TravisTorrent

Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration

Beller, Moritz; Gousios, Georgios; Zaidman, Andy

DOI
10.1109/MSR.2017.24

Publication date
2017

Document Version
Early version, also known as pre-print

Published in
Proceedings - 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR 2017

TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration

Moritz Beller, Georgios Gousios, Andy Zaidman
Delft University of Technology, The Netherlands
{m.m.beller,g.gousios,a.e.zaidman}@tudelft.nl

ABSTRACT

Continuous Integration (CI) has become a best practice of modern software development. Thanks in part to its tight integration with GitHub, Travis CI has emerged as arguably the most widely used CI platform for Open-Source Software (OSS) development. However, despite its prominent role in Software Engineering in practice, the benefits, costs, and implications of doing CI are all but clear from an academic standpoint. Little research has been done, and even less was of quantitative nature. In order to lay the groundwork for data-driven research on CI, we built TravisTorrent, `travistorrent.testroots.org`, a freely available data set based on Travis CI and GitHub that provides easy access to hundreds of thousands of analyzed builds from more than 1,000 projects. Unique to TravisTorrent is that each of its 2,640,825 Travis builds is synthesized with meta data from Travis CI's API, the results of analyzing its textual build log, a link to the GitHub commit which triggered the build, and dynamically aggregated project data from the time of commit extracted through GHTorrent.

1. INTRODUCTION

Since its conception in 1991 and wide-spread distribution as part of Microsoft's and Extreme Programming's development practices [1–3], CI has become a global Software Engineering phenomenon. Over the past five years, TRAVIS CI has emerged as a popular CI environment for OSS projects, having performed hundreds of millions of free builds.

The unbroken trend toward CI in practice came with little backup from the academic side, however. From an academic standpoint, we still lack quantifiable evidence on the implications of introducing and continuing to use CI. While we have used TRAVISTORRENT to drive a first investigation into CI testing practices [4], the questions that can be answered with it reach far beyond:

- Does the use of CI lead to higher-quality products, for example by catching bugs and test failures earlier?
- Does CI lead to fewer test regressions?
- What are CI best practices that successful projects employ (build more often, fail more often, have more test on the CI?
- Do CI-enabled projects switch to a Continuous Delivery?
- How long may a CI build run to still be considered helpful?
- Does a broken build really negatively affect other developers' productivity, as is often claimed?
- Two development models compete over how to use CI: Should developers consider a successful build their "holy grail", or rather embrace breaking (and fixing) it often?
- Does a broken build lead to fewer outside contributions, in accordance with the "broken window" theory?
- Do multiple integration environments lead to fewer defects?

Only recently have researchers begun to discover TRAVIS CI as a data source [4–7]. They have, however, not yet taken advantage of the endless possibilities that the combination of a streamlined, popular and tightly coupled CI environment (TRAVIS CI), version control system (GIT) and collaboration platform (GITHUB) provide, as collecting and aggregating this data in a single data set is logistically and algorithmically complex. By synthesizing all three data sources in one readily accessible data set of more than 1,000 projects, we hope to facilitate more holistic research on CI with TRAVISTORRENT, by giving researchers the opportunity to do "full-stack research" from an analysis of build logs to repositories.

2. THE TRAVISTORRENT DATA SET

In this section, we give an overview of the TRAVISTORRENT data set and ways to access it (more details in Appendix A).

The TRAVISTORRENT Data set. From the 17,313,330 active OSS repositories on GITHUB in August, 2015, our data set contains a deep analysis of the project source code, process and dependency status of 1,359 projects. To be able to do this, we restricted our project space using established filtering criteria to all non-fork, non-toy, somewhat popular ($>$ 10 watchers on GITHUB) projects with a history of TRAVIS CI use ($>$ 50 builds) in Ruby (936) or Java (423). Both languages are very popular on GITHUB (2nd and 3rd, respectively) [4]. Then, we extracted and analyzed build information from TRAVIS CI build logs and the GHTORRENT database for each TRAVIS CI build in its history, detailed in Appendix A. Well-known projects in the TRAVISTORRENT data set include all 691,184 builds from RUBY ON RAILS, GOOGLE GUAVA and GUICE, CHEF, RSPEC, CHECKSTYLE, ASCIIDOCTOR, RUBY and TRAVIS.

Data-set-as-a-service. TRAVISTORRENT[1] provides convenient access to its archived data sets and free analytic resources: Researchers can directly access an in-browser SQL shell to run their queries on our infrastructure, and download SQL dumps or the compressed data set as a CSV file (1.8 GB unpacked). It also provides documentation and a getting started tutorial. We share all tools we wrote to crave the data on TRAVISTORRENT as OSS, allowing for future extensions and bug fixes by the community.

MSR '17

[1] `http://travistorrent.testroots.org`

APPENDIX

In the appendix, in addition to a list of references, we detail the technical challenges we had to overcome when linking Travis builds to GitHub commits and when analyzing the build logs.

A. DATA SAMPLE

In this section, we outline all fields available and describe an abbreviated data sample from TRAVISTORRENT.

General Data Structure. In the TRAVISTORRENT data set, each data point (row) represents a build job executed on Travis. Every such data point synthesizes information from three different sources: The project's git repository (prefixed `git_`), data extracted from GitHub through GHTorrent (prefixed `gh_`), and data from Travis's API and an analysis of the build log (prefixed `tr_`). In total, we provide 55 data fields for each build. These are described in detail in Table 1.

Sample. The last column of Table 1 features an exemplary data point from the famous `rails/rails` project (note that there currently are 2,640,824 data points more like this in TRAVISTORRENT). Here, we are shortly highlighting some key observations.

The data sample we picked is a pretty interesting, as it is quite unusual for Rails. Not surprisingly, Rails's `project_name` is `rails/rails`. When the commit was made, 168 people had made contributions to it (it is important to realize that all metrics are calculated for the point in time in which the commit was made, so `gh_team_size` for example will grow over time). The build we are looking at (1543966) comprises two commits (the latest commit built, c1d9c11, and a predecessor 87a2f021), most likely because both commits were pushed in one go and Travis naturally builds the latest available commit. This commit is not a Pull Request (`gh_is_pr` is false), but made directly onto the stable development branch (4-1-stable). We could resolve a predecessor build, 39557888. By search in TRAVISTORRENT for the predecessor, we could for example see whether this unusual commit directly onto the stable branch was made in order to fix an urgent problem. We can see that our BUILDLOGANALYZER picked up a Ruby build with the testunit framework. 310 tests were executed successfully, until one test (SerializedAttributeTest) failed, which took 28.2 seconds (`tr_testduration`). Very unusual for Rails is that despite the failing test (`tr_tests_fail`), the overall build status was still considered passed (`tr_status`). A deeper investigation could now look into how many times this happens, and if only perhaps on specific tests, which might be ignored.

B. TECHNICAL CHALLENGES

In this section, we describe the technical challenges we faced when extracting and synthesizing the data set for TRAVISTORRENT in order to give miners a better understanding of how we came up with (1) the collection and (2) the analysis of build logs and (2) the mapping between Travis builds and GitHub commits.

We structure this by referring to the tools we used to extract and create the TRAVISTORRENT data set [4]. Our data collection and analysis pipeline is written in Ruby and R. For replication purposes and to stimulate further research, our tools are in the public domain.

B.1 Data Collection

TravisPoker. To find out which and how many projects on GitHub use TRAVIS CI, we implemented TRAVISPOKER. This fast and lightweight application takes a GITHUB project name as input (for example, RAILS/RAILS), and finds out if and how many TRAVIS CI builds were executed for this project.

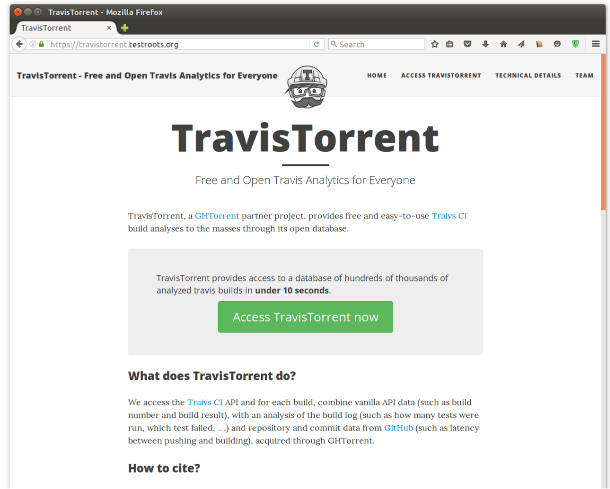

Figure 1: TRAVISTORRENT (`http://travistorrent.testroots.org`) on July, 20th, 2016.

TravisHarvester. We implemented TRAVISHARVESTER to aggregate detailed information about a project's TRAVIS CI build history. It takes as input a GITHUB project name and gathers general statistics on each build in the project's history in a CSV file. Associated with each build entry in the CSV are the SHA1 hash of the GIT commit, the branch and (if applicable) pull request on which the build was executed, the overall build status, the duration and starting time and the sub jobs that TRAVIS CI executed for the different specified environments (at least one job, possibly many for each build). TRAVISHARVESTER downloads the build logs for each build for all jobs and stores them alongside the CSV file.

To speed up the process of retrieving thousands of log files for each project, we parallelize our starter scripts for TRAVIS HARVESTER with GNU PARALLEL.

B.2 Analysis of Build Logs.

BUILDLOG ANALYZER is a framework that supports the general-purpose analysis of TRAVIS CI build logs and provides dedicated Java and Ruby build analyzers that parse build logs in both languages and search for output traces of common testing frameworks.

The language-agnostic BUILDLOG ANALYZER reads-in a build log, splits it into the different build phases, and analyzes the build status and run time of each phase. The fold for the SCRIPT phase contains the actual build and continuous testing results. The BUILDLOG ANALYZER dispatches the automatically determined sub-BUILDLOG ANALYZER for further examination of the build phase.

For Java, we support the three popular build tools MAVEN, GRADLE, and ANT. In Java, it is standard procedure to use JUNIT as the test runner, even if the tests themselves employ other testing frameworks, such as POWERMOCK or MOCKITO. Moreover, we also support TESTNG, the second most popular testing framework for Java. Running the tests of an otherwise unchanged project through MAVEN, GRADLE and ANT leads to different, incompatible build logs, with MAVEN being the most verbose and GRADLE the least. Hence, we need three different parsers to support the large ecosystem of popular Java build tools. As a consequence, the amount of information we can extract from a build log varies per build technology used. Moreover, some build tools give users the option to modify their console output, albeit rarely used in practice.

Table 1: Description of TRAVISTORRENT's data fields and one sample data point from RAILS/RAILS

Column Name	Description	Unit	Example
row	Unique identifier for a build job in TravisTorrent	Integer	1543966
git_commit	SHA1 Hash of the commit which triggered this build (should be unique world-wide)	String	c1d9c11cbe3d20f2...
git_merged_with	If this commit sits on a Pull Request (gh_is_pr true), the SHA1 of the commit that merged said pull request	String	
git_branch	Branch git_commit was committed on	String	4-1-stable
git_commits	Preceding commits that were not built (e.g., transferred in one push, ...) this build comprises	List of Strings	87a2f02199d21a2aa...
git_num_commits	The number of commits in git_commits, to ease efficient splitting	String	1
git_num_committers	Number of people who committed to this project	Integer	1
gh_project_name	Project name on GitHub (in format user/repository)	String	rails/rails
gh_is_pr	Whether this build was triggered as part of a pull request on GitHub	Boolean	false
gh_lang	Dominant repository language, according to GitHub	String	ruby
gh_first_commit_created_at	Push date of first commit in git_commits to GitHub	ISO Date (UTC+1)	2014-04-18 20:12:32
gh_team_size	Size of the team contributing to this project	Integer	168
gh_num_issue_comments	If git_commit is linked to an issue on GitHub, the number of comments on that issue	Integer	0
gh_num_commit_comments	The number of comments on git_commit on GitHub	Integer	0
gh_num_pr_comments	If gh_is_pr is true, the number of comments on this pull request on GitHub	Integer	0
gh_src_churn	The churn of git_commit, i.e. how much production code changed in the commit, based on lines	Integer	4
gh_test_churn	The churn of git_commit, i.e. how much test code changed in the commit, based on lines	Integer	8
gh_files_added	Number of files added in git_commit (this is generally correlated with the churn)	Integer	0
gh_files_deleted	Number of files deleted in git_commit (this is generally correlated with the churn)	Integer	0
gh_files_modified	Number of files modified in git_commit (this is generally correlated with the churn)	Integer	3
gh_tests_added	How many test cases were added in git_commit (e.g., for Java, this is the number of @Test annotations)	Integer	0
gh_tests_deleted	How many tests were deleted in git_commit (e.g., for Java, this is the number of @Test annotations)	Integer	0
gh_src_files	Number of production files in the repository	Integer	
gh_doc_files	Number of documentation files in the repository	Integer	
gh_other_files	Number of remaining files which are neither production code nor documentation	Integer	
gh_commits_on_files_touched	Number of commits that touched (added/deleted/modified) the files in git_commit previously	Integer	93
gh_sloc	Number of executable production source lines of code, in the entire repository	Integer	53421
gh_test_lines_per_kloc	Test density. Number of lines in test cases per 1,000 gh_sloc	Double	2191.011
gh_test_cases_per_kloc	Test density. Number of test cases per 1,000 gh_sloc	Double	188.3342
gh_asserts_cases_per_kloc	Assert density. Number of assertions per 1,000 gh_sloc	Double	535.0143
gh_by_core_team_member	Whether this commit was authored by a core team member	Boolean	true
gh_description_complexity	If gh_is_pr is true, the Pull Request's textual description complexity	Integer	
gh_pull_req_num	Pull request number on GitHub	Integer	
tr_build_id	Unique build ID on Travis	String	23298954
tr_status	Build status (pass, fail, errored, cancelled)	String	passed
tr_duration	Overall duration of the build	Integer (in seconds)	23389
tr_started_at	Start of the build process	ISO Date (UTC)	2014-04-18 19:12:32
tr_jobs	Which Travis jobs executed this build (number of integration environments)	List of Strings	[23298955, ...]
tr_build_number	Build number in the project	Integer	15459
tr_job_id	This build job's id, one of tr_jobs	String	23298981
tr_lan	Language of the build, as recognized by BUILDLOGANALYZER	String	ruby
tr_setup_time	Setup time for the Travis build to start	Integer (in seconds)	0
tr_analyzer	Build log analyzer that took over (ruby, java-ant, java-maven,java-gradle)	String	ruby
tr_frameworks	Test frameworks that tr_analyzer recognizes and invokes (junit, rspec, cucumber, ...)	List of Strings	testunit
tr_tests_ok	If available (depends on tr_frameworks and tr_analyzer): Number of tests passed	Integer	310
tr_tests_fail	If available (depends on tr_frameworks and tr_analyzer): Number of tests failed	Integer	1
tr_tests_run	If available (depends on tr_frameworks and tr_analyzer): Number of tests were run as part of this build	Integer	311
tr_tests_skipped	If available (depends on tr_frameworks and tr_analyzer): Number of tests were skipped or ignored in the build	Integer	
tr_failed_tests	All tests that failed in this build	List of strings	SerializedAttributeTest
tr_testduration	Time it took to run the tests	Double (in seconds)	28.2
tr_purebuildduration	Time it took to run the build (without Travis scheduling and provisioning the build)	Double (in seconds)	
tr_tests_ran	Whether tests ran in this build	Boolean	true
tr_tests_failed	Whether tests failed in this build	Boolean	true
tr_num_jobs	How many jobs does this build have (length of tr_jobs)	Integer	30
tr_prev_build	Serialized link to the previous build, by giving its tr_build_id	String	39557888
tr_ci_latency	Latency induced by Travis (scheduling, build pick-up, ...)	Integer (in seconds)	1408

Example 1: Standard output from MAVEN regarding tests

```
1  -------------------------------------------------------
2   T E S T S
3  -------------------------------------------------------
4  Running nl.tudelft.watchdog.ClientVersionCheckerTest
5  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
     elapsed: 0.04 sec
6
7  Results :
8
9  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
10
11 [INFO] All tests passed!
```

Example 1 shows an excerpt of one test execution from the TESTROOTS/WATCHDOG project. In the output, we can see the executed test classes (line 4), and how many tests passed, failed, errored and were skipped. We also get the test execution time (line 5). Moreover, MAVEN prints an overall result summary (line 9) that the BUILDLOG ANALYZER uses to triage its prior findings. Line 11 shows the overall test execution result. Our BUILDLOG ANALYZER gathers all this information and creates, for each invoked project, a CSV table with all build and test results for each job built. We then aggregate this information with information from the build status analyzer step by joining their output. TRAVISTORRENT provides convenient access to this data. GRADLE is much less verbose than MAVEN, providing us with fewer information.

By contrast, in Ruby, the test framework is responsible for the console output: it is no different to invoke RSPEC through RAKE than through BUNDLER, the two predominant Ruby build tools. For Ruby, we support the prevalent TEST::UNIT and all its off springs, like MINITEST. Moreover, we capture behavior driven tests via RSPEC and CUCUMBER support.

B.3 Data Linearization And Synthesization

If we want to answer questions such as "Does the use of CI lead to higher-quality products?", we need to make a connection between the builds performed on TRAVIS CI and the repository which contains the commits that triggered the build. We call this build linearization and commit mapping, as we need to interpret the builds on TRAVIS CI as a directed graph and establish a child-parent relationship based on the GIT commits that triggered their execution. Although this sounds trivial, since there should be a 1:1 relationship between builds and commit, there are six different scenarios (a–f) arising from GIT's non-linear nature that make this a hard task, as we discuss in the following. During this step, we also assessed the status of the project at the moment each build was triggered by extracting and synthesizing information from two sources: the project's GIT repository and its corresponding entry in the GHTORRENT database.

Figure 2 exemplifies a typical GITHUB project that uses TRAVIS

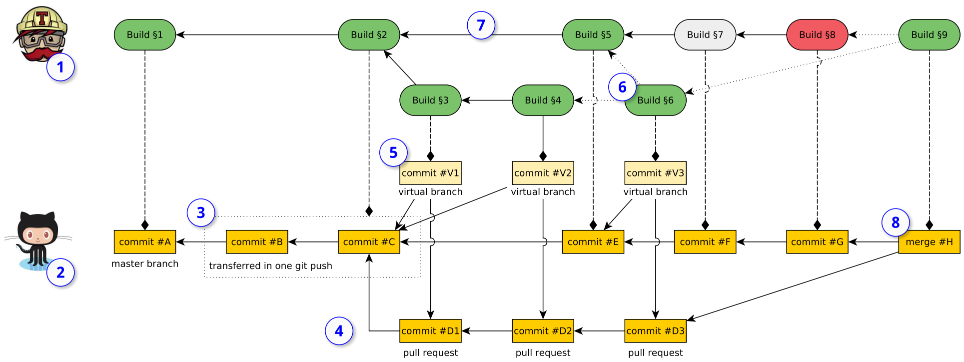

Figure 2: Exemplary of how to match commits from a GITHUB repository to their corresponding TRAVIS CI builds (source: [4]).

CI for its CI. In the upper part ①, we see the TRAVIS CI builds (§1-§9), which are either passed (§1-§6, §9), canceled (§7), or broken (§8). In the lower part ②, we see the corresponding GIT repository hosted on GITHUB with its individual commits (#A-#H). Commits #D1-#D3 live in a pull request, and not on the master branch, traditionally the main development line in GIT.

a) Build §1 showcases a standard situation, in which the build passed and the commit id stored with the build leads to the correct commit #A that triggered build §1. However, there are a number of more complex situations.

b) If multiple commits are transferred in one `git push` ③, only the latest of those commits is built (§2). In order to get a precise representation of the changes that lead to this build result, we have to aggregate commits #B and #C.

c) It is a central function of TRAVIS CI to support branches or pull requests ④, such as commit #D1. When resolving builds to commits, we know from the API that §3 is a pull request build. Its associated commit points us to a virtual integration commit #V1 that is not part of the normal repository, but automatically created as a remote on GITHUB ⑤. This commit #V1 has two parents: 1) the latest commit in the pull request (#D1), and 2) the current head of the branch the pull request is filed against, the latest commit on the master branch, #C. Similarly, when resolving the parent of §4, we encounter a #V2, resolve it to #D2 and the already known #C. We also know that its direct parent, #D1, is branched-off from #C. Hence, we know that any changes from build result §4 to §3 were induced by commit #D2.

d) In the case of build §6 on the same pull request ⑥, its direct predecessor is unclear: we traverse from #V3 to both 1) commit #D2 in the pull request, which is known, and to 2) #E on the master branch, which is unknown and cannot be reached from any of our previous commits #D2, #D1, or #C. This is because there was an intermediate commit #E on the master branch in-between, and pull requests are always to be integrated onto the head commit of the branch they are filed against. In such a case, one build can have multiple parents, and it is undecidable whether the changes in #D3, #E or a combination of both lead to the build result §6.

e) Build §5 shows why a simple linearization of the build graph by its build number would fail: It would return §4 as its predecessor, when in reality, it is §2 ⑦. However, even on a single branch, there are limits to how far GIT'S complex commit relationship graph can be linearized and mapped to TRAVIS CI builds. For example, if a build is canceled (§7), we do not know about its real build status – it might have passed or broken. As such, for build §8, we cannot say whether the build failure resulted from changes in commit #F or #G.

f) When merging branches or pull requests ⑧, a similar situation to c) occurs, in which one merge commit #H has two predecessors.

C. WHY USE TRAVISTORRENT?

In this section, we describe why TRAVISTORRENT lends itself to the MSR'2017 mining challenge.

- TRAVISTORRENT features a novel data set in an emerging research field that is highly relevant in practice but still lacks quantifiable evidence and that has not yet been explored multiple times before.
- TRAVISTORRENT eases work with a huge set of data (downloading and processing of terabytes of build logs) some groups might not otherwise have the opportunity to work with, especially students.
- Our analytics service and tutorial make running custom queries against the data set easy, without any local setup requirements.
- Researchers can easily connect their existing expertise, tools and results from previous research on GITHUB.
- With more than 1,000 projects, TRAVISTORRENT is sufficiently large for multiple researchers to work on sub-aspects, yet still manageable on standard computers.

D. REFERENCES

[1] G. Booch, "Object oriented design with applications. redwood city," 1991.

[2] M. A. Cusumano and R. W. Selby, "Microsoft secrets," 1997.

[3] M. Fowler and M. Foemmel, "Continuous integration," 2006. `http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf`.

[4] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[5] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from GitHub," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pp. 401–405, IEEE, 2014.

[6] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Joint Meeting European Softw. Engineering Conf. & Symp. Foundations of Softw. Engineering (ESEC/FSE)*, pp. 805–816, ACM, 2015.

[7] M. Hilton, T. Tunnell, D. Marinov, D. Dig, K. Huang, *et al.*, "Usage, costs, and benefits of continuous integration in open-source projects," tech. rep., Oregon State University. School of Electrical Engineering & Computer Science, 2016.