Chamulteon: Coordinated Auto-Scaling of Micro-Services

Bauer, André; Lesch, Veronika; Versluis, Laurens; Ilyushkin, Alexey; Herbst, Nikolas; Kounev, Samuel

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

Chamulteon: Coordinated Auto-Scaling of Micro-Services

André Bauer*, Veronika Lesch*, Laurens Versluis†, Alexey Ilyushkin‡, Nikolas Herbst* and Samuel Kounev*

*University of Würzburg, Germany – Email: {firstname}.{lastname}@uni-wuerzburg.de
†Vrije Universiteit Amsterdam, The Netherlands – Email: l.f.d.versluis@vu.nl
‡Delft University of Technology, The Netherlands – Email: a.s.ilyushkin@tudelft.nl

Abstract—Nowadays, in order to keep track of the fast-changing requirements of Internet applications, auto-scaling is used as an essential mechanism for adapting the number of provisioned resources to the resource demand. The straightforward approach is to deploy a set of common and open-source single-service auto-scalers for each service independently. However, this deployment leads to problems such as bottleneck-shifting and increased oscillations. Existing auto-scalers that scale applications consisting of multiple services are kept closed-source. To face these challenges, we first survey existing auto-scalers and highlight current challenges. Then, we introduce Chamulteon, a redesign of our previously introduced mechanism, which can scale applications consisting of multiple services in a coordinated manner. We evaluate Chamulteon against four different well-cited auto-scalers in four sets of measurement-based experiments where we use diverse environments (VM vs. Docker), real-world traces, and vary the scale of the demanded resources. Overall, Chamulteon achieves the best auto-scaling performance based on established user-oriented and endorsed elasticity metrics.

Index Terms—Cloud Computing, Auto-Scaling, Elasticity, Workload Forecasting, Service Demand Estimation, Container, Benchmarking, Metrics

I. INTRODUCTION

"Cloud computing has emerged as a popular computing model to support processing large volumetric data using clusters of commodity computers" [1, p. 1]. Essentially, cloud computing provides elastic on-demand access to data center resources. Although there are threshold-based scaling mechanisms such as the ones used in Amazon EC2, business-critical applications in clouds are usually still deployed with over-provisioned resources. This strategy is pursued to avoid becoming dependent on an auto-scaling mechanism with its possibly wrong or badly-timed scaling decisions. There are exceptions like Netflix, which uses its own auto-scaling engine called Scryer[1].

Current research in the scientific community is focused on novel approaches for reliable proactive auto-scaling to improve auto-scaling mechanisms and increase the trust of the industry in auto-scaling. Generally, auto-scalers can be distinguished based on whether they are capable of scaling applications composed of only one service or applications consisting of multiple services. Examples from the first category include Chameleon, which we proposed in our previous work [2], and the famous open-source auto-scalers React, Adapt, Hist, and

---

[1]Netflix Scryer: https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270

Reg [3]–[6]. In contrast, popular auto-scalers from the second category, such as AutoMap, AGILE, and CloudScale [7]–[9], are closed-source.

Modern applications are often designed based on a micro-service architecture, and thus they consist of multiple services. As the associated auto-scalers mentioned above are closed-source, the straightforward implementation of an auto-scaling mechanism for such applications is to instantiate an open-source individual single-service auto-scaler for every service. That is, each service is observed and scaled independently by an auto-scaler. However, this approach can lead to problems like oscillations and bottleneck shifting. B. Urgaonkar et al. [10] provide a detailed example of these problems.

Another approach is the implementation of a black-box auto-scaler for applications consisting of multiple services [10]. This auto-scaler makes scaling decisions based on the observed end-to-end response time of the application. Whenever one of the services becomes a bottleneck, the response time will increase and cause a violation of the service-level agreement (SLA). The auto-scaler can detect this and trigger the auto-scaling mechanism. However, the problem is how to determine which particular services need to be scaled and by how much.

To tackle the described problem, we formulate the following three research questions: (RQ1) *What mechanisms exist for scaling applications with multiple services and what are the open challenges?* (RQ2) *How can we enable coordinated scaling of applications with multiple services?* (RQ3) *How well does our Chamulteon approach perform compared to state-of-the-art auto-scaling mechanisms?*

Towards addressing the aforementioned questions, our contribution is four-fold:

1) We survey existing auto-scalers for applications consisting of multiple services and discuss challenges for such auto-scalers as well as their benchmarking (Section II).
2) We redesign our white-box auto-scaler Chameleon [2] to enable the coordinated scaling of applications with multiple services. That is, we introduce the components, the decision making logic and the conflict resolution introduced by proactive scaling coupled with a reactive fallback (Section III-A). The redesigned version, called Chamulteon, maintains a performance model of the application, observes and predicts the request arrival rates, and estimates the service time of each service.

This approach is applicable also to classical multi-tiered applications without changes.

3) We conduct measurement-based experiments with real-world system traces in different environments: the stressed application is, first, deployed on virtual machines and then, deployed in Docker containers. Further, we change the scale of peak demand from 20 to 60 to 120 application instances.

4) We compare Chamulteon with the open-source auto-scalers React, Adapt, Hist, and Reg [3]–[6] (Section V). To the best of our knowledge, existing auto-scalers for applications with multiple services are closed-source. We analyze the results with a set of SPEC-endorsed elasticity metrics [11], in addition to metrics capturing the user-perspective. Further, we define a new aggregation metric to capture the coordinated auto-scaling performance overall in a single score (Section IV-D3). The results of the experiments show that Chamulteon exhibits the best auto-scaling performance in comparison to the other auto-scalers.

II. State-of-the-Art and Open Challenges

Based on what we collected from recent [12] and established survey articles [13] in addition to a systematic literature search, we start by giving an overview of existing auto-scalers for applications composed of multiple services or tiers. Then, we discuss the difference between our approach and existing auto-scalers. Finally, we present the challenges that existing multi-service auto-scalers face and the challenges for benchmarking such mechanisms.

A. Scaling Applications with Multiple Services

This section surveys state-of-the-art auto-scaling approaches for scaling multi-service applications. We consider both reactive and proactive auto-scalers. Table I summarizes the surveyed auto-scalers.

In 2018, Khorsand et al. [14] presented the *FAHP* approach for autonomic resource management of multi-tier cloud applications. This work implements a MAPE-K loop leveraging a fuzzy analytic hierarchy process. The performance dependencies between the different tier are black-box. *FAHP* is evaluated in simulation using real workload traces.

The *HybridScaler* [15] presented by Wu et al. in 2016 is designed for multi-tier web applications. This approach is considered as hybrid because it combines vertical reactive scaling to accommodate bursting workloads with proactive horizontal scaling for mid- to long-term workload changes. *HybridScaler* maintains a resource pressure model per tier and is aware of the applied cost-model, i.e., to optimize in the case of an hourly billing interval. There is no dedicated focus on how to realize the coordination of scaling between the different services.

In 2015, Beltrán presented *AutoMAP* [7], a reactive auto-scaler. Its provisioning model is based on response time triggers, so it implements a rule-based approach. It supports vertical and horizontal scaling as well as cost-effectiveness.

Cost-effectiveness in this context means finding the optimal resource configuration via choosing virtual machine image sizes to reduce overall costs.

The *AGILE* auto-scaler is a proactive mechanism introduced by Nguyen et al. in 2013 [8]. It uses a wavelet-based approach for providing medium-term resource demand predictions of up to two minutes. Based on these predictions new application server instances are started up to be prepared for the load peak. AGILE's architecture consists of agile slaves and an agile master. The slaves monitor resource usage, and the agile master maintains a dynamic resource pressure model for each application using online profiling.

Shen et al. [9] introduced in 2011 an elastic resource scaling approach called *CloudScale*. CloudScale is a proactive approach that supports vertical scaling for voltage and memory scaling and migrates virtual machines out of overloaded hosts. It uses a hybrid prediction approach that consists of signature-driven and state-driven algorithms. If SLO violations can be solved by local scaling, scaling of CPU and memory is applied. Otherwise, the approach migrates virtual machines out of the over-utilized host until the utilization of the host is reduced.

Zhu and Agrawal [16] introduced in 2010 an approach based on control theory for vertical scaling of applications in cloud environments. They focus on a series of interacting service components and scale them independently. The vertical scaling is based on CPU cycles and memory allocation. They take a fixed time-limit and a resource budget into account to maximize the Quality of Service (QoS). They developed a system model to capture the relationship between the input of the system and its performance. The model uses an auto-regressive-moving-average with exogenous inputs (ARMAX) of second order to represent the system behavior.

Sharma et al. introduced in their work from 2012 an approach based on queueing theory [17]. Their approach can scale the targeted application horizontally and vertically. The vertical scaling is executed by choosing differently sized virtual machine images. The application is modeled as a chain of M/G/1-PS queues, and the decision logic considers the end-to-end response time. The approach has a cost-aware component that reconfigures the scaling decisions to find the most cost-efficient heterogeneous configuration.

In 2010, Bi et al. [18] proposed an approach that uses a flexible hybrid queueing model to determine the number of virtual machines for each tier. The method collects performance metrics (like arrival rate, average service time and CPU utilization) and analyses them to find appropriate scaling decisions. The first tier is modeled as an M/M/n queueing system and the preceding tiers as multiple M/M/1 systems with FCFS scheduling.

B. Distinctive Features of Chamulteon

Like the majority of the reviewed algorithms, Chamulteon is based on queueing theory and supports horizontal scaling. Though some approaches support vertical scaling like AutoMAP [7], CloudScale [9], HybridScaler [15], and the approach from U. Sharma [17], only CloudScale supports

TABLE I
OVERVIEW OF RELATED WORK

Related Work	Methodology	reactive (r) / proactive (p)	vertical (v) / horizontal (h)	Cost-Efficient	Evaluation experimental (exp) / simulative (sim)	Open-Source
[14]	MAPE loop, fuzzy logic	r	h	no	sim: RUBiS, real workloads	✗
[15]	resource pressure model	r & p	h & v	yes	exp: RUBiS, real workload	✗
[7]	response time, threshold triggered	r	h & v	yes	exp: RUBiS, synthetic workload	✗
[8]	wavelets, resource demand prediction	p	h	no	exp: RUBiS, real workloads	✗
[9]	load pattern extraction (FFT)	p	v	no	exp: RUBiS, real workloads	✗
[16]	control theory, ARMAX	r	v	no	exp: GLFS, Volume Rendering	✗
[17]	queueing network M/G/1-PS	r	h & v	yes	sim & exp: TPC-W, real workload	✗
[18]	que. net. first: M/M/c, rest: M/M/1	p	h	no	exp: RUBiS, synthetic workload	✗

migration of virtual machines. As there is no high demand for migrations of micro-service instances, Chamulteon does not support migration. In contrast to the evaluation/simulation of the reviewed approaches, Chamulteon is evaluated with different realistic experiment setups, resource demands, and real-world traces.

Although a recent survey [12] highlights the importance of combining reactive and proactive auto-scaling mechanisms, most approaches can scale applications either proactively or reactively. In contrast to existing hybrid auto-scalers (e.g., HybridScaler [15]) that combine reactive and proactive mechanisms, Chamulteon: (i) leverages long-term predictions from time series analysis in combination with (ii) predictive models from queueing theory, and also integrates a (iii) reactive fallback mechanism. Due to these two integrated mechanisms, Chamulteon has to resolve conflicts introduced by the different decisions. The reviewed approaches do not explicitly address this issue.

C. Challenges

In contrast to scaling applications with only one service, the scaling of applications consisting of multiple services introduces many more challenges. A short overview is discussed in this section.

1) Auto-Scaling Challenges: When applying auto-scalers to applications that consist of multiple services diverse challenges arise. The first and most crucial challenge is which properties to consider when scaling an application? The individual services have their specific service demands, requirements for scalability, and maximum capacity. All of these constraints need to be considered when scaling the application in a coordinated manner. The second challenge is the selection and configuration of an auto-scaling approach? Most auto-scalers found in the literature are either proactive or reactive. Both approaches have specific advantages and disadvantages. Proactive mechanisms can scale at an early stage of a load spike. However, the workload forecasts they are based on are not always reliable due to the uncertainty of forecasting models. With reactive scaling, this uncertainty is eliminated as all decisions are made based on actual measurements, but such mechanisms can only react after an overload situation occurs. In case an auto-scaler implements both proactive and reactive mechanisms, the process of deciding based on which mechanism the application should be scaled poses a crucial challenge. Finally, another challenge is how auto-scalers that employ both horizontal and vertical scaling can find a trade-off between these two approaches to satisfy the demand?

2) Benchmarking Challenges: Various metrics are used to quantify the scaling behavior of an auto-scaler such as cost, system, and user-oriented metrics. While using the system metrics in applications containing only one service is easy, the usage becomes more difficult with applications containing multiple services. For applications with one service, there is only a limited number of possible configurations that can be measured and quantified. In contrast, an application with multiple services has $\prod n_i$ possible configurations (n_i is the maximum allowed number of resources for service i). So, it takes a long time to measure and quantify each configuration. Furthermore, there may be configurations that are equally optimal regarding the served requests. Thus, the challenge is how to find the optimal configurations for multi-service applications?

III. THE CHAMULTEON APPROACH

We now present Chamulteon, a novel auto-scaling mechanism specifically designed to support the coordinated auto-scaling of multi-service applications. Chamulteon is based on our original auto-scaler Chameleon [2], which is a hybrid proactive auto-scaler combining multiple different proactive methods coupled with a reactive fallback mechanism. The system consists of two independent cycles: (i) the reactive cycle that monitors the application and scales reactively in short intervals and (ii) the proactive cycle which predicts the demand at longer intervals for a set of future scaling intervals.

In Section III-A, we highlight the changes and components of Chamulteon. After that, we explain the decision-making process. In Section III-C, the resolution of conflicts introduced by the two cycles is explained. Finally, the assumptions and limitations of Chamulteon are stated.

A. Redesign of the Original Chameleon

Chamulteon is based on a redesign of the basic architecture and workflow of Chameleon for scaling single services, which consists of four main components: (i) a controller, (ii) a
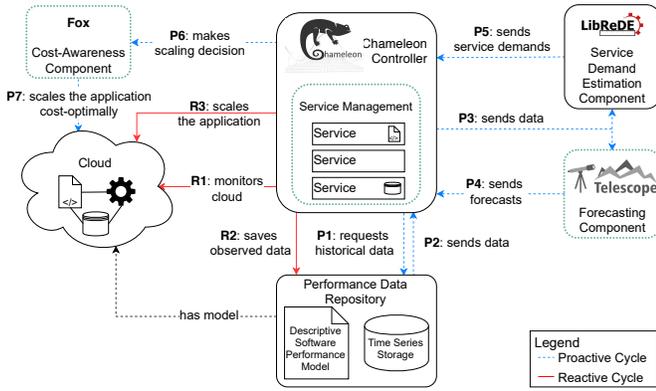
Fig. 1. Components of Chamulteon.

performance data repository, (iii) a forecasting component, and (iv) the service demand estimation component. The performance data repository contains a time series storage and an instance of a descriptive performance model of the dynamically scaled application based on the Descartes Modeling Language (DML) [19], [20]. Figure 1 shows the architecture of Chamulteon. The red lines show the reactive cycle; the blue dashed lines depict the proactive cycle. The newly introduced components, as well as the changed components, are highlighted with a green dotted line.

The first change compared to the original Chameleon is the *service management*. This component allows Chamulteon to make decisions for each service while taking each service and its associated decisions into account. The decisions are made with knowledge about the other services, so that scaling can be triggered earlier on succeeding services. This approach allows removing oscillations. Moreover, a *Cost-Awareness Component* called Fox [21] has been added, which is useful when running an application in a public cloud. This component, if activated, reviews all decisions proposed by the *Controller* and evaluates whether they are cost-efficient or not. Currently, there are two implemented charging strategies that are used by Amazon EC2 and the Google Cloud. Finally, the forecasting component now leverages a hybrid decomposition-based forecasting method Telescope [22], designed specifically for auto-scaling use-cases.

1) Forecasting Component [2]: To enable proactive decisions, the arrival rates for the next reconfiguration intervals are forecast. To minimize the forecasting overhead, this component is only called if an earlier forecast has no more predicted values for future arrival rates or a configurable drift between the forecast and the recent monitoring data is detected. The drift is calculated based on the mean absolute scaled error metric (MASE) [23].

2) Service Demand Estimation Component [2]: The LibReDE library offers eight different estimation approaches for service demands on a per request type basis [24]. We use the estimator based on the service demand law to minimize the estimation overhead. As input, the request arrivals per resource and the average monitored utilization are required.

For complex service deployments, LibReDE requires structural knowledge about the application deployment that is provided by the DML performance model instance.

3) Cost-Efficiency Component: The cost efficiency component, FOX [21], leverages knowledge of the charging model of the public cloud and reviews the scaling decisions proposed by the auto-scaler in order to reduce the charged costs to a minimum. More precisely, FOX delays or omits releasing resources to avoid additional charging costs if the resources will be required again within the charging interval. By using this review logic, the charging interval of each resource is utilized as efficiently as possible.

4) Summary of the Redesign: To enable scaling of applications with multiple services, the main differences are summarized: (i) The request arrival rates are only monitored and forecast for the user-facing service. (ii) The amount of requests arriving at each service is determined based on the scaling decision for the predecessor service. (iii) For the dynamic on-demand forecasting, we use a novel forecasting method called Telescope [22]. This method is specifically designed for such use cases; it has a reliable forecast accuracy and a short time-to-result. (iv) The service management and cost-awareness components are added as new components.

B. Decision Making Process

The decision making process consists of a *proactive* and a *reactive phase*, invoked from corresponding proactive or reactive monitoring cycles for predefined auto-scaling intervals. Both cycles make decisions for each service based on the theoretical queueing theory-based utilization ρ. Therefore, Chamulteon transforms the instance of the DML performance model into a product-form queueing network [19], [25], whereby each service is modeled as an M/M/n/∞ queue. As each service instance is mapped to exactly one resource instance (e.g., container), the resource instance and the service instance are interchangeable in the modeling perspective. If the utilization exceeds/undershoots the predefined service thresholds, the required number of instances is calculated. After all decisions are made, they can further be adjusted for cost-efficiency if the associated component is activated.

Algorithm 1 depicts the proactive decision finding for a specific predicted arrival rate and service. The decisions are made based on the predicted arrival rate λ, the estimated service demand μ, and the number of running instances n (lines 2–4). The arrival rate for each service is estimated according to the forecast. If the service is the user-facing service, the arrival rate is equal to the predicted value. Otherwise, the predicted arrival rate is estimated based on an invocation graph. This graph is extracted from the DML model that also captures the request types and their control flows. More precisely, the algorithm checks whether there are enough instances to process the incoming arrival rate for each service. If there are too few resource instances, the arrival rate λ is set to the maximum arrival rate that can be served by the bottleneck service. Otherwise, the arrival rate λ is equal to the predicted arrival rate (line 5).

Algorithm 1: Proactive decision logic.

---

1 **Decision Logic** for service s at time t in the future
2   λ = getForecast(t);
3   μ = getAvgServiceDemand(s);
4   n = getNumInstances(s);
5   λ = estimateArrivals(λ,s); `// estimates future`
      `arrival rates based on invocation graph`
6    $\rho = \frac{\lambda}{\mu \cdot n}$; `// calc. the future avg. utilization`
7   **if** $\rho \geq \rho\_upper$ **then**
8   **while** $\rho \geq \rho\_upper$ **do**
9          $\rho = \frac{\lambda}{\mu \cdot (++n)}$; `// calc. new avg. util.`
10   n = min(n, maxInstances(s));
11   **else if** $\rho < \rho\_lower$ **then**
12   **while** $\rho < \rho\_lower$ **do**
13          $\rho = \frac{\lambda}{\mu \cdot (--n)}$; `// calc. new avg. util.`
14   n = max(n, minInstances(s));
15   **return** *decision(n, s, t)*;

Based on this information, the service utilization is calculated (line 6). If the calculated utilization exceeds the upper threshold, the number of required instances is computed using the target utilization. If the predicted number of instances exceeds the pre-set maximum allowed number of instances for this service, the prediction is limited to the pre-set maximum (lines 7–10). Analogously, if the calculated utilization falls below the lower threshold, the number of required instances is calculated based on the minimum allowed number of instances for this service (lines 11–14). Differently than shown, the safety clause for jumping out of both while loops is when the maximum or minimum number of instances is reached. Finally, a prediction with the number of required instances for the specific time and service is returned.

C. Decision Conflict Resolution

As Chamulteon consists of a proactive and reactive cycle, management of scaling decisions created by the two cycles is required. That is, conflicts have to be resolved, and the execution of the decisions has to be scheduled.

1) Scope Resolution: Each decision for a service has a valid period in which no other decision is executed. Due to the different reconfiguration intervals of reactive and proactive cycles, there may be a reactive and proactive decision for the same time interval. If the proactive decision is trustable and wants to scale up or down, the reactive decision is omitted. Otherwise, the proactive decision is skipped. In this context, trustable is a threshold that refers to the model accuracy of the underlying forecast on which the decision is based.

2) Time Resolution: As Chamulteon executes a new forecast as soon as a drift between the last forecast and the monitored arrival rates occurs, there may be proactive decisions with different underlying forecasts for the same time period. Assuming that decisions based on the newest forecast contain more up-to-date information, all proactive events for the same time period are skipped.

D. Assumptions and Limitations

We make the following explicit assumptions: (i) To obtain good forecasts with a model of the seasonal pattern, the availability of two days of historical data is required. With less historical data, the forecasts contain only trend and noise components resulting in a decreased accuracy and fewer proactive scaling decisions. (ii) Chamulteon requires an external monitoring component that collects the required values (e.g., the arrival rate). (iii) Chamulteon is currently focused on request-based applications as it relies on the respective service demand estimation models. (iv) The DML model has to be either created manually or by use of an external tool. (v) Each service instance is deployed on homogeneous resources.

IV. Evaluation Methodology and Setup

In this section, we describe the evaluation methodology and setup. First, the evaluation environment is described. Section IV-B summarizes the considered application and respective workloads. Afterwards, Section IV-C introduces the competing auto-scalers considered in the evaluation. We define the set of elasticity and user-oriented metrics for evaluating and comparing the different auto-scalers in Section IV-D.

A. Evaluation Environment

The experiment setup consists of multiple components: the benchmark application, a load balancer, the auto-scalers under evaluation, and the load generator. Our experimental environment is based on CloudStack[2]. CloudStack manages virtualized KVM-server hosts and is running in a cluster of 11 identical servers (HP DL160 Gen9 with eight physical cores @2.4Ghz and 32GB). Eight of them are reserved for CloudStack with overbooking and hyperthreading deactivated. The remaining three servers are used to host the load balancer (Traefik[3]), the CloudStack management system, Chamulteon, and the other auto-scalers, as well as the load driver[4]. For the evaluation, we use up to 20 virtual machines (Debian 4.9.110 with 2 vCPUs and 8 GB) on which either the application or a Kubernetes (Rancher[5] v2.1.0 + kubectl v1.11.3) cluster are deployed. Within the Kubernetes cluster, we run up to 120 Docker (v17.03.2-ce) containers.

B. Workload and Application

We use two existing traces from real-life systems to generate representative load intensity profiles: (i) The *BibSonomy* trace consisting of HTTP requests to servers of the social bookmarking system BibSonomy [26] during April 2017. (ii) The German Wikipedia[6] trace containing the page requests to all German Wikipedia projects during December 2013. For a feasible experiment run duration, we pick a selected subset from the traces covering one day and accelerate them to last either an hour or six hours. Due to the fast provisioning times

---

[2]CloudStack: https://cloudstack.apache.org/
[3]Traefik: https://traefik.io/
[4]Load-Gen.: https://github.com/joakimkistowski/HTTP-Load-Generator
[5]Rancher: https://rancher.com/
[6]Wiki Source: https://dumps.wikimedia.org/other/pagecounts-raw/2013/

of Docker containers, measurements covering one hour are sufficient. In contrast, for the VM setup, the experiments are longer and cover 6 hours.

The stressed benchmark application represents a lightweight micro-service application with three different services: (i) UI service, (ii) validation service, and (iii) data service. The application is written in Java and is deployed on a Tomcat server (v8.5). The load generator requests data by sending HTTP requests to the UI service. The UI forwards each request to the validation service for checking its validity. After that, the request is redirected to the data service. This service provides the requested data and sends the response to the UI for rendering the content. On average, for each request, the UI service needs 0.059 seconds, the validation service needs 0.1 seconds, and the data service needs 0.04 seconds to process the request. These values are determined with LibReDE (c.f. Section III-A2) and represent the *service demand* of requests for each service. The service demand captures the average time required from each service for processing a request, excluding any waiting times. That is, the UI can handle up to 17 requests per second, the validation service ten requests per second, and the data service 25 requests per second.

C. Competing Auto-Scalers

For the evaluation, we select five representative auto-scalers that have been published in the literature over the past decade (2008 [5], 2009 [3], 2011 [6], and 2012 [4]). The auto-scalers can be grouped into two classes: (i) auto-scalers that build a predictive workload model based on long-term historical data [5], [6] and (ii) auto-scalers that only use recent history to make auto-scaling decisions [3], [4]. In the remainder of this section, we describe each of these in more detail.

Each auto-scaler is called periodically and receives a set of input values and returns the amount of resources (number of service instances) that have to be added or removed. The input consists of the following parameters: (i) the accumulated number of requests during the last interval, (ii) the estimated service demand per request determined by LibReDE as used in Chamulteon, and (iii) the number of currently running instances. The competing auto-scalers are available online[7]. We used the default configurations, which are also used in a simulative evaluation [27].

As these auto-scalers are not designed to scale applications with multiple services, we extend these auto-scalers to enable scaling such applications. For each service, we deploy an instance of the associated auto-scaler. Further, we adjust the arrival rates at each service. That is, the first service receives the actual observed request rate as input. As inputs for the second and third instances, the number of requests are calculated using the following formula, where $r(i)$ is the request rate at service i, $n(i)$ is the number of instances of service i, and $s(i)$ is the service rate at service i.

$$r(i) := \begin{cases} \text{measured arrival rate} & \text{if } i = 0 \\ \min(r(i-1), n(i-1) \cdot s(i-1)) & \text{if } i > 0 \end{cases}$$

[7]Competing auto-scalers: https://github.com/ahmedaley/Autoscalers [27]

Hence, the request rate at the second and third service is calculated as the minimum of the request rate at the preceding service and the number of service instances of the predecessor service multiplied by the service rate per instance. Thus, if the capacity of the predecessor service is exceeded, the maximum request rate this service can handle is forwarded to the next service. If the service does not operate at full capacity, the arriving request rate is forwarded to the next service.

1) React: In 2009, Chieu et al. [3] presented a reactive scaling algorithm for horizontal scaling. React provisions VM instances based on a threshold or a certain scaling indicator of the web application. The considered indicators include: the number of concurrent users, the number of active connections, the number of requests per second, and the average response time per request. React monitors these indicators for each VM and calculates the moving average. Afterwards, the number of active VM instances (i.e., with active sessions) above or below the given threshold is determined. Then, if all VMs have active sessions above the threshold, new web application instances are provisioned. If there are VMs with active sessions below the threshold and with at least one VM that has no active session, idle instances are removed.

2) Adapt: Ali-Eldin et al. [4] propose an autonomic elasticity controller that changes the number of VMs allocated to a service based on both monitored load changes and predictions of future load intensity. We refer to this technique as *Adapt*. The predictions are based on the rate of change of the request arrival rate, i.e., the slope of the workload, and aims at detecting the envelope of the workload. The designed controller adapts to sudden load changes and prevents premature release of resources, reducing oscillations in the resource provisioning. *Adapt* tries to improve the performance regarding the number of delayed requests and the average number of queued requests, at the cost of some resource over-provisioning.

3) Hist: Urgaonkar et al. [5] propose a provisioning technique for multi-tier web applications. The proposed methodology adopts a queueing model to determine how much resources to allocate in each tier of the application. A predictive technique based on building *Hist*ograms of historical request arrival rates is used to determine the amount of resources to provision at an hourly timescale. Reactive provisioning is used to correct errors in the long-term predictions or to react to unanticipated flash crowds. We refer to this technique as *Hist*.

4) Reg: Iqbal et al. propose a regression-based auto-scaler (hereafter called *Reg*) [6]. This auto-scaler has a reactive component for scale-up decisions and a predictive component for scale-down decisions. When the capacity is insufficient, a scale-up decision is taken and new VMs are added to the service in a way similar to *React*. For scale-down, the predictive component uses a second order regression to predict future load. The regression model is recomputed using the complete history of the workload each time new measurement data is available. When the current load is lower than the provisioned capacity, a scale-down decision is taken using the regression model.

D. Evaluation Metrics

To compare and quantify the performance of different auto-scalers, we use a set of both system- and user-oriented metrics. The system-oriented metrics consist of elasticity metrics [28]. As user-oriented metrics, we report the percentage of SLO violations, and the user satisfaction reflected by the application performance index.

When using only individual metrics for judging the performance of auto-scalers, the results can be ambiguous. Hence, we define the auto-scaling worst-case deviation of each auto-scaler from the theoretically optimal auto-scaler. Each elasticity and aggregate metric is explained in the remainder of this subsection. For the following equations, we define: (i) T as the experiment duration and the current time as $t \in [0, T]$, (ii) s_t as the resource supply at time t, and (iii) d_t as the demanded resource units at time t. The demanded resource units d_t is the minimal amount of resources required to meet the SLOs under the load intensity at time t. Δt denotes the time interval between the last and the current change either in demand d or supply s. The resource supply s_t is the monitored number of running resources at time t.

1) Provisioning Accuracy θ_U and θ_O [28]: These metrics describe the relative amount of resources that are under-provisioned or over-provisioned during the measurement interval, that is, the *under-provisioning accuracy* θ_U is the amount of missing resources required to meet the SLO in relation to the current demand normalized by the experiment time, whereas, the *over-provisioning accuracy* θ_O is the amount of resources that the auto-scaler supplies in excess of the current demand normalized by the experiment time. Values of this metric lie in the interval $[0, \infty)$, where 0 is the best value and indicates that there is no under-provisioning or over-provisioning during the entire measurement interval.

$$\theta_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(d_t - s_t, 0)}{d_t} \Delta t$$

$$\theta_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(s_t - d_t, 0)}{d_t} \Delta t$$

2) Wrong Provisioning Time Share τ_U and τ_O [28]: These metrics capture the time in percentage, in which the system is under-provisioned or over-provisioned, during the experiment interval, that is, the *under-provisioning time share* τ_U is the time relative to the measurement duration, in which the system has insufficient resources, whereas, the *over-provisioning time share* τ_O is the time relative to the measurement duration, in which the system has more resources than required. Values of this metric lie in the interval $[0, 100]$. The best value 0 is achieved when no under-provisioning or no over-provisioning is detected within a measurement period.

$$\tau_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(d_t - s_t), 0) \Delta t$$

$$\tau_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(s_t - d_t), 0) \Delta t$$

3) Auto-scaler Worst-Case Deviation ς: As described before in Section II-C2, the benchmarking of auto-scaling for applications with multiple services is challenging. To face the issue of how to compare system elasticity in a setting with multiple different services, we introduce the *auto-scaler worst-case deviation ς* metric. The basic idea is to compare the auto-scalers with respect to their worst behavior across all services. This makes sense since the services depend on each other and the system performance is limited by the worst service performance. In other words, the worst elasticity metrics across all services are considered.

In a first step, the elasticity metrics for each service $s_1, \ldots, s_n$ are calculated. Then, for each metric the worst performance across all services is selected:

$$\hat{\tau}_U := max(\tau_{U,s_i}), \quad \hat{\tau}_O := max(\tau_{O,s_i})$$
$$\hat{\theta}_U := max(\theta_{U,s_i}), \quad \hat{\theta}_O := max(\theta_{O,s_i})$$

In order to calculate the deviation, we calculate the overall *worst-case provisioning accuracy* $\hat{\theta}$ and the overall *worst-case wrong provisioning time share* $\hat{\tau}$. We then calculate the average for both metrics consisting of both components:

$$\hat{\theta}[\%] := \frac{\hat{\theta}_U + \hat{\theta}_O}{2}, \qquad \hat{\tau}[\%] := \frac{\hat{\tau}_U + \hat{\tau}_O}{2}$$

In the last step, we compute the Euclidean distance to determine the worst-case deviation of the considered auto-scaler from the theoretically optimal auto-scaler. As the theoretically optimal auto-scaler is assumed to know when and how much the demanded resources change, the values for worst-case provisioning accuracy $\hat{\theta}$ and worst-case wrong provisioning time share $\hat{\tau}$ are equal to zero. In other words, if an auto-scaler is compared to the theoretically optimal auto-scaler, the Euclidean norm can be used.

$$\varsigma[\%] := \|(\hat{\theta}, \hat{\tau}) - (0, 0)\|_2 = \|(\hat{\theta}, \hat{\tau})\|_2 = \sqrt{\hat{\theta}^2 + \hat{\tau}^2}$$

4) Application Performance Index [29]: The *Apdex* (application performance index) is an open standard measure developed by a consortium of companies that measures user satisfaction on a uniform scale of 0% to 100%. The best value of 100% is achieved when all requests are served within the agreed response time (SLO). In addition to the SLO violations that reflect whether a request is served within the required time frame, this metric can be used to provide additional insight on how bad the violations are from a user's perspective. For the calculation, we define: (i) ν as the number of satisfied requests, i.e., requests within the SLO. (ii) ϵ as the number of tolerating

requests, i.e., requests that exceed the SLO within a toleration interval. (iii) Ω as the number of total requests.

$$Apdex[\%] := \frac{\nu + 0.5 \cdot \epsilon}{\Omega}$$

V. EXPERIMENT RESULTS

In this section, we benchmark Chamulteon against the other auto-scalers. Section V-A explains how to interpret the results of the measurements. In Section V-B, we compare the auto-scalers in the virtual machines and Docker container setups. Afterwards, the scalability of the auto-scalers is investigated in Section V-C. Section V-D concludes the evaluation with a list of key findings. Finally, we discuss threats to validity.

A. Introduction to the Results

Before discussing the results, we introduce the experiment format: Figure 2 shows the Wikipedia trace scaled by Reg. The first three plots in the figure show the scaling behavior for each service; the bottom plot shows the request evaluation from the SLO perspective. For each plot, the x-axis shows the time of the measurement in seconds. In the scaling plots, the amount of demanded resources is depicted as a blue dashed curve and the amount of supplied resources as a red curve. In the request evaluation plots, the requests sent per second are depicted as a blue dashed line and the requests that are served within the SLO response time as a green line.

We showed this figure as it is a good example of the bottleneck shifting mentioned above. While the first service is scaled after 60 seconds to satisfy the demand, the second service is scaled 60 seconds later, and the last service 120 seconds later. This behavior can be explained as follows: in the first 60 seconds, the first service can handle a lower number of responses and thus, the underlying services also receive fewer requests. After the first service is scaled, the second service cannot handle all received requests, and therefore, the last service again receives a lower number of requests. This effect can be seen until 900 seconds as the resource supply for each service is increasing slower than the previous service supplies. Similar to the observations in the articles [2], [27], [30], Reg exhibits a high rate of oscillations (between second 1000 and 1600) that cannot be explained. After second 2000, Reg tends to over-provision without any oscillations. In contrast, Figure 3 shows the scaling behavior of Chamulteon. In this and the following experiments, the cost-component is deactivated. The scaling behavior of Chamulteon exhibits neither bottleneck shifting nor oscillations. Due to the configuration of Chamulteon, the system is always allocated slightly more than the required amount of resources, and thus almost all requests can be served within the SLO. Note that during the whole experiments, the cost-awareness component is deactivated. On the one hand, we want to investigate and compare the scaling performance of Chamulteon. On the other hand, the component was evaluated in our previous work [21].

The scaling performance of Reg and the other auto-scalers is shown in Table II. Each column shows an auto-scaler and each row represents a metric: the average provisioning

accuracy ($\overline{\theta}_U$ and $\overline{\theta}_O$) and the average wrong provisioning time share ($\overline{\tau}_U$ and $\overline{\tau}_O$) for each service, the auto-scaler worst-case deviation ς, the SLO violations, and the user satisfaction (Apdex). Due to the bottleneck shifting and the oscillations, Reg exhibits the worst $\overline{\theta}_U$ (15.3%) and also the worst $\overline{\tau}_U$ (52.2%). These metrics are also reflected by the highest SLO violations (37.3%) and the lowest user satisfaction (31.1%). In

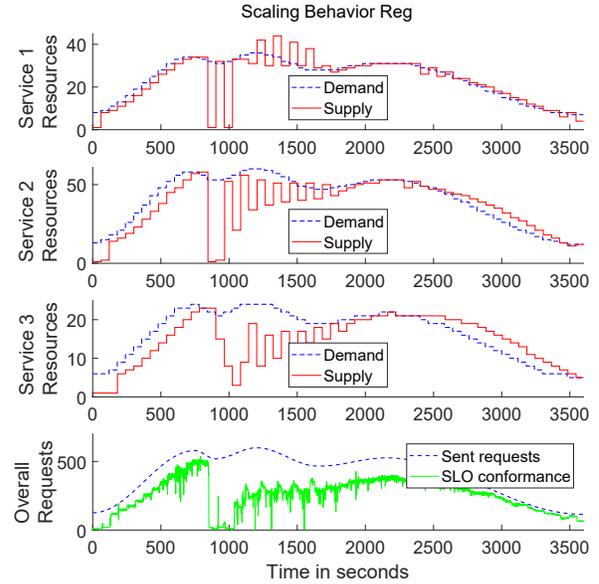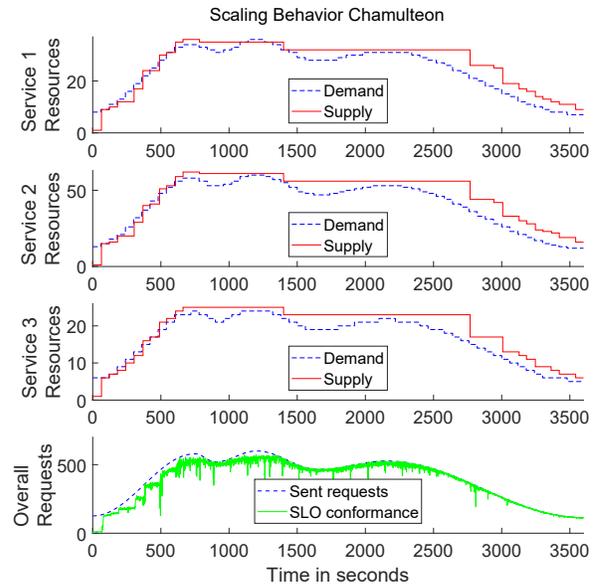Fig. 2. Scaling behavior of Reg on the Wikipedia trace.

Fig. 3. Scaling behavior of Chamulteon on the Wikipedia trace.

B. Docker vs. VM Scaling

In this section, we evaluate how well Chamulteon scales different setups in comparison to the other auto-scalers. In the first scenario, the application was deployed on Docker (with up to 120 containers), and due to the fast container provisioning

TABLE II
EVALUATION ON WIKIPEDIA TRACE (DOCKER).

Metric	Chamulteon	Adapt	Hist	Reg	React
$\overline{\theta}_U$	**3.7%**	12.6%	7.0%	15.3%	5.3%
$\overline{\theta}_O$	29.3%	10.2%	32.1%	**8.8%**	13.1%
$\overline{\tau}_U$	**14.9%**	34.7%	25.6%	52.2%	23.6%
$\overline{\tau}_O$	84.4%	54.9%	69.4%	**41.2%**	69.7%
ς	52.9%	50.6%	58.1%	52.9%	**50.3%**
SLO	**6.2%**	24.2%	12.5%	37.3%	11.2%
Adpex	**77.7%**	51.6%	67.8%	31.1%	72.8%

TABLE III
EVALUATION ON WIKIPEDIA TRACE (VM).

Metric	Chamulteon	Adapt	Hist	Reg	React
$\overline{\theta}_U$	0.9%	9.7%	4.5%	7.3%	**0.2%**
$\overline{\theta}_O$	15.6%	**6.0%**	23.9%	10.2%	47.5%
$\overline{\tau}_U$	3.0%	31.0%	15.7%	24.0%	**0.8%**
$\overline{\tau}_O$	60.6%	**15.7%**	38.7%	24.0%	94.1%
ς	37.0%	34.9%	37.1%	**34.8%**	57.8%
SLO	2.0%	19.1%	5.1%	12.6%	**1.0%**
Adpex	83.2%	30.7%	69.8%	50.3%	**92.0%**

TABLE IV
EVALUATION ON BIBSONOMY WITH SMALL SETUP.

Metric	Chamulteon	Adapt	Hist	Reg	React
$\overline{\theta}_U$	**2.0%**	9.7%	5.43%	11.0%	3.5%
$\overline{\theta}_O$	19.1%	9.3%	18.9%	**4.9%**	14.9%
$\overline{\tau}_U$	**7.4%**	40.6%	23.8%	42.7%	14.5%
$\overline{\tau}_O$	78.8%	40.7%	61.2%	**32.3%**	68.5%
ς	**47.4%**	50.1%	48.7%	48.7%	56.1%
SLO	**7.3%**	17.8%	11.9%	23.4%	10.5%
Adpex	**90.5%**	79.8%	84.6%	71.2%	87.5%

times, the application was scaled every minute. Further, the application was stressed for one hour. In the second scenario, the application was deployed on virtual machines (with up to 20 VMs); it was scaled every 2 minutes and stressed for 6 hours. In other words, the scenarios differ in the amount of scaling instances, the scaling interval, and the experiment duration. The results are shown in Table II for Docker and in Table III for the VM scenario.

In the Docker scenario, Chamulteon exhibits the best $\overline{\theta}_U$ and $\overline{\tau}_U$ followed by React. However, Chamulteon has the second worst $\overline{\theta}_O$ and worst $\overline{\tau}_O$. This is because Chamulteon is configured to slightly over-provision to achieve good user metric values. As discussed in Section IV-D focusing on individual elasticity metrics may lead to ambiguous results, we compare the auto-scalers based on the aggregated metric ς. React achieves the best value for ς and is slightly better than Chamulteon. However, Chamulteon has the lowest SLO violations and the highest user satisfaction. In contrast to the first scenario, in the second scenario (VM), React and Chamulteon swapped in the ranking for the basic elasticity metrics. Accordingly, React exhibits the best $\overline{\theta}_U$, $\overline{\tau}_U$, SLO violations, and user satisfaction metrics followed closely by Chamulteon. While the best ς is achieved by Reg, the worst performance is achieved by React due to its high $\overline{\tau}_O$. That is, almost during the entire measurement period, React keeps the system in an over-provisioned state.

In summary, React and Chamulteon have achieved the best user-oriented metrics with distance to the other auto-scalers. However, taking ς and the user-oriented metrics into account, Chamulteon shows only a slight deviation from the winner, React has by far the worst value for ς in the second scenario. That is, Chamulteon delivers more robust performance in both scenarios compared with React.

C. Scalability

In this experiment, we stressed the application, which was deployed in Docker, with the BibSonomy trace while scaling the demand. In the first scenario, the application required at maximum 120 containers and in the second 60 containers. The results are shown in Table IV for the small setup and in Table V for the large setup. Independent of the scale, Chamulteon achieves for both experiments the best $\overline{\theta}_U$, $\overline{\tau}_U$, SLO violations, and user satisfaction metrics. In the small setup, React has the worst ς, but the second best SLO violation

and user satisfaction. Concerning the user metrics, Hist is slightly worse than React. In the large setup, Hist has the second best SLO violations and user satisfaction followed by React. In both setups, Reg has the worst user metrics.

Regarding scalability, Chamulteon (8.97%) has the lowest relative deviation between both setups, followed by Hist (13.57%). The highest difference has React with 43.88%.

D. Summary of Evaluation Findings

We conducted four different sets of experiments to investigate the auto-scaling performance. We varied the deployment (Docker vs. VM), the scale (20, 60 and 120 instances), and the used workload trace (Wikipedia vs. BibSonomy). We summarize the main findings of our evaluation as follows:

1) Chamulteon exhibits in three out of four experiments the best user-oriented metrics and in two the best worst-case auto-scaling deviation. In the remaining experiments, Chamulteon achieves the second best values. Concerning scalability, Chamulteon exhibits a relative deviation of only 8.97%. In contrast to the other auto-scalers, the

TABLE V
EVALUATION ON BIBSONOMY WITH LARGE SETUP.

Metric	Chamulteon	Adapt	Hist	Reg	React
$\overline{\theta}_U$	**2.4%**	17.5%	5.9%	15.4%	5.6%
$\overline{\theta}_O$	19.5%	7.7%	24.6%	**4.6%**	9.4%
$\overline{\tau}_U$	**6.9%**	50.8%	28.3%	55.4%	32.6%
$\overline{\tau}_O$	89.7%	38.9%	65.7%	**36.0%**	55.1%
ς	**51.4%**	55.8%	56.1%	59.1%	53.3%
SLO	**9.6%**	33.2%	12.9%	36.3%	15.3%
Adpex	**77.1%**	42.8%	75.4%	35.2%	74.1%

bottleneck shifting effect could not be observed when using Chamulteon.

2) Although React is a reactive auto-scaler, it achieves in two experiments the second best user-oriented metrics. However, React tends to strongly over-provision the system and differs in scalability by 43.88% between the small and large scenario.

3) Hist also exhibits in two experiments the second best user-oriented metrics due to its tendency to over-provision. With respect to scalability, Hist exhibits the second lowest deviation.

4) Reg and Adapt tend to under-provision and thus, exhibit the worst user-oriented metrics.

E. Threats to Validity

Although our experimental analysis covers different scenarios and we compare different auto-scalers, the results may not be generalizable to other types of applications or to closed-source auto-scalers. In principle, for the evaluated competing auto-scalers, a comparable behavior has been observed in the related work on auto-scaler evaluation [2], [27], [30].

In our evaluation, we employed for each service the same auto-scaler. Indeed, it is possible to use different mechanisms for each service. However, the selection of the scalers and the order in which they are used is a crucial part. Thus, even though in theory by experimenting with different combinations of auto-scalers in different orders, one may achieve better results, in practice, finding an optimal configuration is extremely challenging, and one has no guarantee that this configuration would remain static as the system and workload evolve.

As Chamulteon is intended to prioritize satisfying the user, the experiments show a slight over-provisioning. This behavior seems to be contrary to the cost-efficiency approach, but in our evaluation, the cost-awareness component was deactivated. Moreover, in our context, cost-efficiency is not equivalent to using as few instances as possible. The cost-awareness component tries to use the accounted instances as efficiently as possible. For example, in the case of an hourly charging, resources would not be released if they are paid and might be needed again in a few minutes to avoid paying double costs for the same resources.

To reduce the risk of a bias in the evaluation, we used an established set of metrics: (i) user-oriented metrics such as the SLO violation and Adpex and (ii) system oriented metrics that are officially endorsed by SPEC [11]. Also, our own worst-case auto-scaling deviation judges over- and under-provisioning equally, not introducing customizable weights that may be configured in a way to favor a given auto-scaler.

VI. CONCLUSION

In this article, we presented Chamulteon, a novel auto-scaler for coordinated auto-scaling of applications with multiple services. Chamulteon is a fundamental redesign of our previously proposed hybrid auto-scaler Chameleon.

Chamulteon combines forecasting and service demand estimation, which is enriched with application knowledge captured in a descriptive software performance model. The service demand estimation is implemented by integrating established open-source tools developed as part of our previous work. The forecasting of arriving requests is realized by a hybrid decomposition based method specifically designed for auto-scaling scenarios. In the evaluation, we employ a set of SPEC-endorsed elasticity metrics, in addition to metrics capturing the user's perspective. We compare Chamulteon against existing popular auto-scalers.

We employ a micro-service-based benchmark application deployed either on VMs or in Docker containers in our private cloud. To emulate realistic workloads, we use two real-world load profiles. In the experiments, we investigate the scalability of the auto-scalers and their scaling behavior on different deployments. For the other auto-scalers, we observe typical scaling behavior characteristics. Chamulteon achieves in three out of four experiments the best user-oriented metrics and in two experiments the best elasticity metrics. In the remaining experiments, Chamulteon exhibits the second best metric results. Also, Chamulteon exhibits the lowest variation between the different setup sizes.

We see potential to extend our proposed Chamulteon auto-scaler, first, by adding support for vertical and nested auto-scaling. The vertical scaling could be combined with horizontal scaling, where a decision logic can evaluate which scaling direction is more efficient. Therefore, a separate cost function needs to be added. Deciding between the two options would require a significant extension to the decision logic that may, for example, be solved with machine learning. Auto-scaling on nested resource layers, for instance, the possibility of adding a new VM or adding a new container in an existing VM, poses a new challenge on its own. Second, the request rate at preceding services is calculated regarding their capacity and the actual request rate arriving at the considered service. Currently, the return path is not considered. If the first service is scaled so that the request rate can be processed, but the maximum capacity is exceeded at a preceding service, this information is not passed to the first service. If this information would be provided to the first service, the auto-scaler could scale down to the maximum capacity of the bottleneck resource and save instance time. Third, besides the existing cost-awareness component, it is possible to add another component responsible for optimizing energy consumption. This component should monitor the energy consumption and execute voltage scaling, e.g., by tuning the CPU frequency or stopping virtual machines in exchange for a higher CPU frequency. The overall target of this component should be to minimize the energy consumption of the whole application.

---

[8]Research Group http://research.spec.org

REFERENCES

[1] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," *INC, IMS and IDC*, pp. 44–51, 2009.

[2] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev, "Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[3] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *2009. ICEBE'09. IEEE International Conference on E-Business Engineering*, 2009, pp. 281–286.

[4] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012, pp. 204–212.

[5] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 1, 2008.

[6] W. Iqbal, M. N. Dailey, D. Carrera, and D. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Elsevier Future Generation Computer Systems (FGCS)*, vol. 27, no. 6, pp. 871–879, 2011.

[7] M. Beltrán, "Automatic provisioning of multi-tier applications in cloud computing environments," *The Journal of Supercomputing*, vol. 71, no. 6, pp. 2221–2250, 2015.

[8] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service." in *10th USENIX International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 69–82.

[9] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *2nd ACM Symposium on Cloud Computing 2011*, 2011.

[10] B. Urgaonkar and A. Chandra, "Dynamic provisioning of multi-tier internet applications," in *2nd IEEE International Conference on Autonomic Computing (ICAC) 2005*, 2005, pp. 217–228.

[11] N. Herbst, R. Krebs, G. Oikonomou, G. Kousiouris, A. Evangelinou, A. Iosup, , and S. Kounev, "Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics," *CoRR*, vol. abs/1604.03470, 2016.

[12] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 73:1–73:33, 2018. [Online]. Available: http://doi.acm.org/10.1145/3148149

[13] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[14] R. Khorsand, M. Ghobaei-Arani, and M. Ramezanpour, "Fahp approach for autonomic resource provisioning of multitier applications in cloud computing environments," *Wiley Software: Practice and Experience*, vol. 48, no. 12, pp. 2147–2173, 2018.

[15] S. Wu, B. Li, X. Wang, and H. Jin, "Hybridscaler: Handling bursting workload for multi-tier web applications in cloud," in *15th International Symposium on Parallel and Distributed Computing (ISPDC), 2016*, July 2016, pp. 141–148.

[16] Q. Zhu and G. Agrawal, "Resource provisioning with budget constraints for adaptive applications in cloud environments," in *19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 304–307.

[17] U. Sharma, P. Shenoy, and D. F. Towsley, "Provisioning multi-tier cloud applications using statistical bounds on sojourn time," in *9th ACM International Conference on Autonomic Computing (ICAC) 2010*, 2012, pp. 43–52.

[18] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center," in *IEEE 3rd International Conference on Cloud Computing (CLOUD) 2010*, 2010, pp. 370–377.

[19] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bähr, "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 5, 2017.

[20] S. Kounev, N. Huber, F. Brosig, and X. Zhu, "A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures," *IEEE Computer*, vol. 49, no. 7, pp. 53–61, 2016.

[21] V. Lesch, A. Bauer, N. Herbst, , and S. Kounev, "FOX: Cost-Awareness for Autonomic Resource Management in Public Clouds," in *9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*, 2018.

[22] M. Züfle, A. Bauer, N. Herbst, V. Curtef, and S. Kounev, "Telescope: A Hybrid Forecast Method for Univariate Time Series," in *International work-conference on Time Series (ITISE 2017)*, 2017.

[23] R. J. Hyndman and A. B. Koehler, "Another Look at Measures of Forecast Accuracy," *International Journal of Forecasting*, pp. 679–688, 2006.

[24] S. Spinner, G. Casale, X. Zhu, and S. Kounev, "LibReDE: A library for Resource Demand Estimation," in *ACM/SPEC ICPE 2014*, 2014, pp. 227–228.

[25] S. Eismann, J. Walter, J. von Kistowski, and S. Kounev, "Modeling of Parametric Dependencies for Performance Prediction of Component-based Software Systems at Run-time," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 135–144.

[26] D. Benz, A. Hotho, R. Jäschke, B. Krause, F. Mitzlaff, C. Schmitz, and G. Stumme, "The social bookmark and publication management system bibsonomy," *The International Journal on Very Large Data Bases*, vol. 19, no. 6, pp. 849–875, 2010.

[27] A. V. Papadopoulos, A. Ali-Eldin, K.-E. Årzén, J. Tordsson, and E. Elmroth, "PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS)*, vol. 1, no. 4, pp. 1–31, 2016.

[28] N. Herbst, A. Bauer, S. Kounev, G. Oikonomou, E. van Eyk, G. Kousiouris, A. Evangelinou, R. Krebs, T. Brecht, C. L. Abad, and A. Iosup, "Quantifying Cloud Performance and Dependability: Taxonomy, Metric Design, and Emerging Challenges," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS)*, vol. 3, no. 4, pp. 19:1–19:36, 2018.

[29] P. Sevcik, "Defining the application performance index," *Business Communications Review*, vol. 20, 2005.

[30] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. Bauer, A. V. Papadopoulos, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscalers for complex workflows," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS)*, vol. 3, no. 2, pp. 8:1–8:32, 2018.