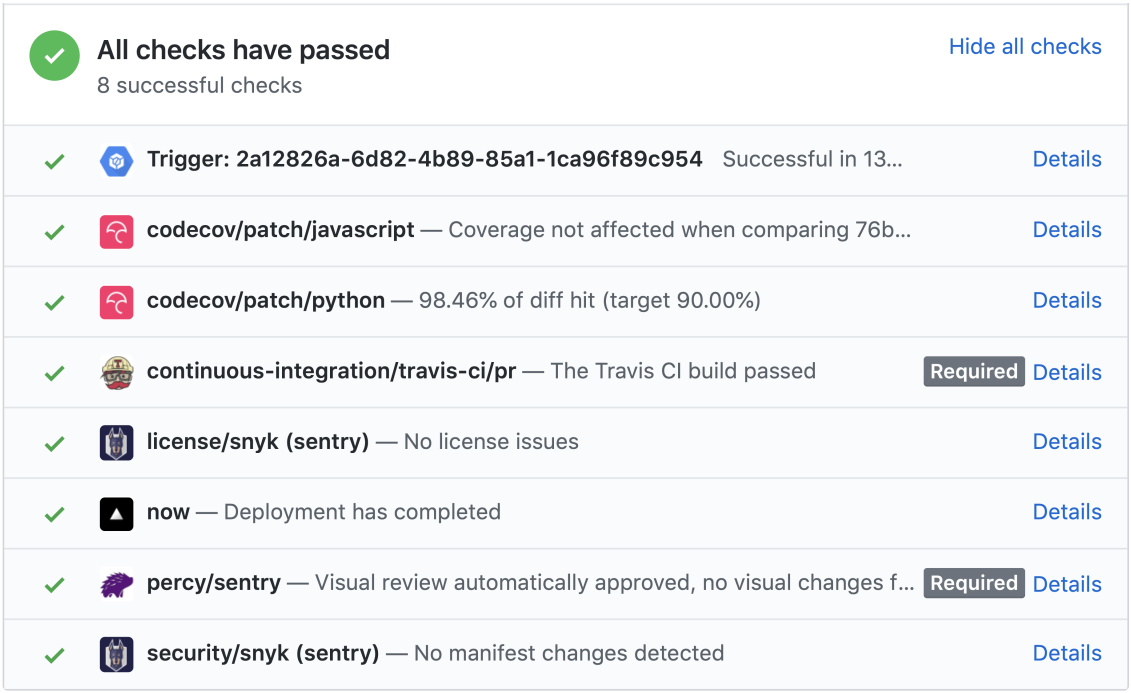


Figure 24.10: Checks on a GitHub Pull Request

regression testing not to verify whether the current code works, but to prevent that future changes to the code might result in issues[35]. They use tools like Jest for JavaScript, pytest for Python, Selenium WebDriver for acceptance testing and the mock library to create mock objects in Python tests. Even though there are quite a lot of tests, they do not seem to have any rules for (external) contributors regarding tests, e.g. every new feature should be tested. Later in this essay, we will discuss some coverage results reported by Codecov.

24.3.1.3 Other automated measures

Another important measure that Sentry has taken is implementing linters and formatters to create clean, readable and consistent code[36]. This helps with writing code with fewer errors, and also allows reviewers to find real issues instead of getting distracted by style issues. Linters include ESLint and flake8 and formatters include prettier and autopep8.

Another very valuable tool for Sentry is... Sentry! Since they continuously deploy and ship new versions of their software, there is a risk that something might go wrong. Sentry allows them to quickly find and resolve these issues and then deploy within a very short time.

[35] Sentry. (2018, October 11). Shipping Clean Code at Sentry with Linters, Travis CI, Percy, & More. Retrieved March 24, 2020, from https://blog.sentry.io/2018/10/11/shipping-clean-code-linters-travis-ci-percy

[36] Sentry. (2018, October 11). Shipping Clean Code at Sentry with Linters, Travis CI, Percy, & More. Retrieved March 24, 2020, from https://blog.sentry.io/2018/10/11/shipping-clean-code-linters-travis-ci-percy

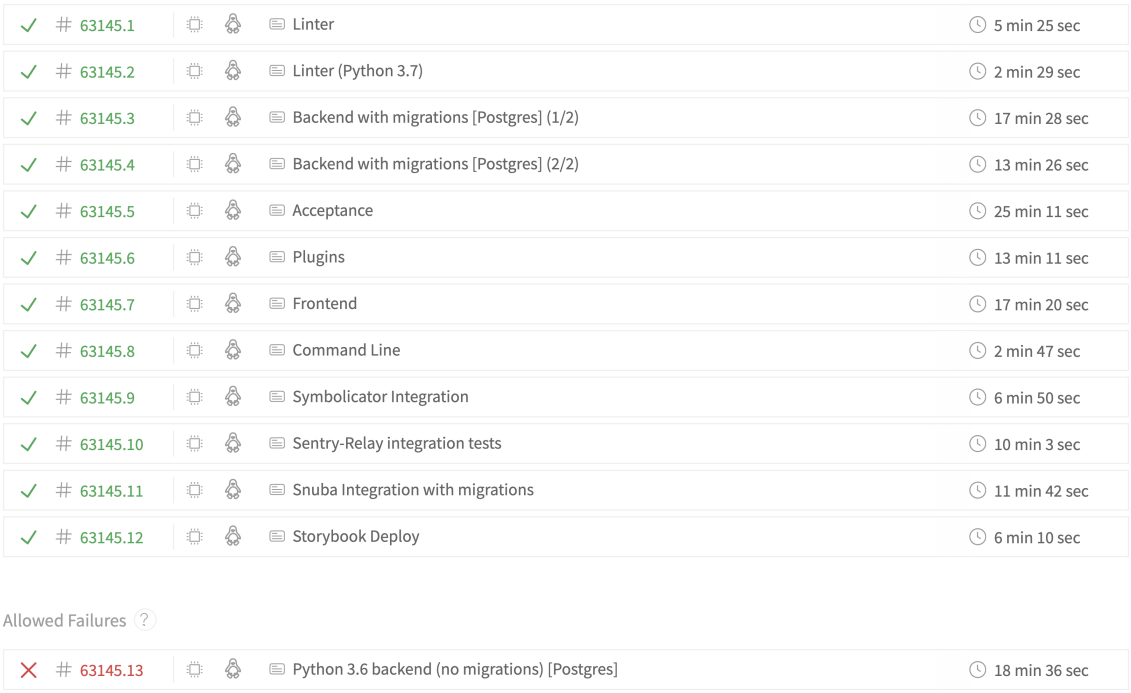


Figure 24.11: Screenshot from Travis showing a variety of jobs and configurations on master

24.3.1.4 GitHub usage

Sentry considers code review to be very important, and they have strict commit guidelines[37]. We analyzed a (relatively small) sample of pull requests and issues, either by randomly picking or by strategically searching using GitHub's querying system, such as: `is:pr is:closed complexity in:comments`. It can be seen that indeed code review is taken pretty seriously by the core Sentry team. Comments vary from warning about specific issues to nitpicks to comments about writing better tests.

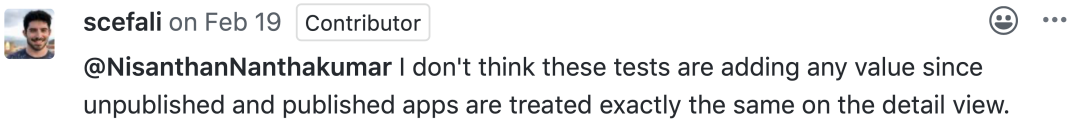


Figure 24.12: Comment on GitHub reviewing added tests

However, when taking another look at the automated checks for pull requests, it can be seen that only the Travis CI and Percy checks are required to pass. This could indicate that they do not consider things like test coverage so important to block a pull request by default. Moreover, we have not noticed any discussion about failing Codecov checks and PRs have been merged regardless of the Codecov status. Also, few discussions could be found about code quality metrics like complexity, module coupling or unit size and

[37] Sentry. (n.d.). Contributing. Retrieved March 24, 2020, from https://docs.sentry.io/development/contribute/contributing/.

Sentry does not make use of any code quality tools like SonarQube. Lastly, they also do not have a clear reviewing strategy (e.g. by using a `CODEOWNERS` file).

24.3.2 Investigating Hotspots

In order to find parts of the codebase that are particularly interesting to analyze, we will try to find parts of the Sentry codebase that are worked on a lot. Using a simple self-written tool we were able to analyze the commits between two given dates. This tool finds the min/avg/max amount of commits for a folder by aggregating the number of commits that 'touch' files within that folder (for the average only the changed files are taken into account). The code for this script can be found here and the full results here.

We ran two experiments, one to act as a baseline for our method and the second to find the current hotspots:

- Between release `v9` and `v10` (Jun 25, 2018 until Jan 7, 2020) as a baseline (results & conclusions), and
- Since the latest release `v10.0.0` (Jan 7, 2020 until Mar 23, 2020).

The baseline experiment is seen as out of scope for this essay but can be found here. The second experiment focuses on the parts of essay that are currently being worked on, we find the following hotspots since the latest release (`v10`):

```
Found 1318 commits in the range 2020-01-07T00:00:00 - 2020-03-23T00:00:00
MIN AVG MAX FOLDER
40  40  40  src/sentry/conf/
14  14  14  src/sentry/eventstore/snuba/
12  12  12  src/sentry/features/
11  11  11  src/sentry/static/sentry/app/views/events/utils/
3   10  17  src/sentry/static/sentry/app/views/alerts/list/
10  10  10  src/sentry/static/sentry/app/components/selectMembers/
10  10  10  src/sentry/static/sentry/app/views/settings/incidentRules/ruleForm/
1   9   13  src/sentry/static/sentry/app/views/settings/projectAlerts/issueEditor/
2   9   32  src/sentry/static/sentry/app/views/eventsV2/
1   9   69  src/sentry/static/sentry/app/types/
2   8   18  src/sentry/static/sentry/app/views/eventsV2/table/
6   8   11  src/sentry/static/sentry/app/views/eventsV2/savedQuery/
...
```

Figure 24.13: Results Analysis Sentry v10

We find that compared with our baseline, even after `v10` Snuba is still a hotspot, next we see a huge focus on the `src/sentry/static/sentry/app/` folder which is where the code for the frontend of Sentry is located. This seems to be one of the hotspots for the last two months since the release of `v10`, specifically the `types` and `views` seem to touch the most.

This would provide three major hotspots:

- `src/sentry/static/sentry/app/views` – Frontend Views
- `src/sentry/static/sentry/app/types` – Frontend Types (related to all aspects of the frontend including the views)
- `src/sentry/snuba` and `src/sentry/eventstore/snuba` – Parts of Snuba

While not investigated further, these hotspots could be related to bugfixes after a new major release.

24.3.3 Architectural Roadmap

Given the roadmap in our first essay "Putting Sentry into Context" we know the future direction of Sentry:

- In the short term improve developer experience, specifically for developers that run their applications on devices not in their control like on mobile, in the browser, or on the desktop.
- In the long term, move into the application monitoring space (APM).

In this section, we discuss how the current architecture would be affected by this roadmap.We will be selecting components from the architectural decomposition (C4 model) from our second essay "The Architecture Powering Sentry" to demonstrate what components are relevant for the roadmap.

First improving the experience for developers, not in control of the hardware their applications run on, can be done by improving the integration, symbolication, and with respect to symbolication the visualization of the symbolication. Improvements in this area, with respect to the C4 model, would have to be seen in the *single-page application*, *symbolication* process, and the *Sentry SDK's* for platforms like native desktop, mobile, and web.

The second part of the roadmap, to broaden the application scope from error monitoring to application monitoring, is a bigger change and therefore seen as a long term goal. Nonetheless, Sentry already contains some of the functionality required for application monitoring, specifically the functionality that overlaps with error monitoring, like cross-stack tracing of events and optimized searching through events using an eventstore. Further development towards this long term goal will affect the parts of Sentry related to handling events as not all events will be errors and there will be a higher amount of events coming in. The parts affected, with respect to the C4 model, will be the *backend* itself (due to changes in the API and higher load), *event writing/reading*, the *event database* and the *single-page application*.

24.3.4 Code Quality & Maintainability

Now that we have identified some interesting components in the codebase, let's take a look at some code quality and maintainability metrics for Sentry. Technical debt is a metaphor that highlights the unnecessary complexity in an application that makes implementing new features more difficult [38]. Estimations of the amount of technical debt are expressed in time and can be used to get an idea about the maintainability of a codebase. Technical debt can be analyzed by SonarQube, which uses the SQALE method to estimate this. Debt ratio is the ratio of technical debt to the total amount of development time [39].

We have applied this analysis to selected parts of the Sentry codebase, namely the main source code folders and the hotspots and roadmap modules outlined earlier based on the C4-model from the previous essay. The results from SonarQube are shown in the table below. According to SonarQube, Sentry scores an A

[38] Fowler, M. (2019, May 21). TechnicalDebt. Retrieved March 25, 2020, from https://www.martinfowler.com/bliki/TechnicalDebt.html

[39] SonarQube. (n.d.). Metric Definitions | SonarQube Docs. Retrieved March 25, 2020, from https://docs.sonarqube.org/latest/user-guide/metric-definitions/

for maintainability based on a low debt ratio (< 5%). The results indicate that the components are of high quality and the low debt ratio indicates that future changes should not result in issues.

The table below also contains test coverage results as reported by Codecov on the `master` branch. Sentry has a fairly high test coverage and even the hotspots and roadmap components mostly have high coverage.

Sentry code measures

Source	Related to	Line coverage (Codecov)	Technical Debt	Debt ratio
src		75.20%	25d	0.1%
src/sentry		86.12%	24d	0.1%
src/sentry_plugins		65.92%	6h 55m	0.1%
-				
Hotspots				
src/sentry/static/sentry/app/views	Frontend views	63.66%	1d 1h	0.0%
src/sentry/snuba	Snuba integration	86.12%	3h 52min	0.5%
src/sentry/eventstore/snuba	Snuba integration	97.5%	6min	0.1%
-				
Roadmap components				
src/sentry/eventstore	Event reading	93.86%	31min	0.2%
src/sentry/eventstream	Event writing	70.66%	3h 57min	0.9%
src/sentry/lang	Symbolication	88.64%	2h 51min	0.3%
src/sentry/api	API	89.32%	5d	0.3%

24.3.4.1 Refactoring Suggestions

SonarQube reports in total 2,729 issues when analyzing the Sentry codebase. These issues are violations of the rules that are defined by SonarQube. In the table below, the rules that are violated more than 50 times are shown.

Language	Rule Violation	# of violations
Python	String literals duplicated three or more times	1378
Python	Cognitive Complexity of functions higher than 15	245
Python	Functions with more than 7 parameters	88
JavaScript	Unused local variables or functions	76
Python	Collapsible `if` statements that are not merged	75
JavaScript	Default export names and file names that are not matched	63
Python	Empty functions or methods	56

By far the most violated rule is the duplication of string literals in Python. When looking at the specific violations, we can see that this happens often with for example configuration strings and dictionary keys. These violations are understandable, but it is recommended to define constants for the strings in these cases, which will improve the maintainability.

Another issue that can be observed frequently in the Sentry codebase is functions with high complexity. SonarQube uses its own metric for this, called Cognitive Complexity, which measures how understandable a

function is [40]. This greatly influences maintainability, so it is recommended to split up functions with high complexity.

Sigrid performs some analyses that SonarQube does not perform, especially related to components and the interaction between components. Sigrid ranks these analyses for each codebase on a scale from 0.5 to 5.5. One analysis where Sentry has a pretty low score (1.7), is called component entanglement. This checks the communication between components. For this analysis, Sigrid gives 66 refactoring candidates, which are mostly cyclic dependencies between components. The presence of cyclic dependencies adds complexity to the architecture, which makes the code more difficult to extend and maintain.

24.4 Supporting Every Language: Sentry's SDKs

No programming language is the same. Different syntaxes, static/dynamic typing, and varying build processes are just a few examples of differences that make it difficult to write a single piece of code that can report errors back to Sentry for all runtimes. Previously we have introduced Sentry, discussed it's architecture, and its development process. Now we broaden our scope to the ecosystem of SDKs supported by Sentry. We will show how Sentry is able to support many different runtimes and we will take a look at these SDKs that report back to Sentry to see how they work.

Sentry currently actively supports SDKs (Software Development Kits) for more than 12 languages. Moreover, they support a wide variety of frameworks for these languages, ranging from front-end frameworks to game engines[41]. Next to the SDKs maintained by Sentry itself, there are also various community supported SDKs available thanks to Sentry being open about how SDKs can and should be developed. Everyone can write or customize an SDK for their own software, as long as it is compatible with Sentry's specifications (which we will discuss soon).

24.4.1 How Sentry's SDKs are being developed

The SDKs developed by Sentry are not included in Sentry's repository itself. Instead, SDKs have their own repositories under Sentry's GitHub organisation. The SDK repositories are mostly made per language, although there are a few exceptions. For example, there is a sentry-laravel repository that makes use of the SDK from sentry-php. However, the Node.js SDK is simply a package within the sentry-javascript repository. It is unclear why Sentry decided to take these different approaches.

Since the SDK repositories maintained by Sentry are separate from the main repository, they also have their own development practices. These practices include code quality assessment and Continuous Integration (CI) – topics that we have discussed about Sentry as well in other essays. Since these topics are not well-documented and vary from repository to repository, we cannot take an in-depth look. We can make some general observations though.

First of all, most if not all of the repositories seem to have automated tests and at least one CI pipeline (mostly Travis CI). The CI builds are often configured to test the SDKs in a variety of ways such as for different versions of the language it is written in. However, some repositories seem to care more about test coverage or code quality in general than others (or Sentry itself for that matter). We have seen some repositories boasting a Codecov badge with a high coverage percentage like sentry-javascript and sentry-dotnet, while

[40] Campbell, G. A. (2017). Cognitive Complexity - A new way of measuring understandability. Retrieved from https://www.sonarsource.com/docs/CognitiveComplexity.pdf

[41] Sentry. (n.d.). Platforms - Docs. Retrieved April 8, 2020, from https://docs.sentry.io/platforms/



Figure 24.14: Some logos of languages supported by Sentry

other repositories have not even added Codecov analysis like sentry-python or sentry-java. It remains unclear why this is the case. We assume this is related to the difference in core contributors of these repositories. Another interesting observation was that the licenses between the different SDKs vary from the BSD (v2 or v3) clause to the MIT license, although this might have to do with compatibility with dependency licenses.

The Sentry forum is an important place for the (initial) development of the SDKs. If people are looking for an SDK of a specific language or framework that does not exist yet, they can indicate this on the forum or decide to make the SDK themselves and ask the Sentry team or the other community members for help.

The Sentry team uses the forum to ask the community for feedback. For example when they released a new version of the Python SDK.

24.4.2 How Sentry's SDKs work

Sentry's SDKs are a crucial part of Sentry as without them no error events would be captured and sent. In this section we take a general look at the working of such an SDK. Our findings are based on analysing a set of the SDKs and public information on the Sentry forums and their blog.

At the end of 2018 the Sentry team made the decision to revamp all existing SDKs to support a single unified API[42]. This decision was made to provide a unified experience across all SDKs, the basic construct being a simple initialisation step that should be similar everywhere:

```
init({dsn});
```

[42] Sentry. (2018, August 8). Join the Discussion on Sentry's Streamlined SDKs. Retrieved April 9, 2020, from https://blog.sentry.io/2018/08/08/new-sdk-unified-api-feedback-requested/

> We want a unified language/wording of all SDK APIs to aid support and documentation as well as making it easier for users to use Sentry in different environments.[43]

One of the reasons is in line with the roadmap we talked about in "Putting Sentry into Context": To move into the APM (application monitoring) domain.

> Design the SDK in a way where we can trivially add new features later that go past pure event reporting (transactions, **APM** etc.)[44]

Due to this revamp a lot of the different SDKs are now more similar and we can define the general workings of them:

- Each SDK will have an init method (in the syntax of the runtime) that supports a configuration, at minimum this should include a DSN code (to have working event reporting). This code is used by Sentry itself to identify the related project.
- Every SDK will have one or multiple "Hub"s which are objects that store state[45].
- The initialisation of the SDK should be done at the start of the application.
- Now behaviour starts to differ between different SDK's as for each runtime the situation is different. However from our findings we can conclude that the SDK will try to capture errors being reported and send them with their context to Sentry with minimal (sometimes no) code added by the developer.

The unified API also makes it possible to manually capture events or add context data like breadcrumbs (used as a trail of events, leading up to an event). Due to the unified API this can be done in similar fashions for all SDKs using the likes of: `capture_event` and `add_breadcrumb`.

24.4.2.1 Integrations

Manually capturing errors and collecting useful information for these errors can be a lot of work, but fortunately the Sentry SDKs help us out with integrations for many popular frameworks.

It is common to use frameworks on top of a runtime/language to help development. Take for example Flask, which is a lightweight web backend framework for Python. Sentry has a integration for the Python SDK that works specifically for Flask[46]. When specifying this integration in the Sentry Python SDK it provides the SDK with more knowledge about your application, which in turn means that you as developer will be provided better insights into the events presented by Sentry. In this example this would (among others) include:

- request data like: HTTP method, URL, headers, form data, JSON payloads
- if using authentication through 'flask-login', user data could be attached to events like an id, email, or username.

Sentry has similar integrations for the popular frontend frameworks (React, Vue, Angular, and more) that automatically detect relevant contextual information such as UI interactions, HTTP requests, etc.

Integrations are therefore a powerful aspect of these SDK's as they remove setup required to retrieve the extra context around events sent to Sentry. At least for the more popular frameworks.

[43] Sentry. (n.d.). Unified API. Retrieved April 9, 2020, from https://docs.sentry.io/development/sdk-dev/unified-api/
[44] Sentry. (n.d.). Unified API. Retrieved April 9, 2020, from https://docs.sentry.io/development/sdk-dev/unified-api/
[45] Sentry. (n.d.). Unified API. Retrieved April 9, 2020, from https://docs.sentry.io/development/sdk-dev/unified-api/
[46] Sentry. (n.d.). Flask - Docs. Retrieved April 8, 2020, from https://docs.sentry.io/platforms/python/flask/

24.4.3 How SDKs communicate with Sentry

Every one of the different SDKs wants to report events (whether they are errors or something else) to the Sentry backend to be analysed and to eventually be shown to the developer. The SDKs do this by sending their events to the `/api/{PROJECT_ID}/store/` endpoint[47].

24.4.3.1 The event object

Events should contain some required attributes, such as an `event_id`, which identifies the event[48]. This identifier is automatically added by most SDKs as a UUID4. Other required fields include a timestamp, the name of the logger that reported the event, and the platform from which the event was submitted. Furthermore, Sentry recommends to add some other attributes with general information to the error object, such as the event level (e.g. `error`, `info`, etc.), the software release version, the environment (e.g. `production` or `development`) and more.

Additionally, more detailed information can be attached to the events via the core and scope interfaces[49]. The core interfaces include the exception interface, the stacktrace interface, the message interface and the template interface. These data objects contain the specific error or message event that is captured. Furthermore, the scope interfaces give more contextual information about the captured events. These include the breadcrumbs, request and UI interfaces that can contain events that happened prior to the captured event, but also a contexts interface that typically contains relevant information about the user or the environment.

An example of an event body can be seen below:

24.4.3.2 Backend processing

Upon receiving events (with identification of the dsn code) the Sentry backend processes the event by:

- Extracting the projectId from the url, and
- Scrubbing sensative data according to project settings, and
- Storing the event in the event database

This will then trigger other components of Sentry to further process the event if needed. This is described in more detail in "The Architecture Powering Sentry".

24.4.4 How you can build your own SDK

Now that we know a little about the existing SDKs, the general working of them and the communication they have with Sentry itself, how do we build an SDK?

Sentry provides a detailed guide explaining what is expected behaviour for our SDK. This includes adhering to the unified API as explained before.

According to this guide the base of our SDK should to be able to automatically capture errors and at least contain the following set of features[50]:

- DSN configuration (Data Source Name: represents the configuration of the Sentry SDK, for example including an auth token and project id.)

[47] Sentry. (n.d.). Event Payloads. Retrieved April 8, 2020, from https://docs.sentry.io/development/sdk-dev/event-payloads/
[48] Sentry. (n.d.). Event Payloads. Retrieved April 8, 2020, from https://docs.sentry.io/development/sdk-dev/event-payloads/
[49] Sentry. (n.d.). Event Payloads. Retrieved April 8, 2020, from https://docs.sentry.io/development/sdk-dev/event-payloads/
[50] Sentry. (n.d.). SDK Development Overview. Retrieved April 8, 2020, from https://docs.sentry.io/development/sdk-dev/

```
{
  "event_id": "fc6d8c0c43fc4630ad850ee518f1b9d0",
  "timestamp": "2011-05-02T17:41:36Z",
  "logger": "my.logger.name",
  "platform": "javascript",
  "level": "error",
  "environment": "production",
  "exception": {
    "values": [
      {
        "type": "ValueError",
        "value": "my exception value",
        "module": "__builtins__",
        "stacktrace": {}
      }
    ]
  },
  "breadcrumbs": {
    "values": [
      {
        "timestamp": "2016-04-20T20:55:53.847Z",
        "type": "navigation",
        "data": {
          "from": "/login",
          "to": "/dashboard"
        }
      }
    ]
  }
}
```

Figure 24.15: Example event body

- Graceful failures (catching failures such as an unreachable Sentry server)
- Setting attributes (such as a list of tags for the event containing context or versioning information)
- Support for Linux, Windows, and OS X (depending on the type SDK you are building)

Further development should target the full list of expected features described in the documentation. Such as the asynchronous transmission of events (in the background) or the automatic collection of stack traces on an error event. Depending on the target language or framework, this can be difficult. For example, for compiled applications the retrieval of the stack trace can be really hard. As debug information (like a stack trace) can be really large (a couple of gigabytes in some cases for a single crash), Sentry encourages contributors to use gzip for encoding the event body to decrease the event message size.

Chapter 25

Signal for Android

In recent years, privacy and security have become topics of focus within the software engineering community as well as in the global discourse on technology. The interest in those topics is not limited to the technological communities, as more and more legislators are introducing laws and regulations to protect the privacy and security of their citizens, such as the European General Data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA).

With regulators doing their best to protect general privacy in terms of the information that organizations store about their clients, we should not forget the means through which we express our deepest desires and greatest wishes: interpersonal communication. We now live in an era where more and more of this communication takes place over the internet through messaging apps. However, most of the popular messaging apps are closed source, meaning that an outside observer can not easily see what the application is doing on the background.

Signal is a secure privacy-central alternative to existing messaging apps. In contrast to its competitors, Signal has its source code available to the public, making it easier to verify what it is doing. Striving to be a secure privacy-central alternative to existing messaging apps, Signal encrypts all messages that its users send and receive through the app. This goes a step beyond the end-to-end encryption used by many other popular messaging apps. Most of these apps encrypt the message content, but Signal additionally encrypts the metadata: Who sent the message at what time. This ensures a higher degree of privacy for the end user. Signal can be used on three different platforms: Android, iPhone, and Desktop. This blog will focus exclusively on Signal's Android implementation.

This blog will take you on a journey under the hood of Signal, through the eyes of a software architect. The four essays that compose this blog will highlight different aspects of Signal's architecture. The first essay details the vision underlying Signal, placing it into the context of its users and analyzing its stakeholders. The second essay describes how this vision resonates with the architecture of the app. The third essay takes a deeper dive into the code that composes Signal. The fourth essay is a deeper analysis of the protocols that enable Signal's unique features and of how they are reflected in the architecture. The goal of this journey is to uncover the objectives for the Signal application. It will furthermore be investigated how these objectives are realized through both the software architecture and development process.

25.1 Who we are

We are a team of four students of the Delft University of Technology in the Netherlands. We have made this blog as part of the CS4315 Software Architecture course, which we have followed as part of our master of Computer Science program.

In the following paragraphs, the members of the team will give a short introduction of themselves.

Martijn van den Hoek After finishing my Bachelor of Computer Science in Delft, I continued with the master in the same discipline. I'm interested in the process of software engineering and the proper software engineering techniques. Besides courses that connect with those interests, my master also includes courses about algorithms.

Wouter Zonneveld Software Engineering is one of the sub-fields in Computer Science which I've always found to be both interesting and practical. This is because having a thorough understanding about how to go from the product requirements to the actual product is essential, no matter in which context you are programming.

Frank Vollebregt My interests lie within the fields of Web Information Systems and Software Engineering. Nowadays, there is an increasing focus on privacy for the end user: Designing and building systems with the additional objective of privacy in mind brings new challenges to the table, for us to solve.

Robin Oosterbaan As all the other group member I finished the bachelor and continued with the Computer Science Master. The Software Engineering group have always given solid courses and as such I wanted more knowledge on the topic of Software Architecture.

25.2 A clear vision or mixed signals?

In this post, we try to discern what the goals of Signal are, how they intend to accomplish these goals and who benefits or suffers from them. We have a look at the unique features Signal offers to its users, and take a leap forward to see what the future may hold.

25.2.1 What Signal aims to achieve

On the Signal website, the project goals are displayed. Signal offers a library which facilitates encrypted messaging. Additionally, they provide Android, iPhone and desktop versions of their messaging app, which uses this library. The goals listed on the website are:

- **Fast**: The computations required for the message encryption and decryption that Signal's library requires inherently cause secure messaging to be slower than regular, non-encrypted messaging. Having an application that either rapidly drains the battery of the users' devices or that becomes sluggish and unresponsive, solely to perform some direct communication, is not acceptable. The Signal applications have also been designed to scale well with multiple users, enabling fast chats, even in groups with many users in them.
- **Simple**: In the development ideology of the contributing file, it is clearly stated that exposing more options to the end user is in most cases undesirable. What's more, during the development one should not consider so-called *power users* who can benefit from having functionality exposed to them. In short, keeping the application simple increases security for all users.

- **Secure**: Signal is best known for its focus on security. The encryption model and library developed by Signal is known for being highly secure. The applications built with the libraries are also designed with a careful eye for security.

25.2.2 Signal's intended audience

When using Signal, the user expects to find all functionality that is conventional for a modern messaging app: Sending messages to individuals or groups, with support to send only text or add attachments like images, locations and such. What sets the mental model of Signal's users apart from other messaging apps' users is the additional expectation of privacy and transparency. Signal specifically targets both the privacy-conscious user, who wants to use instant messaging without anyone looking over his digital shoulder, as well as the user who values transparency, which is provided by Signal through their open-source code bases.

25.2.3 What sets Signal apart

As mentioned before, Signal is all about security and privacy. A feature of Signal is that some of the aspects of its security are visible to the users. Users can, for example, decide to reset the security of conversations. Moreover, they can verify the identity of other users by scanning a QR-code on their device.

A technical feature within Signal is that some metadata of messages sent within Signal is also encrypted. For instance, the *sender* field of a message in Signal is sealed, meaning it is not visible to anyone except the receiver of the message[1]. Along with these unique features, Signal comes with all the features a user expects from a modern instant messaging application, such as document sharing, voice/video chatting, et cetera.

25.2.4 Who is affected by Signal

Since a stakeholder is used in several ways, we define a stakeholder as an entity which is directly affected by the system. Signal is developed by Signal Messenger LLC which was founded by *Moxie Marlinspike* and *Brian Acton* in 2018. However, since it is open-source contributions to the application can be made by anyone who is interested in contributing to the product.

Signal is funded by the non-profit organization *"Signal Foundation"* which was created by Brian Acton in 2018. Furthermore, in the startup phase of Signal, donations and grants from among others the *Knight Foundation*, *Shuttleworth Foundation*, and the *Open Technology Fund*, were used to fund Signal's development.

Users of Signal's applications are also a major stakeholder. They rely on the security and privacy Signal provides for their private conversations. These users can be split up into two categories: *companies* and *individuals*. A company could decide to use Signal for its internal communication and thus rely on Signal's availability.

Where regular users and companies seek privacy and security, intelligence agencies such as the NSA have contradictory wishes. Such agencies want to be able to monitor digital communication, so that they can identify potential threats. In the case of Signal, however, they are unable to do so, since Signal's encryption make this an impossibility. An NSA employee has even said that this encryption is a "major threat" to their mission[2].

[1] Joshua Lund. Technology preview: Sealed sender for Signal. published on 29-10-2018. retrieved from https://signal.org/blog/sealed-sender/. retrieved on 03-03-2020.

[2] Jacob Appelbaum, Aaron Gibson, Christian Grothoff, Andy Müller-Maguhn, Laura Poitras, Michael Sontheimer and Christian Stöcker. Inside the NSA's War on Internet Security. published on 28-12-2014. retrieved from https://www.spiegel.de/international/ger

25.2.5 Current and future context

Two different perspectives can be assumed when looking at Signal: the business perspective and the technical perspective. When considering the business perspective, we can see that Signal has a number of competitors[3]. However, when comparing those competitors in terms of features[4], it becomes clear that only Facebook Messenger, WhatsApp and Telegram offer end-to-end encryption. Because of the transparency that Signal enjoys from its open-source nature, those who value privacy in their means of communication are likely to favor Signal over the competitors that offer end-to-end encryption. Now that people start to value their privacy more and more, it could be possible that the amount of users that choose Signal will increase.

From the technical perspective, we can observe that governments are trying to pressure companies such as Signal to loosen up their promises on privacy and security. For example, the American government ordered Signal in the first half of 2016 to hand over the data it held on two accounts, and to not make public that it had done so or had received the order to[5]. In a resolution proposed by Interpol, the organization seeks the ability for law enforcement to defeat end-to-end encryption as used by messaging apps such as Signal[6].

However, such legislation is not the only threat to Signal's encryption protocol. The development of quantum computers poses a threat to the encryption schemes that are currently being used[7]. In order to keep Signal safe in a future where quantum computers can be used to defeat its encryption protocol, recent research has analyzed different algorithms which can be used to prepare the Signal protocol against attackers who possess a quantum computer[8].

25.2.6 What's in store for Signal

Although Signal developers generally do not talk about features until they are ready[9], Signal's creator Moxie Marlinspike revealed a number of features that the app is going to implement[10]. Those features revolve around further increasing the privacy of the users, while maintaining the functionalities that is offered by other instant messaging apps. The two main features revolve around the creation of groups and the recommendation of friends to a user.

In the case of the creation of groups, Signal wants to be able to create groups in such a way that the Signal

many/inside-the-nsa-s-war-on-internet-security-a-1010361.html. retrieved on 04-03-2020.

[3] Statista. Most popular global mobile messenger apps as of October 2019, based on number of monthly active users. published on 20-11-2019. retrieved from https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/. retrieved on 06-03-2020.

[4] Wikipedia. Comparison of cross-platform instant messaging clients. retrieved from https://en.wikipedia.org/wiki/Comparison_of_cross-platform_instant_messaging_clients. retrieved on 06-03-2020.

[5] Brett Max Kaufman. New Documents Reveal Government Effort to Impose Secrecy on Encryption Company. published on 04-10-2016. retrieved from https://www.aclu.org/blog/national-security/secrecy/new-documents-reveal-government-effort-impose-secrecy-encryption?redirect=blog/free-future/new-documents-reveal-government-effort-impose-secrecy-encryption-company. retrieved on 06-03-2020.

[6] Joseph Menn. Exclusive: Interpol plans to condemn encryption spread, citing predators, sources say. published on 17-11-2019. retrieved from https://uk.reuters.com/article/uk-interpol-encryption-exclusive-idUKKBN1XR0S5. retrieved on 06-03-2020.

[7] Wayne Rash. Quantum Computing Poses An Existential Security Threat, But Not Today. published on 31-08-2019. retrieved from https://www.forbes.com/sites/waynerash/2019/10/31/quantum-computing-poses-an-existential-security-threat-but-not-today/. retrieved on 08-03-2020.

[8] Ines Duits. The Post-Quantum Signal Protocol: Secure Chat in a Quantum World. published on 05-02-2019. retrieved from https://www.semanticscholar.org/paper/The-Post-Quantum-Signal-Protocol-%3A-Secure-Chat-in-a-Duits/ea9216c3c7ab51d74f1d02ea274f656caf3fcbab. retrieved on 06-03-2020.

[9] Joshua Lund. (reaction to) A proposal for alternative primary identifiers. published on 01-06-2018. retrieved from https://community.signalusers.org/t/a-proposal-for-alternative-primary-identifiers/3023/10. retrieved on 03-03-2020.

[10] Andy Greenberg. Signal Is Finally Bringing Its Secure Messaging to the Masses. published on 14-02-2020. retrieved from https://www.wired.com/story/signal-encrypted-messaging-features-mainstream/, retrieved on 03-03-2020.

servers do not know who participate in these groups[11]. In order to show a user which of his friends are on Signal as well, it would be required to scan through the user's address book and match the telephone numbers to the telephone numbers of Signal users. The issue with this is that it requires the user to send his address book to Signal's servers, which might compromise privacy. To work around this issue, Signal wants to process the user's address book in a secure cryptographic enclave, available on Intel processors[12]. By doing so, the user's address book still has to be uploaded to the servers, but now the user can validate that the server deleted the address book once it has scanned it and found the user's friends who are on Signal as well. Besides those improvements, Signal aims to increase its user base to billions of users[13].

As stated before, Signal does not have an official roadmap containing the upcoming features. For the Android version of Signal however, it is possible to make an educated guess about which features will be rolled out in the near feature, as a comparison is being kept between the different platforms on which Signal is available[14].

We are excited to see the growth of Signal and similar privacy-centered initiatives. Online privacy has recently gained a lot of additional public attention with the introduction of new privacy legislation like the Europeans GDPR, and as more people become data-aware, Signal can expect to gain a lot more traction in the years to come.

25.3 Rome wasn't built in a Signal day

In this post we explore how the concepts discussed in our previous post are realized through Signal's architectural elements, deployment, and other design principles. We will also see how key features, such as sending a message, work at run-time.

25.3.1 Architectural views

For identifying the different architectural views, we will use the 4+1 model described by Philippe Kruchten[15].

Logical view The logical view of the system is important to identify functionality which the system provides to the user. Different diagrams are used to visualize the logical view of the system. These diagrams are useful to communicate new features in Signal.

Physical view The physical view contains the hardware aspect of the Signal application. There are two parts of hardware which are connected via an internet connection: the client (e.g. on a user's phone), and the server (hosted by Signal Messenger LLC). It is important to understand how the hardware is connected to each other in order to understand why the software is architected in a certain way.

Development view It can be argued that the development view is crucial for any software system. Modularisation impacts how code is written and how well it can be maintained. This again impacts the ease at which

[11] Andy Greenberg. Signal Is Finally Bringing Its Secure Messaging to the Masses. published on 14-02-2020. retrieved from https://www.wired.com/story/signal-encrypted-messaging-features-mainstream/, retrieved on 03-03-2020.

[12] Andy Greenberg. Signal Has a Fix for Apps' Contact-Leaking Problem. published on 26-09-2017. retrieved from https://www.wired.com/story/signal-contact-lists-private-secure-enclave/, retrieved on 03-03-2020.

[13] Andy Greenberg. Signal Is Finally Bringing Its Secure Messaging to the Masses. published on 14-02-2020. retrieved from https://www.wired.com/story/signal-encrypted-messaging-features-mainstream/, retrieved on 03-03-2020.

[14] klajsdgasjg. [Wiki] Feature Comparison: Android, iOS, Desktop. published on 03-03-2020. retrieved from https://community.signalusers.org/t/wiki-feature-comparison-android-ios-desktop/12003, retrieved on 03-03-2020.

[15] Philippe Kruchten. Architectural Blueprints — The "4+1" View Model of Software Architecture. published in November 1995. retrieved from https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf/. retrieved on 19-03-2020.

features for Signal can be added, and how quickly issues can be fixed.

Process view The process view is important since non-functional requirements such as security and availability are crucial for a privacy centered instant messaging application. When the security is breached, or the up-time is low, users will resort to using a different application. Ease of use is another crucial requirement, as the co-founder Moxie Marlinspike said himself: *"In many ways crypto was the easy part. The hard part is developing a product that people are actually going to use and want to use. That's where most of our effort goes"*[16].

Scenarios This final view contains use cases which describe a sequence of interactions between objects and processes. Using these scenarios, architectural elements can be identified and Signal's architectural design can be illustrated.

25.3.2 Non-functional but still important

Security One of the non-functional properties which Signal is well-known for, is security. The underlying security protocol, i.e. the Signal protocol, is cleared to use for the U.S. Senate[17] and the NSA regards the encrypted voicecalls as a "major threat" to their operations[18]. Keeping the application secure during open source iterative development requires critical analysis of every proposed modification, since any contributor could potentially introduce weaknesses to the system.

Speed The signal protocol has also been developed to be fast. In order to decrease latency, Signal hosts more than a dozen servers around the globe[19] to decrease the distance between the clients and a server.

Reliability Because Signal is also used for business critical communication, reliability is regarded to be an important property. Reliability of the code over time is achieved by following the deployment cycles that use beta testers to verify the workings of important aspects in the application. Since multiple servers are available, an outage of a server solely results in some extra latency, as another server can be used as a backup.

Transparency Convincing users of the trustworthiness of Signal is partially done using the open source initiative. The communication from developers to the users is another way to convey the intentions and guarantees that Signal offers.

Usability The Signal team has decided to keep the application as simple as can be for the end user. This in turn results in strict guidelines for keeping the user interface clean and withholding options from the user.

25.3.3 How is Signal deployed?

The Signal deployment process is as far as we know undisclosed. Nevertheless we can deduce the steps that have to be taken in order to release a new version.

[16] Andy Greenberg. Your iPhone Can Finally Make Free, Encrypted Calls. published on 29-07-2014. retrieved from https://www.wired.com/2014/07/free-encrypted-calling-finally-comes-to-the-iphone/. retrieved on 17-03-2020.

[17] Zack Whittaker. In encryption push, Senate staff can now use Signal for secure messaging. published on 16-05-2017. retrieved from https://www.zdnet.com/article/in-encryption-push-senate-approves-signal-for-encrypted-messaging/. retrieved on 17-03-2020.

[18] Michael W. Macleod-Ball, Gabe Rottman, Christopher Soghoian. The Civil Liberties Implications of Insecure Congressional Communications and the Need for Encryption. published on 22-09-2015. retrieved from https://www.aclu.org/sites/default/files/field_document/encrypt_congress_letter_final.pdf. retrieved on 18-03-2020.

[19] Andy Greenberg. Your iPhone Can Finally Make Free, Encrypted Calls. published on 29-07-2014. retrieved from https://www.wired.com/2014/07/free-encrypted-calling-finally-comes-to-the-iphone/. retrieved on 17-03-2020.

For development of both the client and the server, Continuous Integration is used, but Continuous Deployment is not. Gradle is used to manage packages.

For the deployment view we will consider two parts of Signal: the Android client and the server. The client application runs on a user's phone with Android 4.4 or newer[20]. The server application runs on a server owned by Signal Messenger LLC. The server and the client are connected via the internet.

Client The application receives updates via the Google Play Store. Depending on the user's settings, it either updates automatically or asks for permission from the user to perform an update. Alternatively, the Signal website also hosts the compiled APK as a method to install the application.

Server Also hosted as an open source project, the server is developed to be hosted in a distributed manner. They run in SGX enclaves. These distributed enclaves can verify each other during operation using *MRENCLAVE* attestation checks[21], again increasing the difficulty of compromising the server network. Signal also uses the AWS Global Accelerator in order to load balance server requests[22].

25.3.4 Starting at the top: Architectural style and patterns

We will look at Signal from the highest level possible and then start zooming in on the architecture of the Android application and the design patterns that have been implemented in it.

Considering both Signal's Android app and the servers that run its infrastructure, we can see that a client-server architecture has been deployed. When sending a message to a friend, the message is encrypted by the sending client, and then sent to the Signal servers. Consecutively, the server notifies the receiving client of a new message using Firebase Cloud Messaging. The receiving client then obtains the message from the Signal servers.

Zooming in on the Android app itself, it becomes evident that a layered architecture has been implemented by means of a model-view-controller (MVC) design pattern to separate the *presentation* and *business* layers. The Data Access Objects (DAOs) form the *persistence* layer which separates the *business* layer from the *database* layer. It has to be noted that not the whole Android app uses the MVC patterns, as traces of the model-view-viewmodel (MVVM) pattern are present throughout the source code.

To make the MVC pattern concrete, we will take the architecture of a single message in a conversation as an example. Such a message is represented by the `ConversationItem.java` class and forms the View part of the MVC, together with a number of `xml` files that describe the layout. The controller that controls the view is represented by the `ConversationFragment.java` class. The final part of the MVC pattern is being fulfilled by the `MessageRecord.java` class. This class represents the data in the application and simultaneously serves as a DAO to the local database within the application.

25.3.5 Diving deeper: Development of the application

In the previous section we have seen the architectural styles and design patterns that are used by Signal. By zooming in on the Android app, we discovered that underneath the app was a layered architecture. In this

[20] Signal support page. Installing Signal. retrieved from https://support.signal.org/hc/en-us/articles/360008216551-Installing-Signal. retrieved on 18-03-2020.

[21] Joshua Lund. Technology Preview for secure value recovery. published on 19-12-2019. retrieved from https://signal.org/blog/secure-value-recovery/. retrieved on 19-03-2020.

[22] Joshua Lund. Technology Preview for secure value recovery. published on 19-12-2019. retrieved from https://signal.org/blog/secure-value-recovery/. retrieved on 19-03-2020.

section, we will decompose the architecture of Signal into its main modules and see how the connectivity of the components reflects this layered architecture.

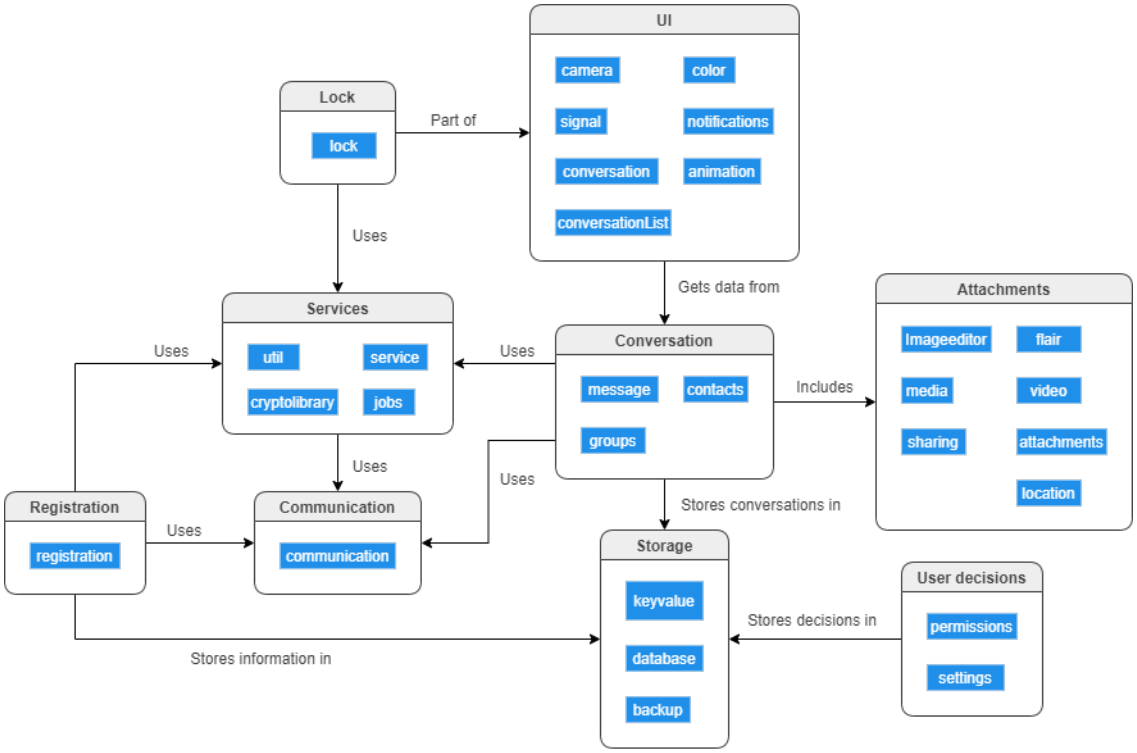


Figure 25.1: High-level decomposition of the Signal app for Android

Deriving the high level decomposition of the module required a number of steps to bring the complexity of the obtained decomposition down. In the first step, folders from the Signal source code were grouped together by functionality in order to obtain the components of the app. In the figure above, these components have been depicted as blue rectangles. However, when analyzing the dependencies between said components, it became clear that they were too complex to be clearly represented by a single figure. In order to make the decomposition more readable, components with the same functionality have been grouped together into the main components that compose Signal, which have been depicted as the boxes with a title.

The decomposition of the main modules nicely shows the layered architecture that has been discussed before. We can see that the *presentation* layer is represented by the `UI` component, which in turn gets its information from the *business* layer, represented by the `Conversation` component. Up until this point, everything is in line with the architectural models. However, the decomposition as shown above does not make an explicit distinction between the *persistence* layer and the *database* layer. Instead, the `Storage` main component (more specifically the `database` component within this main component) does not only contain the code for the *database* layer, but it also contains the code that would normally be included in the *persistence* layer by offering the DAOs that have been described in the previous section.

Now that we have seen the layered architecture in its full glory, we will shortly consider the other components and give a quick description of their functions where deemed necessary. The `registration` component

serves the function of registering a user. The `lock` component allows the user to set up a PIN code, which must be entered the next time a user tries to register with their phone number[23]. This places them somewhat out of the normal operation of Signal. The `communication` component takes care of all communication, be it SMS, MMS or Firebase Cloud Messaging. The `services` component contains different functionalities that are offered as services or jobs to run.

25.3.6 Looking at Signal during run time

We can also look at the application and how it operates during run time. Below we describe how a session is established in Signal, and how messages and calls between users are made.

25.3.6.1 Under the hood: sessions

To provide communication between two users, Signal creates a so-called session. Such a session contains the encryption keys used for communicating with another user. Since a user may have multiple devices, multiple sessions may be created: Suppose Alice and Bob want to be able to send messages: If Bob uses Signal on two devices (e.g. an Android Phone and a desktop computer), Alice has to establish a session to both of Bob's devices. When a user removes the application or adds another device, Signal notifies his contacts in the chat that "the safety number has changed". A user can verify the session by scanning a QR code on the other's device.

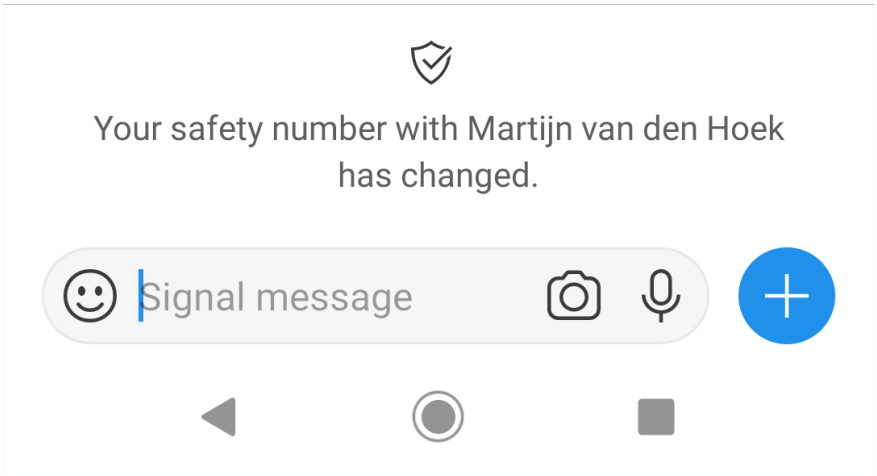


Figure 25.2: The app notifies a user of a session update

25.3.6.2 You got mail

When sending a message to other users, the encryption keys from the sessions that are described above are used. The message is encrypted on the sender's device for each active session with the recipient and sent to

[23] Jim O'Leary. Improving Registration Lock with Secure Value Recovery. published on 27-01-2020. retrieved from https://signal.org/blog/improving-registration-lock/. retrieved on 19-03-2020.

the Signal server. When the recipient opens the app, or the app checks for messages in the background, they will receive the encrypted message, which is stored on the recipient's device(s). With their own decryption key, the recipient is now able to read the message.

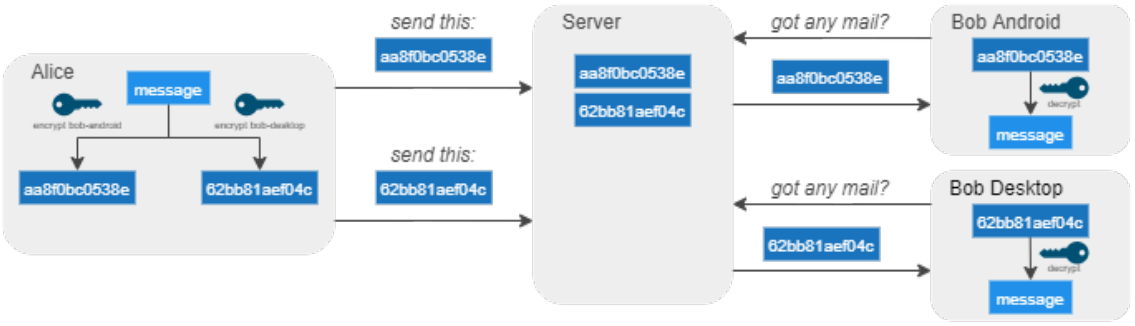


Figure 25.3: Overview of sending a Signal message

25.3.6.3 Call me, maybe?

Signal also enables the user to call other users with audio or video. This uses a peer-to-peer (P2P) system. The request to start a call is sent from the user's device to the server in an encrypted form, whereafter the recipient's phone fetches this request, decrypts it locally and starts ringing. When the recipient accepts the call, an encrypted direct connection between the caller and the recipient is made, over which they communicate.

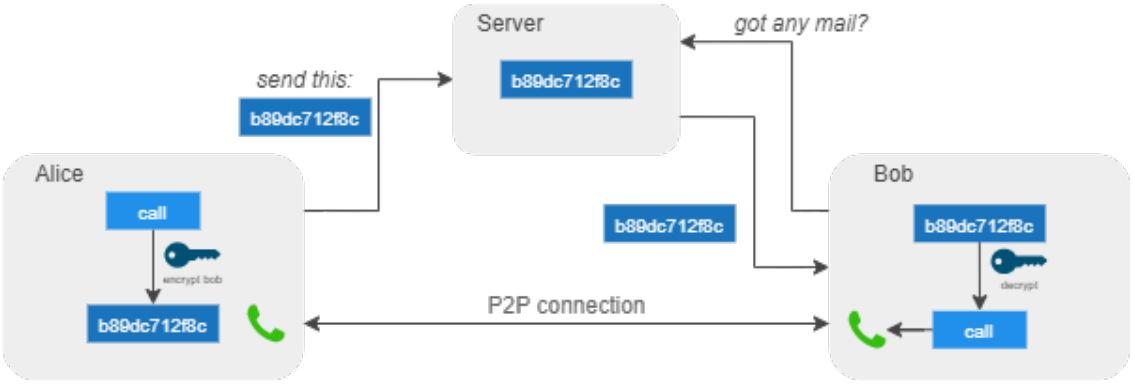


Figure 25.4: Overview of calling another user

It is interesting to see how Signal is constructed. Not only does it have the functionality of a messaging application, but it also provides additional features to ensure user privacy and security. Its use of an MVC ensures Signal can decide to broaden its horizons, and develop new application as new platforms emerge. What's more, if new security standards are adopted, Signal can update its library without affecting the app, to ensure privacy and security in the future.

25.4 Reducing noise to improve Signal quality

In this post we will discuss how software quality is maintained and guaranteed in Signal. We'll go over the use of Continuous Integration, analyse test coverage and explore changes that would improve Signal's code quality. We conclude by estimating, based on the above, the technical debt in Signal Android.

Since we are unaware of any interesting changes happening in the near future, we will not go further into Signal's roadmap.

25.4.1 Ensuring software quality in Signal

The software quality of Signal is a collective effort of both the contributors and the Signal team. Contributors are asked to read and follow the *contributor guidelines*. They describe a number of do's and dont's about creating issues and new features. Furthermore, they recommend those who want to contribute to make sure that their contributions are in line with the *code style guidelines*. Those guidelines can be summarized in three rules. Firstly, methods should be short and self-explanatory, thus not needing comments within them. Secondly, imports should be fully quantified and in a predefined order. Lastly, the naming of variables and fields should follow the conventions as laid out in the guidelines.

To further increase the quality of the code that composes Signal, the built-in Android linter is being used, with a number of rules of which the severity has been modified.

Lastly, Signal deploys both unit tests and UI tests to verify the functionality of the app. Besides the app, a separate set of unit tests make sure that the signal library works as intended.

25.4.2 Ensuring quality, continuously

All the components of the software quality process as discussed before come together in the Continuous Integration (CI) process of Signal. To understand the CI process, we first have to understand how Signal is built.

The build process of Signal is using the Gradle build tool and allows to create different variants of the app. The Gradle configuration file for Signal decomposes the build variant into two components: the build type and the build flavor. A build variant is the combination of a build type and a build flavor. A short summary of the different build types and flavors is given below.

Build types

- *Debug* Shrinks the size of the produced Android package (APK) using Proguard. This type also configures which Proguard rules for optimization are to be used.
- *Staging* Extends the Debug type, but additionally configures values for using the remote staging environment such as the URL of the staging content delivery network.
- *Flipper* Extends the Debug type, but disables the shrinking by Proguard in order to allow the usage of the Flipper debugging tool.
- *Release* Does not extend the Debug type, but does use the Proguard rules that have been configured in it and explicitly enables shrinking.

Build flavors

- *Play* This flavor builds a package that can be offered through the Google Playstore.

- *Website* This flavor builds a package that can be offered through a website. To achieve this, it configures a URL from which the app can fetch its own updates. Although this functionality has been added in 2017[24], there is no sign that this configuration is used at the time of writing.

The CI configuration for Signal is triggered on pushes and pull-requests and consists of two checks. The first check makes sure that all contributors have signed the Contributor License Agreement.

The second check first performs a minor setup of the environment and then executes a single Gradle task to perform quality assurance, which has been visualized below. All used icons are obtained from Pixabay[25]. The quality assurance consists of four separate checks that all have to pass in order for the Gradle task to succeed. The four checks have been summarized below.

Steps in the CI process

- `Signal-Android:testPlayReleaseUnitTest` This step runs the unit tests of the Android app on the *play* flavor of the *release* build type.
- `Signal-Android:lintPlayRelease` Using the same build variant as the previous step, this step runs the built-in Android linter to enforce code consistency.
- `libsignal-service:test` Executes the tests of the Signal service library used for secure communication. This step does not involve a specific build variant.
- `Signal-Android:assemblePlayDebug` Assembles the APK for the *play* flavor of the *debug* build type.

25.4.3 Uncovering the coverage

The Signal-Android project only uses JUnit tests to verify functionality, no integration or device tests are used. As seen in the figure below, the overall coverage of the project is rather low.

The Signal library that implements the Signal protocol also has a total of 6% Class coverage and 5% Method coverage, but important to note is that the crypto package, which arguably is the most important, has the highest coverage (46% Class, 48% Method).

Interesting here is that although the Signal protocol is proven to be cryptographically sound[26], the given implementation has no such guarantees. However, since many institutions have performed independent audits of the implementation[27][28] and all recommended using Signal, the lack of coverage does not imply the software is not secure.

The development cycle that makes use of initial beta releases in order to test new features allows for catching mostly UI bugs and improving usability. Being reluctant with changes to the workings of the protocol, whilst focusing more on new features and bug fixing and relying on the open source verification model keeps the application secure. However the lack of testing in general can be considered to be one of Signal's main sources of technical debt.

[24] Moxie Marlinspike. Support for website distribution build with auto-updating APK. published on 26-02-2017. retrieved from https://github.com/signalapp/Signal-Android/commit/9b8719e2d56a098502475bb5b2295c7a376d4caa. retrieved on 24-03-2020.

[25] All used icons are issued under the Pixabay license. Special thanks to the creators of the broom icon, check mark icon, contract icon, hammer icon, circle procces icon and the checklist icon

[26] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. published in July 2019. retrieved from https://eprint.iacr.org/2016/1013.pdf. retrieved on 26-03-2020.

[27] Laurens Cerulus. EU Commission to staff: Switch to Signal messaging app. published on 20-02-2020. retrieved from https://www.politico.eu/article/eu-commission-to-staff-switch-to-signal-messaging-app/. retrieved on 26-03-2020.

[28] Zack Whittaker. In encryption push, Senate staff can now use Signal for secure messaging. published on 16-05-2017. retrieved from https://www.zdnet.com/article/in-encryption-push-senate-approves-signal-for-encrypted-messaging/. retrieved on 26-03-2020.

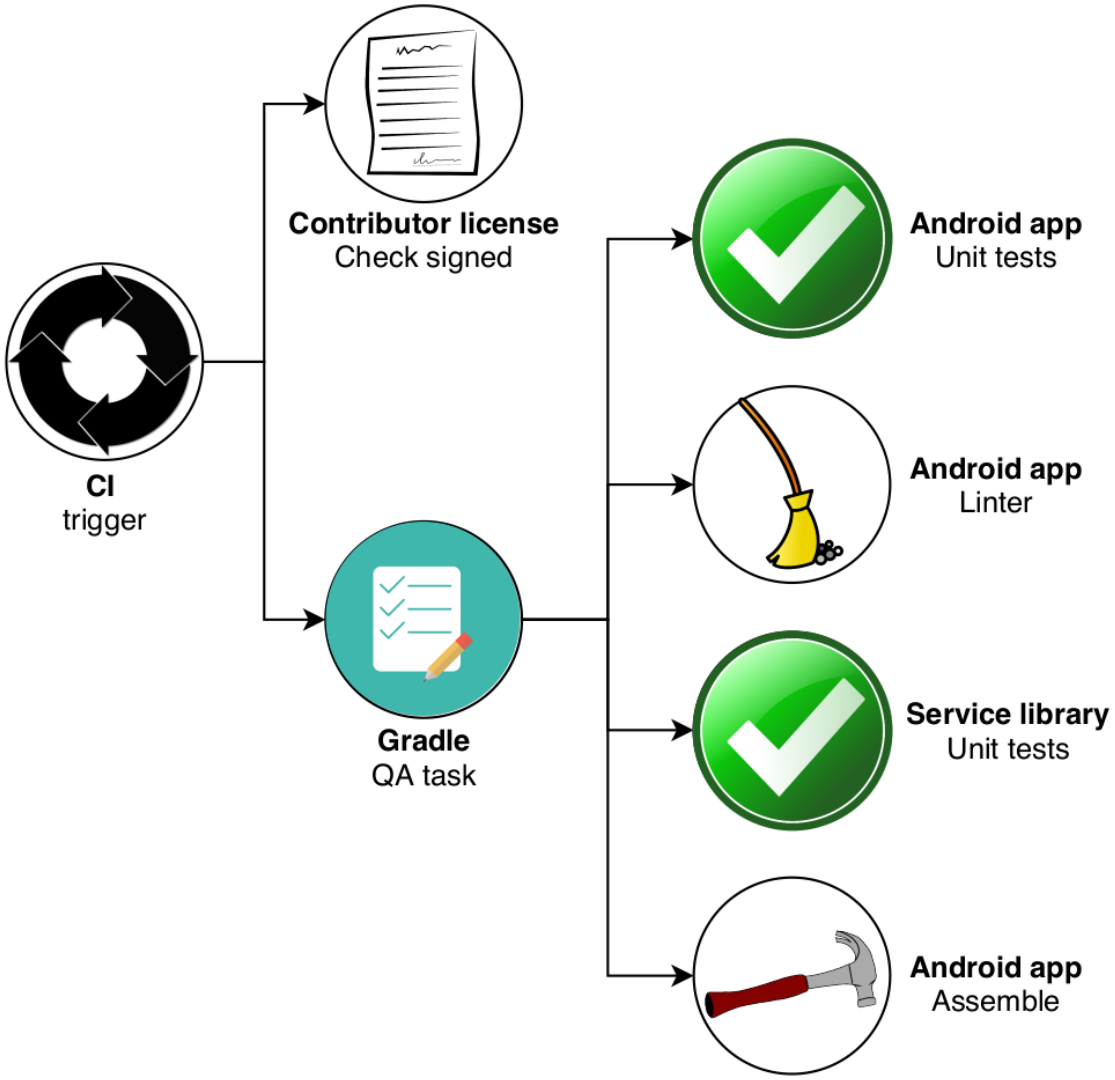


Figure 25.5: Visualization of Signal's CI process

Element	Class, %	Method, %	Line, %
conscrypt	100% (0/0)	100% (0/0)	100% (0/0)
thoughtcrime	7% (184/2533)	3% (503/15534)	2% (1737/67072)
whispersystems	4% (11/267)	4% (62/1394)	3% (206/6042)
signal	0% (0/11)	0% (0/22)	0% (0/37)
apache			
greenrobot			
json			
jsoup			
threeten			
w3c			
webrtc			
xml			
xmlpull			

Figure 25.6: Signal-Android test coverage

Element	Class, %	Method, %	Line, %
AttachmentCipherInputStream	100% (1/1)	81% (9/11)	66% (83/124)
AttachmentCipherInputStream	100% (1/1)	81% (9/11)	66% (83/124)
AttachmentCipherOutputStream	100% (1/1)	75% (6/8)	72% (26/36)
AttachmentCipherOutputStream	100% (1/1)	75% (6/8)	72% (26/36)
DigestingOutputStream	100% (1/1)	85% (6/7)	76% (16/21)
DigestingOutputStream	100% (1/1)	85% (6/7)	76% (16/21)
ProfileCipher	100% (1/1)	75% (3/4)	60% (24/40)
ProfileCipher	100% (1/1)	75% (3/4)	60% (24/40)
ProfileCipherInputStream	100% (1/1)	75% (3/4)	62% (18/29)
ProfileCipherInputStream	100% (1/1)	75% (3/4)	62% (18/29)
ProfileCipherOutputStream	100% (1/1)	71% (5/7)	62% (20/32)
ProfileCipherOutputStream	100% (1/1)	71% (5/7)	62% (20/32)
UnidentifiedAccess	100% (1/1)	25% (1/4)	42% (6/14)
UnidentifiedAccess	100% (1/1)	25% (1/4)	42% (6/14)
SignalServiceCipher	0% (0/3)	0% (0/10)	0% (0/101)

Figure 25.7: Crypto package test coverage

25.4.4 Hot-spots in the codebase

To find out which components are most commonly updated, we used the `git effort` script, which is part of the git-extras set of tools. It iterates over commits and counts how many changes there have been in each file of the repository. After filtering away build files, configuration files, resources and the Signal library, which is a different repository, we end up with the following list.

Filename	# of commits
/util/FeatureFlags.java	22
/conversation/ConversationFragment.java	17
/conversation/ConversationActivity.java	17
/registration/service/CodeVerificationRequest.java	15
/lock/RegistrationLockDialog.java	14
/database/helpers/SQLCipherOpenHelper.java	14
/ApplicationContext.java	14
/util/TextSecurePreferences.java	13
/megaphone/Megaphones.java	12
/database/RecipientDatabase.java	10
/jobs/JobManagerFactories.java	11
/profiles/edit/EditProfileFragment.java	10
/migrations/RegistrationPinV2MigrationJob.java	10
/registration/fragments/RegistrationLockFragment.java	9
/registration/fragments/RegistrationCompleteFragment.java	9
/recipients/Recipient.java	9
/mediasend/MediaSendViewModel.java	9
/lock/v2/ConfirmKbsPinFragment.java	9
/jobs/PushProcessMessageJob.java	9

From this list, it is clear that a few components within the application can be regarded as so-called hotspots: The `ConversationActivity.java` and `ConversationFragment.java` have been altered relatively frequently. This makes sense, as adding new functionality to conversations in Signal might require some refactoring in how the conversation is structured. Secondly, files related to the registration and PIN lock functionality have also been subject to some changes. `RegistrationLockDialog.java`, `RegistrationPinV2MigrationJob.java` and `CodeVerificationRequest.java` are the main files affected. Since Signal has recently improved its registration lock functionality[29], it is logical for these files to have been altered more frequently. At the very top of the list is the `FeatureFlags.java` file. These flags are used to disable functionality that is not entirely complete yet. Overall, it is clear that most code is written once, and not frequently edited thereafter.

25.4.5 What's wrong, and how to fix it

The Software Improvement Group (SIG) has analysed Signal's codebase, distinguishing code quality in 9 different categories: volume, duplication, unit size, unit complexity, unit interfacing, module coupling, component balance, component independence, and component entanglement. Out of those aspects, we will

[29] Jim O'Leary. Improving Registration Lock with Secure Value Recovery. published on 27-01-2020. retrieved from https://signal.org/blog/improving-registration-lock/. retrieved on 25-03-2020.

neither discuss volume, as it is hard to give refactoring recommendations, nor component balance as this is the area in which Signal achieves the highest score. We will address each of the other categories from the category with the highest score down to the lowest. We will describe what each measure means, along with examples of "bad practices" in Signal and a recommendation on how to fix these where applicable.

- *Duplication* Code duplication means that the same exact code is used at 2 or more places in the codebase which means the code becomes harder to maintain. The best refactoring candidate in Signal is an entire java file `Hex.java` which is present in both `libsignal/service` and the `util` folder of Signal. To refactor this, Signal should remove one of the instances of `Hex.java` and redirect all usages to the other instance.
- *Unit Size and Unit Complexity* Unit size describes the size of methods, whenever this grows too large the methods will most likely have more than one responsibility. Unit complexity is when the McCabe complexity[30] of a method becomes too large which makes it hard to understand and test. Although there are several methods which have 100+ lines, or a McCabe complexity of 50+ in Signal, 2 methods jump out, namely `ClassicOpenHelper.onUpgrade` (667 lines, McCabe of 162) and `SQLCipherOpenHelper.onUpgrade` (448 lines, McCabe of 83). As the names would suggest, these methods share similar functionality and implementation. Both methods contain large `if` blocks which would be good candidate places to use helper functions. This would reduce both unit size and unit complexity.
- *Unit Interfacing* Unit Interfacing represents the number of parameters of a method. One of the methods which is indicated to have bad unit interfacing is `PushServiceSocket.uploadToCdn`. We could try to reduce the number of inputs for this method (currently 14!), by for example using a data object to encapsulate some of the used fields. However, if we look at the class as a whole, a bigger problem might be the root cause. The class is almost 1500 lines long and has almost 50 fields. The best approach here would likely be to identify different functionality within the class and split up the class accordingly.
- *Module Coupling* Module coupling relates to how classes interact with each other. A high coupling often indicates that a class has several responsibilities. Even though a lot of module coupling is present in the Signal Android application, upon closer inspection, the majority of it is only logical, by the way the application is and should be structured. For example, the `Recipient.java` file contains a lot of module coupling, but as the recipient is directly related to a contact, a job and the corresponding notification, these couplings are not redundant.
- *Component Independence and Entanglement* A high component independence means that a single component affects many other components which leads to the code being more difficult to reason about. Component entanglement is somewhat similar in that it concerns *how* components affect each other. For both of these aspects it is important to keep in mind that SIG used the componentization we created and thus not the components defined by Signal itself. The score of both is very low due to the same categories of classes, namely communication and database classes. These classes are both highly entangled and have many dependencies. This is however to be expected as they are key components which are necessary for many operations.

In the end, we can conclude that while Signal has a quite extensive continuous integration system, the tests present leave a large chunk of the code base uncovered. While adding a feature may be relatively straightforward, one may not be sure all existing functionality still works properly. There are a few hot-spots in the code base, where features had to be reworked entirely: This hints at the presence of some technical debt in the 'accidental prudent' quadrant [31]. Although there are still improvements to be made, Signal's

[30] T. J. McCabe. A Complexity Measure. published in December 1976. retrieved from https://ieeexplore.ieee.org/document/1702388. retrieved on 26-03-2020.

[31] Martin Fowler. Technical Debt Quadrants. Published on 14-10-2009. retrieved from https://martinfowler.com/bliki/TechnicalDebt

code quality as a whole is adequate.

25.5 Smoke Signals or secure crypto?

Since Signal's unique characteristics revolve around security and privacy, we decided to explore these features more in depth for our fourth and final essay. More specifically, we will examine how these characteristics affect various layers of Signal's architecture. We look into how Signal manages sessions between users and at how Signal encrypts and decrypts incoming messages from a component perspective.

25.5.1 Liberté, egalité, sécurité

25.5.1.1 Open, Sesame

In a previous essay, we mentioned that Signal uses session management. This section takes a deeper dive into the workings of this algorithm, called Sesame, in relation to the different layers in the architecture of Signal.

Firstly we will briefly go over what sessions are and how they are implemented. A session is some secret data stored by a device of a user[32]. Each session has a *UserID* and a *DeviceID* which means that there exists a session per conversation, per device. So if Alice talks to Bob and Charlie, but Bob has two devices on which he uses Signal, Alice has three sessions. A session contains the encryption and decryption keys used for sending and receiving messages. Messages can only successfully be decrypted using a matching session. Signal's implementation employs the X3DH[33] key agreement protocol to negotiate key pairs between both devices, these respective session key pairs are stored in the application database. Sesame, however, supports the use of other protocols too, given those meet certain conditions. Session keys can also be updated as two parties communicate by means of a ratcheting algorithm such as Double Ratchet. Concretely, Signal stores a list of *UserRecord* entries, which are indexed by their UserID. For each UserRecord there is a list of one or more *DeviceRecord* entries, which are indexed by their DeviceID. A device may contain an ordered list of inactive sessions and/or an active session. A DeviceRecord or UserRecord entry may be marked as 'stale' when, for example, a user deletes Signal from one of their devices, or stops using Signal altogether. In this case, the records are not deleted, since there may still be delayed messages that need to be decrypted.

Sending a message Whenever someone sends a message, Sesame is used and is provided with some plain text and a set of recipient UserIDs (the sender of the message is included in this set). The text is then encrypted and the following steps are executed per recipient:

As seen in the figure above, first a check is performed to see if there exists a non-stale UserRecord corresponding to the current UserID. If this is the case, for each DeviceRecord the plain text is encrypted using the corresponding sessions. Should there exist no non-stale UserRecord, the message is not sent. Now the UserID is sent to the server along with a list of encrypted messages and a corresponding list of DeviceIDs. The server first checks if the specified UserID is in-use and if the list of DeviceIDs is up to date. If this check passes, the server sends the messages to the corresponding mailboxes and the process starts over for the next UserID. However if the specified UserID is no longer in use, the server notifies the sender and the sender marks the UserID as stale (the process now starts again for the next UserID). If some of the

Quadrant.html. retrieved on 26-03-2020.

[32] Moxie Marlinspike, Trevor Perrin. The Sesame Algorithm: Session Management for Asynchronous Message Encryption. published on 14-04-2017. retrieved from https://signal.org/docs/specifications/sesame/. retrieved on 09-04-2020.

[33] Moxie Marlinspike, Trevor Perrin. The X3DH Key Agreement Protocol. published on 04-11-2016. retrieved from https://signal.org/docs/specifications/x3dh/. retrieved on 08-04-2020.

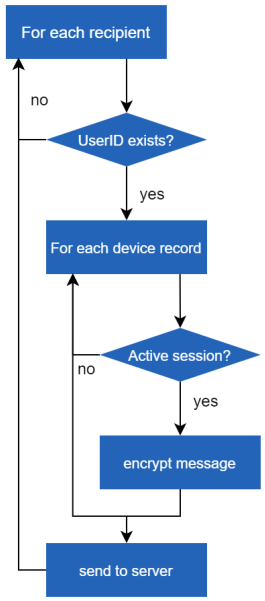


Figure 25.8: The flow of sending a message in the Sesame protocol

DeviceIDs are no longer up to date, the server again notifies the sender and sends a current list of DeviceIDs. The sender marks all old DeviceRecords corresponding to the DeviceIDs as stale and gathers the updated information. The process is now started again for the next UserID.

Mapping to layers As we have now seen how sending a message works in Sesame, we will explore how different layers in the application are used to accomplish sending a message. In the second essay, we decomposed Signal as shown below and concluded that it had a layered architecture.

Whenever a user opens a conversation, the conversation component in the UI module is used to visualize it. The UI component forms the presentation layer. When a message is sent, the UserRecord entries with their corresponding DeviceRecords and sessions are fetched from storage via the database component in the Storage module. This module acts as the persistence layer and database layer in the application. With the sessions that have been retrieved, the message component in the Conversation module, which forms the business layer, handles the message encryption using the Signal library. The encrypted messages for each session can now be sent to the server.

25.5.1.2 What really grinds attackers' gears

This section will provide the reader with a conceptual understanding of the idea behind Signal's Double Ratchet[34] scheme. It is written to be a high-level abstraction. Therefore, some lower-level details have been omitted as they were unnecessary for this concept. We will take the reader along for a ride through the conceptual development of the double ratchet algorithm. For each step in the development, we will point out the vulnerabilities in the algorithm and how they can be prevented. This section assumes familiarity

[34]Trevor Perrin, Moxie Marlinspike. The Double Ratchet Algorithm. published on 20-11-2016. retrieved from https://signal.org/docs/specifications/doubleratchet/. retrieved on 08-04-2020.

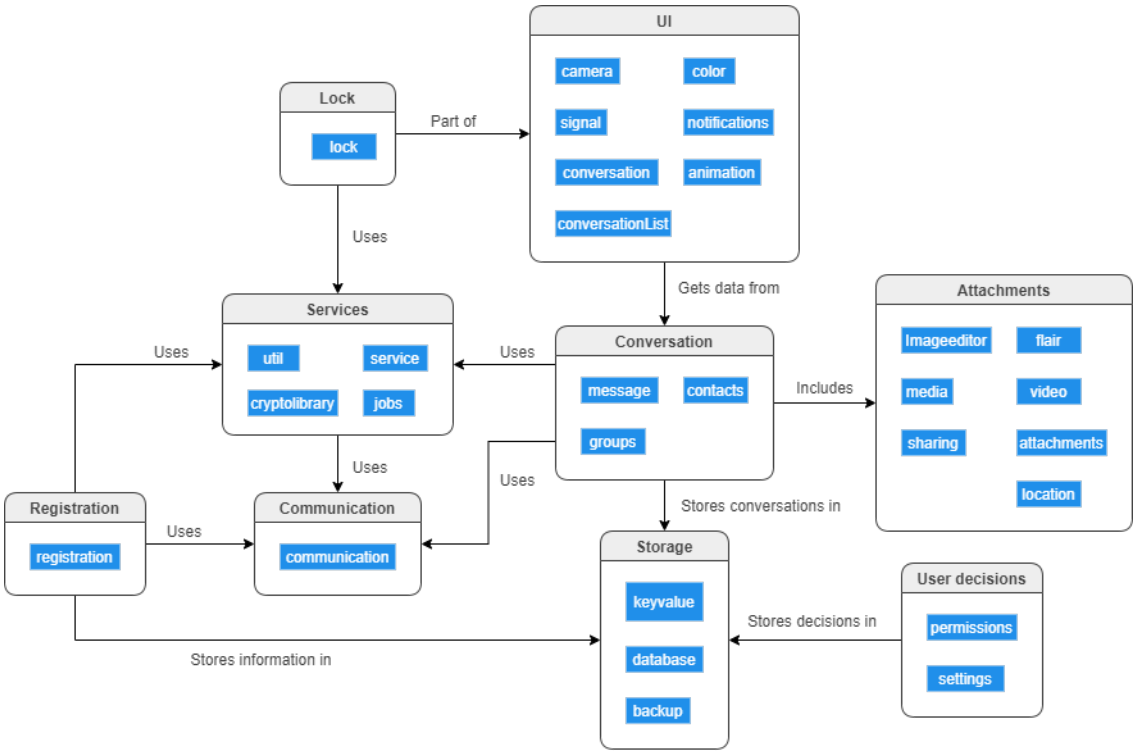


Figure 25.9: High-level decomposition of the Signal app for Android

with symmetric key cryptography[35] and the Diffie-Hellman protocol[36].

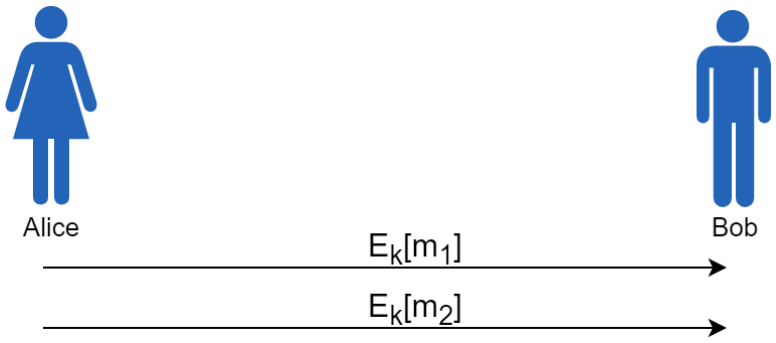


Figure 25.10: Interaction between Alice and Bob when using symmetric key encryption

The figure above shows the interaction between Alice and Bob when Alice sends two messages to Bob using basic symmetric key encryption. We let Ek[m1] denote the encryption of message 1 with the symmetric key k, which Alice and Bob share. If at any point in time an attacker obtains the key, he can read all messages that Alice has ever sent to Bob.

To prevent such an attack, the ratchet scheme below uses Key Derivation Functions (KDFs), which are one-way functions. In short, this means that given an input, the function always returns the same output, but that it is not possible to go back from the output to the input that produced it. Alice starts with an initial chain key. She inputs this key to the KDF, which then produces a new chain key and the encryption key for the first message, Km1. She then encrypts the first message using this key and sends it to Bob.

To send the second message, she uses the chain key that the KDF generated and uses it to create a new chain key and the message key for the second message. If an attacker obtains a single message key Kmx (for any message x), only this message is revealed. However, if an attacker manages to obtain a chain key, he can read all messages that Alice sent after the discovered chain key was used. Due to the one-way nature of the KDF, he can not read messages that have been sent before the chain key was obtained. Bob initializes his own ratchet with the same initial chain key, and can thus obtain the created symmetrical keys to decrypt the messages Alice sends him.

In order to prevent an attacker from reading all subsequent messages after a chain key has been compromised, the scheme introduces the Diffie-Hellman (DH) ratchet. This is another ratchet that uses the DH protocol to share secrets between Alice and Bob. Explaining this protocol is out of scope for this post, as the inner workings of this ratchet are not relevant to the conceptual working of the scheme as depicted below. This scheme is nearly the same as the one we saw before, except that the DH procedure is now used to regularly reset the chain keys. This makes sure that, when an attacker obtains a chain key, he can only read all messages that have been sent until the chain key was reset. As this happens regularly, the amount of messages that an attacker can read is minimal.

Now that we have seen how Signal's encryption protocol works, we will tie it in to the architecture of the app. We will refer back to the decomposition that was derived in the second essay and that has been

[35] IBM Knowledge Center. Symmetric cryptography. retrieved from https://www.ibm.com/support/knowledgecenter/SSB23S_1.1.0.2020/gtps7/s7symm.html. retrieved on 09-04-2020.

[36] David Terr. Diffie-Hellman Protocol.. retrieved from https://mathworld.wolfram.com/Diffie-HellmanProtocol.html. retrieved on 09-04-2020.

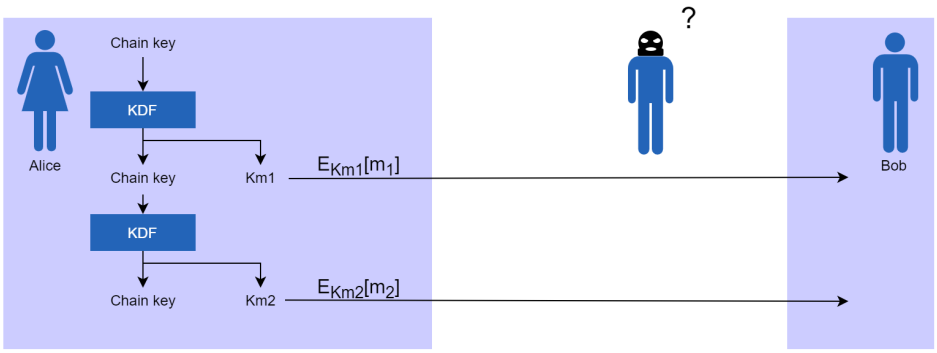


Figure 25.11: Interaction between Alice and Bob when using a single ratchet

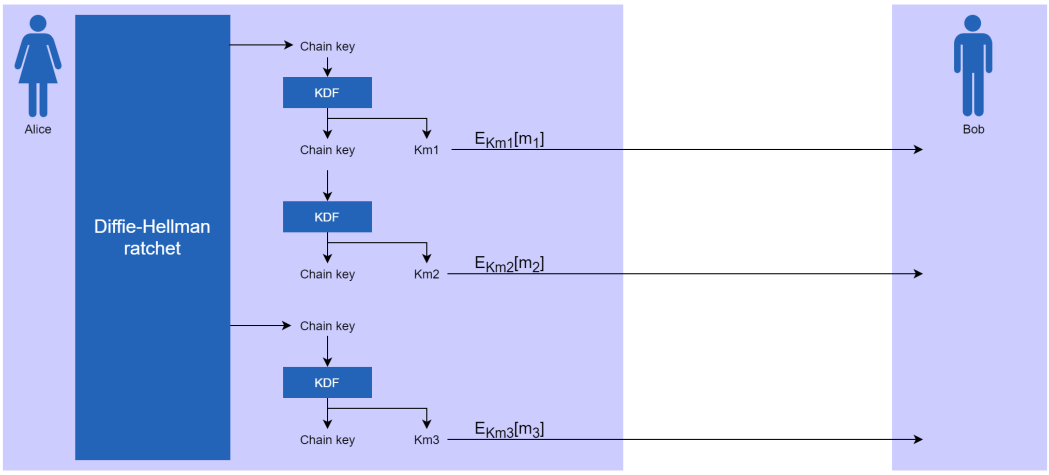


Figure 25.12: Interaction between Alice and Bob when using a double ratchet

mentioned before.

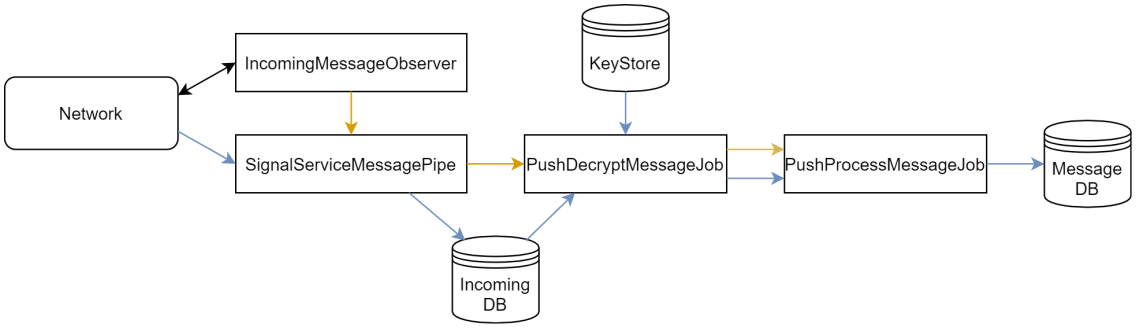


Figure 25.13: The handling of an incoming message by the Signal application. The blue arrows indicate traversal of data, the orange arrows indicate traversal of jobs.

When a new message arrives, it is observed by the `IncomingMessageObserver.java`, which is included in the Services module. The `SignalServiceMessagePipe.java` class from the library is then used to read the data from the network and create an envelope around it. Next, the `IncomingMessageProcessor.java` stores the envelope temporarily in a database and registers a `PushDecryptMessageJob`. This job decrypts the envelope using the library and registers a `PushProcessMessageJob` to process the contents of the message. When this job is executed, it creates an `IncomingTextMessage` (which is included in the communication module) object, and writes it to the database. All classes that have `Job` in their title are executed asynchronously and are part of the Services component.

When we consider the Services module to also be part of the business layer, we can see that the encrypted messages are decrypted in this layer. They are then passed to the persistence layer in the form of an `IncomingTextMessage` object, which is a data access object. Finally, they are stored in the database, which belongs to the equally named layer. Opening a chat in the application will pull the messages from the corresponding database table.

25.5.2 Practically Preserving Privacy

The aforementioned protocols and algorithms ensure privacy across Signal's application. As all messages are end-to-end encrypted using the Sesame sessions, a man-in-the-middle attack or a compromised server does not expose the messages' contents to an adversary. Moreover, since Signal uses the sealed sender, meaning the sender's metadata is also encrypted. While the above properties are mainly active in the business layer of Signal, some properties that ensure privacy are also active in the presentation. Signal allows users to verify their safety number with another user from the conversation, to ensure that the session has not been compromised. Alternatively, a QR code can be scanned to verify this number.

By making privacy options visible and available to the user, Signal strives to demonstrate to the user their focus on privacy. However, no system is perfect, and despite their best effort, it is difficult to provide Signal's services without storing any data on the user. For example, since the sender sends their encrypted message to the Signal server, this server 'knows' the IP address from which the message was sent. In practice, this can be circumvented by the use of a Virtual Private Network (VPN).

In conclusion, Signal employs different protocols across their layered architecture to ensure security and privacy for the end user. The protocols and algorithms have been shown to be highly secure, but are not

perfect. The Signal team continues to work to keep their services secure and to ensure privacy to the end user, in which their main effort is having to store less user data on their servers. We are thrilled to see Signal improve further, in a future where privacy becomes even more central in our society.

Chapter 26

Solidity



"Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state."

26.1 What is Solidity?

Solidity is a contract-oriented programming language for implementing smart contracts designed to run on the EVM (Ethereum Virtual Machine). Such contracts are published on the Ethereum blockchain and Solidity is one of the two main languages meant for this, the other being Serpent. Solidity is Turing complete and compiled to Etherium Virtual Machine (EVM) bytecode, which is executed on the EVM on Ether mining machines. The language is similar in syntax to JavaScript. Solidity is poised to produce a disruptive change in the world of business digitalization, by decentralizing and automating contracts.

26.1.1 What is a smart contract and why are smart contracts interesting?

Smart contracts are build with blockchain technology. The first thing people associate with blockchain are crypto currencies, the hype that surrounded it left many people with a negative view on crypto currencies. However, the underlying blockchain technologies have great potential. Blockchain allows the storage of data in a way that makes the data accessible for many people but without the need for a middle man to ensure security. This feature is of great use in the construction of smart contracts.

A smart contract is "a set of programmable instructions, intended to automatically facilitate, enforce and verify the execution of an agreement". The contracts are inspired by regular legal contracts between multiple parties. The difference with conventional contracts is that they are enforced automatically in a decentralized manner, as opposed to being enforced manually by a central authority. As such, as long as all the inputs and outputs of a contract are digital, a smart contract is fully automated and self-regulated.

The contracts can be used to record agreements and ensure the execution of such an agreement. One can imagine that this will be useful in many industries, ranging from mortgages to voting.

26.1.2 Why is Solidity interesting and what does Solidity offer?

Today, there are four major platforms for smart contracts, namely:

- RSK
- Ethereum
- EOS
- Cardano

When thinking about whether Solidity is the right choice for you, you may ask yourself two questions. First, why Ethereum? Second, why Solidity?

- **Why choose Ethereum?**

  Ethereum was released in 2013 by Russian-Canadian programmer Vitalik Buterin. He envisioned it as a decentralized global supercomputer, allowing developers to pay "gas" to run their decentralized applications on it, known as dApps. It was incorporated as Ethereum Switzerland GmbH in 2014 and raised $18.4 million during its ICO.

  - Ethereum plans to move away from Proof-of-work (POW) into Proof-of-stake (POS)

  Proof of work, also known as Nakamoto Consensus, is the same consensus protocol used by Bitcoin. This comes with the same problems that Bitcoin has, notably considerable operating costs. Notably, the Bitcoin network is due to this very reason prone to reduced scalability, throughput and large power costs (in 2018, it used as much power as the entire country of Hungary!).

  By replacing it with proof of stake, Ethereum estimates a reduction of over 99% in energy consumption.

- **Why choose Solidity?**

  Solidity is one of the first programming languages with support to compile to bytecode that can be run on the Ethereum Virtual Machine. As a result, it has more tools and has grown in popularity compared to other existing languages for the same purpose. The strong developer community is a good reason to choose for Solidity. There is a huge amount of online and offline documentation and help available for anyone who wishes to explore, learn and contribute to Solidity - making it ideal for a project such as ours.

The main perk of using Solidity as compared to another blockchain technologies is its ease of application to smart contracts. It is easy to learn as it shares a large number of programming constructs and perceptions that exist in other programming languages variables such as string manipulation, classes, functions, arithmetic operations, and so on. Solidity code looks surprisingly similar to C++, C#, or JavaScript. It can also be uses to write smart contracts for the RSK (Rootstock) blockchain, which is the most secure platform for smart contracts.

26.2 Meet the Team

We are a diverse team with multiple nationalities, all following a master at the TU Delft. Our interest in blockchain technologies and smart contracts unites us into a solid (pun intended) team.

Our team members:

- **David Cian**

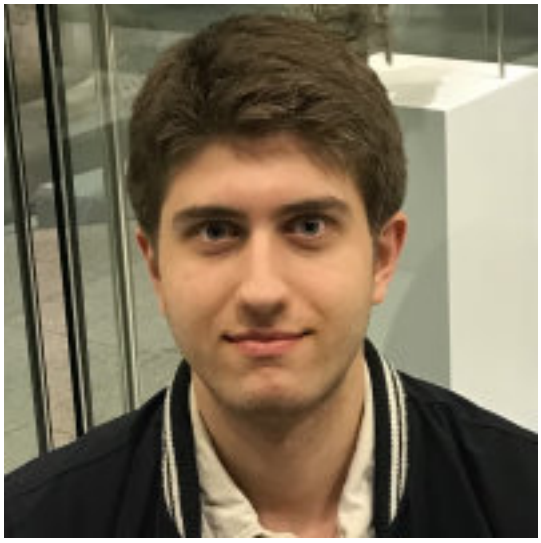


Figure 26.1: Portrait picture of David

David is a CS undergrad on exchange at TU Delft, coming from EPFL. He loves learning while keeping as broad a horizon as possible. He is excited about the novel applications of blockchain technology besides cryptocurrency and the opportunities it offers to build a better world. You can find more info about him on his personal website.

- **Max Groenenboom**

  Max is an Embedded Systems master student with a background in Computer Science. He wants to broaden his view and knowledge of the world, and hasn't worked with blockchain yet. This project would be a great chance to make a first step into blockchain technology, considering Etherium is a very novel and new way of using the blockchain.

- **Luc Lenferink**

Luc is a Computer Science master student interested in sequential prediction. He likes to be in touch with new trends and believes that blockchain and smart contracts have a future in multiple industries.

26.3 Solidity: The Product Vision

This is the first in a series of 4 essays that analyse the architecture of Solidity. We start off with a description of the vision underlying Solidity and a peek into the future of Solidity.

26.3.1 Aim of the project

Solidity is closely tied to Ethereum, so too does Solidity's vision build on Ethereum's.

26.3.1.1 Ethereum's vision

As a blockchain technology, Ethereum is a platform with a couple of remarkable properties. In a nutshell, it is one implementation of a distributed ledger, for all practical purposes immutable. It also has a diverse community of both users and contributors, as opposed to being either targeted at one enterprise user or having its development steered by one product owner.

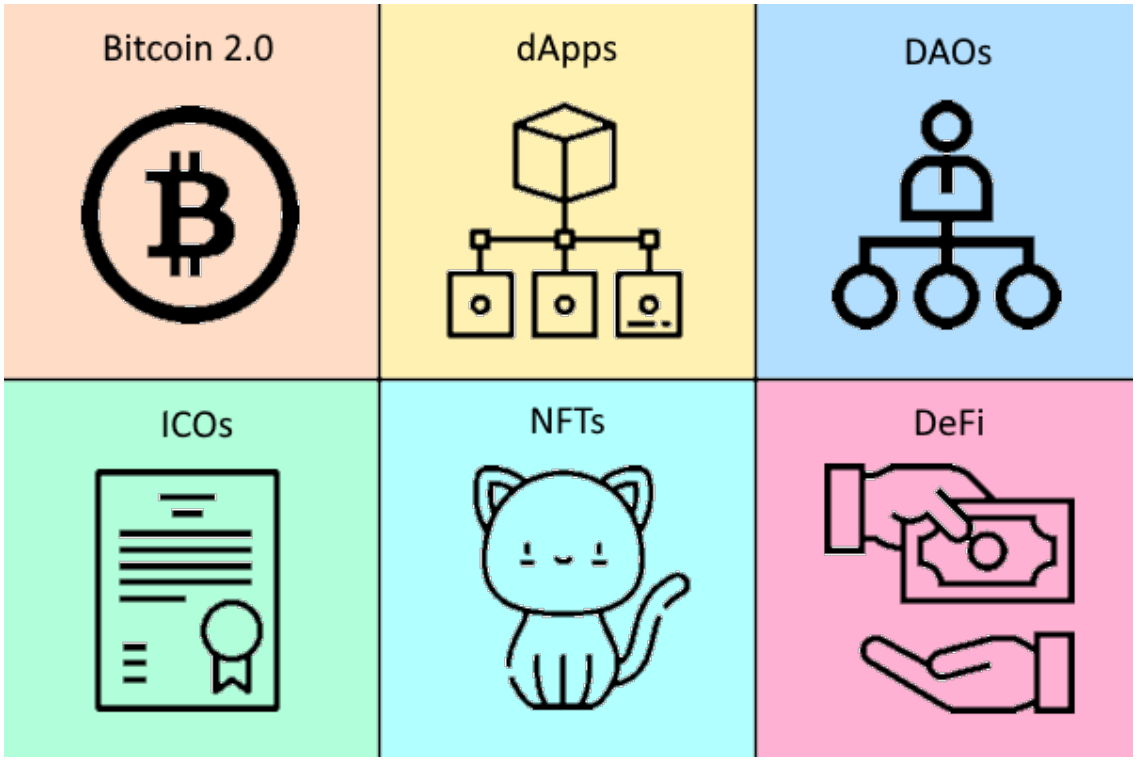


Figure 26.2: The aims of Ethereum

These two reasons explain the variety of visions assigned to it over time, which you can check out below[1]:

- Vitalik Buterin, who published the original Ethereum whitepaper, was a founding editor of Bitcoin magazine and very engaged with its development. When VCs started rushing in to fund Bitcoin 2.0, he was frustrated with the isolation of the new projects branching from Bitcoin, akin to having a *"separate hardware module for solitaire, [...] for Internet Explorer, [...] for World of Warcraft"*. This led to him developing Ethereum as a platform for wildly different applications, an approach reflect in the name *Ethereum*, stemming from the concept of Aether, an imperceptible material supposedly filling the universe.
- This vision was further crystallized with the advent of dApps (distributed apps), presenting the Ethereum network as a "world computer" running on gas (users' computing power).
- Ethereum is also seen as a platform for DAOs (Decentralized Autonomous Organization), which as the name suggests mean to be analogous to real-world organizations, except ownerless.
- Smart contracts are seen as particularly fit for ICOs (Initial Coin Offering), a usage both popular and involving large monetary amounts.
- One of the most visible Ethereum projects ever was CryptoKitties, where the kitties were actually assets stored on the Ethereum blockchain. NFTs (Non Fungible Token) have sizable mind-share with Ethereum enthusiasts.
- The decentralized aspect fed into the narrative of DeFi (Decentralized Finance). A usage this time with the potential of rendering the global economy more efficient, fairer and more stable, in the wake of the 2008 financial crisis it has found wide acclaim as an alternative to central mismanagement of the global economy.

A remarkable feature of Ethereum is its potential to disrupt not only technologically but politically as well.

26.3.1.2 Solidity's vision

Solidity's goal is to promote smart contracts. To do so, it aims to[2]:

1. Let everyone write smart contracts
2. Let them write secure contracts

26.3.2 End user mental model

Solidity is a programming language for constructing smart contracts. Its end users are programmers who want to implement smart contracts on the Ethereum blockchain. As Solidity is a programming language, all requirements and expectations that would be in place for regular programming languages are also the case for Solidity. The way Ethereum handles smart contracts however introduces additional requirements not seen in regular programming languages.

26.3.2.1 Code execution costs

Executing code on the blockchain is not free: it costs ETH in the form of gas. Some instructions are a lot more expensive than other ones. At the same time, some instructions are a lot cheaper than one would

[1] Pereira, F. (2018, November 20). Visions of Ether. Retrieved March 8, 2020, from https://tokeneconomy.co/visions-of-ether-590858bf848e

[2] Ethereum, Ethereum DevCon-0: Solidity, Vision and Roadmap. (2015, January 6). Retrieved March 8, 2020, from https://www.youtube.com/watch?v=DIqGDNPO5YM

expect. For example multiplication operations have the same gas cost as division operations, whereas in most languages division is a lot more expensive than multiplication.

Because of these not always intuitive gas costs, one might want the language's compiler to optimize not only in execution speeds, but also on gas costs. Maybe even on gas costs alone as the smart contract's programmer does only care about their executing costs, and not the speed. Next to these alternative optimizing goals, Ethereum wants to know the maximum gas a program can use. As these exact gas costs are only known at runtime, an estimation system would be a good component to have.

Solidity contains solutions to both problems mentioned above. Its built-in optimizer performs both usual optimizations but also performs gas-based optimizations. Next to this a system is in place for the end user to calculate the maximum gas costs of their contract.

26.3.3 Key capabilities and properties

A few key capabilities and properties of Solidity can be defined:

- It supports a versioning system where the version in which a contract was written is explicitly mentioned in the contract. Furthermore, tools are supplied with which source code can be transformed to newer versions.
- Solidity is object-oriented and high-level. It supports multiple inheritance and complex type definitions.

26.3.4 Stakeholder analysis

The stakeholders of a software project can be divided into five groups[3]: end users, the business, customers, domain experts, and developers. Below we will analyse the different groups and their stakes in the project. To keep the analysis concise, we will leave out the business and the domain experts as Solidity is open source and the domain experts are in this case developers.

Stakeholder	Stakes
End users	Programming languages are used solely by programmers, as using a programming language makes you a programmer. Therefore, the group of end users of Solidity is easily identifiable: every programmer that uses Solidity to implement smart contracts. The end users benefit from the features that Solidity offers and that allow them to implement their projects
Customers	The customers of Solidity are essentially its users in a broader sense, ranging from a programmer that uses the language to write its own smart contracts (the end users) to companies that sell smart contract implementations or use smart contracts for their operations. So in a lot of cases the customers of the language are the consumers of the service that the language provides. Therefore, the resulting language is either directly beneficial to them or as part of a product.
Developers	The developers of Solidity are the contributors of the project, i.e. the programmers that develop the Solidity language and its documentation.

[3] Coplien, J. O., & Bjørnvig, G. (2011). Lean architecture: for agile software development. John Wiley & Sons.

The groups of end users and developers intersects, as a lot of developers will use the language themselves. They directly benefit from (positive) changes they make.

From the analysis it follows that all groups overlap. It is nevertheless interesting to distinguish the groups as they are not proper subsets of each other.

26.3.5 Context of the system

Solidity was created by Dr. Gavin Wood as a language explicitly for writing smart contracts with features to directly support execution in the decentralized environment of the Ethereum world computer. Solidity is now developed and maintained as an independent project on GitHub. In this section, we look at the current and future context in which the system operates.

The main Entities or Actors in the Solidity system are:

1. **Solidity compiler:** The main product of the Solidity project is the Solidity compiler, solc, which converts programs written in the Solidity language to EVM bytecode. The project also manages the important application binary interface standard for Ethereum smart contracts. Each version of the Solidity compiler corresponds to and compiles a specific version of the Solidity language.
2. **Ethereum Virtual Machine:** At the heart of the Ethereum protocol and operation is the Ethereum Virtual Machine, or EVM for short. It is the part of Ethereum that handles smart contract deployment and execution. At a high level, the EVM running on the Ethereum blockchain can be thought of as a global decentralized computer containing millions of executable objects, each with its own permanent data store.

26.3.6 Product roadmap

This section looks at the product roadmap, laying out the main future directions anticipated for the upcoming years.

26.3.6.1 Issues

Solidity currently has 678 open issues with 27 labels varying from translations to difficulty levels to bugs to features. Among these, there are 116 open features reported by various authors.

The current sprint-cycle is 2 weeks. There are 2 current open milestones: 1. Sprint 3: Due March 12, 2020 with 11 issues 2. Sprint 4: Due March 26, 2020

The past releases can be found here

26.3.6.2 Projects

The current bugs and features for the forthcoming releases have been organised into projects.

There are currently 12 open projects:

Project	Description
Natspec	Issues regarding the Natural Specification Format
Wasm/Ewasm	Issues in Ethereum flavored WebAssembly (ewasm)

Project	Description
Reducing Technical Debt	Issues to reduce technical debt in the Solidity code
Sol -> Yul Codegen	Manages tickets related to the effort of compiling Solidity code to Yul code
Emscripten	Issues for Emscripten
Documentation translation	A project to track the progress of Solidity docs translations
Yul	Issues for Yul
Zeppelin Audit	Issues that came up during the audit by Zeppelin
Backlog (breaking)	Breaking changes that the team weakly agreed to implement
Consolidate inheritance rules	Issues to incorporate inheritance rules
SMT Checker MVP	This project is unrelated to release projects and aims to develop the SMT checker towards a minimum viable product.
Backlog (non-breaking)	Non-breaking changes to be made in the code

26.3.6.3 Development

Solidity is a relatively new language with a fast-pace of continuous development.

The guidelines for development are spelled out here. There are public team calls on Monday at 12pm CET and Wednesday at 3pm CET for interested contributors.

26.4 Solidity: From vision to architecture

Previously, we introduced Archie's vision for Solidity. This time, let's dive deeper into Solidity's architecture with our ever-curious software architect apprentice!

26.4.1 Relevant Architectural Views

In 1995, Philippe Kruchten published an article called: "Architectural Blueprints - The"4 + 1" View Model of Software Architecture"[4]. It introduced a model for describing the architecture of a software-intensive system using so-called viewpoints. To get a grasp on Solidity's architecture, we can use an extended version of this model introduced by Nick Rozanski and Eoin Woods [5] [6]:

Viewpoint	Description
Functional	Describes the system's runtime functional elements, their responsibilities, interfaces and primary interactions.

[4] Kruchten, P. (1995). Architectural Blueprints - The "4 + 1" View Model of Software Architecture.
[5] Rozanski, N., & Woods, E. (2005). Applying viewpoints and views to software architecture. Open University White Paper.
[6] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

Viewpoint	Description
Informational	Describes the way that the architecture stores, manipulates, manages, and distributes information (including content, structure, ownership, latency, references, and data migration).
Concurrency	Describes the concurrency structure of the system, and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled.
Development	Describes the constraints that the architecture places on the software development process.
Deployment	Describes the environment into which the system will be deployed, capturing the hardware environment, the technical environment requirements for each element and the mapping of the software elements to the runtime environment that will execute them.
Operational	Describes how the system will be operated, administered and supported when it is running in its production environment. For all but the smallest simplest systems, installing, managing and operating the system is a significant task that must be considered and planned at design time.

Are all of these viewpoints relevant? As it turns out, not exactly. As a programming language, certain viewpoints are more applicable to Solidity than others. Based on the industrial experience report of Woods[7] we can select three of them:

Viewpoint	Consideration
Functional	A compiler is essentially a multi-step translator, where each step has a well-defined responsibility
Development	As a multi-stage process, dependencies between a compiler's components often constrain the development process.
Operational	The Solidity compiler is not, as we'll see, merely a one-trick pony; it is capable of much more than rote code generation

We will lay out the Development view and the Functional view, but we can't ignore the elephant in the room forever: Solidity's architectural style and patterns.

[7] Woods, E. (2004, May). Experiences using viewpoints for information systems architecture: An industrial experience report. In European Workshop on Software Architecture (pp. 182-193). Springer, Berlin, Heidelberg.



Figure 26.3: Image of Archie



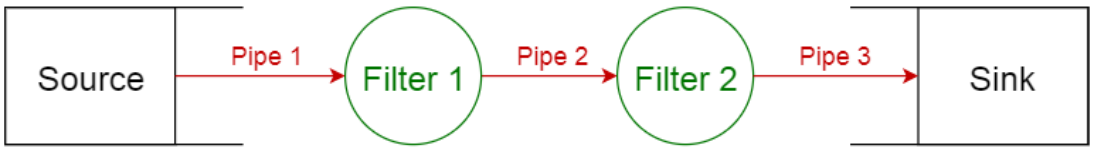
Figure 26.4: Image of Archie



Figure 26.5: Image of Archie

26.4.2 Architectural style and patterns

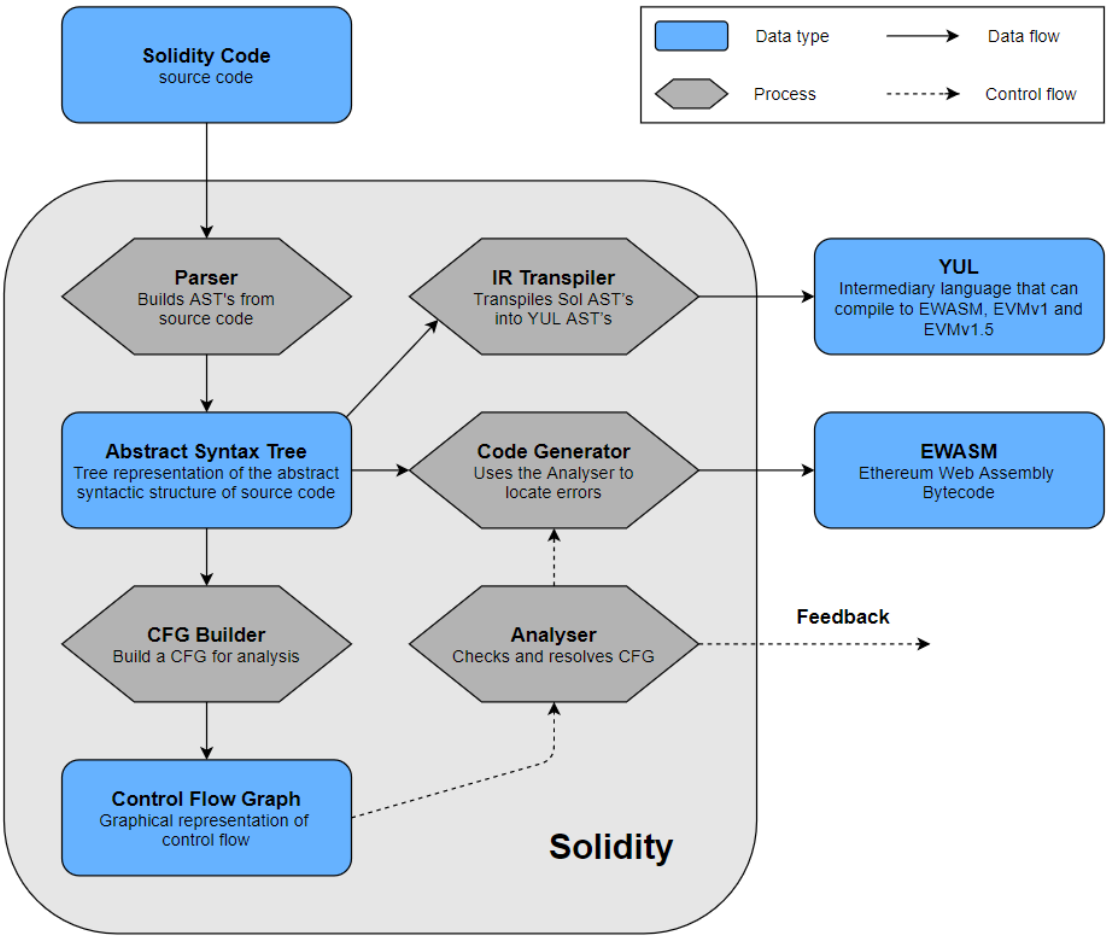
One of the central parts of software architecture is choosing a style and applying design patterns to obtain elegant and robust software. As it is a compiler, it goes through several stages: lexing, syntax checking, type checking, static analysis and code generation. The pipe and filter pattern is thus a natural style for Solidity.



Pipes-and-filter pattern [8]

Check out Solidity's particular compilation pipeline summarized below with a flow chart to get clear view on the data's flow throughout the compilation process:

[8] Mallawaarachchi, V. (2018, April 27). 10 Common Software Architectural Patterns in a nutshell. Retrieved from https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013



Source code to bytecode flow chart

The input of the pipeline is the smart contract's source code, written in Solidity. First, a lexer converts the character stream into a stream of tokens. This stream is then parsed into an Abstract Syntax Tree (ASTs), a tree representation of the code's abstract syntactic structure. The ASTs are then used to build Control Flow Graphs (CFGs), which are used further down the line by the static analyser. The analyser first checks the syntax, then annotates the AST with the types and does type checking and finally analyses the contract levels. It also resolves references and names. If the analysis finds an error, feedback is provided to the user. Otherwise, the code generator generates the target code from the ASTs. The default target is EVM bytecode. Apart from this, Solidity also contains a transpiler that can compile Solidity to Yul, an intermediate representation (IR).

Now back to the viewpoints!

26.4.3 Functional view

Let's take a look at the flow of the program. The code is first parsed by the, well, parser, resulting in an AST, which is then statically analysed.

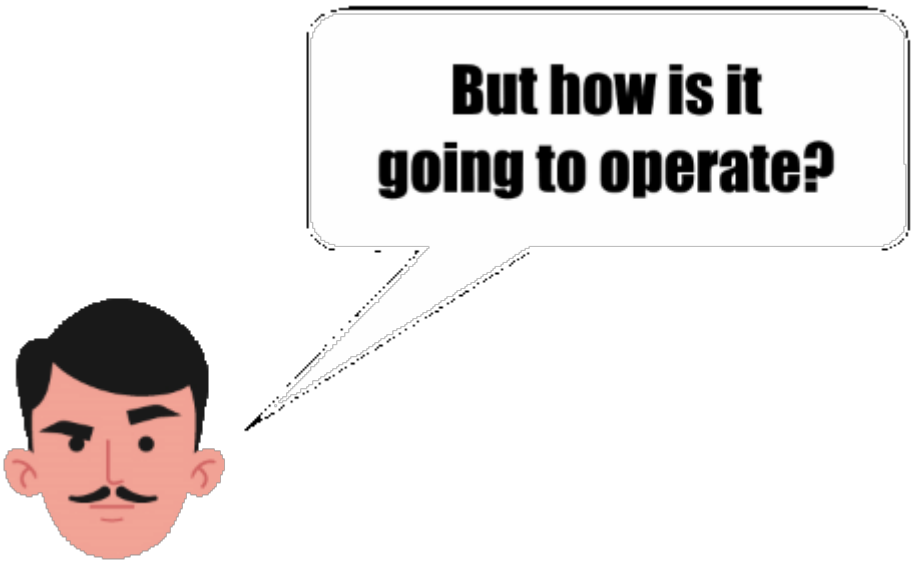


Figure 26.6: Image of Archie

1. First, the syntax is checked, for instance for absent end of statement (eos) tokens.
2. Next, the NatSpec documentation is parsed and added as annotations to the AST.
3. Names and types are checked, and the AST is annotated with the types. This happens in several steps: first, declarations and imports are registered. Next, names and types are resolved, after which inheritance and interface implementations are processed. Then, type requirements are checked, and finally types themselves are verified.
4. The control flow is analyzed. It is here that for instance dead code and uninitialized variables are detected.
5. A static analysis is performed to flag potential code smells.
6. Lastly, a state check is performed to verify whether state transitions are clean, and a model check verifies if the program actually does what it's supposed to do according to its specification.

After this the code generator (what most people consider the compiler's core) is ran. Only targeted files are compiled. First dependencies of the target files are compiled, then the AST is optimized and compiled. Finally, a check whether the gas limitations are honoured is performed (never forget the gas!).

After compilation, the EVM bytecode is ready. However, the user may choose to compile to Yul code instead of EWASM bytecode. This happens in the same way as the EVM bytecode compilation; only targeted files are compiled and dependencies are compiled first.

26.4.4 Development view

The development view describes the constraints that the architecture places on the software development process.

Let's derive this by giving an overview of the system decomposition, the modules of the system and their

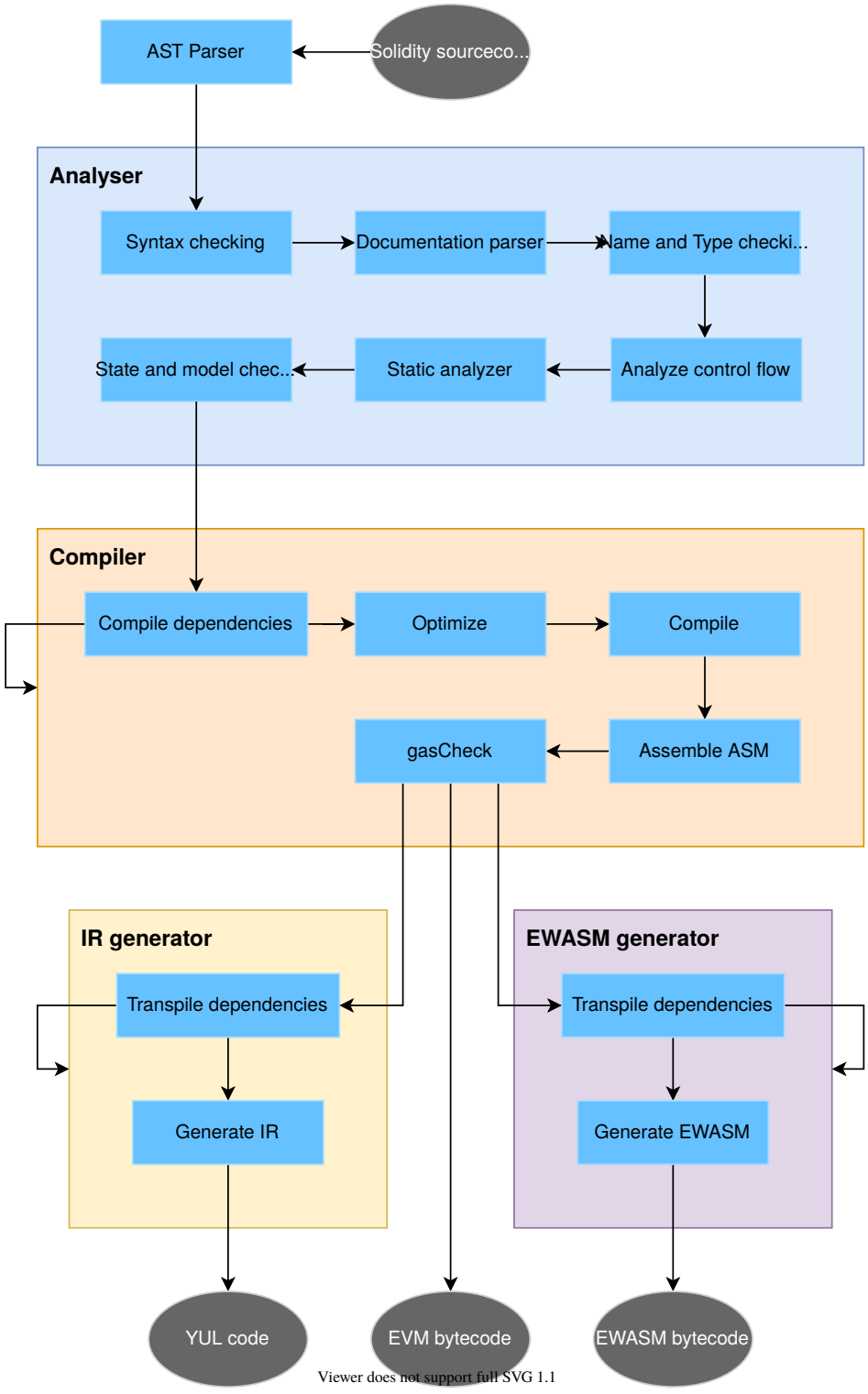


Figure 26.7: Run-time view

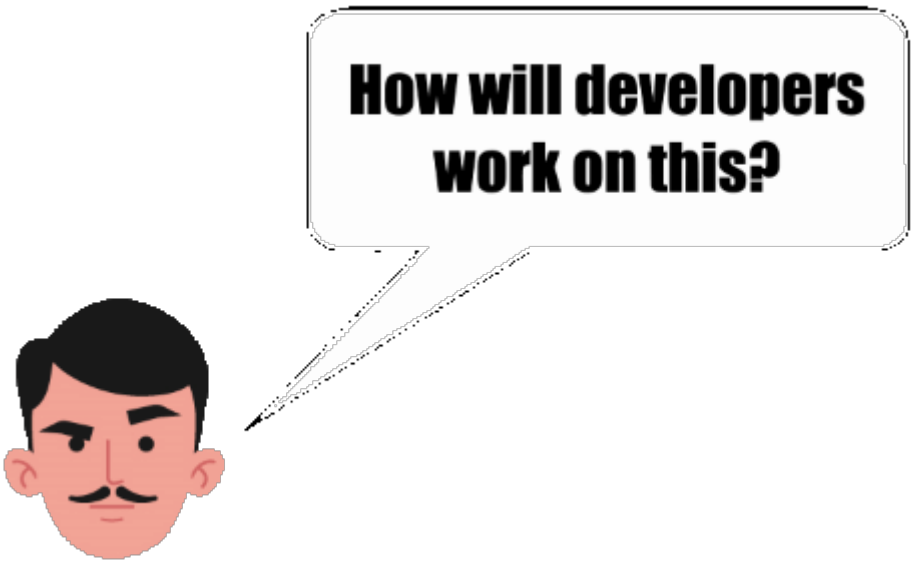


Figure 26.8: Image of Archie

dependencies. The system contains three libraries that are used by almost all the modules: Solidity Utilities, Abstract Syntax Tree, and Language Utilities (The white modules in the image). The three modules have been replicated to keep the image clear.

Module	Description	Contents
Abstract Syntax Tree		ABT, Types, Type provider, ABT storage methods
Analyzer		Control flow graph, Control flow graph builder, Name and type resolver, reference resolver, Syntax checker, Override and other checkers
Command Line Interface		Program entry, Arguments & input parser, some basic application logic
Code Generator		Expression compiler, Contract Compiler, Utility functions, Module with IR
Contract Interface		Full compiler stack
Utilities		
EVM Optimizer		Gas Meter, EVM state, EVM instructions, Control flow graph optimizer, Assembly functions

Module	Description	Contents
Language Utilities		Lexer
Model Checker		
Parser		
Solidity Utilities		
Tools		Version Updater, Yul Phaser
Yul		Yul intermediate language, multiple backend target language

In general, changes can be freely made to the compiler's internals for each module. However, a change to a module's functionality must be propagated to all subsequent stages in the compilation pipeline. This restricts the developers' freedom to independently work on different stages without coordinating, something that they achieve through the Scrum agile methodology[9].

26.4.5 Operational view

There isn't just one way to use `solc`, the Solidity compiler. In fact, in these early stages, both Solidity and the EVM still introduce breaking changes sometimes[10]. `solc` can compile for different versions of the EVM, it can do dry-runs, and quite interestingly also transpile to Yul! Yul is an intermediate representation that can be used for high-level optimization. A nice architectural feature of the command-line compiler is that in can also be fed the options as a JSON file for automated workflows by using te *JSON-input-output interface*.

26.4.6 Quality properties

The Solidity compiler gets the job done: after all, it is the most popular language in use for smart contracts today, being responsible for the handling of millions in assets. But how well does it get the job done?

In the context of smart contracts, a realistic ranking of quality properties by importance would be: 1. Security 2. Gas cost 3. Compilation speed (which we'll skip for now)

26.4.6.1 Security

A smart contract can go wrong in three ways, as security issues can pop up in: the platform (EVM), the compiler (Solidity), or your own smart contract. The platform isn't our concern here, but it's important to note that the Solidity dev team must write a compiler which compiles source code correctly and thus securely while at the same time making it as easy as possible to write secure contracts. The second concern is in fact openly stated during the very first ETH DevCon [11]. There's also a bug bounty program for the code generator, and we can take a look at the list of security bugs [12] until now to convince ourselves that security bugs do happen at a non-negligible rate (Serpent, Solidity's predecessor, was actually abandoned en masse after an OpenZeppelin security audit revealing gaping flaws in its security [13]).

[9] Ehlert, S. (2015, January 9). Retrieved from https://www.youtube.com/watch?v=tOwhUkp38bI

[10] Retrieved from https://solidity.readthedocs.io/en/v0.6.2/060-breaking-changes.html

[11] Wood, G., & Reitwiessner, C. (2015, January 6). Ethereum DevCon-0: Solidity, Vision and Roadmap. Retrieved from https://www.youtube.com/watch?v=DIqGDNPO5YM

[12] Retrieved from https://solidity.readthedocs.io/en/v0.5.3/bugs.html#known-bugs

[13] OpenZeppelin. (2017, July 28). Retrieved from https://blog.openzeppelin.com/serpent-compiler-audit-3095d1257929/

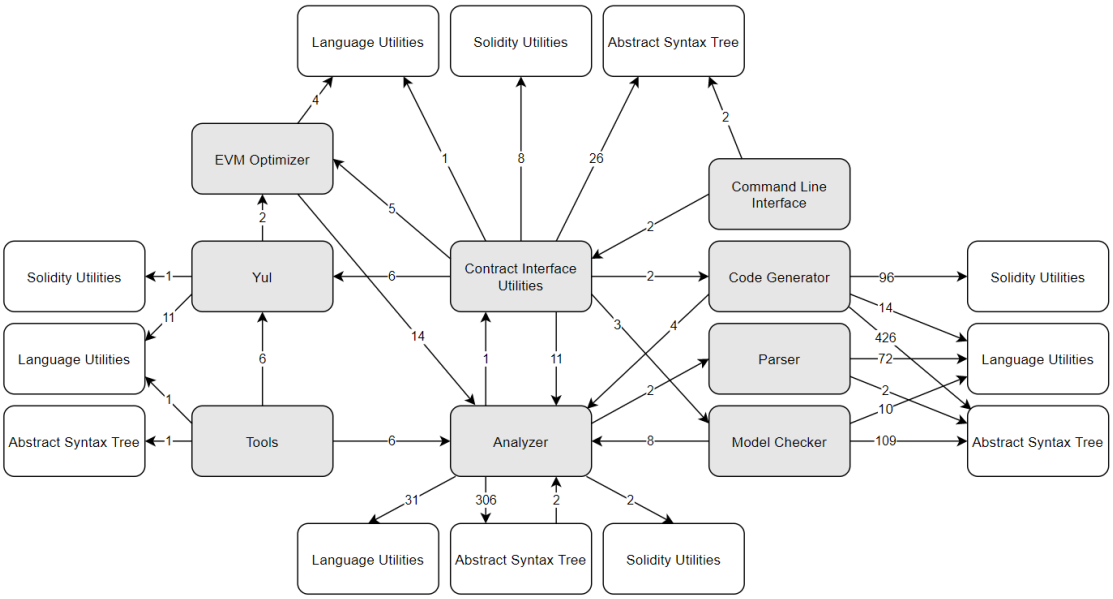


Figure 26.9: Modules



Figure 26.10: Image of Archie

Fear not, Solidity has done a lot at the architectural level to reduce security risks: security is an ongoing concern throughout the compilation chain. The syntax is checked by a separate module (SyntaxChecker.cpp). Furthermore, Solidity is not only statically typed (as types need to be known at compile time), but also strongly typed, and types are also checked by a separate module (TypeChecker.cpp). As pointed out in the OpenZeppelin audit, Serpent's utter lack of a type system was one of its most flagrant security shortcomings. Finally, during static analysis, warnings are emitted by the compiler, and following their advice is an easy way to make your contract more secure. Solidity also offers language features such as function modifiers (`pure`, `view`, `payable`), encapsulation, etc...

One of the very cool security-oriented features of Solidity is also its builtin formal verification capabilities: it offers a model checker allowing you to verify if your implementation follows the specification (mind you, this still leaves you open to insidious errors in the specification itself).

26.4.6.2 Gas cost

In Solidity, you pay ETH tokens in the form of gas to run your contract. The more you pay, the more incentive miners have to add your transaction to the blockchain, the faster it goes through. Solidity addresses this in three ways: it lets users get a gas estimate with a command-line option, it offers a range of types so as to allow efficient usage and it tries to optimize gas costs during compilation.



Figure 26.11: Image of Archie

Image credit: pikisuperstar

26.5 Solidity: Solid code or not?

This is the third essay on the Solidity project. This essay is about the code quality of Solidity. It will give an answer to the important question: Is the code of Solidity of solid quality or not, and how it this quality maintained?

26.5.1 Defining software quality

What does writing quality code mean? Without centering on a specific domain, the main meaning it registers with developers pertains to change. Grady Booch defines software architecture as the "significant design decisions that shape a system, where significant is measured by cost of change". But resistance to change can be generated at any level of abstraction, down to the implementation details themselves. This is the so-called technical debt, which causes you to pay interest on it in the form of extra work when you add new features.

This is but one of the quality properties that code might have. Indeed, there are other quality properties desirable for a system, such as speed or security.

26.5.2 Determining software quality

Now that we have defined two different types of software quality, we are going to look at how Solidity tests these types of qualities.

Most of the packages in solidity have unit tests in the `test` folder. These tests can be ran by running a shell script: `scripts/tests.sh`. These are however a lot of tests so it might be faster to only run specific tests during implementation.

Non-functional requirements are not as easy to test, but luckily the repository contains a number of scripts in the `scripts` and the `test` directories used to automate non-functional requirement testing. These tests are important as having a good universal coding style, documentation etcetera improve the readability of the code a lot.

1. `test/buglistTests.js` tests if known bugs are not correctly avoided in the code.
2. `test/docsCodeStyle.sh` tests if documentation is styled correctly.
3. `scripts/checkStyle.sh` performs a C++ style check on all C++ code files.
4. `scripts/test_antlr_grammar.sh` tests if all sol files are parsable and thus grammatically correct.
5. `scripts/pylint_all.py` performs syntax testing on all python scripts.

26.5.3 Continuous integration

Like most repositories, Solidity uses Continuous Integration to ensure lasting code quality standards. The CI performs most tests mentioned above, and some additional tests to ensure functionality on different platforms.

The first category of tests performed by CircleCI are simply builds. Here the CI tries to build the system on different platforms with a standard `Release` target.

The second category performs all defined unit tests on the platforms mentioned above. Being able to build the system on a platform is one thing, but knowing that the system behaves the same is crucial in a program such as this one.

Lastly style, syntax and bug checks are performed. These are not platform specific as they run in bash and python, or use third-party software to rely on, such as antlr, Z3, Asan etc. Following is a list of checks that are performed by CircleCI.

- check spelling
- check documentation examples
- check C++ coding style
- run operating systems checks
- run buglist test
- run pylint (pylint: checker/code analysis)
- check antlr grammar (antlr: tool for processing structured text)
- run z3 proofs (z3: theorem prover)
- ASan build and tests (Asan: tool that detects memory corruption bugs)
- Emscripten build and run selected tests (Emscripten: toolchain for compiling C/C++ to asm.js and WebAssembly)
- OSSFUZZ builds and (regression) tests (fuzzing: test correctness by generating program inputs to reach certain execution states)

Next to these CircleCI tests, there are also Appveyor and Travis tests. Appveyor is used exclusively for Microsoft Windows tests (Presumably because CircleCI doesn't work well with Windows). The same build and tests as mentioned above are performed here. Travis is used for deploying code and services to, for example, docker and other repositories.

26.5.4 Test quality

Defining tests and implementing continuous integration is one thing, but those tests also need to be of high quality to make any difference. How can we determine if tests are of high quality?

To do this, we will first look at the functional tests, the unit tests. The easiest and dirtiest way to look at unit test quality is to look at the line coverage and the branch coverage of those tests. Unfortunately the maintainers of Solidity decided that code coverage is too misleading to use, so they did not implement it in their continuous integration.

However Solidity does use Codecov for coverage analysis. This application can be launched by continuous integration, but the maintainers of Solidity decided to only run it for certain pull requests and commits. The newest coverage report we could find is presented here, but nevertheless it is unfortunately pretty outdated.

Another indication of test quality is ratio of test code. According to SIG the test code ratio of Solidity is 111.9%. This means for each line of code there are 1.119 lines of test code on average, and that 52.8% of the repository is test code. As the repository contains more test code than functional source code, this is a pretty good ratio. However this still doesn't mean those tests are of high quality, and there is no good way to determine this without looking at each and every test, and the code.

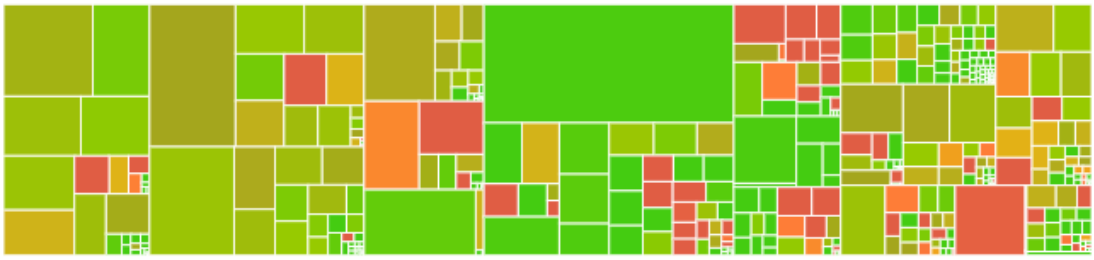
Determining the quality of non-functional tests is a whole different story compared to unit tests. The best way we can do this is by looking at what the tests check, and testing if they break when deliberately introducing styling flaws.

We will look at some of these tests to analyse what they do and if we think that adds something to the repository.

An important readability tool is the code style checker, `scripts/checkStyle.sh`. This checker simply

Codecov Report

> Merging #6678 into develop will **not change** coverage.
> The diff coverage is `n/a`.



```
@@           Coverage Diff           @@
##           develop    #6678   +/-   ##
=========================================
  Coverage    87.85%   87.85%
=========================================
  Files          399      399
  Lines        38942    38942
  Branches      4577     4577
=========================================
  Hits         34213    34213
  Misses        3179     3179
  Partials      1550     1550
```

Flag	Coverage Δ	
#all	87.85% <ø> (ø)	⬆
#syntax	26.07% <ø> (ø)	⬆

Figure 26.12: Solidity code coverage

performs a bunch of greps on all `.cpp` and `.h` files. It checks if whitespace is used correctly and at the right places, and if certain keywords are in the right order.

Most other checkers make use of existing software to check for example if files can be parsed. These parsers and linters are used a lot and thus we can trust that these do what they should do. Because python and solidity files are not built or tested by the build targets, these files have to be tested separately on syntax errors. These are important tests as broken scripts or even solidity example scripts might slip through the tests if these aren't performed.

26.5.5 Activity

One of the interesting things to look at is the recent code activity and how that activity relates to the architectural components. We looked at the number of code changes in the last month of the development branch to get an idea of this.

module	number of changes
circleci | 151	151
code generator	778
model checker	576
contract interface	109
abstract syntax tree	105
analyser	349
yul phaser	2760
docs	797
scripts	603
EVM optimizer	474
yul library	75
commandline compiler	67 |
language utilities	3
utility library	156

test module	number of changes
smt checker tests	709
semantic tests	6454
syntax tests	543
abstract syntax tree JSON tests	473
yul phaser tests	3096
yul library / object compiler tests	15
yul library / yul optimizer tests	1561
yul library tests	285
command line tests	77
utility library tests	53

We can see some hotspots in the coding activity. The majority of the changes have been made in the yul

phaser, the yul library, the yul phaser tests and the semantic tests. This observation matches with the open projects in Solidity (https://github.com/ethereum/solidity/projects), here we see a big Yul project that is 70% done. Another big project is called 'SMT Checker MVP' where the authors aim to develop the semantics checker towards a minimum viable product. The checker proves if the written code is correct or not and aims to catch bugs at compile time. This most likely relates to the majority of the changes made in the code generator, model checker, contract interface, smt checker tests, semantics tests and syntax tests modules. We can also see other projects that match with the observed changes, such as: wasm, documentation translation, ect.

We can also conduct a high-level analysis based on the roadmap projects. We assume that projects most likely to be worked on are those with at least one issue *in progress* and rank them by a combination of the number of issues in progress and issues to do. We consider issues in progress as a multiplier for issues to do: `likely_amount_of_work = #_issues_in_progress * #_issues_to_do`. The tie between *Consolidate inheritance rules* and *Natspec*, which have no issues in progress, is broken by the total number of issues.

The ranking is: 1. Yul 2. SMT Checker MVP (not taken into account) 3. Zeppelin Audit (not taken into account) 4. Sol -> Yul codegen 5. Wasm/Ewasm 6. Consolidate inheritance rules 7. NatSpec

We can assign a partial score from 0 to 6 to each component identified by Sigrid Says everytime it is a part of one of these projects (except the SMT checker and Zeppelin audit): `score = 7 - project_rank`. We can then sum the partial scores to yield a total score for expected change magnitude:

1. libsolidity (12)
2. codegen (9)
3. libyul; solc; interface (8)
4. libevmasm (6)
5. analysis; ast (1)

26.5.6 Architectural roadmap

The roadmap for Solidity features is a list of GitHub projects.

- Natspec: NatSpec (Natural Specification) is a rich documentation format for functions, return variables and other constructs.
  - liblangutil
    - ErrorReporter.cpp: it reports errors when parsing docstrings
    - Scanner.cpp: the lexer must scan comments
  - libsolidity
    - analysis
      - DocStringAnalyser.cpp: does static analysis on docstrings
- Wasm/Ewasm: Ewasm (Ethereum WebAssembly) will replace the EVM (Ethereum Virtual Machine) as the state execution engine of the Ethereum network as part of Eth 2.0.
  - libyul
    - backends/wasm
      - EVMToEwasmTranslator.cpp: a translator facilitates passage from EVM to Ewasm
      - BinaryTransform.cpp: transforms internal wasm to binary
    - AssemblyStack.cpp: handles Ewasm assembly

  - libsolidity
    * interface
      · CompilerStack.cpp: the compiler stack generates Ewasm
      · StandardCompiler.cpp: as the standard compiler, it can take requests for Ewasm code
  - solc
    * CommandLineInterface.cpp: the CLI accepts Ewasm command line parameters

- Reducing technical debt: Accumulated technical debt is to be reduced through refactoring.
- Sol -> Yul CodeGen: The project concerns the code generation step for compiling from Solidity to Yul, an intermediate representation language.
  - libsolidity
    * codegen
      · YulUtilFunctions.cpp: utility functions for Yul
    * codegen/ir
      · IRGeneratorForStatements.cpp: this generates Yul for statement tokens
      · IRGenerationContext.cpp: an expression token is situated in a context, taken into account during Yul generation
- Emscripten: The project concerns the configuration of Emscripten, which is used to compile to WebAssembly.
- Documentation translation: As the name states, this is about translating documentation to various languages.
- Yul: The project concerns the Yul format, optimizations that can be done to the code in Yul format as well as compilation from Yul to other targets.
  - libevmasm
    * AssemblyItem.cpp
  - libsolidity/interface
    * CompilerStack.cpp: the compiler stack handles the source mapping
    * StandardCompiler.cpp: the standard compiler interacts with the source mapping
  - libsolidity/codegen
    * ir
      · IRGenerate.cpp
  - libyul
    * AssemblyStack.cpp: the source mapping's is from assembly to source
    * AsmAnalysis.cpp: this analyzes the assembly code
  - libyul/backends
    * evm
      · EVMDialect.cpp
      · AsmCodeGen.cpp
      · EVMCodeTransform.cpp
    * wasm
      · EVMToEwasmTranslator.cpp
      · WasmDialect.cpp
    * optimiser
      · ConditionalSimplifier.cpp
      · ControlFlowSimplifier.cpp

  - solc
    * CommandLineInterface.cpp

- Zeppelin Audit: The project aims to fix security issues found during the audit conducted by Zeppelin.
- Backlog (breaking): All breaking changes (non-backward compatible) are here.
- Consolidate inheritance rules: As a contract-oriented programming language, contracts in Solidity can inherit from one another. This project aims to make the inheritance rules more comprehensive.
  - libsolidity/analysis
    * TypeChecker.cpp: the type checker needs to take into account inheritance rules
    * OverrideChecker.cpp
  - libsolidity/ast
    * AST.cpp: the AST needs to conform to the inheritance rules
    * ASTJsonConverter.cpp
- SMT Checker MVP: The project aims to reach a minimum viable product (MVP) for a satisfiability modulo theories (SMT) automated checker. It is unrelated to Solidity releases and is fairly independent.
- Backlog (non-breaking): All non-breaking (backward compatible) changes are here.

26.5.7 Code quality assesment

Let's look at the Sigrid Says measurements. They're given on a scale from 0 to 5.5, which we split into five categories:

- 0-0.9: worrying
- 1-1.9: poor
- 2-2.9: average
- 3-3.9: good
- 4-5.5: excellent

Sigrid can only do so much, so meaning should only be assigned to these values in context.

Component	Overall maintainability	Volume	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling	Component independence
all	average	excellent	excellent	poor	poor	average	average	average
libsolidity	good	excellent	good	average	average	average	excellent	excellent
codegen	average	excellent	excellent	worrying	poor	average	average	good
libyul	good	excellent	excellent	average	average	average	excellent	good
solc	good	excellent	excellent	worrying	worrying	excellent	excellent	excellent
interface	average	excellent	excellent	poor	poor	good	poor	poor
libevmasm	average	excellent	excellent	poor	poor	poor	good	average
analysis	average	excellent	excellent	poor	worrying	good	average	worrying
ast	average	excellent	excellent	good	average	good	worrying	worrying

On average, Solidity is ranked as an average project. Its strong suits on average seem to be *volume* and *duplication*, and it additionally ranks very well in terms of *component balance*, scoring a 4,6. Three components register two metrics in the worrying category, namely `solc`, `analysis` and `ast`. Luckily, these

are rather unlikely to register major changes, so according to Ward Cunningham's technical debt allegory, it is probable that not a lot of interest in terms of work will be spent changing these components.

26.5.8 Importance of code quality

A central considerent for Solidity in terms of code quality is security, as Solidity is used to write contracts handling actual financial assets. In fact, there's an entire chapter in the documentation pertaining to it. The contributing guide gives insight regarding what is considered essential for code quality. A Git flow is enforced, and it is recommended that each contribution come with its own tests. For larger changes, it is recommended to consult the development team on the development Gitter. The coding style is well defined and tested for in the CI process. American Fuzzy Lop (AFL) is used for fuzz testing.

26.5.8.1 The testing label

The GitHub issue tracker for solidity has a dedicated *testing* label to be added to all issues related to testing. At the time of writing, there are 55 open issues and 84 closed. Out of 680 open issues, 8% are testing issues. As issues 7860 and 7861 show, alternatives for current code coverage reporting, which is limited, are being considered.

26.5.9 Refactoring suggestions

When we look at the refactoring candidates resulting from the Sig analysis, we see that the code generator module pops out. The code generator module has a lot of long functions and methods. It also contains a lot of complex functions (measured with McCabe complexity[14]). This is probably where most of the improvements are possible. Long and complex functions are difficult to understand and require more test cases, and even then it is much more difficult to cover all possible cases. Aside from this we can conclude the following from the analysis:

- Duplication is not a problem in the Solidity code base, there is actually very little duplication.
- Unit interfacing: The analysis shows there are some functions and methods with many parameters that are good candidates for refactoring but that this number is fairly small.
- Separation of concerns: The abstract syntax tree module and the language utilities module have a strong coupling between their files. Specifically, the AST files are very strongly coupled, making them difficult to analyse, test and modify.
- Component independence is a point of attention throughout the codebase especially in the analysis and code generator modules, which makes system maintenance more difficult.

26.6 Solidity and its variability

This is the fourth and the last essay on the Solidity project. It covers an analysis on the variability of the Solidity compiler. Like most compilers, Solidity allows a lot of command line arguments that alter the expected input and output files and the expected command line arguments themselves. Because in the first place we are talking about a compiler, and in the second place security plays a crucial role in the project, it is rather important that the process and the output is trustworthy in all possible variances. This is why we decided that a variability analysis would be an interesting final vision on the Solidity project.

[14] McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering, (4), 308-320.

26.6.1 Variability modeling

The first step in determining how variability is handled in a system, is by determining what variability actually exists. This part is dedicated to that. We will first look at what types of variabilities are present in Solidity. Then we will show how these variabilities influence eachother, and finally we will show a feature model of the system.

26.6.1.1 Main features

We can define different kinds of variability. The first, most obvious kind is command line arguments. When we run the command `solc.exe --help` we find there are 46 different command line arguments. 10 of these allow one or multiple additional arguments. This is already a massive pool of variability, and it's only the first type of variability.

Another variability in Solidity is the operating system it compiles on. Solidity is designed to function on Windows, Linux and MacOs. The program is intended to run the same on all of these platforms, however considering that Solidity is written in C++, that may not always be guaranteed. We will do an analysis on how Solidity handles the differences in operating systems, and possible differences in libraries used depending on the operating system.

The last variability in Solidity is the choice of C++ compilers. GCC, Clang and MSVC can be used to build the Solidity code base. MSVC is only available on Windows.

26.6.1.2 Incompatibilities

There are little real incompatibilities as we are dealing with a compiler, however we have found some interesting things when searching through the codebase. One of the things that stand out is the number of Microsoft software related design choices. The code is full of comments that explain how certain choices relate to Windows or MSVC.

Take for example the batch file for installing pre-requisite packages for solidity on Windows. There we found the following: "The lack of a standard C++ packaging system for Windows is problematic for us". The developers have considered various options for improving this situation, one of them was switching to NuGet C++ packages. Another option that they have considered is to add dependencies as git-submodules so that Solidity does not depend on platform specific packaging systems. It would add more control and robustness but increase the duration of the build process. Currently the issue is still open on the Ethereum Aleth repository: a collection of C++ libraries and tools for Ethereum which is also used in Solidity.

Another interesting finding regarding configurations is the existence of a special header file UndefMacros.h which "should be used to #undef some really evil macros defined by windows.h which result in conflict with our Token.h".

Not only windows but also the MSVC compiler demands certain configurations. It for example requires MSVC specific Ethereum dependencies. All of the compilers require compiler specific compile options. But it seems that the focus is mainly on GCC an Clang as the compiler options used for MSVC for the most part disable warnings.

We also see a special flag for the MSVC compiler as it has difficulties with compiling large objects. This flag will be removed as soon as the files concerned are reduced in size which is being worked on.

26.6.1.3 Feature model

An easy and clear way to visualize features, is by constructing a feature model. We created this model in FeatureIDE for Eclipse.

This model shows the different options for compiling solidity itself. This includes different possible compilers and platforms.

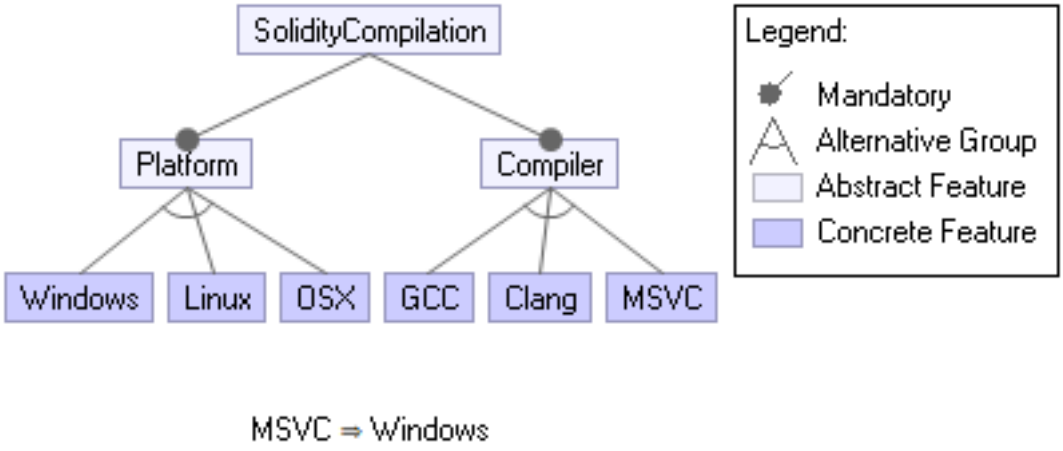


Figure 26.13: Solidity compilation feature model

We did not make a feature model of the command line options solidity offers. This is because these options are all completely distinct. The only way some flags influence others is by ignoring them completely if they are present (thus allowing them to be anything instead of constraining them).

26.6.2 Variability management

Now that we know what types of variability exist in the system, we want to know how these are managed. This includes both how different stakeholders can find out what types of settings and toggles exist in the system, and how these are managed in the system itself. Secondly, how are these variabilities managed in the long-term? Is for example the performance and functionality for different platforms tested?

26.6.2.1 Information sources

The groups of users and developers overlap in the case of Solidity, this became clear from our stakeholder analysis in our first essay. The users and developers therefore share the same sources of information.

The main source of information regarding variability and solidity is the documentation. In the solidity documents there is a lot of information on the different platforms that Solidity supports. There are specific prerequisites for the different operating systems and compilers.

26.6.2.2 Management mechanisms

As mentioned in our previous essay, during the continuous integration, there is a category of tests that performs all defined unit tests on the different platforms, i.e. Windows, Linux and MacOS.

There is also a windows specific upgrade tool and the Visual C++ compiler is only available on windows.

26.6.3 Variability implementation mechanism and binding time

Finally, we want to know how the variability is implemented, and how (and when) the different versions are bound into the system.

26.6.3.1 Mechanisms and binding time

There is a variation point at compile time, for instance where you choose which C++ compiler you use to compile. All choices made here are to be done by modifying the files contained in the cmake folder, as well as the CMakeLists.txt file, although discouraged. Here, constraints for specific compilers and flags that are only needed on some compilers are maintained.

For instance, at compile-time (of the compiler itself, not to be confused with the compile-time of actual Solidity files!), you can for instance choose the target platform, and whether to also build supporting tools for the compiler in the EthOptions file.

As mentioned before, apart from the compiler choice, Solidity offers a slew of command line parameters. These parameters are all read in the `solc` module, which handles all command-line related code. The parameters are then managed by the `boost::program_options` library. This library ensures parameters are all managed the same way and can all be reached from anywhere in the program.

As these program_options are only set when reading the command line arguments, these variables are fixed after the command line interpreting is done. This means that after this these variables are bound and cannot be changed anymore, the entire program after the command line interpretation runs with the same command line arguments.

Although you can build the project locally as mentioned previously, the project is also built automatically on CircleCI. This makes use of the CircleCI config file, which allows you to choose for which platforms to build (e.g. ArchLinux, OSX, Ubuntu, etc.).

At run-time, the compiler presents a choice of flags to the user in order to compile his program, this is one of the main variation points. Information which can be specified at this point is which target to compile to (e.g. the version of EVM if compiling to EVM bytecode), whether to optimize, to produce the AST in JSON, etc.

It is important here to not mistake the definition of the target at two variation points as binding tim variability, as these are two different targets: at compile-time, you define the target for which you compile the compiler so it can run on a designated machine, whereas at run-time, you specify the target on which you want to run your smart contract. Pretty meta, right?

26.6.3.2 Design choices

Looking at the code base you can see that the design choices with respect to the operating system and compiler are focused on facilitating Solidity for Windows, Linux, MacOs with GCC, Clang and MSVC. However it looks as if most is designed to work on Linux and MacOs in combination with GCC and Clang, support for Windows and MSVC is added with flags and some parts of the code are changed to be compatible with the former.

Chapter 27

spaCy



Figure 27.1: spaCy Logo

spaCy is a free, commercially open-source library for industrial-Strength Natural Language Processing

(NLP) in Python, released under the MIT license. Part of the Explosion Company. It is designed to do real work, build real products or gather real insights. The library is designed in such a way that it tries to avoid wasting your time. The owners like to think of Space as the “Ruby on Rails” of Natural Language Processing. spaCy is the way to go when dealing with large volumes of text, whether it is used for Deep Learning, information extraction or dealing with words in context.

spaCy is compatible with 64-bit CPython 2.7 / 3.5+ and runs on Unix/Linux, macOS/OS X and Windows. It is already trusted by Airbnb, Uber, Quora, Retriever, Stitch Fix, Chartbeat, the Allen Institute for Artificial Intelligence and many more. In 2015, independent researchers from Emory University and Yahoo! Labs even showed that spaCy offered the fastest syntactic parser in the world and that its accuracy was within 1% of the best available (Choi et al., 2015). spaCy is maintained by two people, and they are welcoming help. They even want questions, issues, bugs etc. to be shared publicly, so more people can benefit from it. They even made a very detailed contributing file.

In the coming sections, an in-depth analysis of spaCy’s architecture will be included. First the stakeholder analysis and merging pipeline based on existing pull requests will be laid out. Afterwards, the architecture will be examined from three different perspectives (Context, Deployment and Development). Afterwards, the project’s Technical and Testing debt will be analyzed and the evolution of the project from the moment it was released will be identified. Lastly, we will summarize the findings and conclusions regarding the architecture of spaCy.

27.1 The Team

Meet the team:

- Anwesh Marwade
- Michael Leichtfried
- Nikhil Saldanha
- Thijs Timmer

27.2 spaCy: For All Things NLP

In this blog post, we take a sneak-peek into a leading-edge Natural Language Processing (NLP) library for Python, its stakeholders and its journey up until now. Hence we decided to get some words together (pun intented) to describe the insipiration behind ExplosionAI’s NLP project: **spaCy**

27.2.1 The Vision

spaCy is a free, open-source library for advanced Natural Language Processing or NLP (written in Python and Cython), developed by Matthew Honnibal. To quote Matthew, "*I wrote spaCy because I think small companies are terrible at natural language processing (NLP). Or rather: small companies are using terrible NLP technology.*" [1]

[1] Introducing SpaCy, https://explosion.ai/blog/introducing-spacy

27.2.1.1 What spaCy tries to achieve

spaCy strives to provide a production level Natural Language Processing pipeline that is fast and efficient. It aims to bridge the gap in cutting-edge NLP between production and academia. Libraries like NLTK provide the necessary tools but are intended to support research and teaching rather than focus on production-level functionalities. This is where spaCy earns its coin, by making production level NLP accessible and deployable. It is a very popular NLP library owing partly to its regular, cutting-edge updates (especially with implementations of state-of-the-art models of Big Transformers like **BERT, GPT-2**) and constant improvements in performance.

What spaCy is:

- It can be used to build information extraction systems
- Or Natural Language understanding systems.
- It can be used to pre-process text for deep learning.

What spaCy is NOT:

- It is not an API or an NLP platform of sorts.
- It is also not an out-of-the-box chat-bot or a conversation-engine.
- It is not a research software. (as mentioned earlier)
- Finally, it is NOT a company. **ExplosionAI** is the company responsible for publishing spaCy.

Importantly and admittedly spaCy is for production-use and hence involves fairly different design decisions than NLTK or CoreNLP; which we will deep-dive in the later essays[2].

Code: Python, Cython

Platforms: Linux, Windows, Mac

27.2.2 End User Mental Model

Academic improvements in quality of models in the NLP domain hasn't been concurrent with commercial NLP. This is largely attributed to the fact that reaching production-level with NLP models requires a huge training effort, an intricate pipeline and not to mention complicated model implementations. This has generally deterred individuals and companies alike in terms of developing real and applied NLP products.

Matthew Honnibal (founder ExplosionAI) developed spaCy as a tool particularly for this purpose: bringing advanced and applied NLP into real products. It is designed to be an 'Industrial Strength NLP' library with efficiency, state-of-the-art implementations, fast deployment (in production) and easy model integrations as its core characteristics. The intention behind spaCy was to provide production level NLP tools for the applied data scientist and/or relatively smaller companies/teams, where NLP was stuck in a limbo due to esoteric implementation of algorithms and exorbitantly expensive training required for language data.

spaCy is hence built for simplicity in production and deployment, performs incredibly fast (Implemented in Cython and admittedly one of the fastest syntactic parsers available) and does away with a lot of the complexity of liguistic models; thus earning it high praise as the *'Numpy'* of NLP! [3]

[2] spaCy 101: Everything you need to know, https://spacy.io/usage/spacy-101
[3] Explosion AI Blog, https://explosion.ai/blog/

27.2.3 Key Capabilities and Properties

spaCy aims to be one-stop-shop for all tasks NLP. Providing production-level code while being easy to apply are two pillars at the forefront of its development. To this end spaCy's features and key capabilities are as follows:

(*Some of them refer to linguistic concepts, while others are related to more general machine learning functionality*)

- Tokenisation
- Part-of-speech (POS) Tagging
- Dependency Parsing
- Lemmatization
- Named Entity Recognition (NER)
- Entity Linking
- Similarity
- Text Classification
- Sentence Boundary Detection
- Word-to-vector transformations
- Out-of-the-box methods for cleaning and normalization of textual data
- Cutting-Edge pre-trained models for SOTA-NLP

27.2.4 Stakeholders

Stakeholders of the spaCy Project:

27.2.4.1 Matthew Honnibal

spaCy is Matthew's brainchild, forged with a vision to create an production-level library of utility functions for NLP. After spaCy's release in 2016 and its eventual success, ExplosionAI was founded as a software company specializing in developer tools for AI and Natural Language Processing with spaCy as their showcase product.

27.2.4.2 ExplosionAI

Known as "The makers of spaCy", ExplosionAI is a company founded by Matthew Honnibal along with co-founder Ines Montani[4]. They now maintian spaCy as an open source repository under the MIT license with regular feature development.

27.2.4.3 End-Users

Based on the use-case requirements we can broadly categorize the end-users as,

- **Commercial NLP**: These are the companies/teams that are actively using spaCy in production for their NLP based applications. They represent the main target audience of spaCy given that it allows them to leverage cutting-edge NLP for real-world products. Example: Uber, Airbnb, Quora [5]

[4] Introducing Explosion AI, https://explosion.ai/blog/introducing-explosion-ai
[5] Spacy main website, https://spacy.io/

- **Data Scientists and Enthusiasts** (NLP): Both professional Data Scientists and hobbyist developers have a similar use-case with spaCy for their NLP based requirements. spaCy is widely used for both personal/professional projects as well as for academic research.

The end-users mentioned above are the main consumers and stakeholders of the product and drive the development roadmap for spaCy as per Matthew.

27.2.4.4 GitHub

As the *spaCy project* is maintained publicly on GitHub and encourages active contribution from the community, we hence consider GitHub to be an integral stakeholder in the successful development of the project.

27.2.4.5 spaCy Community

- **Contributors**: spaCy has a vibrant community of developers and contributors as espoused from the 408 direct contibutors and 24,574 contributors in its dependency graph (at the time of writing). It maintains a thorough issue tracker and actively monitors Gitter chat and the reddit/stackoverflow support groups.
- **Users** (Current and Past): We consider that users play an active role in the spaCy community by participating in its development through reporting of bugs, recommending new and relevant features and suggesting certain improvements. Since spaCy is constantly under development, these users are an important source of feedback.
- **Ecosystem**: There are several projects (or extensions) that primarily use or even build on spaCy and hence are considered as stakeholders. Example: sense2vec, neuralcoref, displaCy etc.

27.2.5 Current & future context

In order to provide a clear understanding of the operation context of the spaCy project, we have an illustration that depicts various dependencies, relationships and intersections of spaCy as a product.

The spaCy project is written majorly in Python (Cython) and is actively maintained by the team at ExplosionAI with an open source MIT license on Github where it encourages contributions from the dev-community with an exhaustive list of code conventions and how-tos.

spaCy has dependencies in terms of libraries from which it derives some functionality like numpy, pandas and also certain libraries/softwares that are dependent on spaCy i.e. THiNC and sense2vec; these are typically products rolled out by ExplosionAI to build upon spaCy. For instance, sense2vec is a twist on traditional word2vec (word embeddings) based on spaCy.

There are notable alternatives to spaCy in the NLP for python domain namely NLTK and TextBlob. While these alternatives like NLTK, which is easy to use and possibly great for NLP research, certainly deserve the credit, spaCy is extremely optimized and understandably preferred in production environments. The main differences between spaCy and other alternatives is explained in the following table from their website.

Additionally, spaCy runs on windows, linux and macOS distributions alike.

[6] Facts and Figures, https://spacy.io/usage/facts-figures

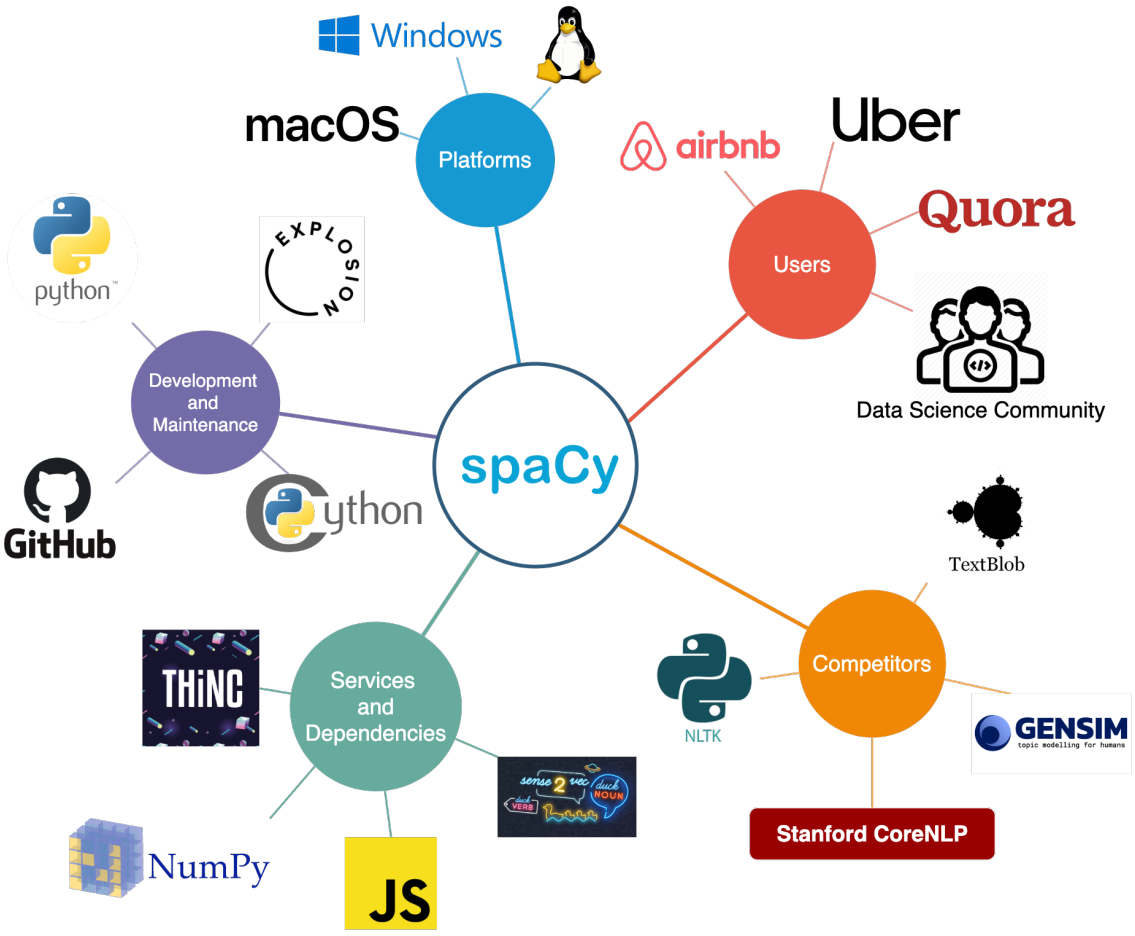


Figure 27.2: Context view

	SPACY	NLTK	ALLEN-NLP	STANFORD-NLP	TENSOR-FLOW
I'm a beginner and just getting started with NLP.	✓	✓	✗	✓	✗
I want to build an end-to-end production application.	✓	✗	✗	✗	✓
I want to try out different neural network architectures for NLP.	✗	✗	✓	✗	✓
I want to try the latest models with state-of-the-art accuracy.	✗	✗	✓	✓	✓
I want to train models from my own data.	✓	✓	✓	✓	✓
I want my application to be efficient on CPU.	✓	✓	✗	✗	✗

Figure 27.3: Alternative Comparison[6]

27.2.6 RoadMap

SpaCy does not maintain an explicit future roadmap, neither on its repository nor its website, but it does have a list of pinned issues. This pinned issue list can be seen as a proxy for the actual roadmap, since these issues are the most important features/problems being tackled at any time.

Firstly, they want to add more pre-trained models for even more already supported languages. As of now, spaCy supports over 50 languages, but has pre-trained models for only 10 of them. The untrained languages are in "Alpha support", which only includes tokenization and other rule-based methods[7].

Example sentences for multiple languages are yet to be included. This would come in handy while testing in unfamiliar languages. They are gradually filling these in, but the issue is open since 2017[8].

Furthermore, ExplosionAI keeps a close contact with its stakeholders through regular blog posts[9]. Based on a recent post (regarding spaCy), a new data augmentation system is under development. This system should eventually translate to having a higher accuracy on texts with inconsistent casing and punctuation. [10]

There are still more issues in their repository and as technical debt. Most of them are as an improvment on existing features, but some make suggestions for new features.

[7] Increased support for new languages, https://github.com/explosion/spaCy/issues/3056
[8] Example sentences for testing languages, https://github.com/explosion/spaCy/issues/1107
[9] Explosion AI Blog, https://explosion.ai/blog/
[10] Introducing spaCy v2.2, https://explosion.ai/blog/spacy-v2-2

27.2.7 Stay Tuned!

spaCy strives to be a *one-stop-shop* for all things NLP. It truly merits its spot as the most popular and domain-leading NLP library. We intend to deep dive into *how and why* spaCy serves a core purpose in any NLP pipeline. Next, we pen down our research on how spaCy's vision influenced its architectural decisions and the story behind its development!

To be continued...

27.3 From Vision to Architecture

SpaCy is the brainchild of Matthew Hannibal, who has a background in both Linguistics and Computer Science. After finishing his PhD and further 5 years of research in state-of-the art NLP systems, he decided to leave academia, created SpaCy and started interacting with a wider development community. [11]

Being a NLP library for processing text, spaCy's core is it's processing pipeline, into which not only core NLP functions like tokenizer, lemmatizer and tagger integrate: spaCy supports custom pipeline components, allowing a big ecosystem of plugins and extensions to grow. [12]

27.3.1 Architectural Views: A Discussion

In this article we examine the architecture of spaCy through the lens of architectural views as described by Kruchten [13]. The four views are as follows:

- **Logical View** is a description of the design model that captures the functional requirements of the application. This is usually done through a clear decomposition of structural elements or abstractions. *UML class diagrams* or *ER(Entity-Relationship) diagrams* are often used as representations of the logical view.
- **Process View** describes the process i.e. the behaviour, concurrency and information-flow of a system. It usually deals with the non-functional aspects for example using *Data Flow Diagrams* or DFDs.
- **Physical View** is often seen from a system engineer's point of view and is concerned with the software topology as well as its hardware mappings.
- **Development View** addresses the software management aspect with its development environment usually as seen from a programmers perspective.

Examining a library instead of a full bodied software program, we try to fit aspects of Kruchten's view model to spaCy's architecture and use-case. To this end, in terms of the **logical view**, the processing pipeline of spaCy seems to be the central element. All of spaCy's multitude of modules like the *tokenizer, NER-module etc.* are built around it. This allows spaCy to have a *single source of truth* by avoiding multiple channels of data flow and saves memory as well. Since spaCy does not directly deal with database I/O or distributed-system functionalities, the **process view** seems to be an unimportant aspect for analysis. The **physical view** of a library (like spaCy) is also **not** of utmost importance to the developers as libraries are inherently built towards being platform/hardware agnostic. That being said, the physical resources of a system do affect the processing capabilities of spaCy's various NLP models (especially deep neural models).

[11] Matthew Honnibal & Ines Montani: spaCy and Explosion: past, present & future (spaCy IRL 2019), https://www.youtube.com/watch?v=Jk9y17lvltY&list=PLBmcuObd5An4UC6jvK_-eSl6jCvP1gwXc&index=13

[12] Matthew Honnibal & Ines Montani: spaCy and Explosion: past, present & future (spaCy IRL 2019), https://www.youtube.com/watch?v=Jk9y17lvltY&list=PLBmcuObd5An4UC6jvK_-eSl6jCvP1gwXc&index=13

[13] 4+1 Architectural Views, https://en.wikipedia.org/wiki/4%2B1_architectural_view_model

Addtionally, since the spaCy project is open-source, it attaches great importance to a high standard of development and collaboration which seems to form a part of the **development view**. This is evidenced from its nuanced and clear documentation on its code repository [14].

Furthermore, we also present two other architectural views (imporant to spaCy) that do not have an overlap with Kruchten's theory namely, deployment and operational view. The **deployment view** describes the environment in which the system will be deployed. By doing most of the heavy-lifting i.e. by abstracting the code-complexities (of complicated transformer models, dependency-parsers etc.), spaCy projects itself as a developer-friendly library. This is an important consideration for its packaging (of spaCy's various modules) and eventual deployment.The **operational view**, which describes system-operation post deployment, doesn't seem relevant to spaCy as it is deployed as a package that can be updated (for eg: using imports). It's **not** a service which needs regular monitoring.

Finally, the arc42 template [15] gives us food for thought about its **runtime view** and its application to spaCy. Since spaCy has quite a few dependencies (explicit as well as underlying) it is critical to consider the relevance of performance-variations that are well captured in the **runtime view**.

27.3.2 Development View

Chapter 20 of the Software Systems Architecture book [16] describes the Development Viewpoint. According to this chapter, this particular viewpoint describes the architecture which supports the software development process. It describes 6 main concerns.

1. Module Organization: This concerns the organizational part of the code. Arranging code structurally logical will help dependency management.
2. Common Processing: Identifying and isolating common processing into seperate coding modules.
3. Standardization of Design: Using design patterns and off-the-shelf software elements in order to benefit team-work.
4. Standardization of Testing: Creating a consisting approach to speed up the testing process.
5. Instrumentation: Practice of using special logging information code.
6. Codeline Organization: Used to ensure that the system's code can be managed, build and tested.

In terms of a development view for spaCy:

There are some coding conventions[17] which need to be followed whenever a contribution is made. This includes, but is not limited to; loosely following pep8[18], following a regular line length of 80 characters, with tolerance up to 90 characters. Also, they state that all code needs to be written in an intersection of Python 2 and python 3.

They have some information regarding the module organization as well. When someone wants to fix a bug, they first have to create an issue (and check if there is none yet), then create a test file (test_issue[ISSUE_NUMBER].py) in the spacy/tests/regression folder. This also includes the standardization of testing, the tests all have to be put in a certain file and have to be given a certain name. [19]

[14] Github: SpaCy, https://github.com/explosion/spaCy
[15] Arc42 Template Overview, https://arc42.org/overview/
[16] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
[17] Contributions Code conventions,https://gitlab.com/connectio/spaCy/-/blob/master/CONTRIBUTING.md#code-conventions
[18] Style Quide for Python, https://www.python.org/dev/peps/pep-0008/
[19] Contributions Adding Tests,https://gitlab.com/connectio/spaCy/-/blob/master/CONTRIBUTING.md#code-conventions

27.3.2.1 System Decomposition

For analyzing dependencies between different modules in spaCy, we defined the main modules as: *cli, data, displacy, lang, matcher, ml, pipeline, Remainder, spacy, syntax, tokens, website*.

Using the tool Sigrid, we analyzed the dependencies between the components:

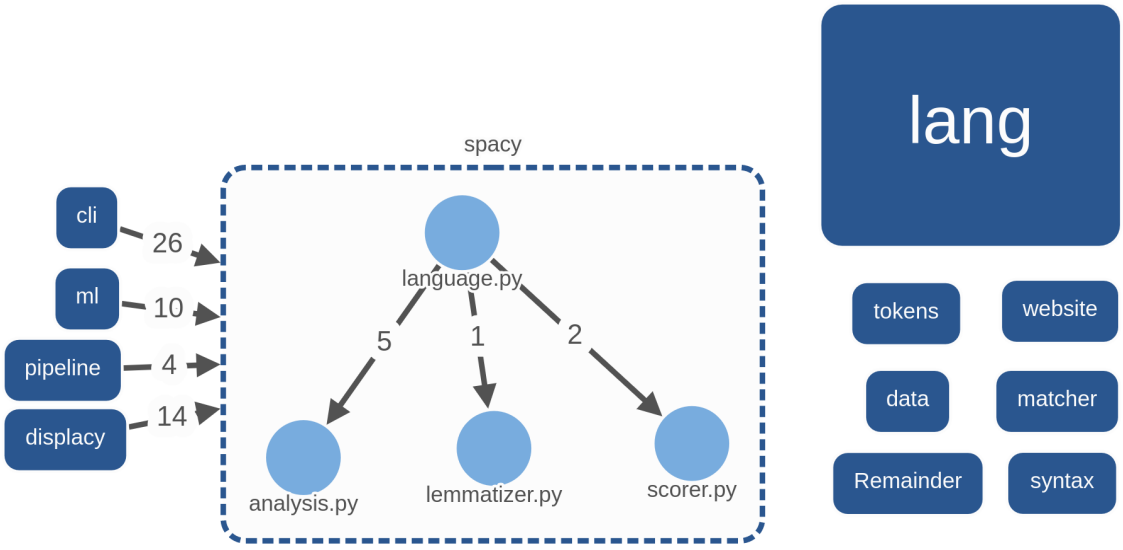


Figure 27.4: System Decomposition

Considering that spaCy is a tool that supports multiple languages the *lang* module is the biggest in code size by far. It contains rules and exceptions for all languages, which inevitably leads to code duplication. Different languages contain different (sub)modules, which makes it difficult to split it up further.

27.3.3 Runtime View

The runtime view describes the behaviour and interactions of system building blocks in the form of scenarios. [20] In this section, only some of the architecturally relevant scenarios are discussed.

27.3.3.1 Pipeline

In order to process a given text with the nlp, spaCy uses a basic pipeline. [21] To this pipeline, custom components can be added in order to make the eventual Doc object more suitable to your needs. In the following section, this basic pipeline is explained.

Tokenizer: Tokenization is one of the most used functions of spaCy, whenever spaCy processes a text, it uses tokenization first. Since this is the most used used function, we will take a look at how this tokenization works. Tokenization first segments the texts into words, punctuation and other characters. This segmentation is carried out by specific language rules. One example they give is that "U.K." should remain as one token, whereas a "." to close off the sentence should be split off.

[20] arc42 Documentation 6 Runtime View, https://docs.arc42.org/section-6/
[21] Language Processing Pipelines, https://spacy.io/usage/processing-pipelines

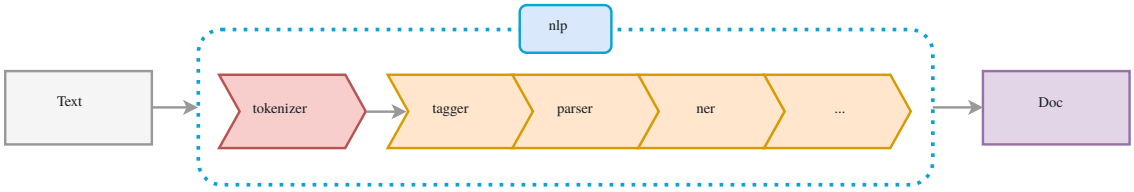


Figure 27.5: spaCy Pipeline

In order to do this segmentation, first the text is split on whitespace characters. Then the program goes through the splitted from left to right, performing two checks on every substring. First it checks if the substring matches a tokenizer rule (U.K. should remain as one, but don't should be split in do and n't). Then it checks if a prefix, suffix or infix can be split off. If there is a match, the rule is applied and the tokenizer continues the loop untill the whole text is done. [22]

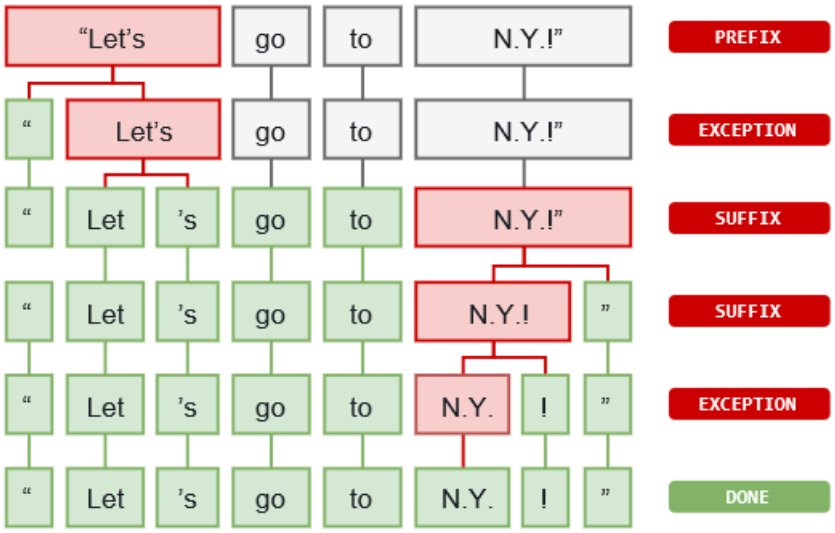


Figure 27.6: spaCy Tokenization

Tagger: After the tokenization, spaCy also makes it possible to parse and tag the newly found tokens with the specific contents. It assigns *part-of-speech* tags such as *verb* or *noun* to the tokens. Additionally it checks for dependencies between the tokens based on existing models. [23]

Named Entity Recognizer (NER): After this tagging and parsing, the pipeline detects and labels named real-world entities. Examples of these entities are "Apple" (*Organisation*) , "U.K."(*Geopolitical entity*) or "$1 Billion"(*Monetary value*). [24]

[22] spaCy 101: Linquistic features, https://spacy.io/usage/spacy-101
[23] spaCy 101: Linquistic features, https://spacy.io/usage/spacy-101
[24] spaCy 101: Linquistic features, https://spacy.io/usage/spacy-101

Components: From spaCy v2.x, the statistical components, like the tagger or parser do not depend on each other, which means they can be switched around, or even removed from the pipeline. However, some of the custom components might need annotations from other components. [25]

Compatibility: spaCy is compatible with the 64-bit CPython 2.7/3.5+ and it runs on Unix/Linux, macOS/OS X and Windows. [26]

27.3.4 Deployment View

The section will describe the deployment process from the time a new major version is developed to the end-user receiving it on their local computer. It also describes SpaCy's runtime requirements.

The maintainers of SpaCy have setup a Continuous Deployment pipeline that that pushes a new version to PyPI (python package index) everytime a new release is made. The infrastructure for making the packaged application available at a single location is maintained by The Python Packaging Authority(PyPA). Once deployed to PyPI, it is available for downloaded through standard Python package managers like pip and conda.

The decision to make SpaCy available through already existing infrastructre for Python package deployment has two main advantages. First, it allows the maintainers to focus improving the core features of the project and second, it reduces cognitive load on the end-user to build the package on their own, including building all related dependencies(of which there could be hundreds!). Most python packages distributed through PyPI have very similar workflow of installation.

SpaCy uses the SemVer policy of versioning[27] with additional labels for pre-release/alpha features. Since SpaCy is quite an old project now (first commit back in 2015), it does not have a regular release cycle[28]. Instead, it keeps end-users engaged by publishing bleeding edge alpha releases through a separate channel, spacy-nightly. These are features that are not fully baked and may not make it to the next major version of the project.

Although there are some system requirements to able to deploy SpaCy, any system that can run Python with enough space to install SpaCy and all of its dependencies should be sufficient. Practically speaking, language modelling and Natural Language Processing is a highly compute heavy task which involves complex operations on large datasets. This could require high-end hardware which are specifically designed for performing such operations.

27.3.5 Trade-Offs and Non-Functional Properties

The modularity of the architecture allows for clear distinctions between functionality and provides centralized data retrieval check-point (single source of truth)

All the C-level functions (written in Cython) are designed for speed over reliability (safety) which is a clear trade-off made for internal code only. This means a program-crash due to an array-out-of-bounds error, although rare, is a bearable consequence for not having strict code-checking that could lead to a higher overhead i.e. slower pipeline.

[25] Pipelines, https://spacy.io/usage/spacy-101#pipelines
[26] Install spaCy, https://spacy.io/usage/
[27] SpaCy Versioning. https://github.com/explosion/spaCy/issues/3845
[28] SpaCy Releases. https://github.com/explosion/spaCy/releases

A lot of the basic data-structures like arrays/vectors are purpose-typed in pure C/C++, which allows for a more elaborate error-checking by the compiler than in python.

Using Cython allows the developers to skip all the extravagant python optimization which usually requires a lot of experimentation to debug and eventually get right. This is an interesting but brave development choice as Cython isn't as ubiquitous and has a fairly steep learning curve.

Matthew believes in iterative development based on active-learning and good tooling that makes experimentation faster; this is essential beacause a lot of the modelling and pipelining in spaCy is quite state-of-the art and often untested in theory. He worries about being able to scale down (rather than up) in order to facillitate faster and easier iterative development.

The developers use A/B evaluation to get quick feedback over small changes.

The people behing spaCy designs around (avoids) using crowdsourced annotations like mechanical turk for improving it's language model due to multiple intermediate problems with this approach. Using it is said to generate low-quality data and makes it difficult tod inspect/interpret results in terms of distinguishing the actual pain-points. I would be unclear whether the wrong data, a poor annotation schema or the model is the limitation. ExplosionAI took the following measures:

- moved annotation in-house
- complex annotation tasks are broken up into simpler/smaller pieces
- usage of semi-automatic workflows

27.3.6 Conclusion

SpaCy is relatively mature library with an extensible architecture that has allowed a rich ecosystem[29] to grow around it.

27.4 Quality and Technical Debt

In the previous blog, we zoomed into spaCy through the lens of the architectural views as described by Kruchten[30]. After discussing the product vision and the architecture, it is time to take a look at the *quality safeguards* and the *architectural integrity* of the underlying system.

27.4.1 Which process is used to ensure quality?

spaCy depends on the community to find bugs or missing functions. Once someone finds a bug or a possible enhancement, they can raise an issue on github in order to notify the maintainers of the bug and try to fix it themselves. spaCy asks their contributors to follow certain steps when contributing [31].

When noticing a bug, first check if it has already been reported. If it has, they "better" leave a comment and continue on that issue, instead of creating a new one. The next thing is to create a test issue, test for the bug and making sure the test fails. The latter seems rather counterintuitive at first. When thinking about it though, the only way to report a bug is to show the bug is there. When the test would pass, the bug would

[29] SpaCy Universe, https://spacy.io/universe
[30] 4+1 Architectural Views, https://en.wikipedia.org/wiki/4%2B1_architectural_view_model
[31] Contributing, https://github.com/explosion/spaCy/blob/master/CONTRIBUTING.md

not be present. Once the test fails, add and commit the test file to the issue. Once the bug is fixed, and the test passes, reference the issue in the commit message. [32]

A bot automatically checks if the contributor has signed the spaCy Contributer Agreement(SCA) which makes sure spaCy may use the contribution across the whole project.

27.4.2 Continuous Integration (CI) Pipeplines: Checks and Config

spaCy uses Azure pipelines to maintain the checks and configurations to be tested during CI. There are two kinds of checks done[33]:

Validate Check: These checks use flake8 on Ubuntu 16.04 and python 3.7 to check code style. They check for errors:

- F821 undefined variable,
- F822 undefined variable in `__all__`
- F823 local variable name referenced before assignment

Module and Functionality Tests:

spaCy is tested to run on the following operating systems:

- Windows: Windows Server 2016 with Visual Studio 2017 - python3.5, python3.6, python3.8
- Linux: ubuntu-16.04 - python3.5, python3.6, python3.8
- MacOS: macos-10.13 - python3.6, python 3.8

All of the above test environments have a 64 bit architecture. They stopped testing on python3.7 temporarily to speed up builds, to this date, testing has not been resumed.[34]

27.4.3 How are checks carried out?

SpaCy uses the pytest testing framework[35]. All tests are collected in a single folder `test` whose structure is similar to the source, except for the fact that the file names are named in the pattern `test_[module]_[tested_behaviour]`. After looking through many of the tests, we found that this is not strictly followed and a majority of the tests are named `test_[module]` and each behaviour is tested as part of a separate test inside the file.

SpaCy has detailed guidelines around writing tests[36], some of the important ones are:

- Tests that are extensive and take too long to complete are marked as slow tests using the `@pytest.mark.slow` decorator. Pytest allows to skip all slow tests.
- The organization of test folders means that we can run all tests for a specific module/directory, if we need it.
- Regression tests are required to be added when a bug corresponding to an issue is fixed. This ensures that future changes will not result in the same bug re-surfacing.

[32] Fixing Bugs, https://github.com/explosion/spaCy/blob/master/CONTRIBUTING.md#fixing-bugs
[33] Azure .yaml file, https://github.com/explosion/spaCy/blob/master/azure-pipelines.yml
[34] Don't test o 3.7 for now to speed up builds, https://github.com/explosion/spaCy/commit/e232356f413f29c2cf9c2ddc86f623bdc05a0ade
[35] spaCy Tests, https://github.com/explosion/spaCy/tree/master/spacy/tests#spacy-tests
[36] spaCy Tests, https://github.com/explosion/spaCy/tree/master/spacy/tests#spacy-tests

Currently, spaCy does not have an automated code-coverage pipeline setup or any code coverage documentation at all. According to our estimates using python library coverage[37], SpaCy has 63% line coverage and 58% branch coverage, with `spacy/cli` module having the lowest coverage of 19%. We make an overview of this below:

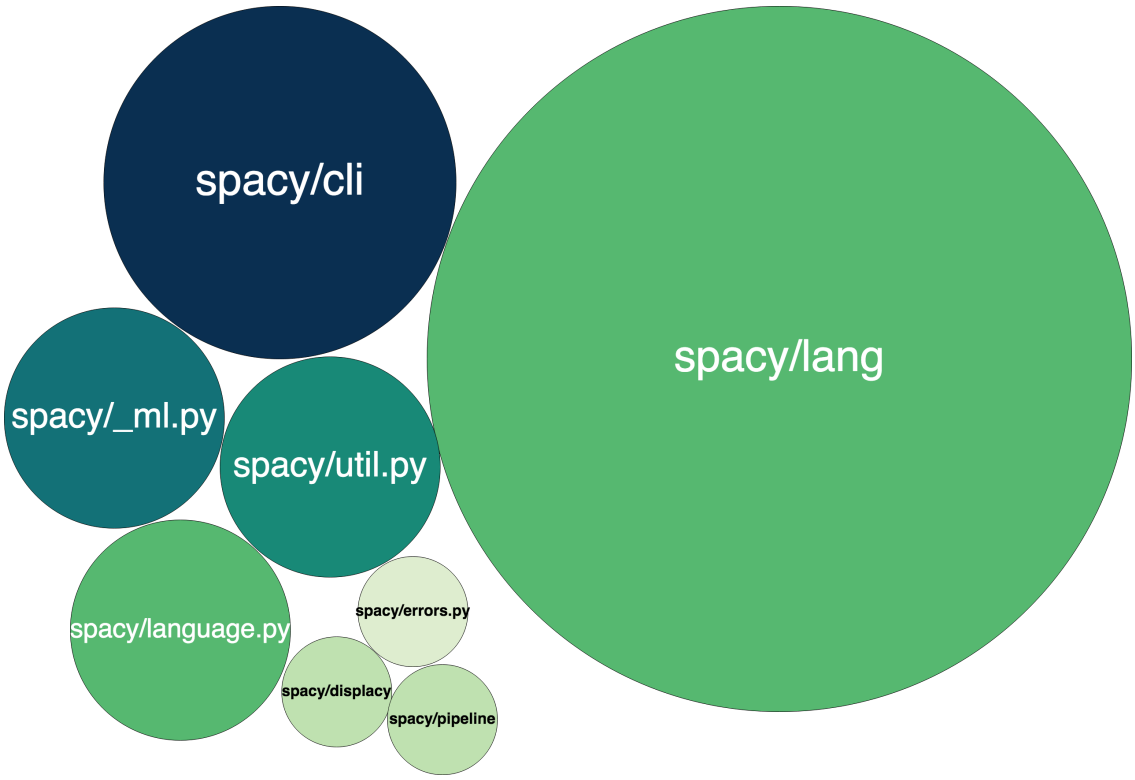


Figure 27.7: An overview of the test coverage. Each bubble is a major module or class whose size is proportional to the number of lines in the code. The darker and bluer a bubble is, the lesser is its coverage. `spacy/lang` which is one of spaCy's largest modules is covered quite well (line coverage of 71%), however `spacy/cli` and `spacy/_ml` have poor coverage metrics (19% and 53% respectively).

> **Sidenote:** While making this infographic, we realized that there is no good open-source visualization tool that can present an overview of module-wise code coverage so this might be an interesting opportunity.

We state the following for reproducibility: To get these numbers, we ran coverage on all files in `spacy/`, excluding `spacy/tests/*`(the tests themselves), checking both basic and slow tests. Specifically, we ran `coverage run --branch --source=spacy --omit=spacy/tests/* -m py.test spacy --slow`.

Based on our limited analysis, a major portion of the low coverage in `spacy/lang` is due to example code. This would better belong in the documentation and improve the code coverage metrics.

In many cases, coverage numbers were low because of literal definitions as in this file. We did find that

[37] Coverage.py, https://coverage.readthedocs.io/en/coverage-5.0.4/

spaCy gives a list of Dos and Don'ts in order to keep the behaviour of tests consistent and predictable. This list gives a summation of the basic conventions which need to be followed as much as possible[38].

The current coverage is only satisfactory and could be improved.

27.4.4 Where are most of the contributors?

To find possible coding hotspots, we decided to take a look at the issue page once again. We looked at all open issues and then zoomed in further with two labels. First we took a look at the most recently updated issues, but this returned mostly bugs which needed to be fixed and were therefore opened less than a day ago. Therefore we decided to take a look at the pinned issues. These issues were opened in december 2018 by ines, one of the maintainers for spaCy. These issues are for the model adding for new languages and making inaccurate pre-trained models more predictable. These pinned issues are still very frequently used, with one of the latest mentions being only *4 hours ago* (at the time of writing). One other issue which is 3rd in line of most commented, is the issue to add example sentences in different languages, for the training of models.

27.4.5 Reducing Technical Debt? How-To...

An assessment for quality and maintainability does not necessarily have to be done manually. There are multiple tools available on the market for creating an analysis and providing refactoring suggestions. For this essay, we will discuss the output of the tools Sigrid and SonarQube. While Sigrid [39] is a commercial tool, SonarQube is open-source in it's most basic variant. For open-source projects like spaCy, it's available for free as a web-app at sonarcloud.io.

27.4.5.1 Analysis by Sigrid and SonarQube

While Sigrid and SonarQube both provide an analysis for maintainability, they are different in multiple manners. Sigrid subdivides maintainability as can be seen in the second table and gives a lower general rating than SonarQube. In addition to maintainability, SonarQube also provides a rating on reliability and security - all of this ratings are based on number of bugs, number of "code smells" and number of "security hotspots" found by SonarQube's analyzer. In the following sections we are discussing some of the suggestions by Sigrid and SonarQube.

Analysis	Rating by Sigrid (0.5 to 5.5 (best))	Rating by SonarQube [40]
Reliability		1
Security		5
Maintainability	2.8 (see below)	5

Analysis	Maintainability ratings by Sigrid
Volume	4.1
Duplication	2.9

[38] Dos and Dont's , https://gitlab.com/connectio/spaCy/-/blob/master/spacy/tests/README.md#dos-and-donts

[39] sigrid says, https://sigrid-says.com

[40] Sigrid uses a star-rating reaching from 0.5 stars to 5.5 stars (best). SonarQube uses a rating from A (best) to E. In the comparison we map letter A to 5 stars down to 1 star for letter E.

Analysis	Maintainability ratings by Sigrid
Unit size	0.5
Unit complexity	2.7
Unit interfacing	2.7
Module coupling	4.9
Component balance	2.0
Component independence	2.2
Component entanglement	5.5

27.4.5.2 Duplications

Both tools provide a list of duplicated code-blocks. Duplicated code can be seen as technical debt - should a developer for example want to improve the implementation of a specific function, he/she has to change it at multiple locations and might not be aware of that fact.

Sigrid shows 100 cases of duplicated code, consisting of 10 to 100 lines of code, sometimes with up to 20 occurences[41]. Most of them are between files with the same filename (in different folders for different languages).

SonarQube provides a better user experience for showing duplicated code than Sigrid, however we have not found any code with it that should be refactored. None of the duplications within the lang-module are duplicated for all languages.

Many of these duplications are in the language-specific part of spaCy. Duplicated code barely contains any algorithms, but mostly just a list of parameters. Unless there is a complete architectural overhaul we consider it unlikely that someone touches code for multiple languages at the same time.

27.4.5.3 Unit Size, Complexity and Interfacing by Sigrid

Sigrid analyzes files and functions/methods according to lines of code, McCabe complexity and number of parameters.

Many of the suggestions by Sigrid are for files like `tag_map.py`, `morph_rules.py`, `tokenizer_exceptions.py` and `norm_exceptions.py`, obviously containing rules and exceptions for different languages. Some of them are huge - e.g. `_tokenizer_exceptions_list.py` with over 15 kLOC. The most complex files in spaCy seem to be `model.js`, `universe.js` and `Scorer.score`.

Sigrid also found the **~400** LOC function `debug_data`. It seems to contain debug code for printing and might be split up into different parts for `ner`, `textcat`, `parser` and `tagger`. This is not related to the big-picture architecture of spaCy, nevertheless our team intends to take a closer look.

Sigrid suggests to avoid functions with many parameters (up to 35 is spaCy for `train.py`). As most of these parameters have default values associated with them, they need not be specified at invocation and we consider those issues unimportant.

[41] https://sigrid-says.com/maintainability/tudelft/spacy/refactoring-candidates/duplication/7251141

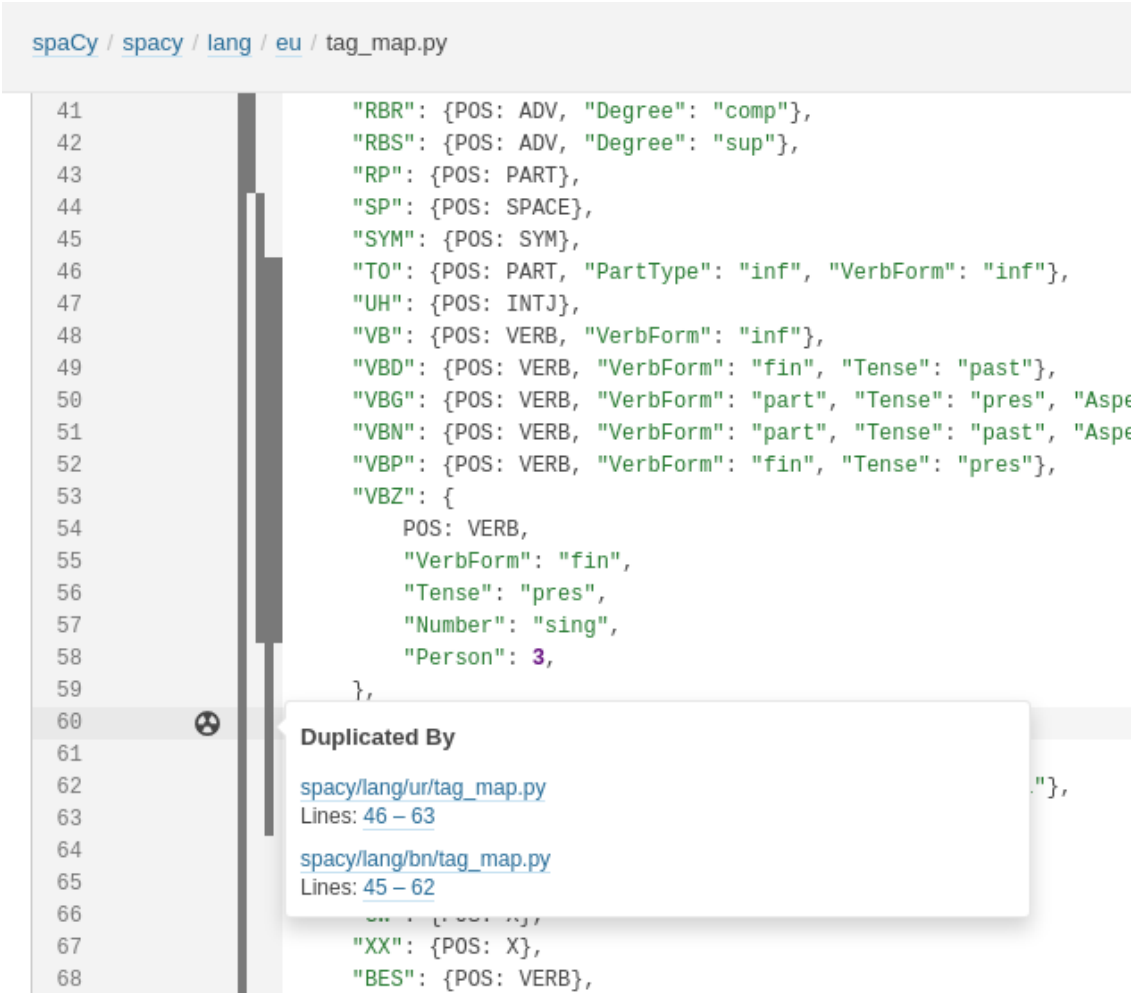


Figure 27.8: Screenshot of SonarQube showing details for duplicated lines

27.4.5.4 Module Coupling, Component Independence and Entanglement by Sigrid

The results for this analyses by Sigrid heavily depends on the compartmentisation that was initially done when importing the project in Sigrid (see previous blog posts). While not necessarily being representative, they still provide some further insights.

Sigrid gives component entanglement the best rating possible, however finds some issues with module coupling and component independence. It detects that `util.py`, `util.js`, `errors.py` are heavily used by different modules, which seems pretty obvious for us humans. However there are also the files `_ml.py` and `compat.py` which are used by multiple components. This is not not clear from the naming of those files.

27.4.5.5 Results

Both evaluated tools provide new insight for the spaCy codebase, even thought some of their functionality like analyzing coding activity was not tested. With the help of Sonarqube we have found a small duplicated if-else branch that we are now trying to fix, Sigrid suggested to change some debug-print-code. We are currently taking a closer look into whether splitting up this functionality into corresponding modules makes sense.

27.4.6 I wish to contribute... where do I sign?

The approach towards integrating new features (as employed by the team behind spaCy) is well documented in the *contributing.md* file on github. This page gives an overview of how the spaCy project is organised and how developers, can get involved. They make a clear distinction between bugs-fixes and feature contributions. Alongwith Matthew Honnibal and Ines Montani (co-founders, ExplosionAI) this repository is maintained by a couple of core contributors namely, Sofie Van Landeghem and Adriane Boyd. Issues are to be submitted based on a custom issue template. The issues are well categorized with carefully curated labels. The team reviews these annotations from time to time, suggesting or adding missing labels to submitted issues as we found out first-hand! There are clear directives and distinctions (of what constitutes spaCy or is a third party dependency) towards opening an issue or making a pull request, which should be followed. A PR needs to be made from a forked repository which additionally requires building spaCy from source using a pre-provided requirements file that helps ensure the added changes do not break the source code.

The proposed code is required to follow `PEP8` style-guide for python. Additional linting and formatting modules (used by spaCy v2.1.0) like `flake8` and `black` are bundled while building spaCy from source. There is an exhaustive documentation available about the code conventions that need to be followed in both Python and Cython related code. In case of a feature development type contributions, individual unit-tests are expected to be made in the pull request in addition to updating the documentation of the related feature.

27.4.7 Making ourselves useful!

As mentioned in our anaylsis of spaCy's product roadmap, the team maintains a separate *'plugins and project ideas'* thread as a proxy for pinning issues related to new feature development. To this end we have proposed that spaCy maintains a separate roadmap for clarity. It has been categorised as a *'good first issue'* by Sofie, which seems to be promising![42]

[42] Good First Issue list, https://github.com/explosion/spaCy/contribute

27.5 Dependencies and Modular Software Design

Research shows that work dependencies – i.e., engineering decisions constraining other engineering decisions – is a fundamental challenge in software development organizations, especially those that are geographically distributed.[43]

Modular Design, the traditional technique intended to reduce interdependencies among components of a system (and between teams developing them), imposes certain limitations in the context of software development. The theory around modular design of software revolves around the assumption that by reducing the technical interdependencies among the modules, task interdependencies are reduced, thereby allowing teams to work in parallel on different parts of the system without needing to communicate among themselves.[44]

There are problems with these assumptions though. Research by Garcia A et. al. suggests that existing modularization approaches consider only a subset of all technical dependencies [45]. Additionally, minimal communication between teams causes variability in the evolution of projects and their subsequent integration. It is also common for software systems to develop or reveal their requirements over time which challenges the determinism of assumptions in the modularity approach. [46] [47]

27.5.1 Socio-Technical Congruence

In the paper by Cataldo et. al.[48] the authors argue that the traditional software modularization is broken and that past work has not taken into consideration, both the technical and work components of dependencies. They propose a new framework for assessing the impact of dependencies on software development productivity called "Socio-Technical Congruence" which shows that development time is reduced significantly when developers' coordination patterns are congruent with their needs.

They do this by conceptualizing a product development project as a socio-technical system, where the technical and the social components need to be aligned in order to have a successful project. The concept of congruence has two components. First, the needs of coordination that are determined by the technical dimension of the socio-technical system and, secondly, the coordination activities carried out by the organization representing the social dimension. If the needs match the activities being carried out, we have congruence!

In order to compute socio-technical congruence, we need deep insight into the repository. We need syntactic dependencies among files, files that each developer modified and coordination instances between developers. In the interest of time, we substitute each of these with our own alternatives, which we discuss in more detail below. The subtitutes are:

---

[43] Cataldo, M. et al. 2007. On Coordination Mechanism in Global Software Development. In Proceedings of the International Conference on Global Software Engineering (ICGSE'07), Munich, Germany, http://casos.cs.cmu.edu/publications/papers/cataldo_p1.pdf

[44] Conway, M.E. 1968. How do committees invent Datamation, 14, 5, 28-31, http://www.melconway.com/Home/pdf/committees.pdf

[45] Garcia, A., et al. 2007. Assessment of Contemporary Modularization Techniques, ACOM'07 Workshop Report. ACM SIGSOFT Software Engineering Notes, 35, 5, 31-37, https://dl.acm.org/doi/10.1145/1290993.1291005

[46] Grinter, R.E., Herbsleb, J.D. and Perry, D.E. 1999. The Geography of Coordination Dealing with Distance in R&D Work. In Proceedings of the Conference on Supporting Group Work (GROUP'99), Phoenix, Arizona, https://dl.acm.org/doi/10.1145/320297.320333

[47] Kraut, R.E. and Streeter, L.A. 1995. Coordination in Software Development. Communications of ACM, 38, 3, 69-81, https://dl.acm.org/doi/10.1145/203330.203345

[48] Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity, https://herbsleb.org/web-pubs/pdfs/cataldo-socio-2008.pdf

- spaCy's architecture with modules INSTEAD OF syntactic dependencies among files
- Issues/PRs related to a module INSTEAD OF files that each developer modified
- interactions on comment threads of issues related to a module INSTEAD OF coordination instances between developers

We can get all this information quite easily from the spaCy website (architecture) and additionally, using the Github API.

We do not intend to be as exact as the authors in computing the socio-technical congruence since that requires a rigorous statistical analysis. This seems to be beyond the scope of this blog post. Instead, we will do a more qualitative analysis of the dependencies.

We first identify the tightly and loosely coupled components of spaCy's architecture, after which we present the communication network of the developers in raising issues / pull requests and interacting on their threads via comments. Finally, we contrast the two and comment on the result in relation to Conway's Law[49]. One would expect the more loosely coupled components of the architecture to have fewer developers in common, whereas a strong network of interactions is expected between tightly coupled components.

27.5.2 Components of SpaCy's architecture

The relationship between modules as seen in the figure act as a fair proxy for syntactic dependencies between files, preserving the same folder/file structure as the actual code while also abstracting each file behind modules. This decreases resulution, but allows for a vizualization that is easier to understand.

***Loosely* Coupled Components:**

- Text Categorizer
- Tokenizer
- Tagger
- Documentation (which is not shown here as part of the architecture)
- Parser

***Tightly* Coupled Components:**

- Doc
- Tokenizer
- Language

27.5.3 Communication in SpaCy

To analyze the communication network in SpaCy, we wrote a few python scripts[51] that use the GitHub API to fetch users who interacted with each other via comments on issues and pull request threads in last 2 years. We filtered issues for labels matching: `feat / doc`, `feat / ner`, `feat / tagger`, `feat / parser`, `feat / textcat`, `feat / tokenizer`, and `docs`. This was done in order to compare the collaboration of users in different parts of the architecture. Different issue labels with their respective colors are shown in the legend of this network. A longe time span for an analysis might lead to discreptancies due to labels and their usage changing over time, which is why we've chosen to analyze two years.

[49] Conway, M.E. 1968. How do committees invent Datamation, 14, 5, 28-31, http://www.melconway.com/Home/pdf/committees.pdf
[50] SpaCy's architecture, https://spacy.io/api#architecture
[51] Python scripts to analyze the communication network, https://gitlab.ewi.tudelft.nl/in4315/2019-2020/desosa2020/-/tree/group-7-essays/projects%2Fspacy%2Fscripts

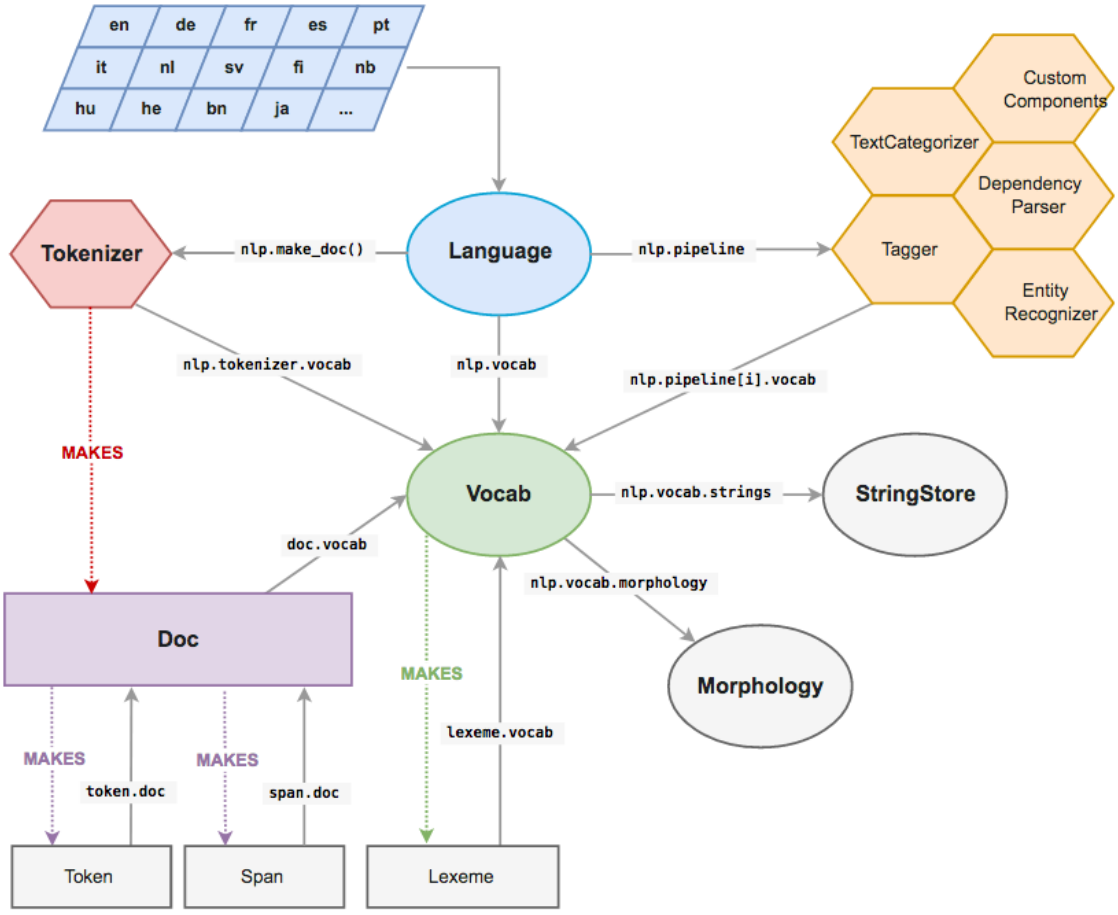


Figure 27.9: Spacy's Architecture - major modules of spaCy and their relationship with each [50]

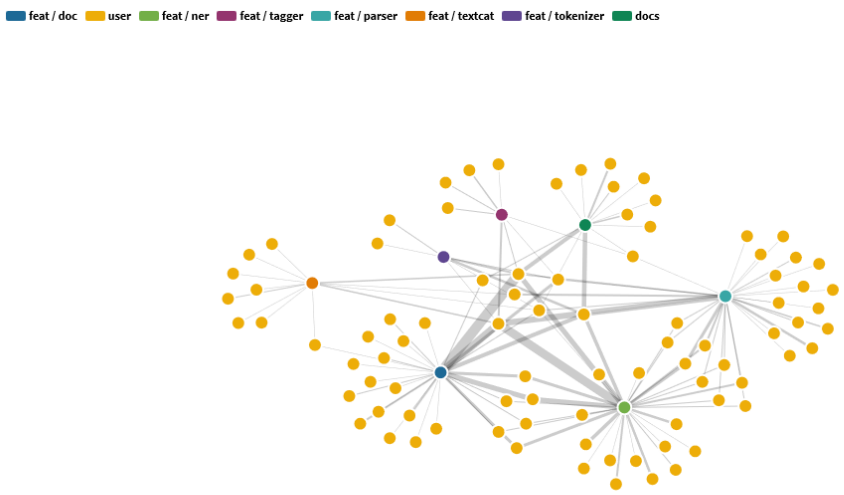


Figure 27.10: Communication Network of Developers in SpaCy

Getting to the visualization itself, each yellow node represents a user and all other nodes are a module from the architecture as shown in the legend. Connections are made *only* between users and modules. A connection signifies that the said user made comments on an issue or a PR thread related to that particular module. The thickness of the connection tells *how many* comment threads the user was involved in. The visualization was developed using Flourish. Click here for an interactive version.

It can be seen from the network plot that there is an emergence of both *in*dependent and *inter*dependent groups working together in spaCy. The loosely coupled components such as Text Categorizer, Tagger, Documentation, and Parser are mostly independent with a few shared collaborators. The tightly coupled components such as the Doc, Tokenizer, and Language have many shared collaborators, but there is no strong pattern. A more interesting pattern that emerges is that the maintainers of SpaCy (the core members which are the user nodes in the center of the image) contribute across modules. This leads us to conclude that there are no obvious roles that each maintainer is playing (or none that are apparent through our analysis, at least).

27.5.4 Conway's Law... Is it still relevant?

Conway Law states that the component structure and organizational structure are in a homomorphic relationship. More than one component can be assigned to a team, but each component must be assigned to only a single team. This means that the organizational structure ends up mimicking the component structure of the software[52].

Based on our analysis, eventhough SpaCy's architecture comes close to obeying Conway's Law, it is NOT

[52]Conway, M.E. 1968. How do committees invent Datamation, 14, 5, 28-31, http://www.melconway.com/Home/pdf/committees.pdf

convincing enough in our opinion. This is mostly owing to the absence of a strict distinction amongst the departments working on different parts of spaCy's architecture. In hindsight, the scenario is better described as a small group of super enthusiastic developers working on almost all parts of the architecture with additional (even bigger) clusters of developers (i.e. contributors) making a few contributions to one or two modules. We believe that Conway's Law still applies to organizations with a strict distinction between departments but not so much in an **open-source** project which favours flexibility in terms of what one can work on.

Chapter 28

Spyder



Figure 28.1: Spyder — The Scientific Python Development Environment

Spyder is an open source IDE (integrated development environment) for python programming language. It offers easy to use editor, debugging tool with unique data exploration features which makes Spyder a great tool for data analytics use and it is included in the Anaconda toolkit by default.

Beyond its built in features, additional features can be added to Spyder using its plugin tool. It can also be separately used as a PyQt5 extension library meaning developers can use Spyder features in their own PyQt application.

28.0.1 Main Components

The most important components of Spyder are as follows:

- *Editor*
- *IPython Console*
- *Variable Explorer*
- *Profiler*
- *Debugger*
- *Help*

28.0.2 Submitted Pull Requests and Issues

- Updating the Wiki page can be seen here *(merged)*
- Adding Persian (Farsi) as new language through Crowdin platform for crowd-sourcing translation *(merge after 98% completion)*
- Correct some mistakes in the code comments here *(merged)*

28.0.3 Team

The project is analyzed by the following team during the Software Architecture course in 2019/2020.

- Soroosh Poorgholi
- Michel Woo
- Pradyot Patil
- Purvesh Baghele

28.0.4 Useful Links

- Spyder-Website
- Spyder-Github

28.1 Your friendly neighborhood SPYDER

Some spiders change colors to blend into their environment. It's a defense mechanism. But the "spider" we are going to talk about changes environment to blend in the user. Spyder is an open-source IDE (integrated development environment) for Python programming language. It offers easy to use editor, debugging tool with unique data exploration features which makes Spyder a great tool for data analytics use and it is included in the Anaconda toolkit by default. Spyder is short for Scientific Python Development Environment.



Figure 28.2: Spyder — The Scientific Python Development Environment

28.1.1 What Spyder tries to achieve

The vision of Spyder is to provide a powerful scientific environment written in Python, for Python developers. It thrives and evolves to offer a combination of the advanced editing, analysis, debugging, and profiling functionality of a comprehensive development tool with the data exploration, interactive execution, deep inspection, and beautiful visualization capabilities of a scientific package.

Moreover, the Spyder IDE has been developed as an extensible framework with many built-in features. Its abilities can be extended even further via its plugin system and API. Furthermore, Spyder can also be used as a PyQt5 extension library, allowing developers to build upon its functionality and embed its components, such as the interactive console, in their PyQt software.

Spyder is included by default in the Anaconda toolkit and runs in the Anaconda environment. In 2018, Spyder became one of the projects funded by NUMFOCUS. NUMFOCUS is a nonprofit organization dedicated to supporting key scientific computing projects; promoting sustainability in the open-source ecosystem; educating the next generation of scientists, engineers, developers, and data analysts.

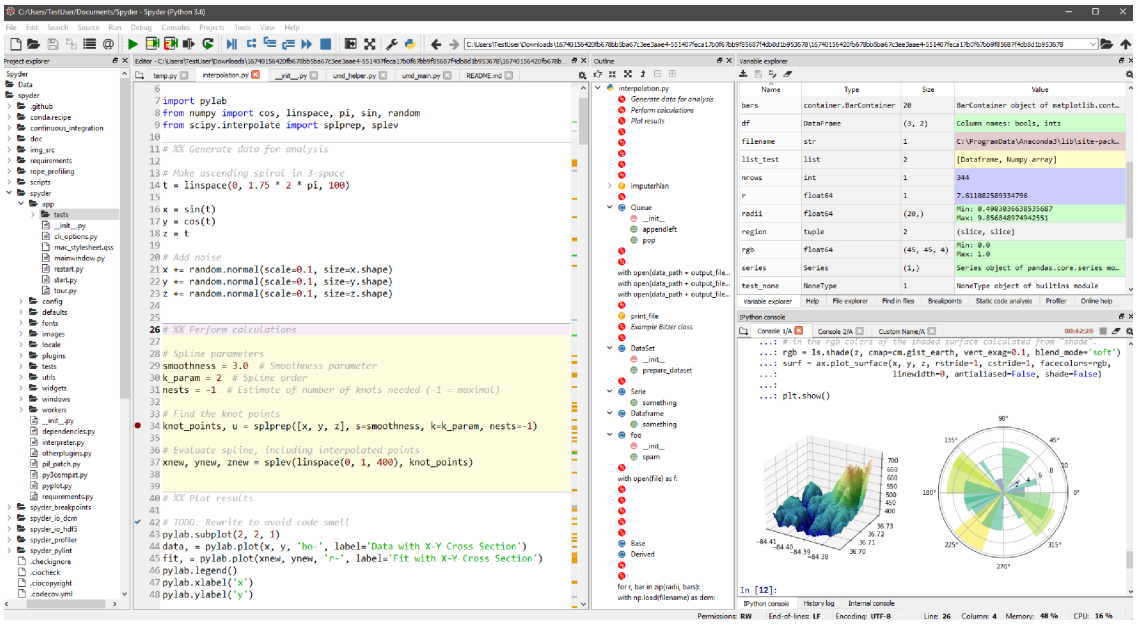
28.1.2 End-user mental model

The end-user mental model can be envisioned as form of application and functionality of the application [1].

The form of the system can be viewed as five main components :

- *Editor*
- *Interactive console*
- *Documentation explorer*
- *Variable explorer*
- *Development tools*

These five components form the integrated development environment of Spyder.



Spyder IDE user interface

The functionality of Spyder mainly focuses on scientific analysis of any kind using the mentioned five components. The end-users for Spyder mainly consist of scientists, engineers, data analysts and students.

[1] James O. Coplien and Gertrud Bjrnvig. 2010. Lean Architecture: for Agile Software Development. Wiley Publishing, link

Hence the Spyder platform continuously evolves and innovates plugins to support statistical analysis carried out in python.

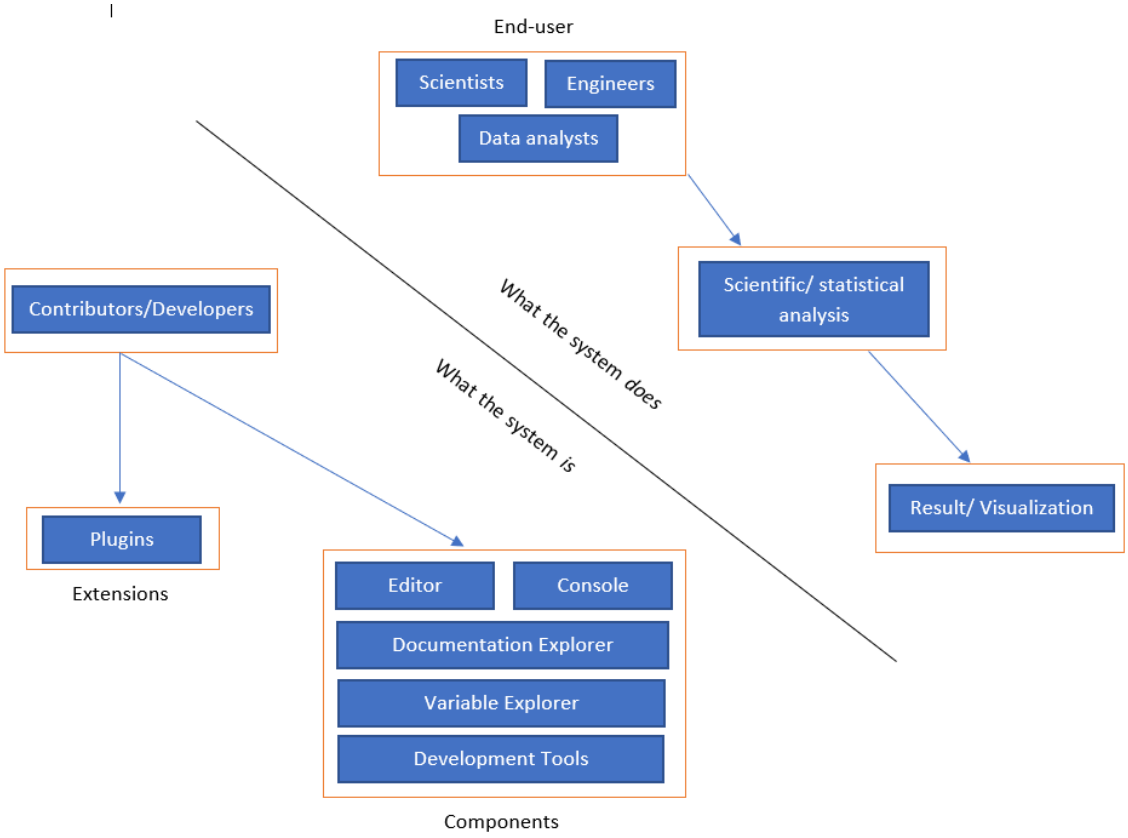


Figure 28.3: Spyder — Simplified End User mental Model

28.1.3 Key Capabilities and Properties

28.1.3.1 Editor

Sypder has a multi-lingual editor that natively incorporates a number of powerful tools for ease of use and for an efficient editing experience. The key features include real-time code and style analysis using pyflakes and pycodestyle, syntax highlighting using pygments, on-demand completion, calltips and go-to-definition features using rope and jedi, a function/class browswer, and horizontal and vertical splitting just to name a few.

Additionally, within the editor, the user can define a “code cell” . A cell can be easily executed all at once in the IPython console which will be discussed below. This feature is much like a cell in MATLAB, however, without need to enable a “cell mode” because in Spyder cells are detected automatically.

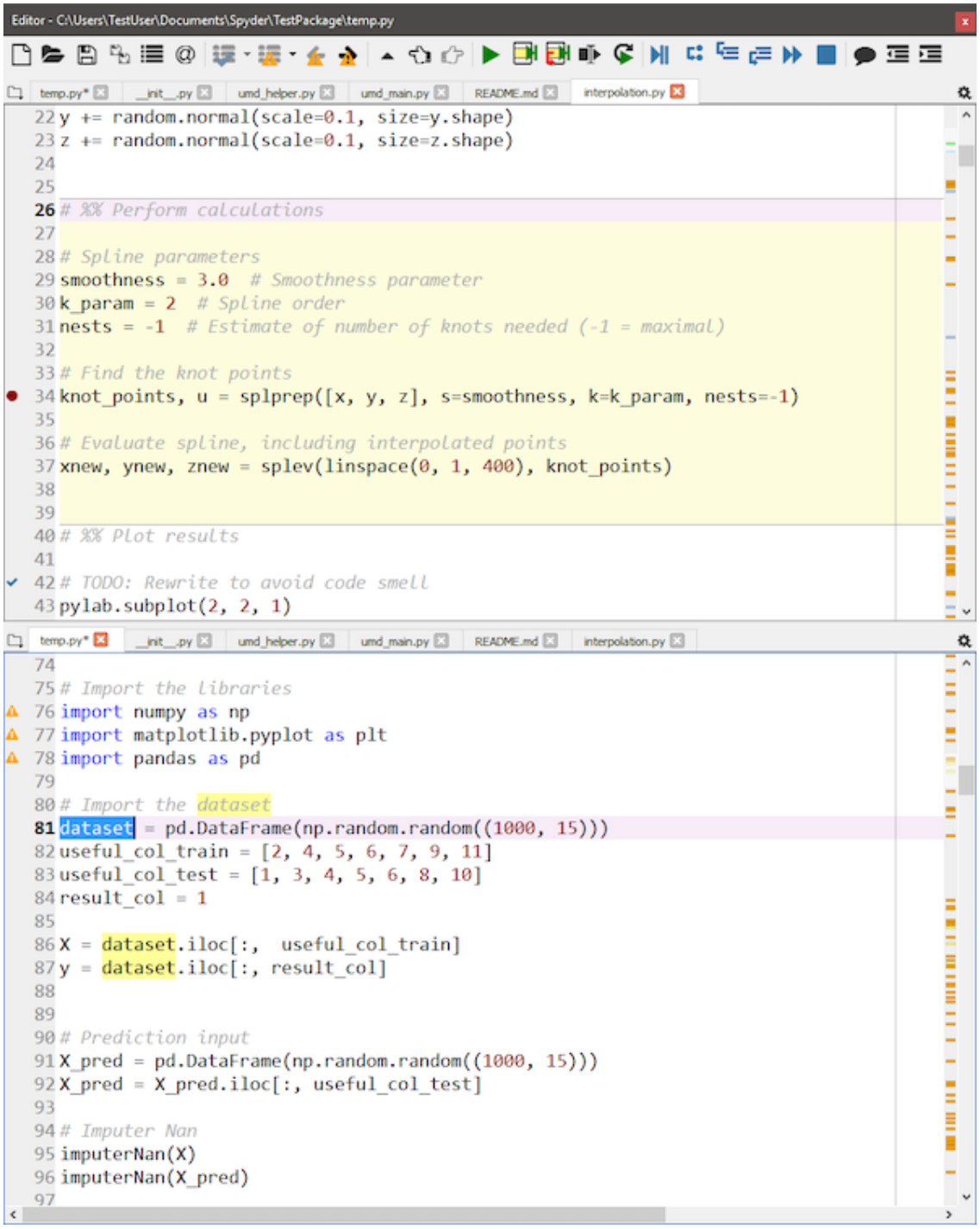


Figure 28.4: Spyder - Editor

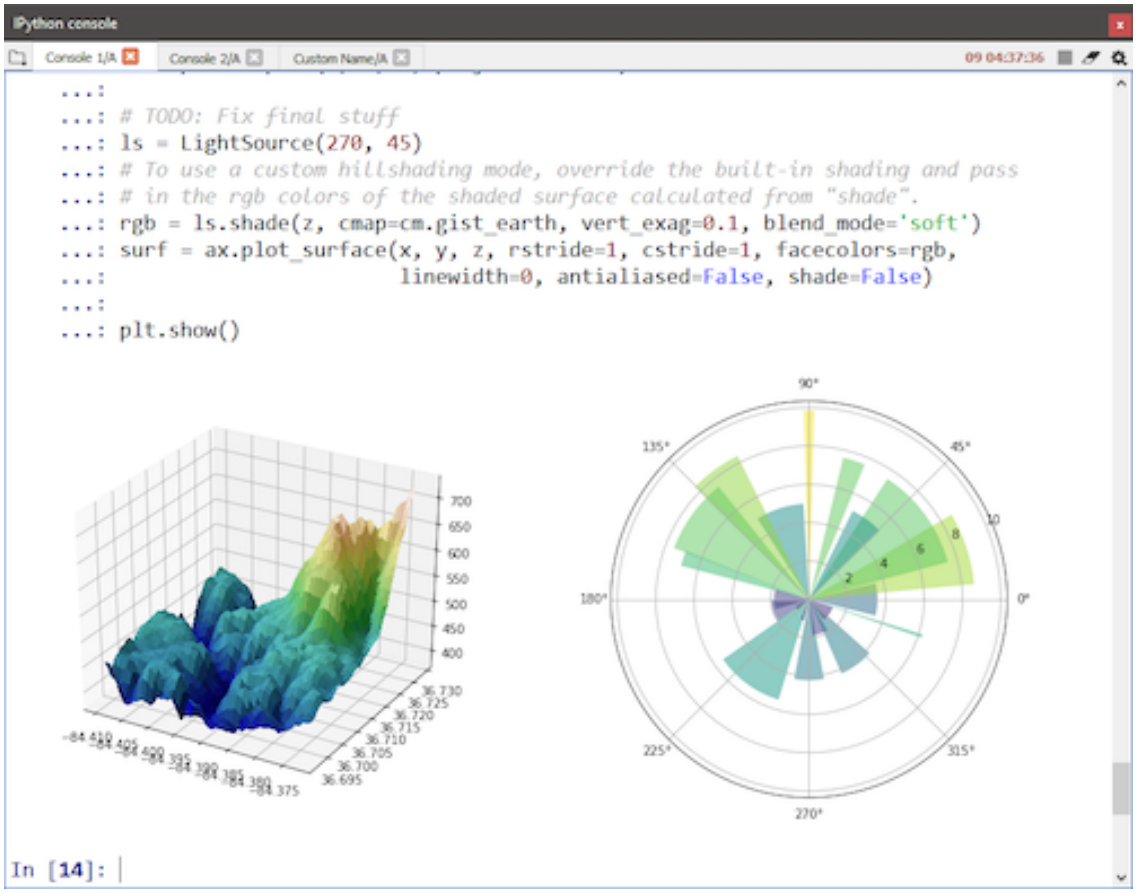


Figure 28.5: Spyder - Console

28.1.3.2 IPython Console

The IPython console is another powerful tool that allows the user to execute commands, interact with and visualize data within a number of fully-featured IPython interpreters. A user can have as many terminal instances that all run in a separate process, allowing them to execute code, interrupt, restart, and terminate a shell without affecting any other process. It also allows for easy testing code without interrupting the primary session.

28.1.3.3 Variable Explorer

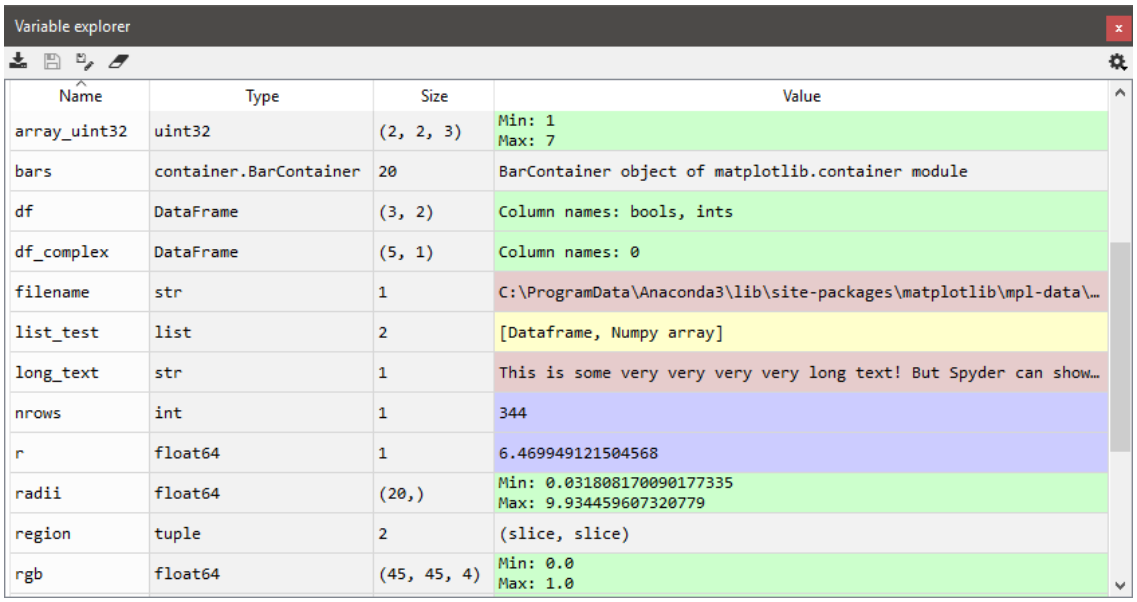


Figure 28.6: Spyder - Variable Explorer

In Spyder, the variable explorer visualizes all the namespace contents; all global object references (variables, functions, modules, etc.) of the current IPython Console. It also allows the user to interact with the contents through GUI-based editors. The variable explorer also offers support for editing lists, arrays, etc. It also allows for instant visualizations of the data as shown in the image below.

28.1.3.4 Profiler

The profiler recursively determines the run time and number of calls for each function and method called within a file. It breaks down each procedure into their individual units. This tool allows the user to easily identify bottlenecks in the code, points at the exact statements that need optimization and will calculate the performance delta.

28.1.3.5 Debugging

Spyder supports debugging through integration with the ipdb debugger within the IPython Console. Ipdb allows breakpoints and execution flow to be visualized within the Spyder GUI as seen above.

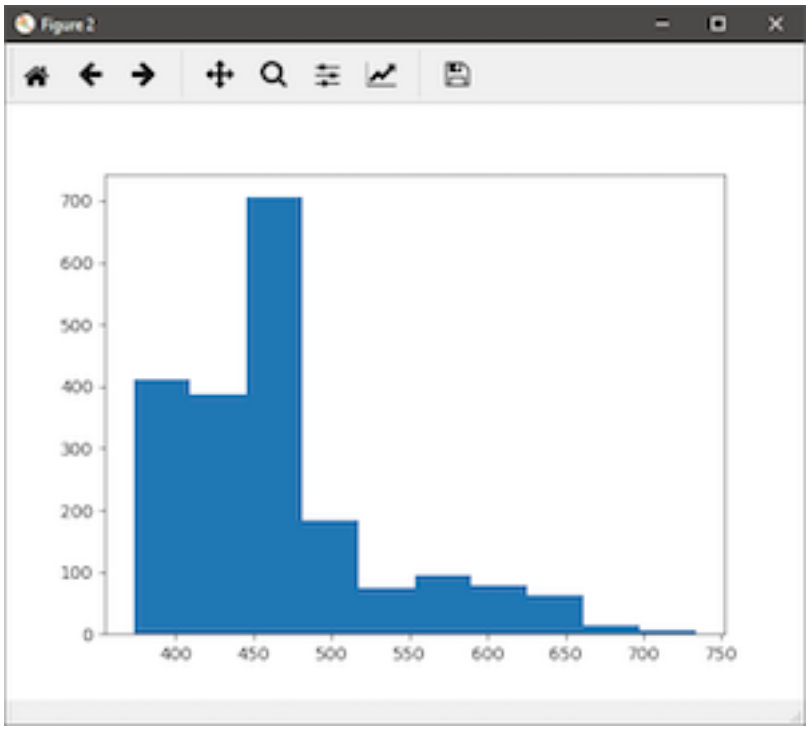


Figure 28.7: Spyder - Histogram

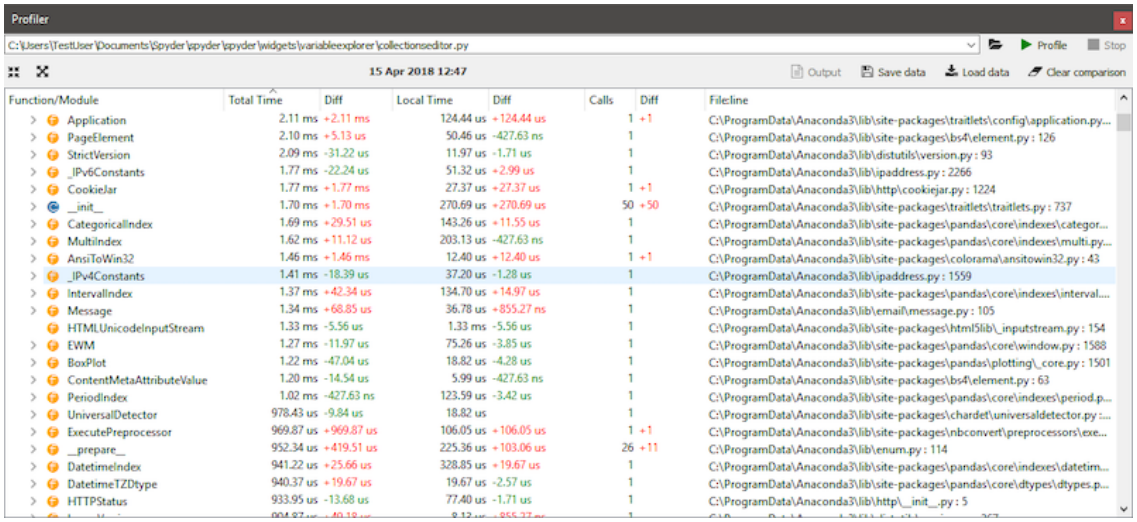


Figure 28.8: Spyder - Profiler

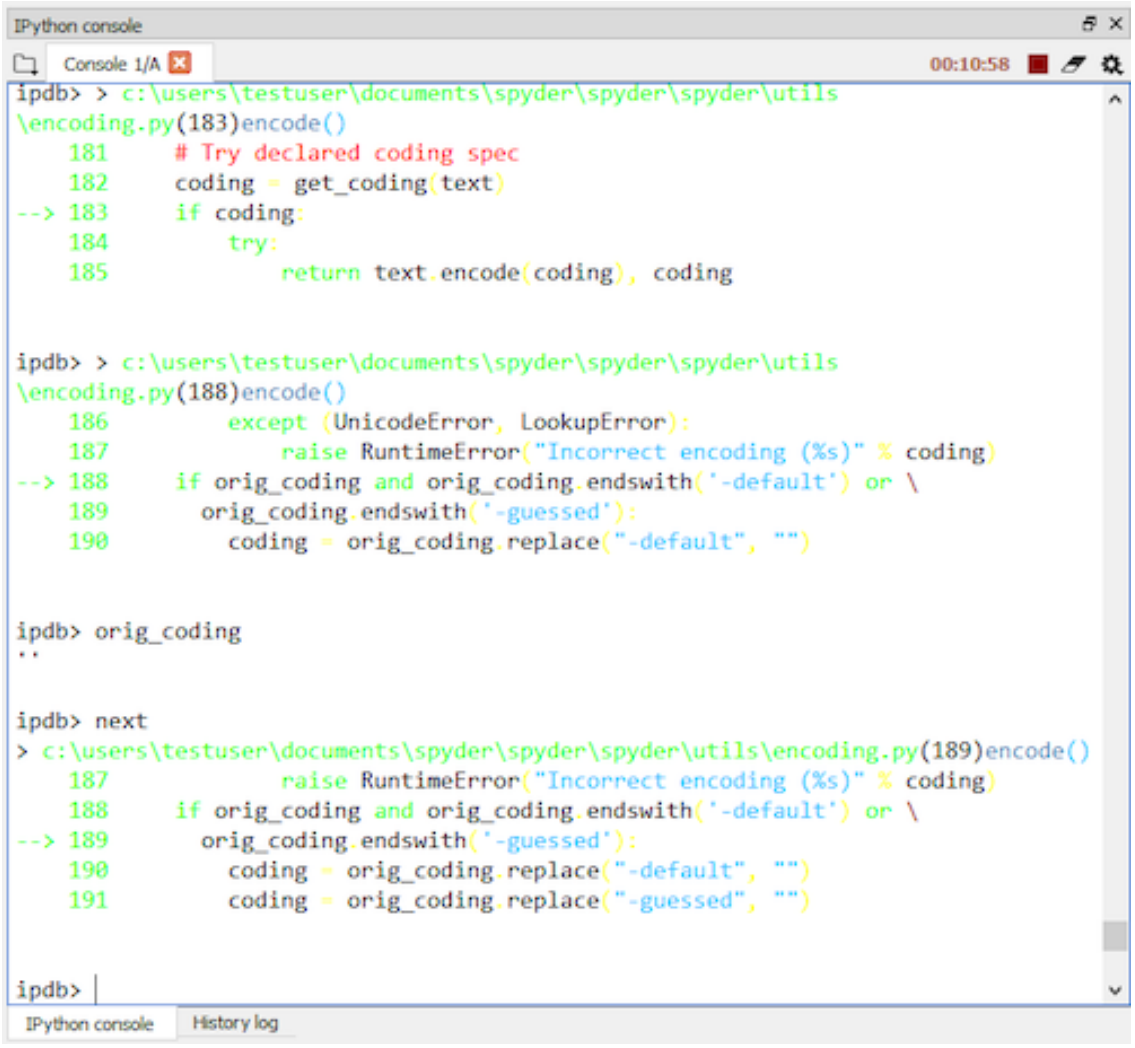


Figure 28.9: Spyder - Debugging

28.1.4 Stakeholder Analysis

The stakeholders has been identified based on the Spyder Official Repository and the official website of Spyder. The stakeholders has been classified using the classes defined by Rozanski and Woods in the book [2].

28.1.4.1 Acquirers

This class of stakeholders are usually responsible for strategic decisions such as planing the road map for the system or search for funding for further development[ˆSoftware Systems Architecture].

According to the Spyder Official Repository Spyder is mainly funded from two main source. The first source of funding is from QUANSIGHT and NUMFOCUS in form of sponsorship. The second source of funding comes from public donation through Open Collective. At the time of writing this report (March 2020) the project got about 310 public contributers and the estimated public contribution in a year is about 6,300 USD.

28.1.4.2 Assessors

Assessors usually monitor the legality concerns regarding the product and make sure that it meets the standards according to the requirements of the system. According to [ˆSoftware Systems Architecture] Accessors mainly come from company internals or from a qualification company.

Since Spyder is an open source software, open to public contribution, they provided a code of conduct in their repository. The project maintainers (Carlos Cordoba and Gonzalo Peña-Castellanos) have put a set of standards (e.g. "*Gracefully accepting constructive criticism*") and it is noted that they are allowed to remove/ edit any contribution and comment in case of violation of these standards. This make these maintainers the Assessors.

28.1.4.3 Communicators

These are the group of stakeholder that explain the system to other parties. It can be explaining for developers or giving out information about the product to the public according to [ˆSoftware Systems Architecture].

For identifying the communicators we analyzed the top maintainers that answer to the comments in the issues section of the repository and in the Spyder Gitter chat. The result shows that (Carlos Cordoba and Gonzalo Peña-Castellanos) are most responsive for communicating with the public. They are also most responsible for assigning bugs and issue fixes to other main developer of the project.

28.1.4.4 Developers

Build and deploy the system from a set of requirement and specifications. They are also concerned with platforms, maintainability, flexibility of the system.

According to the Spyder git insight the top 5 developers (*Based on number of commits*) are:

- Carlos Cordoba
- Pierre Raybaut
- Daniel Althviz Moré
- Gonzalo Peña-Castellanos
- Jean-Sébastien Gosselin

[2] Rozanski, Woods Software Systems Architecture. website. Retrieved February 24, 2020.

By the time of writing this report the project has 149 contributers.

28.1.4.5 Maintainers

Track the status and progress of the system after being deployed and when it is being used by the end user. Maintainers are responsible for development documentation, debug environment, fix issues and preserve the knowledge over time.

The maintainers of Spyder are mainly (Carlos Cordoba and Gonzalo Peña-Castellanos). However some of the questions and bug fixes are also being answered by the other users.

28.1.4.6 Suppliers

Suppliers build and deploy the infrastructure necessary for the product to run properly. According to [^Software Systems Architecture] this group is usually not involved in building, running or using the system and they act more as a bridge between the product and the end user.

Some of the main platforms that are necessary for Spyder to run properly are listed below:

- PyQt GUI development toolset
- CircleCI CI/ CD tool
- Kite Autocompletion of the code

28.1.4.7 Support staff

This group is responsible for providing support in case the end user have any problem with the product. Since Spyder is an open source project, most of the questions and problems are answered by the open source community in platforms such as Stackoverflow, Github, Gitter, etc.

28.1.4.8 System administrators

System administrators run the system once it has been deployed. Since Spyder is an open source project the users are the administrators.

28.1.4.9 Testers

The role of this group is to evaluate the system in order to establish whether or not it is suitable for deployment and use or it needs further improvement before release. As mentioned in the [^Software Systems Architecture]*"Although developers also perform testing, testers should be independent and do not have the same sense of ownership of the system's implementation."*

The maintainers of the Spyder already perform test on their platform. However since it is an open source software, some of the end users might also perform some test to find bugs and report it to the maintainers or send a pull request. In order to merge the pull request, the developed code by the open source community needs to pass the tests which comes with the master branch code.

28.1.4.10 Users

Users define the system's functionality and will ultimately make use of it and their needs should be satisfied by the product. Since Spyder is an open source IDE (Integrated Development Environment) for Python with

more focus on data analysis use cases, most of it's user are people working in the field of data analysis, machine learning.

28.1.4.11 Competitors

A list of some other projects with the same use case as Spyder is mentioned below.

- PyCharm
- Ecilpse
- Microsoft Visual Studio

28.1.4.12 Stakeholder mapping

Can color code the stakeholders based on if you think they are a supporter or a critic.

One of the main reasons to do stakeholder analysis is to categorize different stakeholders in order to properly address their needs. According to [3] the stakeholders can be divided into 4 categories based on their power and interest in the project.

28.1.5 Current context

After Anaconda, Inc's sponsorship for the project ended in mid-November 2017, developer efforts spent on Spyder Plugins were refocused on core Spyder IDE which caused a significant delay in the release of Spyder 4. The latest version of Spyder viz. Spyder 4.0.1 was released in January 2020 which had some improvements over the previous major release of Spyder 4. This release was made possible with the support of Quainsight, Kite, and other open source supporters. The main features of this release include:

- Dark theme for the whole application
- A separate plot pane to browse all inline figures generated by IPython console.
- Integration with Kite completion engine.
- Full support for inspecting any kind of Python object through the Variable Explorer
- Simplified Files pane interface

and many more

Currently, Spyder is funded by contributors on Opencollective. Along with this, the project also receives some financial support from Quainsight and NumFOCUS.

28.1.6 Future roadmap

As we interpret from looking at the issues and release notes, Spyder is quite complete as far as functionality and current need is concerned. A good thing about Spyder is that it incorporates the use of third-party plugins like Spyder Notebook, Spyder Terminal etc. These plugins make Spyder more modular so any functionality that doesn't seem like a core functionality but sounds important can be added as a plugin. An example of this is the inclusion of IPython notebook as a plugin into Spyder. According to the Spyder's github page, the future roadmap includes working towards the development of Spyder 5. Some major tentative features that could be incorporated are:

- Python 3 only support

[3] Aubrey L. Mendelow, Environmental Scanning-The Impact of the Stakeholder Concept, link

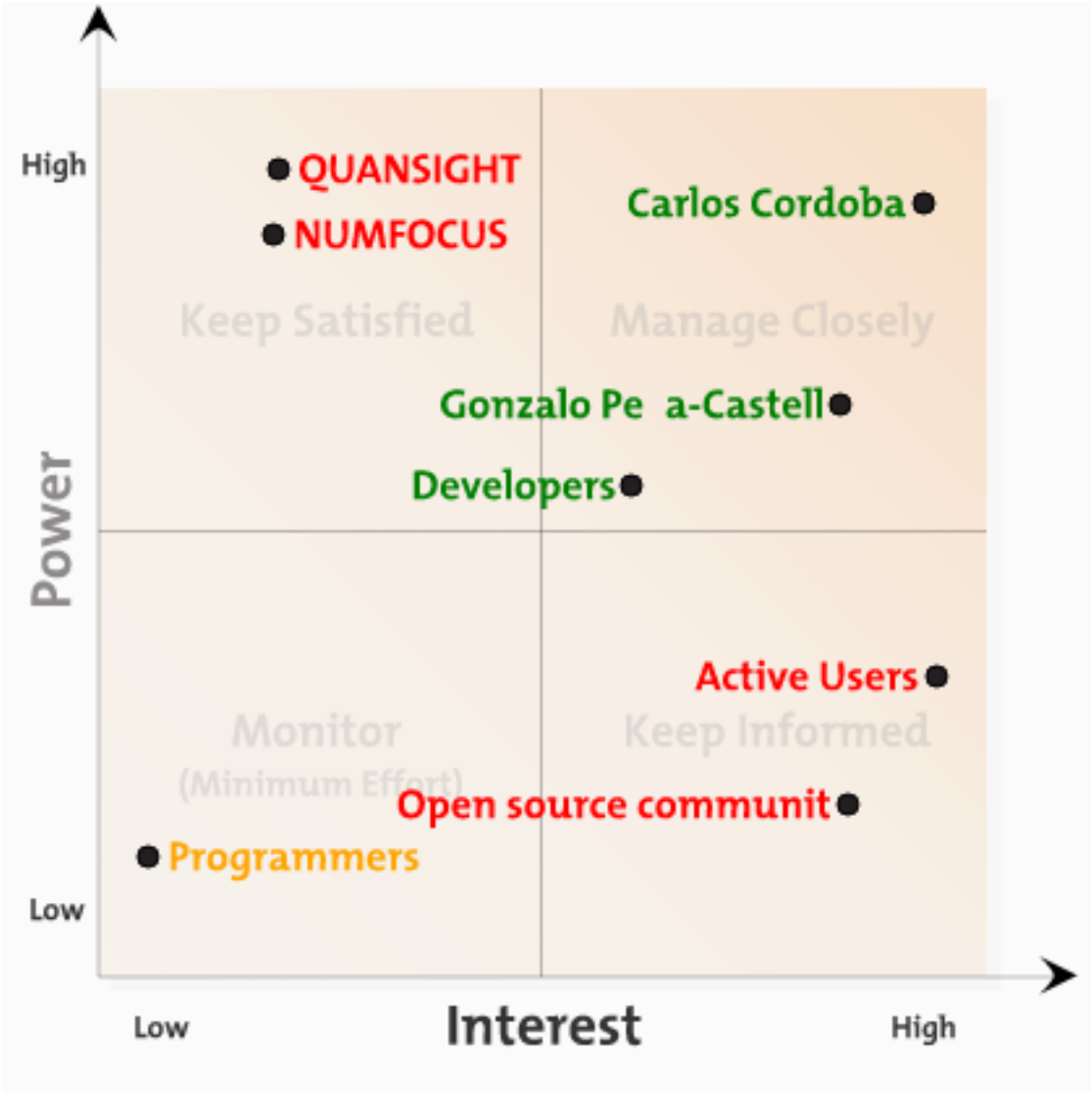


Figure 28.10: Stakeholder Mapping

- New “Viewer” pane to display HTML content
- Improved git support
- docrepr integration.

We would like to end with :

> With great Python, comes great Spyder.

28.2 How Spyder Knits its Architecture from Vision

Hi folks! we are back again with some more insights on your friendly neighborhood Spyder. In our first blog, we talked about the product vision of Spyder which focused on the core elements of the system along with the end-user mental model and analysis of the stakeholders. This essay explores how these elements are realized through its architecture and relationships, and the principles of its design and evolution.

28.2.1 Architectural Views

We start with architectural views in context of Spyder . The fundamental features of a complex system cannot be explained using a single model. Even if one tries to do so, it may happen that the model either becomes too difficult to comprehend or fail to identify the important features of the architecture. It is therefore advisable to break down the system in terms of separate but interrelated views which aim to explain different aspects of the architecture.

There are a number of views that can be used to describe the architecture of a system. Philippe Kruchten[4] designed the “4+1 architectural view model” based on multiple and concurrent views. Furthermore, Rozanski and Woods[5] have also proposed some additional views viz. Context, Functional, Information, Concurrency, Development, Deployment, and Operational views.

We now define the views that are relevant in the context of Spyder :

Now let’s see why these views are relevant in the context of Spyder :

- Contextual view : As we’ve seen in essay 1, there are a lot of stakeholders involved in Spyder and therefore it becomes important to define what the system does and does not do, and how the system interacts with other systems, organizations, and the people involved.
- Process view : Spyder interacts a lot with other dependencies at run time(e.g. PyQt5, Pyflakes etc.). Moreover, Spyder makes use of various plugins(e.g. profiler, IPython console etc.) to build on the main components in the system. Hence, a process view showing all the system processes and how they communicate between each other becomes very important.
- Development view : There are a lot of modules in Spyder. One of the most important stakeholders in Spyder are the developers and the open source community. Therefore it becomes highly essential to have a development view to provide a programmer’s perspective on the system components.
- Deployment view : Spyder is available on various platforms. It comes bundled with Anaconda Python distribution and is also available to run and install individually. Hence Spyder is a system with complex runtime dependencies and complex runtime environment. It is therefore very crucial to address these concerns in the context of Spyder.

[4] Kruchten, Philippe B. “The 4+ 1 view model of architecture.” IEEE software 12.6 (1995): 42-50.

[5] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition. website. Retrieved March 10, 2020.

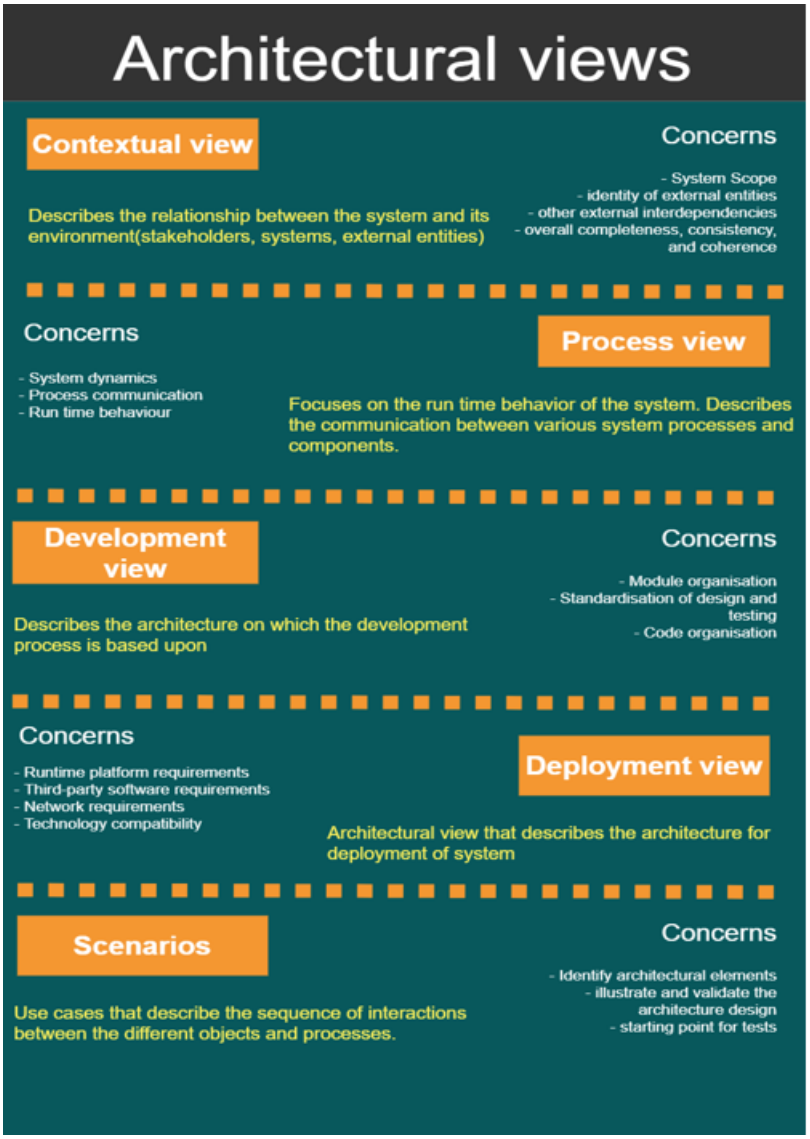


Figure 28.11: Architectural views of Spyder

- Scenarios : Spyder has many types of users and stakeholders. It thus becomes important to visualise how these people interact with the system and how the system reacts to the actions performed by these people.

28.2.2 Architectural Styles and Patterns

Spyder employs an architecture where “plugins” are used to compartmentalize the main components of the whole system which resembles a microservice architecture. Some examples of plugins, as mentioned in the first essay; the editor component, profiler, IPython console, and variable explorer to name a few. Essentially, Spyder is completely comprised of these plugins. Plugins consist of QT widgets (GUI elements) with some extra code to embed the widget into the Spyder application[6]. The architecture of which can be seen in the image below. All these plugins are then called back in the main Spyder library. The manner in which these plugins communicate resemble a microservice system.

Additionally to the built-in plugins, third party plugins are a large part of Spyder’s functionality. Therefore Spyder offers an API that allows the user to create their own or extend the built-in editor, panel, manager etc. [7]. And as seen, in the image below, the API makes calls to all the plugins.

In this way, the overarching architecture style/pattern seen in Spyder is the microservice architecture and can be visualized as the figure below [8].

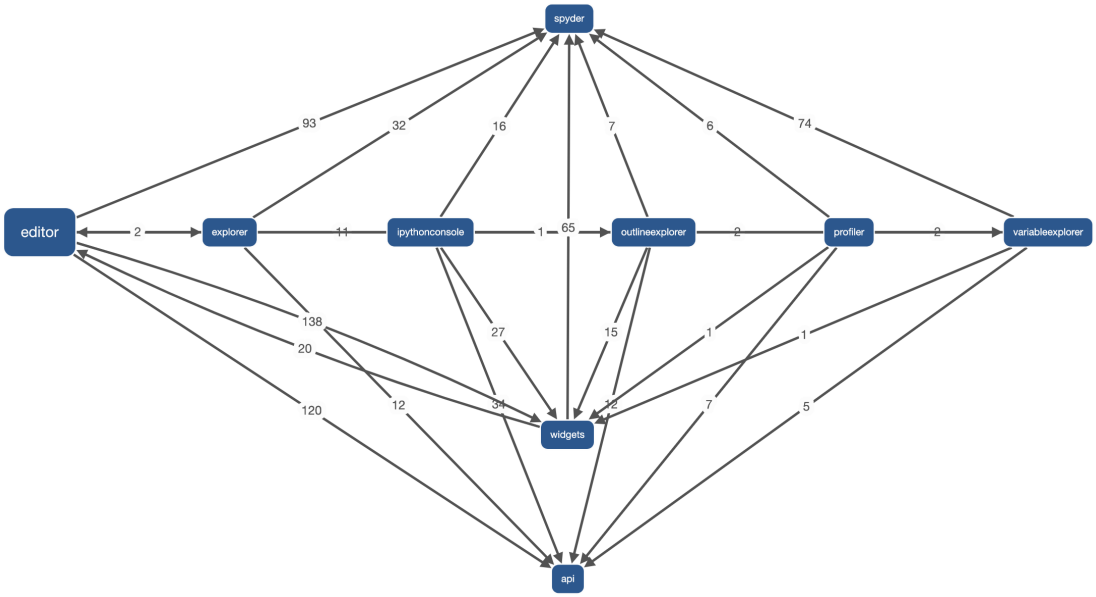


Figure 28.12: Architecture of Spyder

[6]Spyder Wiki. website. Retrieved February 24, 2020.
[7]Spyder Wiki. website. Retrieved February 24, 2020.
[8]Software metric analysis. website

28.2.3 Development view

The main goal of the development view is to describe the architecture that the software is based on and analyze how it is developed. In the following sections we dive deeper into the system decomposition of Spyder and how they perform test for their codebase.

28.2.3.1 System Decomposition

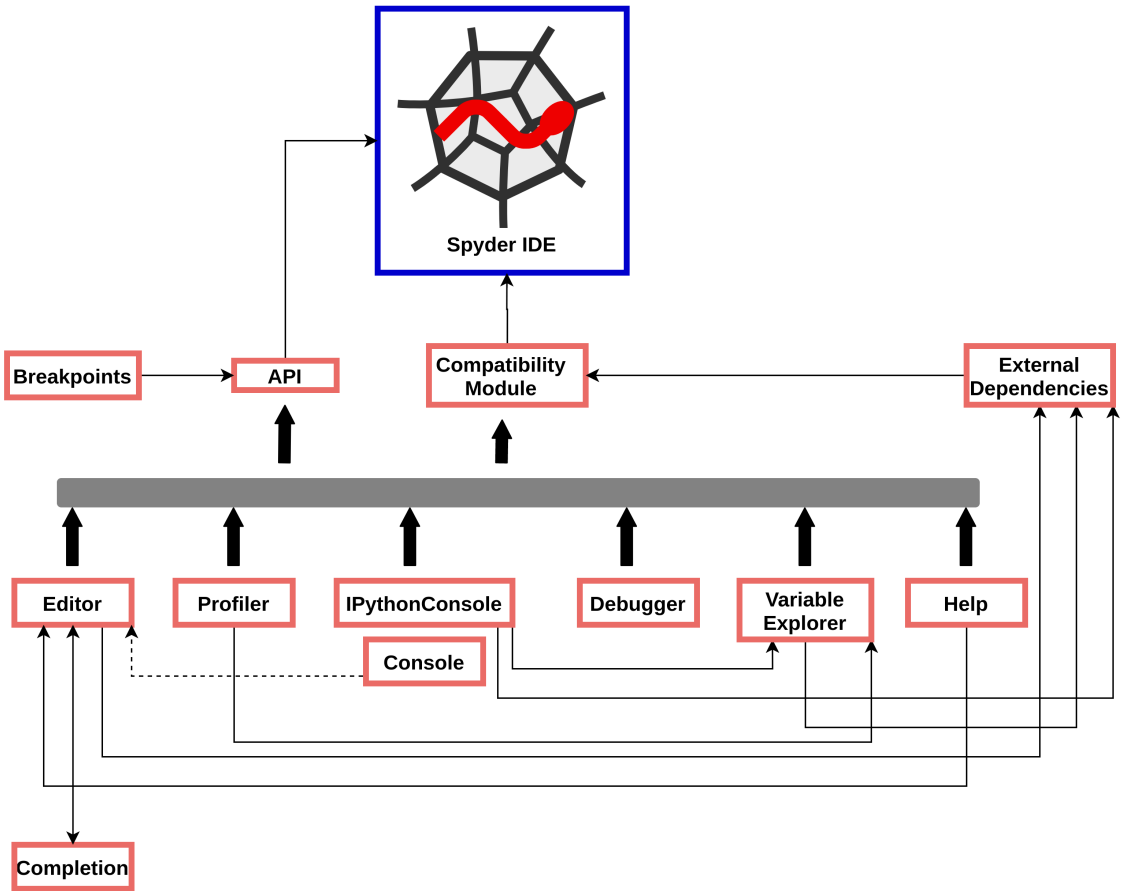


Figure 28.13: Main modules of Spyder

In the figure above the main modules of Spyder and their and their relations can be seen. The codebase analysis has been inspired by Sigrid. In the following section a more detailed description of the main modules are given.

- Compatibility Module

This module makes sure that all the developed code in different modules are compatible between Python 2 and Python 3. This module is dependent on other built-in modules from Python such as *os, sys and operator.*

- Editor

This module is the editor in the IDE. As can be seen in the figure above it is in middle layer between the IDE and the sub modules it inherits. Not all the submodules (e.g. code completion) are mentioned here to avoid clutter. This module is mainly based on PyQt module which is mainly for GUI design.

- Profiler

This module is responsible for analyzing the code performance and finding the bottlenecks. This module also uses QVBoxLayout routine from the PyQt widgets as a 3rd party module.

- IPythonConsole

This is a third party module imported for running the Python code. It is used as a plugin inside the Spyder. It has 3rd party dependencies such as jupyter_core, jupyter_client, qtconsole and zmq for ssh connection from the application to the IPythonConsole.

- Console

This is the internal console in the Spyder which can be used instead of the 3rd party IPythonConsole. It is dependent on PyQt, os, sys, logging libraries which the last three are built in in Python.

- Debugger

This is the Spyder debugger which can be used to debug the code or execute it line by line. This module calls the breakpoints from the API manager and it passes them to the Python built in debugger pdb.

- Variable Explorer

The variable explorer enables inspecting the normal and nested code variables on the fly. It used some remote settings from 3rd party libraries and it gets called from the consoles as can be seen in the figure above.

- Help

The help plugin is implemented and shown in the Spyder using a web view. For this it is using WebEngineWidget from PyQt among some other features from this library.

- API

This package includes some of classes which can be used to create 3rd party plugins and extend Spyder. It is still considered pre-released until the developers release Spyder 4.0 officially.

28.2.3.2 Testing

For testing Spyder the developers designed test file separately for each plugin in their specified folder. According to the Sigrid also the test to code ratio for Spyder is 25%. Also for every pull request some test are run automatically. There is also a *runtest.py* file provided by the developers for open source community. The goal is to pass the test after making changes to the code and before submitting it. This script will run some test automatically on the CI server. One can also use the pytest module and write their own test code which provide more flexibility for the developer. There is also a debug flag argument to run the code for the main IDE in case one wants to debug and see the debug prints with different Verbose.

28.2.4 Deployment view and Run-time view

After development of Spyder, we need to deploy it and run it. So, let's have look at Deployment view and Run-time view. Rozanski and Woods [9] defined deployment view as "aspects of the system that are important after the system has been built and needs to be validation tested and transitioned to live operation". A Deployment view is useful for any information system with a required deployment environment that is not immediately obvious to all of the interested stakeholders.

28.2.4.1 Deployment of Spyder

For using and installing Spyder, the Spyder development team recommends to use Anaconda Python distribution, which comes with everything Spyder needs to get started in an all-in-one package. However, it is also possible to install and run Spyder individually on popular operating system such as Windows, Linux and macOS. As an alternative to Anaconda one can also use miniConda, which just a lighter version of Anaconda. For the sake of simplicity and to avoid specificity of different OS and distributions, we consider the recommended Anaconda distribution as a *run-time Software* requirement for Spyder.

The above figure gives a concise overview of Spyder's deployment view. The Anaconda distribution is available on Windows, Linux and macOS, which makes Spyder available on these platforms too. Since most of systems today have the required configuration for installing a Python distribution, the *hardware requirements* pretty much depends on the task an individual wants to accomplish using Spyder. As far as *network requirement* is concerned, Spyder requires a network connection to check and download updates for improved experience.

28.2.4.2 Run-time view of Spyder

The Spyder IDE depends on numerous third party dependencies. The Anaconda distribution takes care of managing these dependencies for Spyder. These dependencies constantly interact with each other at run-time. The dependencies include [10] :

The runtime view shows an overview of all systems and nodes that are involved when Spyder is running. It is a schematic overview of the aforementioned runtime requirements, third party dependencies and network requirements. The figure below shows the run-time view of Spyder. Parts of the diagram are based on an overview of the components and layers in the scientific computing environment for Python explained by Robert Johansson [11].

28.2.5 Spyder adapting to improve non-functional properties

In the first blog we discussed the product vision of Spyder and the functionality it provides to meet the goals. In this blog up till now we discussed the architectural style, development view, run-time view and deployment in context of Spyder to support it's functionality. However, design choices and trade-off have to be made to satisfy non-functional properties of software. A non-functional property (NFP) of a software system can be defined as a constraint on the manner in which the system implements and delivers

[9] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition. website. Retrieved March 10, 2020.

[10] Spyder Docs. website. Retrieved February 24, 2020.

[11] Robert Johansson. Numerical Python, chapter 1, Introduction to Computing with Python pp 1-24. website. Retrieved March 10, 2020.

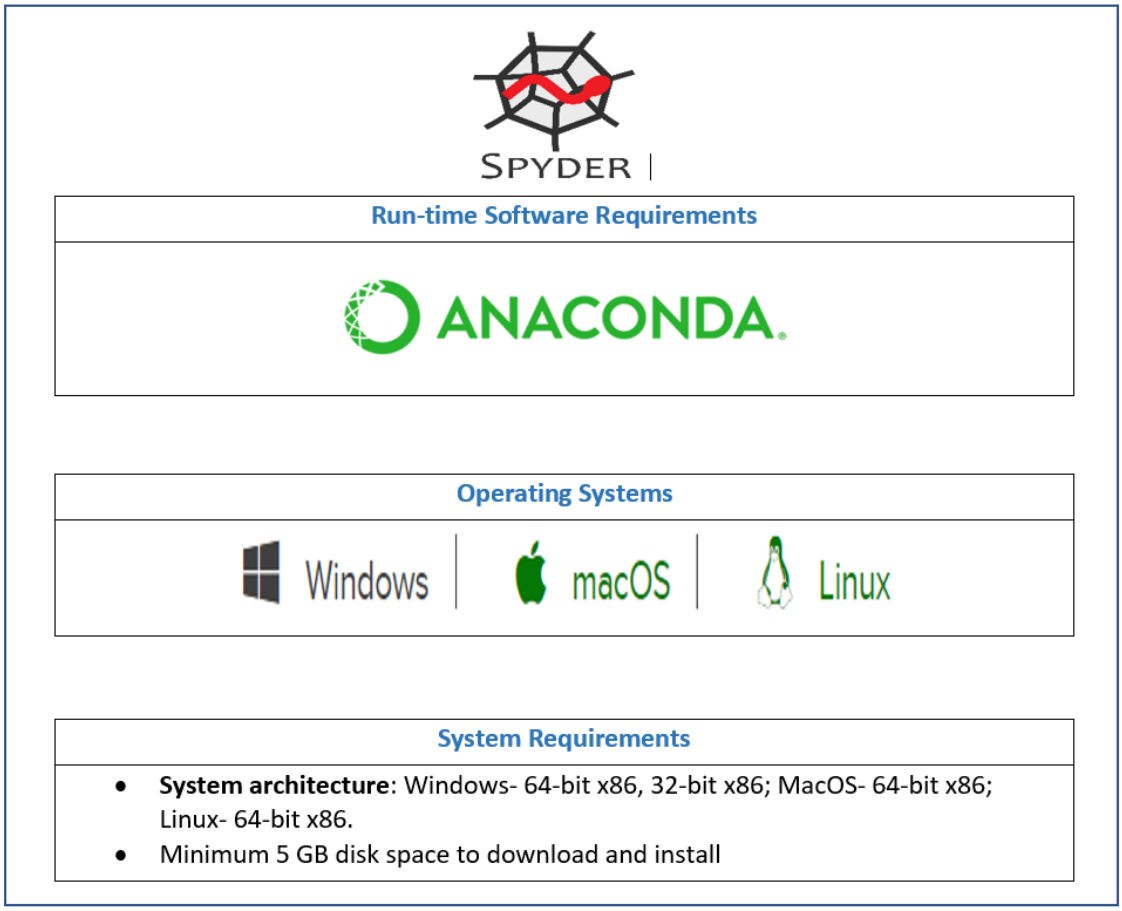


Figure 28.14: Deployment view of Spyder

- Python 2.7 or >=3.3
- PyQt5 >=5.5
- Qtconsole >=4.2.0 – for an enhanced Python interpreter.
- Rope >=0.9.4 and Jedi >=0.9.0 – for code completion, go-to-definition and calltips in the Editor.
- Pyflakes – for real-time code analysis.
- Sphinx – for the Help pane rich text mode and to get our documentation.
- Pygments >=2.0 – for syntax highlighting and code completion in the Editor of all file types it supports.
- Pylint – for static code analysis.
- Pycodestyle – for style analysis.
- Psutil – for memory/CPU usage in the status bar.
- Nbconvert – to manipulate Jupyter notebooks on the Editor.
- Qtawesome >=0.4.1 – for an icon theme based on FontAwesome.
- Pickleshare – To show import completions in the Editor and Consoles.
- PyZMQ – To run introspection services in the Editor asynchronously.
- QtPy >=1.2.0 – To run Spyder with different Qt bindings seamlessly.
- Chardet >=2.0.0– Character encoding auto-detection in the Editor.
- Numpydoc Used by Jedi to get return types for functions with Numpydoc docstrings.
- Cloudpickle Serialize variables in the IPython kernel to send them to Spyder.

Figure 28.15: Dependency list of Spyder

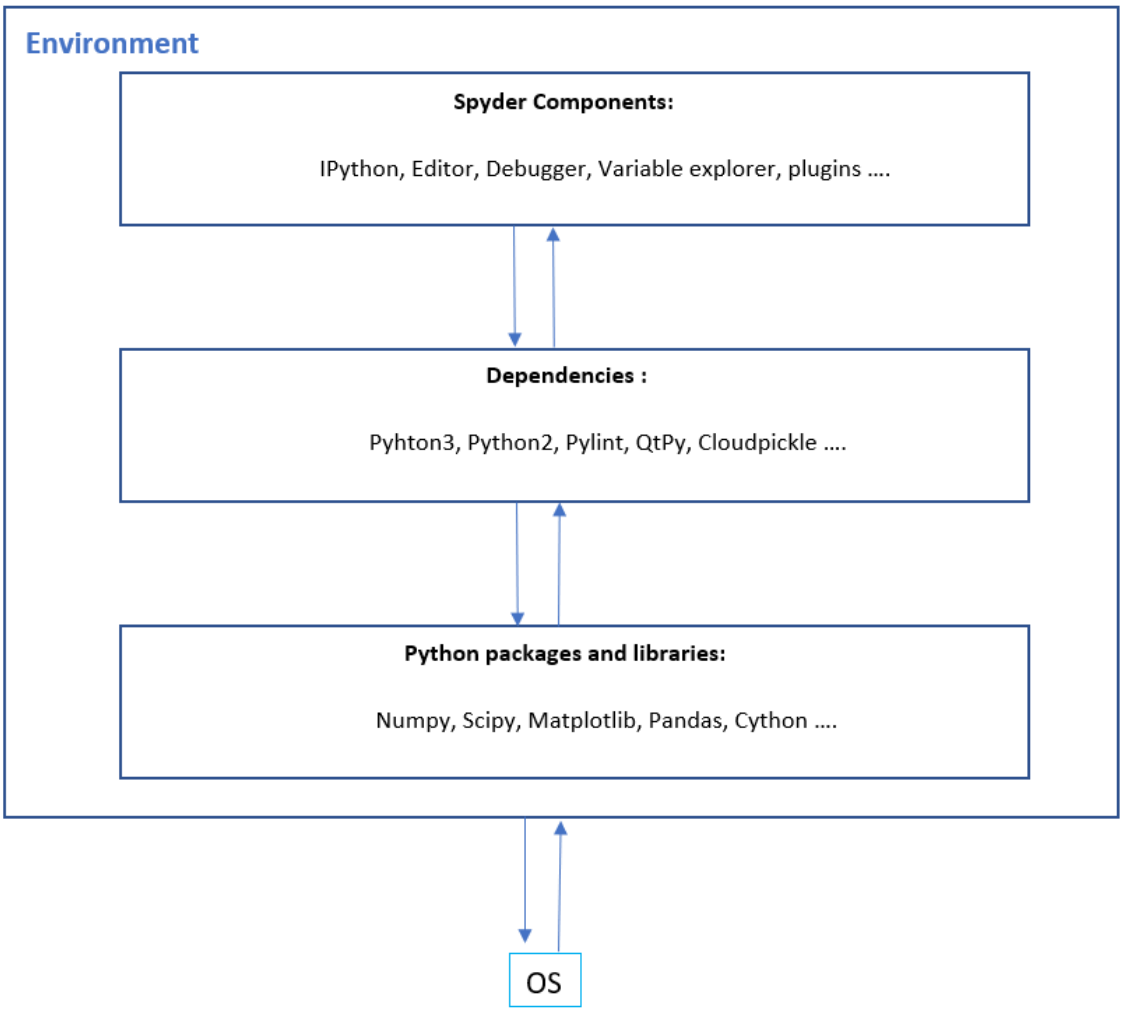


Figure 28.16: Run-time view of Spyder

its functionality [12]. Some of the several NFP's are efficiency, complexity, scalability, adaptability and dependability.

Over the years Spyder team has made some design choices and trade-off's to satisfy the non-functional properties. Despite the desire to serve the end-users with Spyder distribution themselves, to increase the adaptability and maintainability, the Spyder team released the stable version 4.0.1 as an anaconda distribution and recommends it rather than different distributions for different platforms. Moreover, to improve the efficiency and decrease the software complexity, the Spyder team implemented api gateways in version 4.0.1 for separating the components and manage the interaction between them. Like most of the softwares, Spyder has it's continuous integration (CI) pipeline. The CI pipeline is supported by CircleCI, Azzure pipelines and Codecov which run test scripts to monitor the code quality.

28.3 Analyzing the Quality of Spyder

Hi folks! We are back again with more insights on your friendly neighborhood Spyder. This time, we are going to focus on how Spyder safeguards the quality and architectural integrity of the underlying system.

28.3.1 Continuous Integration of Spyder

Spyder has employed three Continuous Integration (CI) pipelines namely CircleCI, TravisCI and Azure pipelines which run test scripts to monitor the code functionality. Moreover, Spyder also uses Codecov and Coveralls to monitor the coverage of test scripts.

Now that we have seen the CI pipelines used by Spyder, you might be thinking : **What do the test scripts associated with these CI pipelines check/cover?**

Since Spyder is almost 100% developed in Python, it uses Pytest and unittest frameworks which enables writing small test scripts for each component. The Spyder team follows a very simple convention, each component has a *test* folder which contains all the test scripts concerning that component. In general, almost every component has a test script which cover scenarios that the user might encounter while interacting with that component. For example, the Spyder editor [13] component has tests to check : *auto save errors, auto-indentation feature, breakpoint feature etc*. Similarly, other components have test scripts to check their functionality respectively. We won't be talking about tests concerning other components because they are quite monotonous and can be easily understood by reading them.

Moreover, there are some checks that are conducted on all the components working integrated. For example, Spyder has tests script for dependency checks to check if all the dependencies required for Spyder are installed in the environment. Similarly, there are tests for Python compatibility checks for *Python 2* and *Python 3*.

However, there still remains a question : **Why does Spyder have three different CI pipelines?**

Well, according to the Spyder team (we explicitly asked them on Gitter), the reason behind this are issues with Operating Systems (OS) and Python distribution. Spyder distribution is available on Windows, Linux and macOS. So, for each of the following combinations there is a different CI pipeline :

- Azure pipelines -> macOS, Windows

[12] Nick Eric M. Dashofy, Nenad Medvidovic, Richard N. Taylor. Software Architecture: Foundations, Theory, and Practice. Wiley, 2009. website. Retrieved March 16, 2020.

[13] Editor, Spyder repository. website

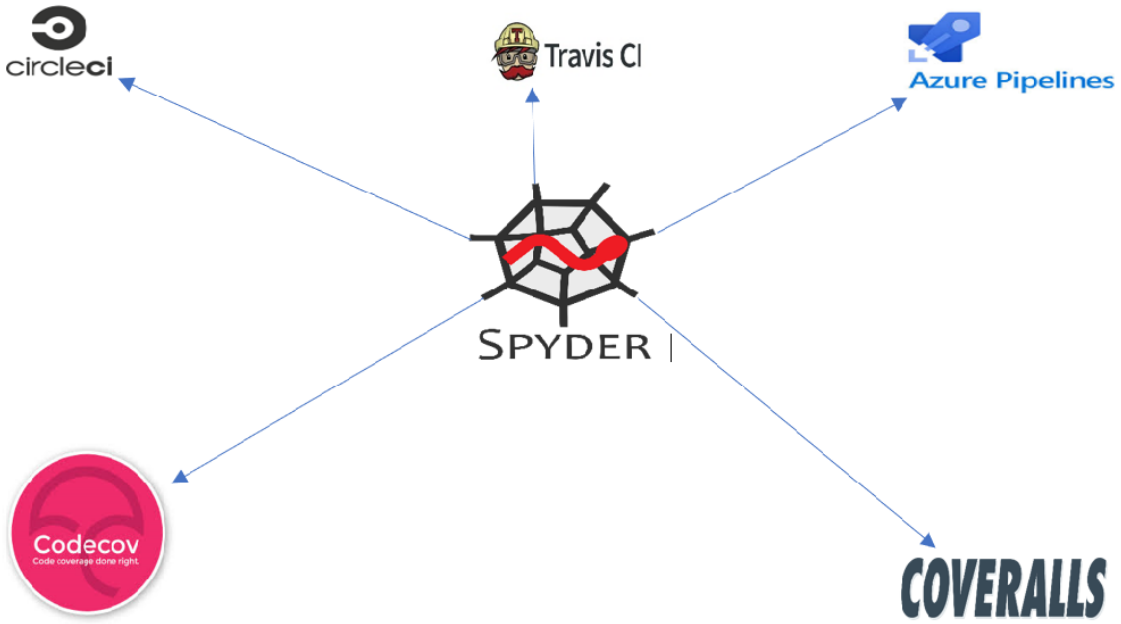


Figure 28.17: Spyder CI pipelines and code coverage platforms

- CircleCI -> Linux with *Python 2, Python 3.6*
- Travis -> Linux with *Python 3.7*

However, Spyder team acknowledges the effort to maintain three different CI pipelines and have started migration to Github Actions [14] , which is a CI pipeline provided by Github.

By now you must be thinking : **What about software quality checks?**

Surprisingly, Spyder does not employ any sort of continuous inspection of code quality. In our opinion this is something Spyder needs to adopt. Hence, we are poised to recommend and set-up a pipeline for continuous inspection of code quality with SonarQube or Codacy for Spyder repository.

Lastly, in this section, we discuss the quality of the software quality processes we mentioned above. We already saw Spyder lacks a pipeline for continuous inspection of code quality. On top of this, according to Sigrid [15] analysis it has a low test-code ratio of 25.8%. Moreover, the code coverage [16] of Spyder is currently 69%, which is just about a reasonable goal to have [17]. Discussions on test quality and new test scripts clearly lack in the issues of Spyder repository. However, the architecture and protocols followed by Spyder for adding test scripts make it easier to add new tests. Generally, the protocol is to create an issue for the bug and then create pull request with correction and a test in the *test* folder of the component to fix the bug.

Overall, Spyder is already following most of the contemporary practices (CI pipelines, code coverage) for

[14] Pull request, Configure CI to use github actions. link
[15] Sigrid, Software metric analysis. website
[16] Coveralls, Code coverage of Spyder. link
[17] Martin Fowler, Test Coverage. website

automatic tests which implies they are concerned about code quality to some extent. In our opinion, they just need to increase the performance with respect to code quality metrics after setting up a proper pipeline for continuous inspection of code quality.

28.3.2 Code Quality Assessment

Different measures can be used to measure code quality qualitatively and quantitatively. Some of the qualitative measures can be *Extensibility, Maintainability, Readability* of the code and quantitative measures can be such as *Complexity or LOC (Lines of Code).*

In this post we are going to evaluate the code quality and maintainability of Spyder according Sigrid [18] analysis. The analysis will be done for the most maintained modules in the repository and according to the developers roadmap [19].

According to Sigrid [20] Spyder has overall maintainability score of 2.7. This measure is an average of different metrics used by Sigrid *(Duplication, Unit Complexity, Component Independence, etc.)*. Spyder scores well in *Volume, Duplication and Component Balance*. This is expected as the developers properly distributed the architecture of the software into different modules and they are evenly developed. Since the code is also properly commented it makes the development cycle for the new developers easier.

However, Spyder is not well designed architecturally considering component independence and component entanglement as it scores less than 1.5 in Sigrid analysis. The main reason is there are lots of cyclic dependencies between the main modules of the architecture such as *Editor, Code Completion* and other modules which can be seen in the Figure below.

In this Figure the red modules with more red opacity (*Editor, Widgets, Utils*) has more cyclic dependency and they are harder to maintain. Therefore it will be a good improvement point for team Spyder to decouple these modules. These modules are described more in details in the previous posts.

28.3.3 Recent coding activity

The last major release of Spyder was `Version 4.0.0` which was released on 06/12/2019. It includes changes like dark theme for entire interface, a new plots pane to browse all inline figures of IPython console etc.(See full list). The main architectural elements that were modified were : `Editor`, `Main window`, `IPython console`, `Debugger`, `Variable explorer`, `Files`, and `Preferences`.

Spyder released three more versions (v4.0.1, v4.1.0, v4.1.1), with the most recent version being released on 18/03/2020. If we take a look at the CHANGELOG.md file, it becomes clear that the major changes have been made to the Editor, Main window, Plots, and IPython console. The code for these components can be considered as "hotspots" because they've had much more activity in the previous releases. To obtain a more concrete claim on hotspots, we list the files based on the number of commits they have based on git commit logs (since 4.0.0)1 : 1

[18] Sigrid, Software metric analysis. website
[19] https://github.com/spyder-ide/spyder/wiki/Roadmap
[20] Sigrid, Software metric analysis. website

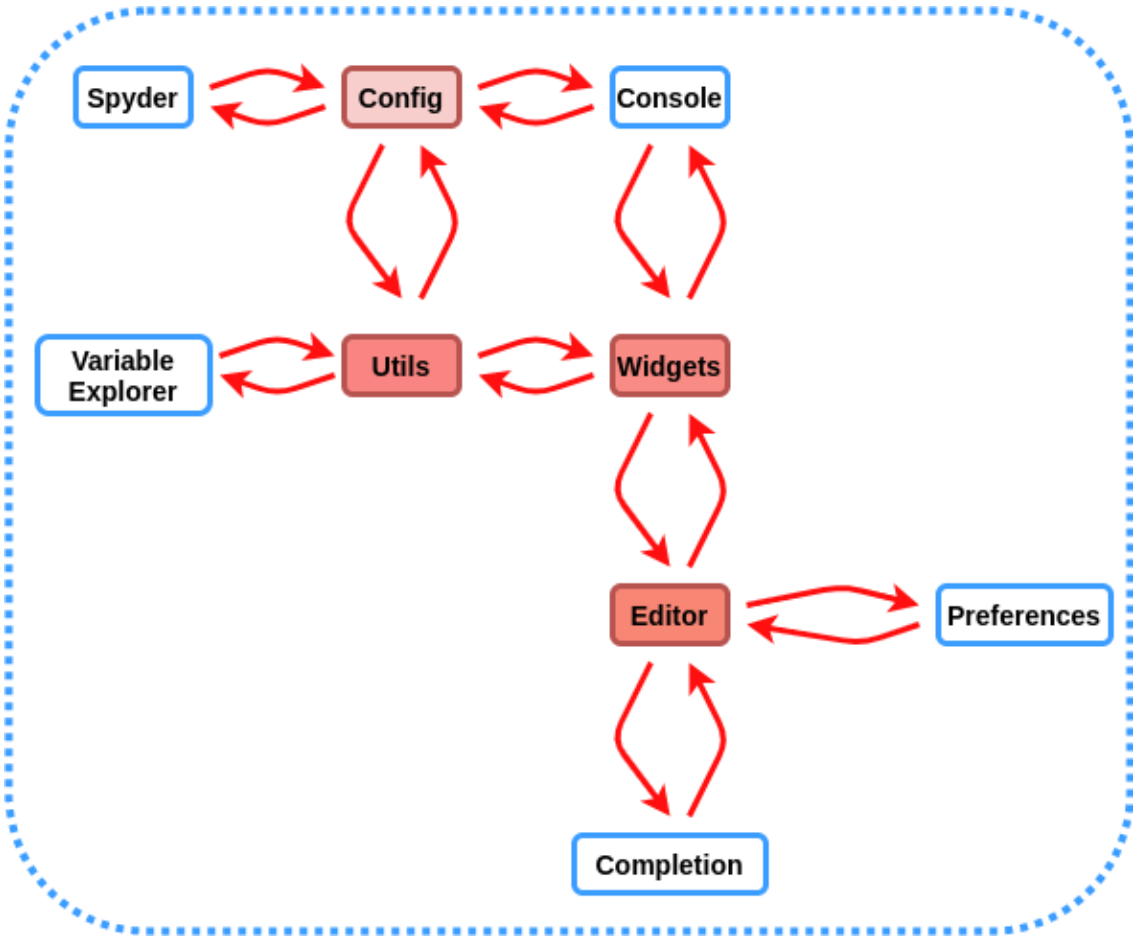


Figure 28.18: Dependency graph of Spyder modules

```
#commits    file
49          spyder/plugins/editor/widgets/codeeditor.py
30          spyder/plugins/plots/widgets/figurebrowser.py
27          spyder/app/mainwindow.py
22          spyder/app/tests/test_mainwindow.py
19          spyder/plugins/editor/widgets/editor.py
19          spyder/plugins/editor/panels/scrollflag.py
18          spyder/plugins/ipythonconsole/widgets/client.py
18          spyder/dependencies.py
17          spyder/plugins/editor/widgets/tests/test_introspection.py
16          spyder/plugins/ipythonconsole/utils/kernelspec.py
```

It is evident that the highest number of commits(87 excluding tests) have been made in `spyder/plugins/editor` folder, followed by `spyder/plugins/ipythonconsole` (34), `spyder/plugins/plots`(30) and `spyder/app/mainwindow.py`(27).

28.3.4 Architectural roadmap

In this section we discuss if the current architecture is indeed ready for this roadmap. We've already seen in essay 1 the future roadmap of Spyder. The main focus for the Spyder team is to work on Spyder 5 and the tentative features that would be worked on are:

- Python 3 only support
- New "Viewer" pane to display HTML content
- docrepr integration
- A Problems pane

We saw the development and deployment views in detail in essay 2. In the development view, we looked into the system decomposition of Spyder. The components in the system are defined independently in itself and still connect together into forming a coherent application (namely Spyder). This is why these components are quite modular and changes can be made individually to them. For example, a new `viewer pane` and the `problems pane` can be implemented into the `Editor` plugin without worrying to much about its dependencies with other plugins. Similarly `docrepr` can be integrated as a third party plugin to Spyder. For Python 3 support, Spyder can make use of Anaconda distribution that now ships with Python 3. All in all, we can say that the current architecture of Spyder can very well support the inclusion of the tentative features that the Spyder team are currently working on.

28.3.5 Technical Debt and Code Quality Analysis

We think team Spyder did a pretty good job regarding technical debt and how they designed the architecture of the system. The reason why adding new features are easy is due to the use of API based system and the widgets that the user can just write and add it to the editor.

The Azure CI pipeline as discussed in the first section is used to run checks on the PR to see if there is any error. If it passes all the checks, then for the new features and bug fixes the head maintainer of the project Carlos Cordoba is responsible for the first look into the *pull request* and assigning it to a reviewer. Afterwards, if it passes the requirements it will be merged to the master branch. Which means they take into account the quality of the code written and how well it is documented. Also some of the developers always take time to comment on the issues and help contributers with better quality solutions as can be seen here.

The maintainers also now use the Crowdin platform which makes adding new languages to Spyder easier through crowd sourcing. A chart of activity for adding new languages can be seen below.

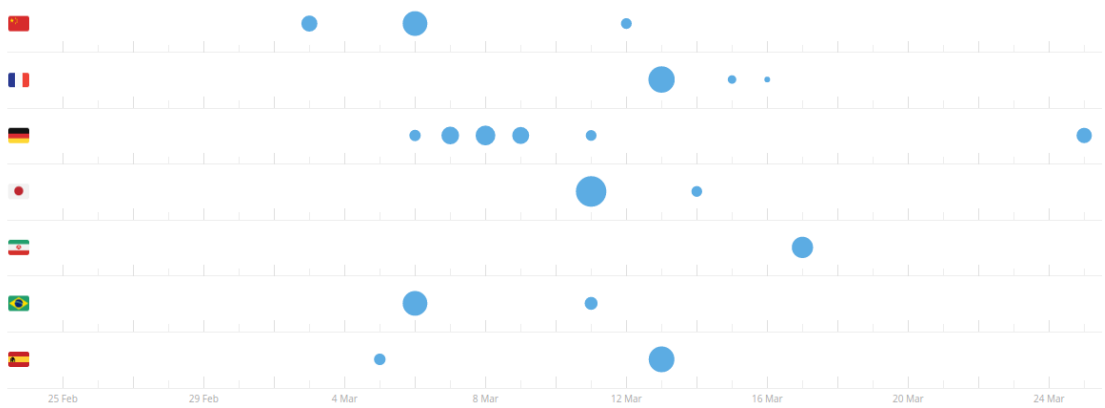


Figure 28.19: Language progress of Spyder

As explained by the maintainers they will add the language once it reaches 98% completion. If there are enough contributers, a vote is taken to see how good the translation is. As can be seen in this link, this is the case for French that has proofreading activity.

28.3.6 Refactoring Candidates

Overall, Spyder does not receive good scores from the Sigrid assessment system. Spyder lacks in 6 of the 7 criteria; unit size (2.0), unit complexity (2.2), unit interfacing (2.0), module coupling (2.2), component independence (1.3), component entanglement (1.4). The one criteria Spyder does well in is code duplication. Below we discuss the main 2 refactoring candidates we feel are more realistic to be able to change.

28.3.6.1 Unit Size

Unit size is a measure of the software's maintainability. More specifically, the unit size is the size of single functions or methods. Having a low unit size score means that there are functions that have many lines of code [21]. The worst offending function discovered by Sigrid is "highlight_block" within spyder/utils with 740 lines of code as seen below. This function highlights the different coding languages, e.g. Cython, C/C++, Fortran just to name the first 3. There are 11 totally different languages highlighted in this function which is why this function is so large. To minimize the unit size of this function, each language could be split into its own function. Other functions like main.py which has 527 lines of code are large because it contains the configuration of the program.

28.3.6.2 Module Coupling

Module coupling is the measure of the interdependence of one module to another. Modules should have low coupling to minimize the "ripple effect" where changes in one module create errors in other modules [22]. Additionally, modules with low coupling are easier to test, change and evaluate. As seen in the image below, most of these modules that are highly coupled are a consequence of the architecture of Spyder. As

[21] Sigrid, Software metric analysis. website
[22] Intermodule coupling. website

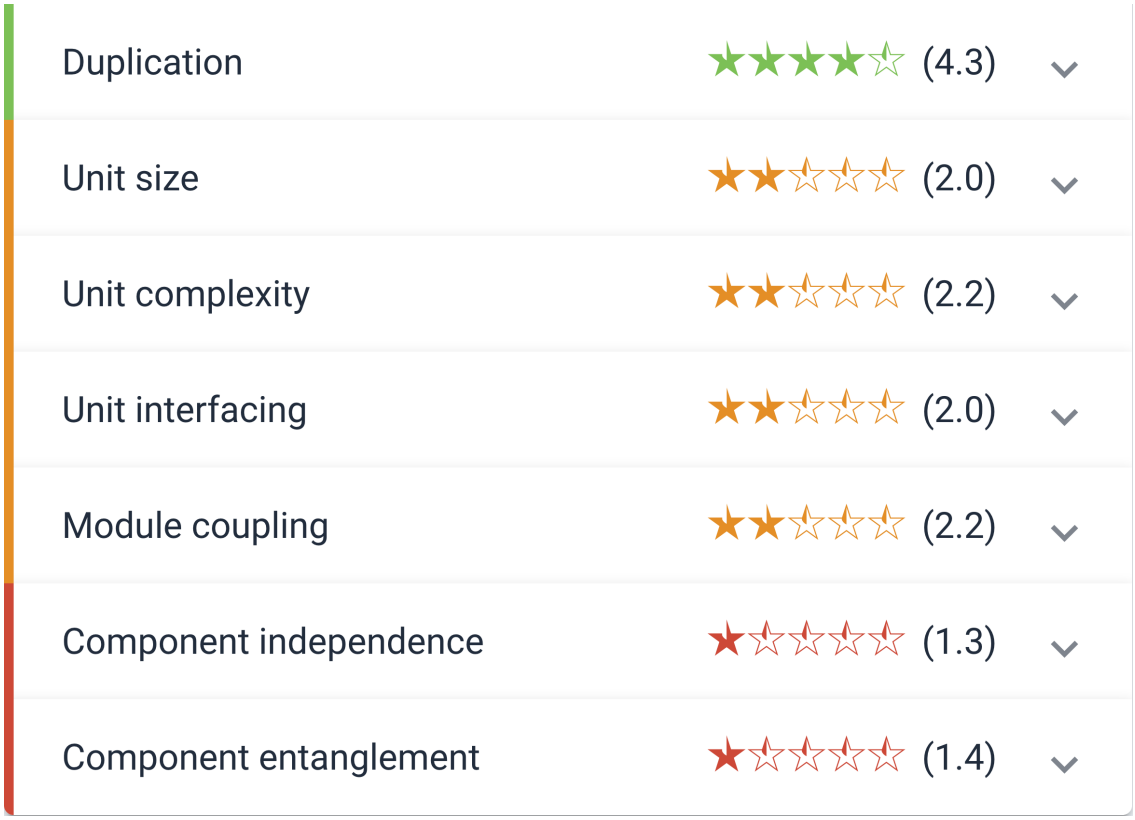


Figure 28.20: Refactoring Candidates

Name	Lines of code	McCabe complexity	Number of parameters	Component
syntaxhighlighters.py:highlight_block(text)	740	3	1	spyder/utils
Editor.get_plugin_actions()	578	3	0	spyder/plugins/editor
MainWindow.setup()	543	64	0	spyder/app
main.py	527	6	0	spyder/config

Figure 28.21: Unit size

discussed in the second essay, Spyder is built on a "plugin" architecture where all the main components such as the editor, variable explorer, and profiler to name a few are separate plugins. These plugins are built from "widgets". As seen in the image below, the top refactoring candidates are from these modules.

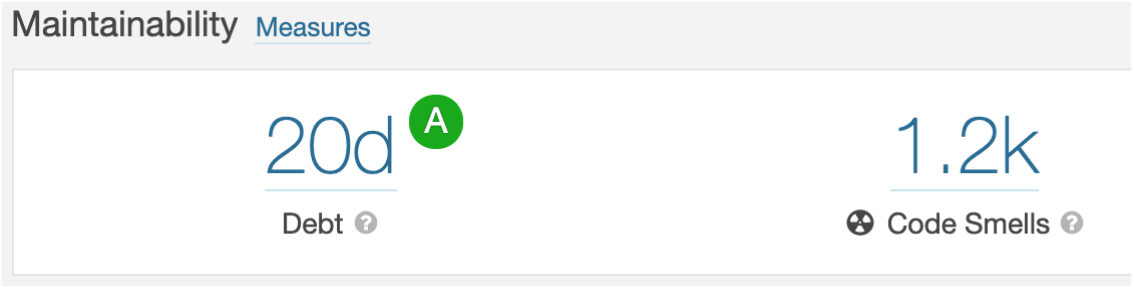
Name	Lines of code	Fan-in	Component
codeeditor.py	3235	52	spyder/plugins/editor
mixins.py	1067	212	spyder/widgets
configdialog.py	745	202	spyder/preferences
base.py	737	102	spyder/plugins/editor

Figure 28.22: Module Coupling

Spyder also faces issues with component entanglement of modules, unit interfacing and component independence and entanglement. These issues as mentioned in module coupling are a consquence of the architecture and therefore we believe they are not realistic refactoring candidates.

28.3.7 Assessment of Technical Debt

Technical debt refers to the extra development costs for rewriting code that is confusing or difficult to maintain, i.e. code smells. As seen in the image below, Spyder receives a score of "A" for maintainability. It would take approximately 20 days according to Sonarqube to re-factor the code smells. A score of "A" translates to a technical debt ratio of less that 5%. A majority of these code smells comes from one math extension that contains all the syntax rules (339 lines of confusing code) which makes sense as its all symbols.



28.3.8

Chapter 29

TensorFlow



Figure 29.1: TensorFlow

Tensorflow is a machine learning framework which helps both beginners and experts in the field to design and deploy their machine learning applications. The framework provides a common basis through many pre-built functions, ready to use instantly, saving developers a lot of time by not having to implement everything from scratch. TensorFlow is a flexible library designed to work on any platform, even mobile devices.

TensorFlow was originally developed by the Google Brain team for internal google use. The first open-source version was released in 2015, a powerful machine learning library with support for several languages. Features that have been added since the first release include machine learning in *JavaScript*, deployment on *Kubernetes*, and deep learning in computer graphics. Tensorflow itself is written in *Python*, *C++* and *CUDA*.

Some noteable companies that make use of Tensorflow include Coca-Cola, Intel and Twitter. More information can be found at https://www.tensorflow.org/.

29.1 The Team

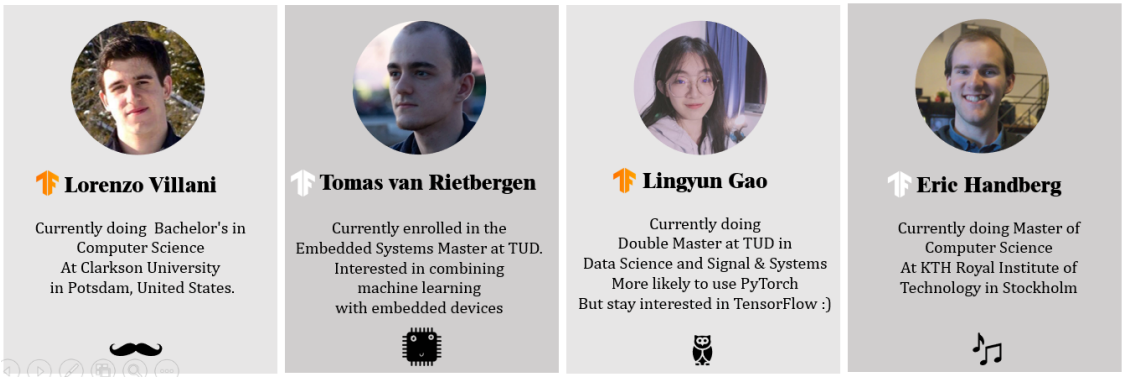


Figure 29.2: team

29.2 TensorFlow: making machine learning manageable

Over the last decade, machine learning has become an increasingly popular solution to solve complex modern-day challenges. Its applications include a diverse range of topics such as image recognition, diagnosing medical conditions or fraud detection. As more distinct fields and industries gain interest in incorporating machine learning techniques into their products, the need for a comprehensive and adaptable machine learning platform arises. This is where TensorFlow comes in, an end-to-end open-source platform for machine learning to easily build and train models, supported by a large ecosystem of tools, libraries and community resources. TensorFlow enables researchers to experiment and push the boundaries of this AI technique while developers can easily build and deploy their machine learning powered applications. With its many options for development and deployment reflected by the number of supported programming languages and hardware platforms, TensorFlow is a very versatile solution applicable to various different domains.

29.2.1 How users perceive TensorFlow

End users of Tensorflow can be beginners and experts working in machine learning research and applications of desktop, mobile, web, and cloud[1]. Specifically, with regard to what Tensorflow is, end users expect

[1]Tensorflow Websites: https://www.tensorflow.org/, 2020/3/8

Tensorflow to be an open-source platform that provides API to quickly build a machine learning workflow in their familiar working environment, typically command-line interface. In terms of user interaction and system functionality.[2] Usually, end users will install and run Tensorflow on the operating system of their computers or cluster of computers. Most of them are accustomed to writing codes and scripts to describe their requirements (build, train, test and analyze machine learning models) for Tensorflow. Thus, Tensorflow should be able to interpret coding languages of the platforms it supports, and transfer them into specific machine learning workflows and provides corresponding machine learning tools.

29.2.2 Core capabilities: What can it do?

Tensorflow provides users with the possibility to easily build, train and deploy machine learning powered applications using one of the several programming languages supported by the available APIs. It allows users to specify and construct their models using a front-end of the chosen languages which the API connects to the underlying high-performance core of Tensorflow that executes the applications efficiently.

29.2.2.1 Spare me the details

Tensorflow leverages the concept of so-called 'dataflow graphs' in which data moves through a series of consecutive nodes connected through edges. Each individual node represents a certain mathematical operation on the data, whereas the edges represent a multidimensional array (also known as a 'tensor') of the data before or after processing. The user can then specify their model in terms of nodes linked together. In essence, the system provides a layer of abstraction sparing the users from having to deal with technical details such as implementing an algorithm or correctly transferring data between functions as this is taken care of by the core library. As a result, the users of Tensorflow can focus all of their attention on the higher-level logic of their application while TensorFlow handles the details.

29.2.2.2 Multilingual and versatile!

As of 2020, in addition to the language options listed in section 2.1.5 of the 2016 version of Desosa [3] , Tensorflow now has API support for several other languages including: Javascript, Go, Swift and a few others maintained by the community. However, one should keep in mind that only the Python and the C++ API are covered in the API stability promises according to the TensorFlow GitHub readme. TensorFlow's services can be accessed on a wide range of operating systems and hardware platforms making it a highly portable system. These platforms include: Windows, MacOS, Linux, Android and for hardware: CPUs, GPUs, Raspberry Pis and even some experimental support for TPUs (Tensor Processing Unit). As a result, Tensorflow is suitable for a wide set of use cases ranging from mobile applications to high-performance devices.

29.2.3 Stakeholders: Who is involved?

In this section, we will analyze 11 different types of stakeholder, we will provide some understanding as to what they are, and then we will observe these roles in the context of TensorFlow. These classes of stakeholders and their descriptions as described in Rozanski and Woods [4].

[2] Jim Coplien Gertrud Bjørnvig. Lean Architecture. Wiley, 2010.

[3] Carmen Chan-Zheng et al, Desosa 2016: Tensorflow, https://delftswa.gitbooks.io/desosa2016/content/tensorflow/chapter.html

[4] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

Stakeholder Class	Stakeholders
Acquirers	TensorFlow was initially developed by the GoogleBrain team for internal use at Google, but has since been released for public use.
Assessors	As well as being legally maintained by the team at Google, TensorFlow is licensed under the Apache License 2.0 which provides a legal guideline for open-source software.
Communicators	The developers of TensorFlow have developed a set of tutorials for the software, as well as extensive documentation.
Developers	The development team of TensorFlow is led by GoogleBrain but there are thousands of contributions from open-source developers on GitHub.
Maintainers	TensorFlow is maintained by GoogleBrain and the open-source developers on GitHub.
Production Engineers	TensorFlow is a client-side, end-user application, and as such has no significant hardware component to speak of, however the software applications used in TensorFlow are managed by the development team.
Suppliers	As TensorFlow is a client-side model, the software in which it runs is downloaded to the user's machine and thus runs in whatever environment it is used in.
Support Staff	There is a global TensorFlow community on Stack Overflow that provides support to users of the software around the world.
System Administrators	As TensorFlow is a client-side program there are no system administrations needed.
Testers	TensorFlow is tested by the users and developers of the system.

Stakeholder Class	Stakeholders
Users	Many people, from hobbyists to companies, use TensorFlow in their projects. Some examples of companies which use TensorFlow are Google, Intel, PayPal, GE, and Airbus.

29.2.3.1 Conclusion

As we can see from the table, due to TensorFlow's nature as a client-side application, there are some types of stakeholders that don't apply, particularly in the context of system administration and hardware-focused stakeholders. TensorFlow's strong foundation and support by Google make it a well-supported system in terms of it's stakeholders due to Google's dependency on the software needing it to be as stable and capable as possible.

29.2.4 The context of TensorFlow

In this section, we will take a closer look at the context in which Tensorflow functions. The context is visualized in the context diagram presented below. In addition to the diagram, the details of the context are described in the accompanying table.

Context Branch	Description
Platforms	The full version of Tensorflow is able to operate on desktop (Linux, macOS and Windows), web, cloud, mobile and IoT devices.
Users	There are several companies using Tensorflow, while many of them are global and well-known. A selection of them are shown in the context diagram. Apart from larger companies, Tensorflow is also used by scientists for various research.
Communication	GitHub is used for version control. Communication within the community happens both online in forums and in real life at summits.

Context Branch	Description
Available languages	Stable Tensorflow API is available in Python and C++. Though not guaranteed to be stable, API:s also exist for a number of other languages, such as Java, Javascript, C, Go and Swift.
Competitors	Tensorflow has several competitors, one example being Pytorch. However, some of them have been officially supported to work on top of Tensorflow, such as Keras.
License	Tensorflow is released under an Apache License 2.0.
Developers	Tensorflow is managed by Google, but it relies heavily on contributions through GitHub.
Trusted Partners	Tensorflow is piloting a Trusted Partner Program, where they have partnered with a few companies they recommend other companies less experienced with machine learning to acquire assistance from if needed.
Extensions	Several extensions to Tensorflow exists, further building on the capabilities of Tensorflow. A selection of them are shown in the context diagram.

29.2.4.1 Future context

Since Tensorflow is an open-source project, its future relies heavily on its contributors and users. Therefore, Tensorflow is asking the community for future directions, for example using this form.

29.2.5 Roadmap: What's next?

Below a summary of the work for each SIG group connected to Tensorflow is displayed which represent the future goals of the project for the year 2020.

29.2.5.1 Roadmap in 2020

29.2.5.1.1 Towards TF2.2:

- Cleaning up and standardizing the high-level API surface

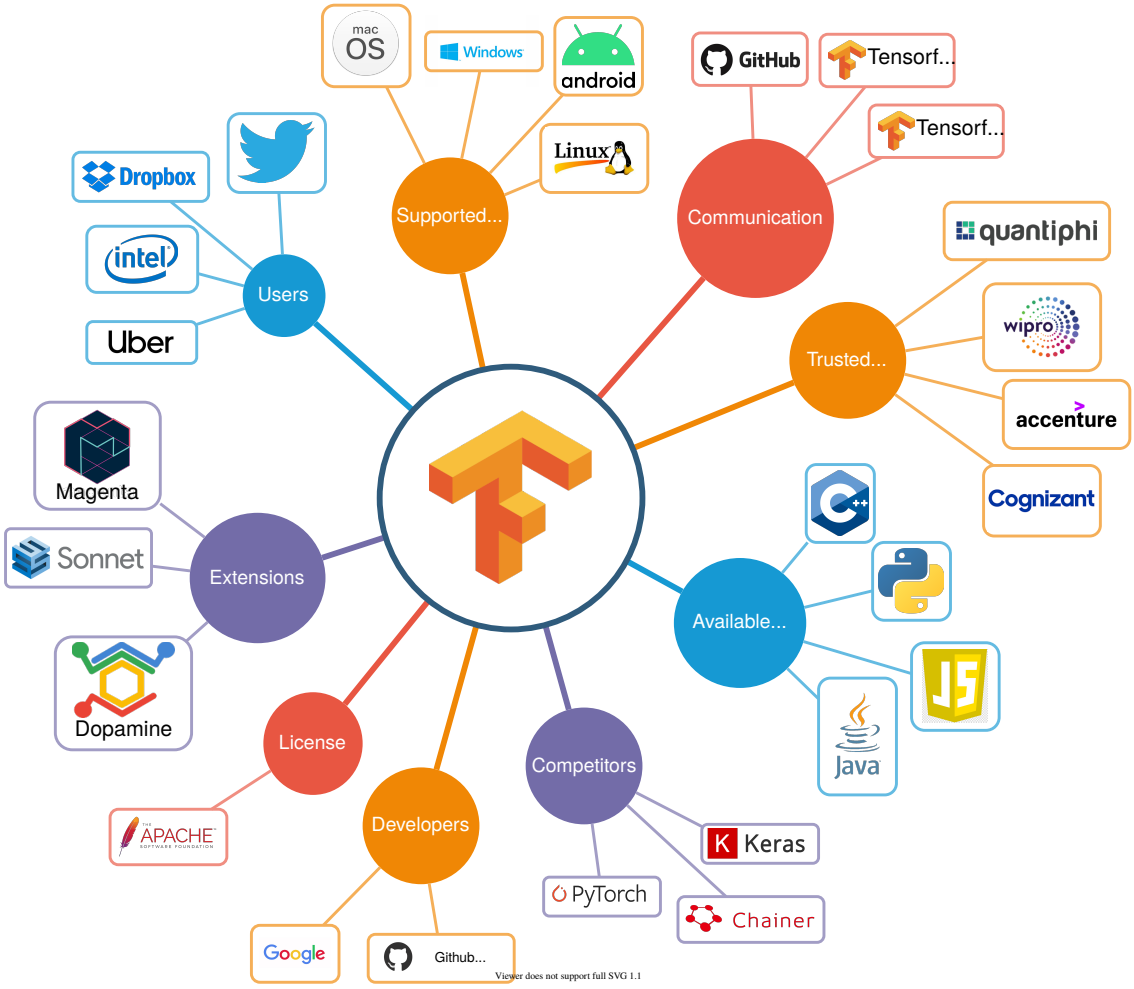


Figure 29.3: A context diagram for Tensorflow

- Making TensorFlow more intuitive and easier to debug
- Continuing to enable scalable production deployment

29.2.5.1.2 Usability

- Support for new development
- Support for more file systems and file formats
- Develop additional tools to enhance the support of machine learning development in JVM languages
- Development and evolution of the Keras API
- Support for different network fabrics and protocols
- Development for TensorFlow Rust bindings project
- Development for TensorFlow Swift
- Development for TensorBoard and other visualization tools

29.2.5.1.3 Performance

- High Performance Compilers and Optimization techniques that can be applied to TensorFlow graphs
- Create common intermediate representation (IR) that reduces the cost to bring up new hardware

29.2.5.1.4 Portability

- Support for TensorFlow models on microcontrollers, DSPs, and other highly-resource constrained embedded devices

29.3 Embedding Vision into Architecture

During the previous chapter we discussed **what** TensorFlow aims to achieve as a product both in its present and future context. As we have seen, this ambitious project tries to suit different use cases with a high degree of flexibility and efficiency making for a not so trivial objective. To realize such a vision, TensorFlow just like any other major project requires a solid foundation supporting all of the smaller building blocks. In this chapter we will focus on **how** these goals are made possible through means of proper architectural components and the interconnections between them. First, let's see what kind of different perspectives there are on architecture and how they relate to TensorFlow.

29.3.1 The different Viewpoints of TensorFlow

In their book[5] and on their website[6] Rozanski and Woods define a set of architectural viewpoints which encompass the different perspectives. Architectural viewpoints provide a framework which captures reusable knowledge to guide architects when designing their application. Essentially, it provides designers with a starting point that suggests from which directions they should approach their project. Now that we have an idea of what these viewpoints entail we can assess how they are applied in the context of TensorFlow and see which viewpoints are more relevant than others. Viewpoints for information systems(IR) will not be discussed, as TensorFlow functions mainly as ML algorithm training and testing platform rather than a data manipulating software.

[5] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

[6] https://www.viewpoints-and-perspectives.info/home/viewpoints/

29.3.1.1 Context viewpoint

TensorFlow is used by many and developed by many. The context viewpoint provides useful information about where and how the application is used. This certainly is an important viewpoint, but for a developer, other viewpoints might be more important. Many details a developer would need are more likely to featured in the development viewpoint or deployment viewpoint. For users however, this is likely a critical viewpoint as the number of agents involved in TensorFlow demonstrates it's capabilities.

29.3.1.2 Functional/Runtime viewpoint

According to [7], the functional viewpoint defines the key components which carry out the main functions of a system and their interactions. This is very similar to how the runtime view is defined in [8]. It is usually easily understood by stakeholders and regarded as a cornerstone of an architectural description. Thus, the functional/runtime viewpoint is suitable to give people insights into TensorFlow.

29.3.1.3 Development viewpoint

TensorFlow is an complex application with many functionalities including several APIs and compatible third-party applications. The application is constantly in motion and under development with many competitors in the field. The development viewpoint is perhaps most crucial to TensorFlow and its developers.

29.3.1.4 Deployment viewpoint

Being a complex application, TensorFlow deployment also becomes an involved process. The many different options of how and where to deploy the application add to the complexity of this viewpoint and necessitate a clear deployment strategy.

29.3.1.5 Operational viewpoint

Since TensorFlow is a popular open source library it is constantly undergoing changes and updates. Therefore, it is important to systematically handle the constant flow of contributions and deployment of updates.

29.3.2 Architectural Style: The different Layers of TensorFlow

After having discussed the architectural viewpoints we now turn to the architectural style which expresses a structural organization schema for software systems. TensorFlow makes use of a **layered** architecture. It consists of many layers which built on top of each other and request and provide services to and from other layers[9]. The following diagram presents a high level overview of the layer structure for TensorFlow.

As can be seen from the image, TensorFlow's top most layer consists of multiple libraries for using the software which is built on top of the clients, like those for Python and C++, which define the computation as a dataflow graph and initiate graph execution using a session.

Next is the API layer which contains the C API. This is built on the Distributed master layer, as well as the dataflow executer. The Distributed master layer prunes a specific subgraph from the graph, partitions the

[7] Software System Architecture, the functional viewpoint, last accessed 2020/3/17 https://www.viewpoints-and-perspectives.info/home/viewpoints/functional-viewpoint/

[8] arc42 Runtime view, last accessed 2020/3/17 https://docs.arc42.org/section-6/

[9] Lamberta, Billy. "Tensorflow/Docs." GitHub, 8 Jan. 2020, github.com/tensorflow/docs/blob/master/site/en/r1/guide/extend/architecture.md.

subgraph into multiple pieces that run in different processes and devices, distributes the graph pieces to worker services and initiates graph piece execution by worker services.

Next are the Kernel Implementations, which perform the computation for individual graph operations.

Lastly, there are the Networking and Device layers, at the lowest level of the architecture.

29.3.3 Development View

Now diving more into the actual architectural viewpoints themselves we start with the development view. The code of TensorFlow is decomposed into multiple modules contained within the TensorFlow directory. The following section will provide a brief analysis of the source code organization and the module dependencies with the help of the presented diagram.

29.3.3.1 Which folder does what?

For comparison, we have included the source code organization diagram of TensorFlow as visualized during the 2016 edition of Desosa[10]. While the two diagrams are quite similar, there are a few notable differences as can be seen from the image below.

TensorFlow 2016	TensorFlow 2020

The following subsections will give a very brief explanation of the contents of each sub-folder in the 2020 source code organization.

29.3.3.1.1 .github/ISSUE_TEMPLATE Contains template files for different kinds of issues for submitting to GitHub. This folder was not yet present in 2016.

29.3.3.1.2 third_party Contains many different instances of third-party libraries used for TensorFlow. This folder has remained since 2016 and still serves the same purpose.

29.3.3.1.3 tools Contains a script file which generates a file with system information details used to populate GitHub issue templates. This folder was also present in 2016, but the purpose of the script file seems to have changed.

29.3.3.1.4 tensorflow Contains the components for the main application. The documentation for some of these folders is unclear, but the contents can be summed up roughly in the following way:

- **c:** Contains files for the C API for TensorFlow.
- **cc:** Contains files for the C++ API for TensorFlow.
- **compiler:** Contains several modules for compiling.
- **core:** Contains code making up much of the main functionality of the TensorFlow application.
- **docs_src:** Formerly used to contain documentation, but documentation has since been moved to its own dedicated repository.

[10] Carmen Chan-Zheng et al, Desosa 2016: TensorFlow, https://delftswa.gitbooks.io/desosa2016/content/tensorflow/chapter.html

- **examples:** Contains demos and examples for different platforms and tasks of how to write TensorFlow code.
- **g3docs:** Was also used for documentation, but is now empty.
- **go:** Contains files for the Go API for TensorFlow
- **java:** Contains files for the Java API for TensorFlow
- **js:** Contains files for JavaScript code generation.
- **lite:** Contains modules for TensorFlow Lite, the lightweight version of TensorFlow aimed for android and embedded devices.
- **python:** Contains files for the Python API for TensorFlow.
- **security:** Contains security advisories for using TensorFlow.
- **stream_executor:** Contains the GPU executor library.
- **tools:** Contains various tools, such as a benchmarking tool and a tool for running all important builds and tests.

29.3.3.2 Module dependencies

In order to find the module dependencies, we looked in the document ion and used the architecture visualization provided by Sigrid. A high level overview of the dependencies are shown in the diagram below.

29.3.4 Run Time View: How do TensorFlow components work together?

In this runtime view, we made a brief description of runtime between the components of TensorFlow, including compiler, stream executor, core engine and API for other languages.

Tensoflow uses a data flow programming model. Specifically, to run a machine learning task, users need to first build a graph whose nodes indicate TensorFlow operations and edges indicate tensors. An example graph is shown in figure below. [11] Users can construct and execute a graph by APIs available in 6 languages. Noted that they are all running based on the core engine. Then, a typical work loop is running in which TensorFlow finds a node ready to execute in the graph, run it and repeat these two steps until the nodes are all executed.

The interaction of TensorFlow components in this workflow is shown in figure below[12]. When users hand in a graph, based on whether it is required to run in a distributed way, the compiler will assemble the codes into cpu/gpu/tpu expression and then send them into stream executor.

29.3.5 Deployment View: How to get up and running

After software has been developed and tested the next step is for it to be deployed onto the end user machines. TensorFlow provides several ways for deployment to end users as mentioned in section 3.1 of the Desosa 2016 edition.[13] Installing through pip is the easiest method and allow users to deploy the TensorFlow on their computer with a single command. Alternatively, the docker container method can be useful for GPU support as it provides an image with all the required GPU libraries pre-installed. The third option is to clone the repository and build the system from source using the build tool Bazel. While this approach takes more work, it is the only way to gain access to the C++ API and gives the user more control over the final software configuration. TensorFlow offers both a 'stable' distribution as well as a 'nightly' build. The nightly build is

[11] TensorFlow example, last accessed 2020/3/15https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/tensorflow.html
[12] Runtime view of TensorFlow, Lingyun Gao
[13] Carmen Chan-Zheng et al, Desosa 2016: TensorFlow, https://delftswa.gitbooks.io/desosa2016/content/tensorflow/chapter.html

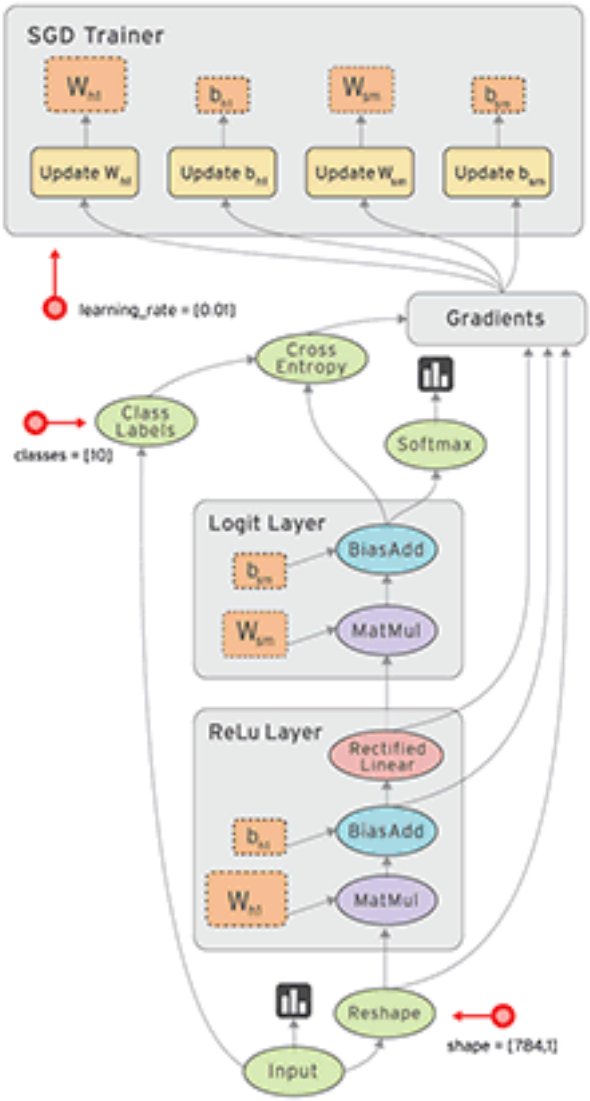


Figure 29.4: Example of TensorFlow Graph

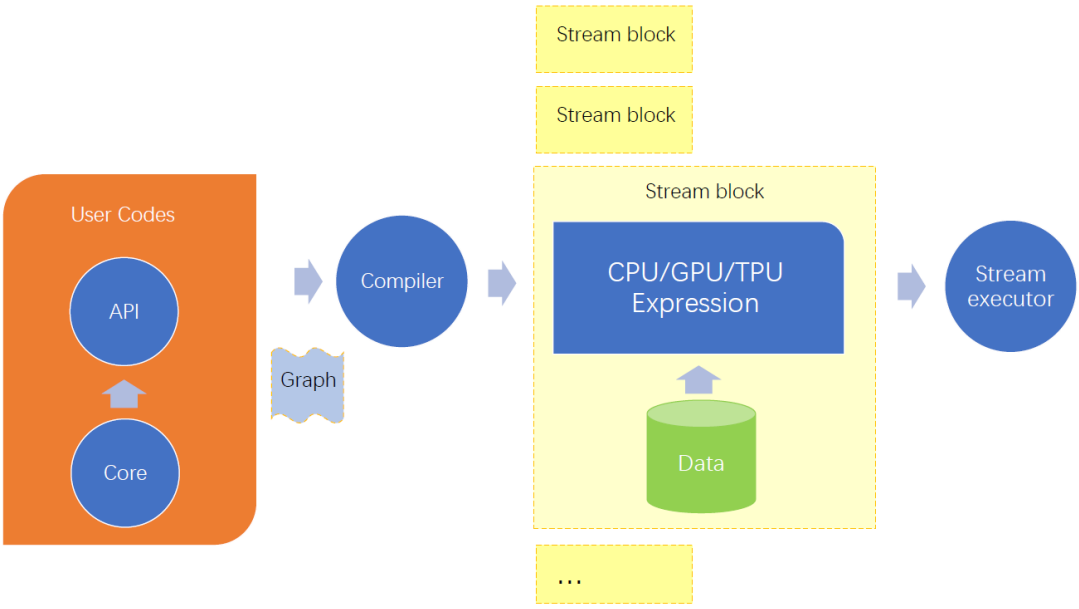


Figure 29.5: Runtime view of TensorFlow

created daily from the master branch and includes the latest changes including the latest new features. The stable version is released far less often making it more reliable.

29.3.5.1 Hardware requirements

According to Rozanski and Woods[14], the deployment view also defines which hardware components are required for the system to run. TensorFlow requires either a CPU, GPU or TPU in order to perform its main functionalities in addition to standard general-purpose hardware. However, in the case of GPU it only supports NVIDIA® GPU cards with CUDA Compute Capability 3.5 or higher. Additionally, operating TensorFlow with the help of GPUs requires some additional software packages: * NVIDIA® GPU drivers - CUDA 10.1 * CUDA® Toolkit 10.1 * cuDNN SDK 7.6 or higher.

In the case of TPU things work a little different. Currently TensorFlow provides experimental support for Keras with Cloud TPUs owned by Google. The user uploads their model through the provided API after which it is executed on the specialized hardware and returns the results.

29.3.5.2 Third-Party software requirements

After having sorted out the hardware there are also many third-party software dependencies that TensorFlow relies on. Below we list the important dependencies for the current major version of TensorFlow (2.x):

First, with respect to the operating system TensorFlow requires one of the following options[15]:

[14] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

[15] TensorFlow official installation guide web page https://www.tensorflow.org/install

Operating System	Version
Ubuntu	>= 16.04
Windows	>= 7
macOS	>= 10.12.6 (Sierra)+ (no GPU support)
Raspbian	9.0

In addition, the following software components are needed to cover certain aspects[16]:

Software	Version	Description
Python	3.5–3.7	The fundamental base that supports all of the Python packages
NumPy	>= 1.16.0 - 2.0	Provides support for large multi-dimensional arrays and associated operations
Six	>= 1.12.0	Provides compatibility between the Python 2 and Python 3 library
Wheel	>= 0.26	A built-package format for python to pack and unpack files
Mock	>= 2.0.0	Provides support for testing parts of the system using 'mock' objects

29.3.6 Non-Functional Properties and Trade-offs

So far in this chapter we have mostly observed TensorFlow from a very technical perspective where every element has a dedicated function. But TensorFlow is more than just a collection of functions to perform machine learning tasks. Other than the technical properties there exist non-functional properties which are more abstract yet still contribute significantly to the meaningfulness of the system. When it comes to TensorFlow, the following topics and their trade-offs come to mind:

- Open-source vs Supportability : TensorFlow is an open-source project giving public permission to codes and design documentation. Furthermore, it encourages the developer community to maintain and contribute to the project. This property boosts the development in project and application, but also brings trouble for management. Currently they have built several special groups in charge of contributions in total, this would increase the communication cost but lead to a systematic contribution and better supportability.
- Adaptability vs Stability: To adapt to a diverse programming language environment, TensorFlow is currently maintaining and building API for at least 12 programming languages. In this way, TensorFlow can take advantage of the variety of different languages. However, there are not enough developers so the stability of those APIs can not be guaranteed. Currently, the only API of python is

[16]TensorFlow pip setup file https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/pip_package/setup.py

promised stabilized. [17]

- Extensibility vs Security : TensorFlow can build dependencies easily on third party libraries such as NumPy, PNG parsers, however, with the cost of several security issues. Extending more libraries also means more vulnerabilities from these included libraries which could trigger unexpected or dangerous behavior with specially crafted inputs.[18]

29.4 Combining quality and collaboration in TensorFlow

As we have seen earlier, TensorFlow is a project of substantial size both in terms of codebase as well as the community surrounding it. With so many people working on such a large application it becomes very hard for anyone to carry a complete view of the entire system in their mind. Developers working on certain semi-isolated parts of the software might introduce modifications that unknowingly impact functionalities elsewhere in the system leading to bugs or even failures. Maintaining a high level of quality becomes an undeniable challenge itself, especially for an open-source project involving countless different contributors. This article will provide you with some insight as to which maintenance methods and tools are being employed by the TensorFlow team to keep their project healthy.

29.4.1 Overview of software quality management: What is it like to be a contributor in Tensorflow?

Assuming you have gained some insights of what Tensorflow is and how Tensorflow works, now it's time to contribute! Tensorflow follows a Github-centered software quality management and asks for the involvement of all contributors, developers, and maintainers.[19] **If you are a code contributor**, before your contribution, you are required to have a Github account, sign in certain agreements and communicate actively with other developers to avoid duplicating efforts. Next, A *high-quality unit test* for checking the validity of the code and *adequate testing* are required. You can always find a detailed test procedure in the documentation of each subproject. Then, when your codes and tests are ready, you are asked to open a *Pull Request* (PR), and maintainers and other contributors will review this PR and your code. Finally, if your PR was proved, you might get a medal for your contribution!

As a contributor, you must learn and think before your actions! Sections below will offer you more details of testing and also provide some ideas of what you could dig into and avoid when you start a new contribution. Have fun!

29.4.2 Compulsive Tests: CI and Sanity Check

All pull requests (PR) must pass the sanity check and TensorFlow unit tests. [20]A sanity check is to catch potential issues of license, coding style, and BUILD files, while unit tests are to guarantee that your PR does not destroy TensorFlow (too much). Unit tests can be done locally, but usually, new changes will be asked to go through TensorFlow Continuous Integration (CI), in which you can trigger builds and tests of your PR performed on either Jenkins or a CI system internal to Google.

[17]TensorFlow API, last accessed 2020/3/19 https://www.tensorflow.org/api_docs
[18]TensorFlow security https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md
[19]Contribution to Github Code , last accessed in 2020/3/26 https://www.tensorflow.org/community/contribute/code
[20]Contributing Guidelines, last accessed in 2020/3/26 https://github.com/tensorflow/tensorflow/blob/master/CONTRIBUTING.md

To be more specific, CI test is a fixed processed testing for each new contribution of Tensorflow. It contains the test and checks as following[21]:

- Build TensorFlow (GPU version)
- Run TensorFlow tests:
  - TF CNN benchmarks (TensorFlow 1.13 and less)
  - TF models
- (TensorFlow 2.0): ResNet, synthetic data, NCCL, multi_worker_mirrored distributed strategy

In addition, the configuration test is not exactly mentioned in the documents. This is probably caused by too many platforms used for Tensorflow and the configuration test is also partly included in the CI test. Thus we will not discuss it in this essay.

29.4.3 Test Coverage

As previously said, tests are an important part when contributing to the Tensorflow repository, and are mainly necessary in two cases[22]. The first case is when a new feature is added, proving that the feature will not lead to future breakdown. The second case is when fixing bugs, since the existence of the bug most likely means that the testing was insufficient in the first place. This process likely assures good test coverage, and in combination with above mentioned CI processes and sanity checks, this is how TensorFlow continues to be stable despite the size of the project and the many contributors. According to Sigrid, the test code ratio is 24.3%.

Looking around in the repository of TensorFlow [23], test files are usually placed in the same folder as the files they are supposed to test. A common pattern, for example in the case of C++ files, is to have a file for testing named `X_test.cc` if there is a file named `X.cc` in that folder. For files written in Python, doctests are used.

29.4.4 Coding Hotspots

With a project of this size, it is interesting to see where developers are actually making contributions. Does the TensorFlow repository have any *"hotspots"*? A hotspot could be defined as a part of your code base that is more complicated than the rest. Therefore, it requires more attention and results in a lot of time spent at that particular part[24]. To find any hotspots in the TensorFlow repository, we used CodeScene[25], which generated the hotspot map over code activity seen in the figure below.

The figure shows the contents of the folder "tensorflow" in the TensorFlow repository (see *Embedding Vision into Architecture* for more details on the folder structure). The lack of red dots in the image is an indication of that *Tensorflow almost completely lacks hotspots*. One file, `wrappers.go` in the folder for the Go API, is clearly marked red by CodeScene. Looking into the history of that file, we can see that this file has received several updates per day by the tensorflow-gardener bot, see image below.

The updates the bot is making seem to only be to remove and add spaces in comments at the same places over and over again. It is definitely questionable if this behavior is really intended.

[21] Tensorflow Builds, last accessed in 2020/3/26 https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/ci_build

[22] TensorFlow Github repository, Contribution guidelines and standards, https://github.com/tensorflow/tensorflow/blob/master/CONTRIBUTING.md#contribution-guidelines-and-standards

[23] TensorFlow Github repository, https://github.com/tensorflow/tensorflow

[24] CodeScene definition of a hotspot, https://codescene.io/docs/guides/technical/hotspots.html#what-is-a-hotspot

[25] CodeScene tool for analysing repositories https://codescene.io/

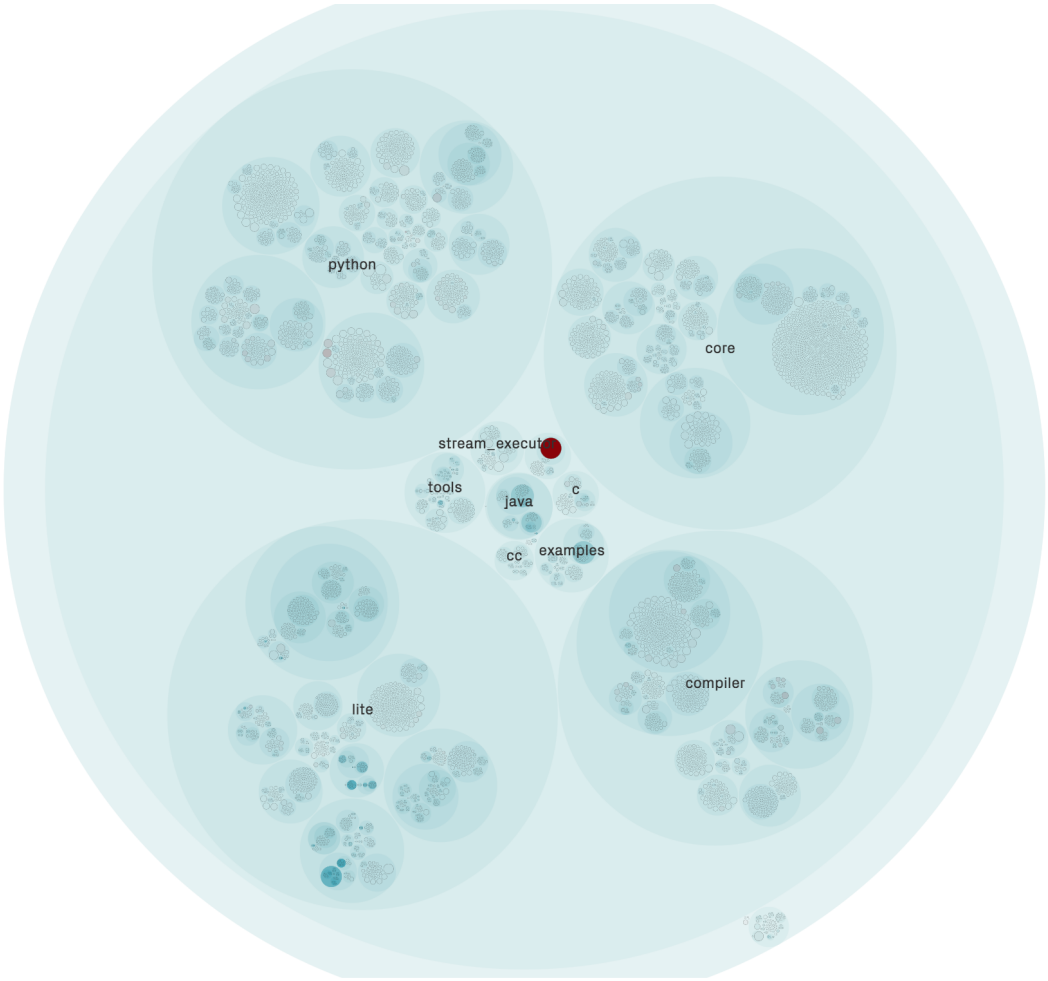


Figure 29.6: Hotspot map for TensorFlow

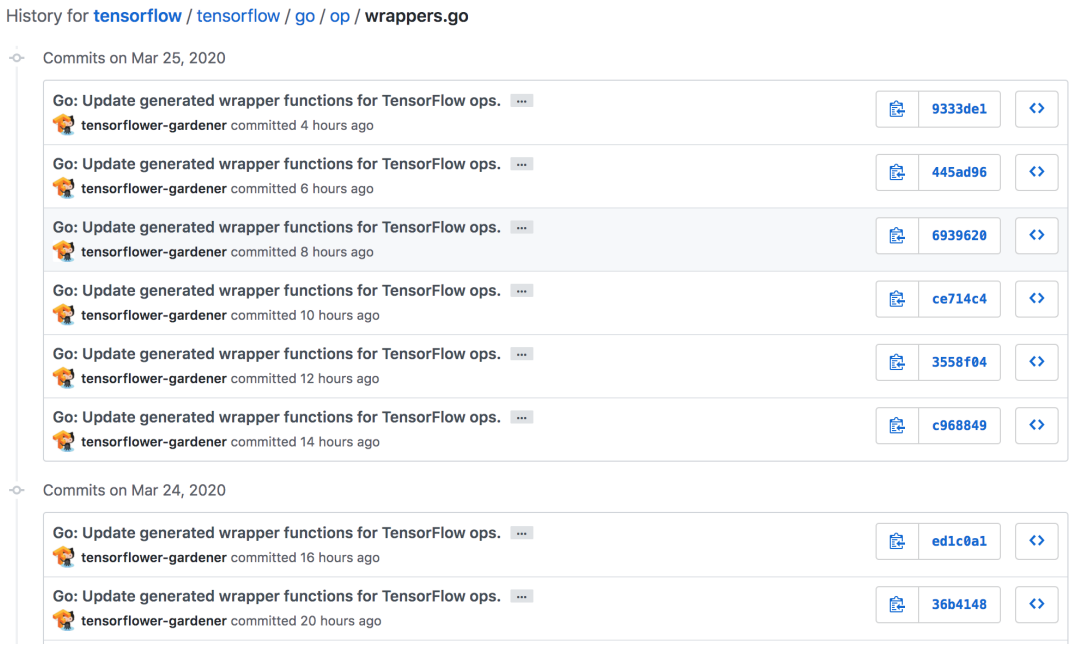


Figure 29.7: History for the file `wrappers.go`

There are a few files in the `python/ops` folder that have been given a slight hint of red. Taking an example, one of these files is `maps_ops.py`. It contains mathematical functions, many of them simple and probably well used, for example `abs()` and `argmax()`. Looking into the history of this file, about 40 commits have been made since the beginning of 2020. Most of these are just updates to comments and docstrings. This does not look like a severe issue and should probably not be considered a hotspot.

29.4.5 Where will the code for the Roadmap 2020 be written?

As seen in our post *TensorFlow: making machine learning manageable*, much of the future development of TensorFlow is centered around usability. As we can see several APIs mentioned here, we can assume that these additions will happen in the respective module for that API. The roadmap also mentions updates to performance. These additions can be assumed to happen in both the compiler module and the core module.

29.4.6 Quantity and, hopefully, quality

Even if testing and continious integration processes are good, contributing would be very difficult if the code didn't live up to a certain quality. However, maintaining quality code in a system such as TensorFlow is doubtless a Sisyphean task. Simply looking at the dependency graph provided by SIG shows an enormous, tangled web of hundreds of components with thousands of dependencies and interconnections, each with up to tens of thousands of lines of code.[26] Given this, it is no surprise that the code quality can tend to suffer at times.

[26] "Sigrid TensorFlow Analysis." Sigrid, Software Improvement Group, 2020, www.sigrid-says.com/portfolio/tudelft.

Firstly, there are so many dependencies that one can hardly begin to understand the effect that one small change to a component will have, and as such, there are many components with very low scores for modularity. The issue with this, at this scale when you make a change to code, not only will you be affecting countless other components, but it will also mean that you now must test many component interactions along with the component itself. This all contributes to the hundreds of man-months put into TensorFlow. Finally we question the quality of those components most likely affected by future change.

As we have mentioned previously, size and dependency are massive burdens to quality, especially in how they link so many components. Many core features, for example the core framework, have low quality scores and will be largely affected by future change. However, the interconnectedness of the component, and of all of the central components of TensorFlow mean that even if these components had better technical quality, it will still be impacted by those components upon which they rely. Therefore it is inevitable that as architectural changes are made to TensorFlow in the future, the ripple effects this will have mean that it will be necessary to redesign or refactor huge amounts of the codebase.

Sigrid attributes to TensorFlow thousands of maintainability violations, especially those pertaining to duplication. This is because with a technology as big and with as many contributors as TensorFlow, it can be difficult to coordinate everyone's knowledge and prior work on the software, so oftentimes the code is written somewhat inefficiently. The nature of these violations are closely tied to the properties which SIG suggests be refactored, and in doing so, the maintainability will be much improved.

29.4.7 Refactoring suggestions, the bad and the less bad

SIG also offered some ways that TensorFlow could be refactored[27]. They describe several areas in which the system could be refactored to provide clarity to the code, make it easier to read, and to improve certain qualities of the code.

Refactorable System Property	Reason	SIG Code Quality
Duplication	Writing code once makes it easier to edit and reuse.	2.4
Unit size	Smaller units typically have a single responsibility and are easier to change without affecting other components.	1.7
Unit complexity	Simple units are easier to understand and test.	2.3
Unit interfacing	Smaller, simpler interfaces are easier to modify and easier to make error free.	0.9
Module coupling	Modules that are loosely coupled are easier to analyze, test, and modify.	2.4
Component independence	Independent components make for easier system maintenance.	2.3

[27] "Sigrid TensorFlow Analysis." Sigrid, Software Improvement Group, 2020, www.sigrid-says.com/portfolio/tudelft.

Refactorable System Property	Reason	SIG Code Quality
Component entanglement	Minimal component communication means less complex architecture.	1.4

TensorFlow does not seem to score particularly high for any of the investigated factors. Moreover, it scores particularly bad with respect to Unit interfacing, Unit size and Component entanglement. In the following section we try to analyze one of the aspects which might be one of the causes for the system's poor performance on the mentioned categories.

29.4.8 Technical Debt: Shedding light on the dark side of TensorFlow

Up until this point, most of the topics we have discussed focus on the objective and positive properties of TensorFlow. However, as with almost any software project of this size, eventually decisions which might be sub-optimal from a design perspective will have to be made in favor of other objectives. It is important that we consider this 'dark side' of TensorFlow as it can bear significant implications for future development. In this section we will discuss several negative aspects of the framework and also analyze how the community takes the technical debt into account when discussing future development.

TensorFlow was influenced by the Theano, a library available since 2007 that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Like Theano, TensorFlow originally worked by first creating a graph statically specifying the types of operations and the order in which they will be executed. After having defined your graph, the real data is fed in through the `tf.Session` and `tf.Placeholder` objects at runtime. This lack of flexibility was a big concern for researchers exploring innovative techniques, one of the main target user groups of TensorFlow. In 2018 former Google scientist Liang Huang stated on a discussion board that the strategy of static graphs akin Theano was a major design mistake which might have serious consequences for the future of TensorFlow[28]. An alternative is to use a more imperative approach like dynamic graphs which allow you to change and execute computation nodes 'on the go' at the cost of some overhead. This strategy, employed by major competitors such as PyTorch[29], greatly aids in debugging as intermediate results are returned immediately providing feedback to fix or improve individual nodes.

Another important aspect of dynamic graphs is that they facilitate the use of Recurrent Neural Networks (RNN). Essentially, RNNs are neural networks with feedback loops typically implemented in code as a for-loop. The input sequence length of dynamic neural networks can vary with each iteration which was inconvenient for the TensorFlow's static graphs with fixed input lengths[30].

However, The TensorFlow team was aware of these issues and not long ago in September 2019 TensorFlow 2.0 was officially released including the promising new feature of eager execution[31]. Eager execution is an interface enabling dynamic graph execution and essentially move TensorFlow into the direction of PyTorch. While the new feature is stable, users have reported significant slowdowns of the execution speed of their

[28] Liang Huang, Former Google Scientist, https://www.quora.com/What-is-the-future-of-TensorFlow

[29] Kirill Dubovikov, CTO at Cinimex DataLab, https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b

[30] Kirill Dubovikov, CTO at Cinimex DataLab, https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b

[31] TensorFlow Team, TensorFlow blog, https://blog.tensorflow.org/2019/09/tensorflow-20-is-now-available.html

models when switching from static to dynamic models[32]. According to one of the team members[33] currently the eager execution mode is mostly meant for debugging purposes. However, he does promise that updates will be coming in the future which will improve the running time of the eager execution method.

As of TensorFlow 2.0 eager execution is now the default context, but can be easily disabled by calling the `tf.compat.v1.disable_eager_execution()` function. Also, the official guide page[34] states that performance of eager execution suffers for models which involve lots of small operations and more optimization work needs to be done. In conclusion, while the team has made good efforts to liquidate some of the technical debt through the addition of eager execution, some negative effects of legacy decisions still remain and will require more work in order to be amended. Hopefully the TensorFlow team will continue to light their (Py)torches through the darker districts of TensorFlow.

[32] TensorFlow Issue #629: 100x slow down with eager execution in tensorflow 2.0, https://github.com/tensorflow/probability/issues/629

[33] TensorFlow Issue #33487, Comment by teammember qlzh727, https://github.com/tensorflow/tensorflow/issues/33487#issuecomment-548071133

[34] TensorFlow eager execution official guide page, https://www.tensorflow.org/guide/eager#benchmarks