This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2018.2870389

Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field

André Bauer, Nikolas Herbst, Simon Spinner, Ahmed Ali-Eldin, and Samuel Kounev

Abstract—Auto-scalers for clouds promise stable service quality at low costs when facing changing workload intensity. The major public cloud providers provide trigger-based auto-scalers based on thresholds. However, trigger-based auto-scaling has reaction times in the order of minutes. Novel auto-scalers from literature try to overcome the limitations of reactive mechanisms by employing proactive prediction methods. However, the adoption of proactive auto-scalers in production is still very low due to the high risk of relying on a single proactive method.
This paper tackles the challenge of reducing this risk by proposing a new hybrid auto-scaling mechanism, called Chameleon, combining multiple different proactive methods coupled with a reactive fallback mechanism. Chameleon employs on-demand, automated time series-based forecasting methods to predict the arriving load intensity in combination with run-time service demand estimation to calculate the required resource consumption per work unit without the need for application instrumentation.
We benchmark Chameleon against five different state-of-the-art proactive and reactive auto-scalers one in three different private and public cloud environments. We generate five different representative workloads each taken from different real-world system traces. Overall, Chameleon achieves the best scaling behavior based on user and elasticity performance metrics, analyzing the results from 400 hours aggregated experiment time.

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

1 Introduction

Over the past decade, the cloud computing paradigm gained significant importance in the ICT domain as it addresses manageability and efficiency of modern Internet and computing services at scale. Cloud computing provides on-demand access to data center resources (e.g., networks, servers, storage and applications). Infrastructure-as-a-Service (IaaS) cloud providers promise stable service quality by leveraging trigger-based auto-scaling mechanism to deal with variable workloads. However, depending on the type of resources and deployed software stack, scaling actions may take several minutes to be effective. In practice, business-critical applications in clouds are usually still deployed with over-provisioned resources to avoid becoming dependent on an auto-scaling mechanism with its possibly wrong or badly-timed scaling decisions.

Sophisticated, state-of-the-art auto-scaling mechanisms from the research community focus on proactive scaling, aiming to predict and provision required resources in advance of when they are needed. An extensive survey [1] groups auto-scalers into five classes according to the prediction mechanisms they use: (i) threshold-based rules, (ii) queueing theory, (iii) control theory, (iv) reinforcement learning, and (v) time series analysis. With a few exceptions, like at Netflix, proactive auto-scalers are not yet broadly used in production. This might be a result of the need for application-specific fine-tuning, and the lack of knowledge about the performance of an auto-scaler in different contexts. Auto-scalers employed in production systems are responsible to dynamically trade-off user-experienced performance and costs in an autonomic manner. Thus, they carry a high operational risk. We pose ourselves the following research questions: (RQ1) *How can the risk of using auto-scaling features in operation be minimized by leveraging multiple different proactive mechanisms applied in combination with conventional reactive mechanisms?* (RQ2) *How can a level-playing field for state-of-the-art auto-scalers be established to increase the trust towards a broader adoption of auto-scalers in production?* (RQ3) *How well does the proposed Chameleon approach perform compared to state-of-the-art auto-scaling mechanisms in realistic deployment and application scenarios?*

In this paper, we propose a new hybrid auto-scaling mechanism called Chameleon combining multiple different proactive methods coupled with a reactive fallback. Chameleon reconfigures the deployment of an application in a way that the supply of resources matches the current and estimated future demand for resources as closely as possible according to the definition of elasticity [2]. It consists of two integrated controllers: (i) a reactive rule-based controller taking as input the current request arrival rate and service demands estimated in an online fashion, and (ii) a proactive controller that integrates three major building blocks: (a) an automated, dynamic on-demand forecast execution based on an ensemble of the seasonal ARIMA [3] and tBATS [4] stochastic time series modeling frameworks, (b) an optional descriptive software performance model as an instance of the Descartes Modeling Language (DML) [5] enabling the controller to leverage structural application

- *André Bauer, Nikolas Herbst, Simon Spinner, and Samuel Kounev, University of Würzburg, Germany. E-mail: [first].[lastname]@uni-wuerzburg.de*
- *Ahmed Ali-Eldin, UMass, Amherst, USA and Umeå university, Sweden.*

This work was funded by the German Research Foundation (DFG) under grant No. KO 3445/11-1. Ali-Eldin was funded by the Swedish Research Council (VR) under the project Cloud Control. This research has been supported by the Research Group of the Standard Performance Evaluation Corporation (SPEC). Manuscript received Oct, 2017.

knowledge by transformation into product-form queueing networks, and (c) the LibReDE resource demand estimation library [6] for accurate service-time estimations at run-time without the requirement of application instrumentation. Chameleon is available as open-source project[1] together with the experiment data presented in this paper.

In the evaluation, we benchmark Chameleon against four different proactive auto-scaling mechanisms covering the mentioned domains, as well as one commonly used reactive auto-scaler based on average CPU utilization thresholds and finally, a scenario without the use of an active auto-scaler. We conduct seven rows of extensive and realistic experiments of up to 9.6 hours duration in three different infrastructure cloud environments: (i) in a private CloudStack-based cloud environment, (ii) in the public AWS EC2 IaaS cloud, as well as (iii) in the OpenNebula-based IaaS cloud of the Distributed ASCI Supercomputer 4 (DAS-4) [7]. We generate five different representative workload profiles each taken from different real-world system traces: BibSonomy, Wikipedia, Retailrocket, IBM mainframe transactions and FIFA World Cup 1998. The workload profiles drive a CPU-intensive web application as benchmark scenario that is comparable to the LU worklet from SPEC's Server Efficiency Rating Tool SERT™2[2]. As elasticity measurement methodology and experiment controller, we employ the elasticity benchmarking framework BUNGEE [2] enabling extensive and repeatable elasticity measurements. The results are analyzed with a set of SPEC endorsed elasticity metrics [8] in addition to metrics capturing the user-perspective. For each experiment row, the metric results are used to conduct three competitions: (i) a rating based on the deviation from the theoretically optimal auto-scaling behavior, (ii) a pairwise competition, and (iii) a score-based ranking of metric speedups aggregated by an unweighted geometric mean. The results of the individual metrics and the three competitions show that Chameleon manages best to match the demand for resources over time in comparison to the other proactive auto-scalers.

The contributions of this work are manifold and we summarize the highlights as follows: In Section 2, we address RQ1 and present the Chameleon approach that integrates reactive and proactive scaling decisions based on time series forecasts, combined with online service demand estimates used as input to product-form queueing networks. Afterwards, in Section 3, we address RQ2 and present the design of a broad auto-scaler evaluation for compute intensive workloads. We propose the auto-scaler deviation metric as a new way to aggregate and rank auto-scalers. Section 4 addresses RQ3 by discussing the results of extensive experiments that demonstrate the superior auto-scaling capabilities of Chameleon compared to a set of state-of-the-art auto-scaling mechanisms. We include a discussion of threads to validity in Section 4.4, summarize related work in Section 5, before we conclude and outline ongoing and future work in Section 6.

1. Chameleon: http://descartes.tools/chameleon
2. SPEC SERT™2: https://spec.org/sert2

2 Chameleon Auto-Scaler

This section describes the operating principle of Chameleon: the generation and optimized combination of reactive and proactive scaling decisions. A design overview is depicted and explained in the following subsection. In Section 2.2, we discuss the proactive decision logic based on an algorithm in pseudo code. Then, in Section 2.3, we present the logic behind conflicting scaling event resolution. Finally, our assumptions are listed.

2.1 Design Overview

The Chameleon mechanism consists of four main components: (i) a controller, (ii) a performance data repository, (iii) a forecast component and (iv) the service demand estimation component based on LibReDE [9]. The performance data repository contains a time series storage and an optional descriptive performance model instance of the application to be scaled dynamically in form of the Descartes Modeling Language (DML) [5]. The design and the flow of information is depicted in Figure 1. The central part of Chameleon is the controller. It communicates with the three remaining components and the managed infrastructure cloud. The functionality of the controller is divided into two parallel sequences: the reactive cycle (red) and the proactive cycle (dashed blue).

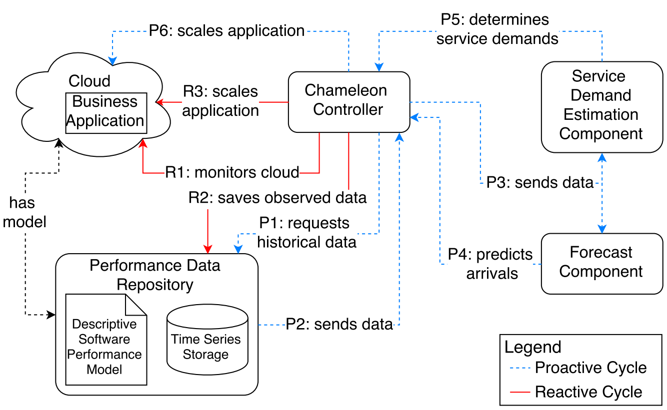


Fig. 1: Design overview of Chameleon.

During the reactive cycle, the controller has three main tasks: (R1) at first, the controller communicates with the cloud management and periodically polls (e.g., every minute) information on the current state of the application delivered via asynchronous message queues. Within the application run-time environment, a monitoring agent [10] is deployed to fill the message queues. The collected information includes CPU utilization averages per node and the number of request arrivals. Average residence and response times per request type on a node can be provided (but are not required to). (R2) Then, the new information is stored in the performance data repository for the current time window. (R3) With this information, the controller decides, if the system needs to be scaled, based on the computed average system utilization and a standard threshold-based approach. The average system utilization is derived from the arrival rate and the estimated service demand based on the service demand law from queueing theory.

The proactive cycle is planned in longer intervals, e.g., 4 minutes, for a set of future scaling intervals. It involves six tasks: (P1) at first, the controller queries the performance data repository for available historical data and checks for updates in the structure of the DML performance model. (P2) Then, the available time series data is sent to the controller. (P3) Afterwards, the time series of request arrival rates is forwarded to the forecast component and data about the CPU utilization and request arrivals per node (plus residence and response times if available) is sent to the service demand estimation component. (P4) Then, the new available forecast values are sent to the controller. (P5) The LibReDE service demand estimation component estimates the time a single request needs to be served on the CPU of a node and sends the estimated value to the controller. (P6) Finally, the controller scales the application deployment based on the estimated service demands, the forecast request arrivals and structural knowledge from the DML descriptive performance model.

Having the flow of the two cycles described, the following two paragraphs now focus on the forecast and service demand estimation components.

FORECAST COMPONENT: The forecast component predicts the arrival rates for a configurable number of future reconfiguration intervals. In order to reduce overhead, the forecast component is not called in fixed periods. If an earlier forecast result still contains predicted values for requested future arrival rates, no new time series forecast is computed. In case, a drift between the forecast and the recent monitoring data is detected, a new forecast execution is triggered. To detect a drift between monitoring and forecast values, we compare the forecast accuracy with the mean absolute scaled error metric (MASE) [11] considering a configurable threshold value, e.g., 0.4. The MASE metric is suitable for almost all situations and the error is based on the in-sample mean absolute error from the random walk forecast. For a 20% forecast, the random walk forecast would predict the last value of the history for the entire horizon. Thus, the investigated forecast is better than the random walk forecast if the MASE value is < 1 and worse if the MASE value is > 1.

For the dynamic on-demand forecast executions, we select an ensemble of the following two stochastic time series modeling frameworks as implemented in R forecast package [12]: (i) sARIMA seasonal, auto-regressive, integrated moving averages [3] and (ii) tBATS trigonometric, Box-Cox transformed, ARMA errors using trend and seasonal components [4]. Due to the capability of both methods to capture seasonal patterns as soon as the data contains two full periods (in our auto-scaling context days), they are considered as complex. We consider the more lightweight approaches that can only estimate trend extrapolations (like splines) as insufficient for auto-scaling decisions in the order of minutes and even hours into the near future. We observe that the time overhead for the forecast executions can vary significantly dependent on the data characteristics and that longer running forecast execution tend to have a lower forecast accuracy. For applicability in an auto-scaling context, timely and accurate forecast results with reasonable overhead are required. Thus, we design the forecast component in a way that the two methods are not run in parallel, but the method that is more likely to have the more accurate result is automatically selected before execution. This selection is performed based on a re-implementation of the meta-learning approach for forecast method selection using data characteristics as in [13].

SERVICE DEMAND ESTIMATION COMPONENT: The LibReDE library offers eight different estimation approaches for service demands on a per request type basis [9]. Among those eight, there are estimators based on regression, optimization, Kalman filters, and the service demand law. LibReDE supports to dynamically select an optimal approach via parallel execution and cross-validation of the estimated error. Furthermore, configuration parameters of the estimation approaches can be tuned automatically for a given concrete scenario [14]. To minimize estimation overheads, the service demand law based estimator is used. As input, the request arrivals per resource and the average monitored utilization are required. Request response and residence times can be provided optionally as they are required by some of the estimators and for an enhanced cross-validation based on utilization and response time errors. For complex service deployments, LibReDE requires structural knowledge about the application deployment to have a defined mapping of what services are deployed on which resources. Chameleon can provide this information from a DML performance model instance.

2.2 Decision Management

In the proactive cycle, the controller determines events (i.e., scaling actions) for each forecast. Decisions are created based on the rules in the Decision Logic (see the simplified Algorithm 1). Then, these decisions are improved/optimized and finally, they are added to the *Event Manager*, see Section 2.3. There, the decisions are scheduled according to their target execution time.

The *Decision Logic* determines for each event the number of instances that need to be removed or added. Algorithm 1 shows the step-by-step approach. The associated parameters that a user needs to specify are $up\_resp\_threshold$ (the upper response time threshold related to the SLO), $pro\_up\_util\_threshold$ (the system utilization threshold for upscaling), $down\_resp\_threshold$ (the lower response time threshold related to the SLO), $target\_utilization$ (the target system utilization for downscaling, and slo (the acceptable response time as SLO).

In the second line, the algorithm loads the user services. These services form the application's interface to the users, e.g., a login request. Then, the future system utilization is calculated by multiplying the predicted arrival rate with the average service demand of all services (Line 3). Afterwards, the utilization of a single VM is calculated, the response time of all user services are calculated, and the maximum response time is returned. This is done by adding the residence times of each called service. Hereby, we model each service as an M/M/1/∞ queue. The residence time of each internal service is computed with the formula: residence time $R = \frac{S}{100-\rho}$ where S is the service demand and ρ the system utilization. The calculated response time can lie in one of the following intervals: (i) $[0, down\_resp\_threshold \cdot slo]$,

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2018.2870389

ALGORITHM 1: Proactive decision logic.

```
1  Decision Logic at time t in the future
2      services = model.getServices(); // gets all services
3      ρS = getArrRateForecast() · AvgServDemand(services);
           // calculates the future system utilization
4      ρ = ρS / getRunningVMs(); // calculates the future average
           utilization on each VM
5      response = calcEnd2EndRespTime(ρ, services);
           // calculates the maximum response time of all services
6      amount = 0; // the number of VMs for adding or releasing
7      if response ≥ up_resp_threshold · slo then
8          while response ≥ up_resp_threshold · slo or
             ρ ≥ pro_up_util_threshold do
9              amount++;
10             ρ = ρS / (getRunningVMs()+amount); // calculates
                   the new average utilization
11             response = calcEnd2EndRespTime(ρ, services);
12     else if response ≤ down_resp_threshold · slo then
13         while response ≤ down_resp_threshold · slo and
             ρ ≤ target_utilization do
14             amount−−;
15             ρ = ρS / (getRunningVMs()+amount); // calculates
                   the new average utilization
16             response = calcEnd2EndRespTime(ρ, services);
17             if response > up_resp_threshold · slo or ρ >
                 pro_up_util_threshold then
18                 amount++; // undo as one condition is violated
19                 break;
20     return decision(amount, t);
```

(ii) $(down\_resp\_threshold \cdot slo, up\_resp\_threshold \cdot slo)$ or (iii) $[up\_resp\_threshold \cdot slo, \infty)$.

If the response time lies in the second interval, the calculated response time has an acceptable value. The algorithm skips the if-else block and returns a NOP decision (as the amount is zero) (Line 20). When the response time lies within the last interval, i.e., the response time is greater than $up\_resp\_threshold \cdot slo$, the number of VMs are iteratively increased until the new response time drops below $up\_resp\_threshold \cdot slo$ and the new average VM utilization ρ is less than $pro\_up\_util\_threshold$ (Line 8-11). The new average VM utilization ρ is calculated by dividing the system utilization ρ_S by the new amount of VMs and the response time is computed based on this utilization. Finally, the algorithm returns a new decision with the number of the additional VMs required for this service (Line 20). If the response time is less than $down\_resp\_threshold \cdot slo$, the number of VMs are iteratively decreased until the new response time is greater than $down\_resp\_threshold \cdot slo$ or the average VM utilization ρ is greater than $target\_utilization$ (Line 13-16). The recalculation of the response time is analogous to Line 8-11. Finally, a new decision is returned with the amount of VMs that can be released (Line 20).

During an interval of the proactive cycle (we use 4 minutes), two proactive decisions are made based on the logic of Algorithm 1. The time between the two decisions and the respective scheduled events are equidistant (we use 2 minutes), see Figure 2. As proactive decisions are calculated based on the current observations and predictions, some rules are needed to adjust/improve these decisions before adding them as events to the *Event Manager* (see Section 2.3). Note that we improve the pair of events in each interval of the proactive cycle (and not more) as a trade-off between decision stability and reactivity of the approach. Basically, there are three possibilities when combining two decisions. (i) Both decisions want to scale up the system, (ii) want to scale down the system, (iii) or they have contrary scaling decisions. If one of the decisions is a NOP, no combination is required. Hence, six different cases can be distinguished.

The first possibility (both decisions plan to scale up) has two cases. Firstly, the first decision wants to scale up n VMs and the second one wants to scale up m instances, where $n \geq m$. The resulting first event allocates n extra VMs. The second event triggers the allocation of 0 new VMs ($\neq$ NOP). In the second case $m > n$ and so, the first event scales up n VMs and second one allocates $m - n$ VMs.

The second possibility (both decisions want to scale down the system) has also two cases. Firstly, the first decision wants to scale down n VMs and the second one wants to scale down m instances, where $n \geq m$. As the down-scaling policy of Chameleon is conservative, the first event releases m VMs and the second one triggers the releasing of 0 VMs ($\neq$ NOP). In the second case, $m > n$ and thus, the first event scales n VMs down and the second one releases $m - n$ VMs.

The last option is that the decisions request opposite scaling actions. There are also two cases. Firstly, the first decision wants to release n VMs and the second one wants to allocate m VMs with $n \geq m$ or $m > n$. To handle contrary decisions, Chameleon uses a shock absorption factor $0 < \xi \leq 1$. Thus, the first event scales $\lfloor(\xi \cdot d_1)\rfloor$ VMs down and the second one scales $\lceil(\xi \cdot (d_1 + d_2))\rceil$ VMs up. The second case is complementary to the first case. The first event allocates $\lceil(\xi \cdot d_1)\rceil$ VMs and the second one releases $\lfloor(\xi \cdot (d_1 + d_2))\rfloor$ VMs. If ξ = 1, the contrary actions are executed without modifications. With decreasing ξ the distance between the opposite actions decreases. In other words, ξ influences the degree of oscillation in a proactive interval.

2.3 Event Manager

As Chameleon consists of a proactive and reactive cycle, the management of events created by the two cycles is required. The event manager of Chameleon has to accept events, resolve conflicts, and schedule events. An event carries information on its type, either proactive or reactive, the amount of VMs to allocate or release, its trustworthiness, and its planned execution time. In contrast to reactive events that are always considered as trustable, a proactive event is trustable only when the MASE (mean absolute scaled error) [11] of the associated forecast is lower than a tolerance value, see Section 2.1. A reactive event should be executed immediately, whereas a proactive event has an execution time in the future. As the proactive and reactive cycle have different interval lengths, their respective events may have different execution times. An overview of how events are planned and scheduled is shown in Figure 2. In order to handle all the constrains, the manager has to resolve the following conflicts:

SCOPE CONFLICT: Each proactive event has an associated scope, which is a time interval before the event execution in which no other event should occur. That is,

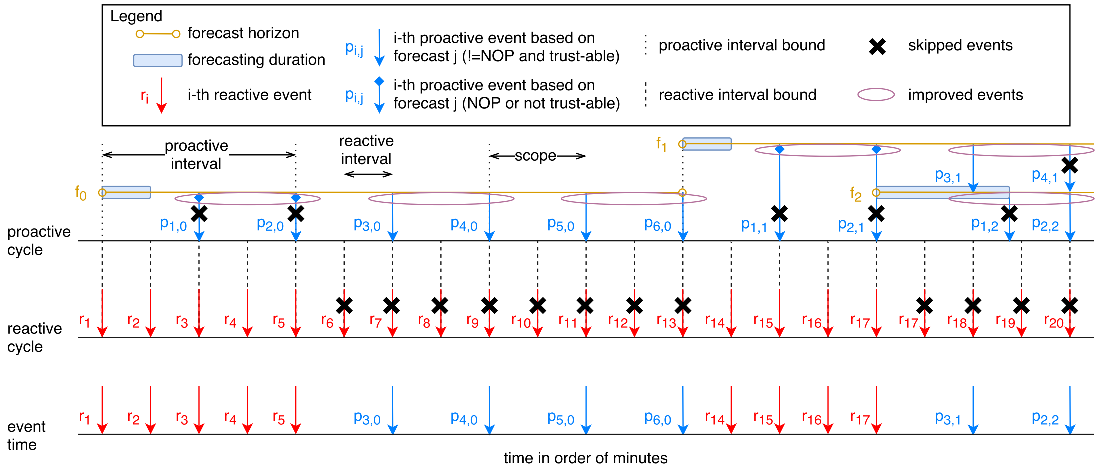


Fig. 2: Exemplary illustration of Chameleon's working mode.

the scope has a fixed length (based on the equidistant time intervals between two events) and ends when the associated event is executed. As proactive events are scheduled in longer intervals, reactive events can be triggered during the scope of a proactive event. This leads to a scope conflict with two cases: (i) If the proactive event is trustable and the associated action is UP or DOWN, then the reactive events are skipped. Figure 2 shows an example of this case. In the scope of the proactive event $p_{5,0}$ for instance, two reactive events r_{10} and r_{11} are triggered. As $p_{5,0}$ is trustable and its action is not a NOP, $p_{5,0}$ is scheduled and both reactive cycle events are skipped. (ii) If the proactive event is not trustable or contains the action NOP, skip the proactive event and execute the reactive one. In Figure 2, the proactive event $p_{1,0}$ for instance, is not trustable and hence, it is ignored. That is, the reactive events r_2 and r_3 that are triggered during the scope of $p_{1,0}$ are executed without modification.

Time conflict: Each event can be identified by its execution time. The time conflict describes the problem when two proactive events with the same execution time appear. This conflict occurs since Chameleon plans proactive events throughout the forecast horizon, however, a new forecast is executed as soon as a drift between the forecast and the monitored load is detected. In such a situation, for some intervals there may be proactive events based on the old forecast and respective events based on the newly conducted forecast. Given that the proactive events based on the new forecast have more recent information, the proactive events based on the older forecast are simply skipped. In Figure 2, the values of forecast f_1 have a deviation from the measured values greater than the tolerance limit. Therefore, the forecast f_2 is executed although the forecast horizon of f_1 is still active. In this situation, the proactive events $p_{4,1}$ and $p_{2,2}$ are scheduled at the same time. As $p_{2,2}$ has more recent information, e.g., the current service demand, $p_{2,2}$ is executed and $p_{4,1}$ is skipped accordingly.

Delay conflict: The execution time of the forecast component ranges between seconds and minutes. Thus, some forecasts may take longer so that the creation of proactive events is delayed, i.e., the event can not take place in the equidistant time interval. In order to prevent such delays, the proactive event of the previous forecast is used for this execution time. An example of this conflict is depicted in Figure 2. Here, through the long computing time, illustrated by the blue box of forecast f_2, the proactive event $p_{1,2}$ is delayed. Thus, the proactive event $p_{3,1}$ of forecast f_1 is executed and $p_{1,2}$ is ignored.

2.4 Assumptions and Limitations

We make following assumptions explicit: (i) in order to obtain sARIMA or tBATS forecasts with a model of the seasonal pattern, the availability of 2 days of historical data is required. With fewer historical data, the forecasts cover only trend and noise components resulting in a decreased accuracy and fewer proactive scaling decisions. (ii) The monitoring values of request arrivals per resource of the application are accurately provided by the monitoring infrastructure, e.g., by polling from a load-balancer or from an instrumented run-time middleware. Chameleon does not rely on utilization measurements. (iii) The service level objective at the application level, which is monitored, is based on the response time of the application. (iv) Chameleon is currently focused on scaling CPU intensive, request-based applications as it relies on the respective service demand estimation models that perform best for CPU-bound workloads. (v) The optional DML descriptive performance model instance can be transformed to a product-form queueing network, whereby each service is modelled as an $M/M/1/\infty$ queue. (vi) As a possible limitation to usability for the Chameleon approach in comparison to existing simplistic reactive auto-scaling mechanisms, we are aware of a setup effort that comes in connection with the forecast and service demand estimation components. However, Chameleon is designed in a way that it would also work with replaced forecast mechanisms and service demand estimation components as long as they work compliant to the defined required interfaces. The overhead of running Chameleon is optimized and comparable to the other proactive policies considered in the evaluation. Especially, the compute intense forecast executions are only triggered when the forecast arrivals drift away from the monitored ones.

3 Evaluation Methodology and Setup

In this section, we introduce the evaluation methodology and setup. First, we summarize the methodological rationale of the BUNGEE experiment controller, the selected workloads used in the evaluation, and the auto-scaled application. Then, in Section 3.3, the five auto-scalers considered in the evaluation are introduced. Finally, we define the set of elasticity and user-oriented metrics used to evaluate and compare the different auto-scalers.

3.1 Elasticity Benchmarking Framework

In order to evaluate the two approaches, we use the BUNGEE Cloud Elasticity Benchmark controller [2]. The working principle is depicted in Figure 3. On the left side, the system under test (SUT) is depicted. It contains the IaaS cloud that hosts the multi-tier applications and the scaling controller. On the right side, the experiment controller (BUNGEE) with its four phases is illustrated. First, the controller constructs for the SUT a discrete mapping function that determines for each load intensity the associated minimum amount of resources required to meet the SLOs (Service Level Objectives). Then, the second phase, called benchmark calibration, uses the mapping from step one to generate identical changes in the curve of the demanded resource units on every platform under comparison. Based on this mapping and a predefined workload profile, the measurement phase stresses the SUT while BUNGEE monitors the supplied VMs. Finally, in the elasticity evaluation phase, the elasticity and user-oriented metrics based on the collected monitoring data are calculated.

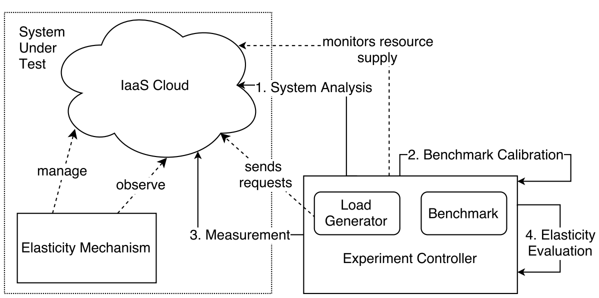


Fig. 3: Experiment setup.

3.2 Workload and Application

To conduct representative experiments, authentic workloads with time-varying load intensity profiles are required. To this end, we collect existing traces from real-life systems that cover up to several month. For a feasible experiment run duration of up to 10 hours, we pick a randomly selected subset from the traces covering up to three days and accelerate the replay time by the factor 7.5 during the experiments so that one day in the traces corresponds to 3.2 hours in the experiments. This way, we trade-off experiment duration and covered time intervals for a realistic setup stressing the auto-scaling mechanisms. A higher time speed-up factor to replay more days within 10 hours experiments time would render the experiments unrealistic, e.g., as the induced changes in demand might exceed the provisioning delays and frequency of resource allocations technically supported by the cloud platforms.

Fifa: The FIFA World Cup 1998[3] trace is a popular trace that represents the HTTP requests to the FIFA servers during the world championship between April and June 1998. This trace was analyzed in the paper of Arlitt and Tai [15]. For our experiments, we use a sub-trace of three days.

Bib: The BibSonomy trace consisting of HTTP requests to servers of the social bookmarking system BibSonomy (see the paper of Benz et al. [16]) during April 2017. Here, we use 2 days for benchmarking the auto-scalers.

IBM: The IBM CICS transactions trace capturing four weeks of recorded transactions on a z10 mainframe CICS installation. From this trace, one weekday was extracted for the experiments.

Wiki: The German Wikipedia[4] trace containing the page requests to all German Wikipedia projects during December 2013. Here, we use two days from this trace.

Retail: The Retailrocket[5] trace containing HTTP requests to servers of an anonymous real-world e-commerce website during June 2015. Similar to German Wikipedia, we use 2 days for the evaluations.

All traces contain 96 data points per day. In case of German Wikipedia, we needed to transform hourly samples using interpolation so that it ends up with 96 data points per day.

The auto-scaling mechanisms are configured to monitor and auto-scale a CPU-intensive Java web application - an implementation of the LU worklet from SPEC's Server Efficiency Rating Tool SERT™2 - as a benchmark application. The application calculates the LU Decomposition [17] of a random generated $n \times n$ matrix, where n is the GET parameter of each HTTP request. The application is deployed on WildFly application servers in all three private and public infrastructure cloud environments. We deploy the application in our private cloud infrastructure that is Apache CloudStack[6] cloud that manages 11 identical, virtualized Xen-Server hosts (HP DL160 Gen9, 8 cores @2.6GHz). To cover setups with background noise, the application is also deployed in both the public AWS EC2 IaaS cloud and in the OpenNebula[7]-based IaaS cloud of the Distributed ASCI Supercomputer 4 (DAS-4) [7]. For all experiments, the auto-scaling mechanism, the load-driver, and the experiment controller are not part of the SUT and are located outside the cloud. The VMs in each scenario have the specification of a m4.large (2 cores and 8GB) instance. To avoid measurement perturbations due to VM image copying, all VMs are initialized ahead of the experiments. All VMs except for one are shutdown at the beginning of each experiment. In the public AWS EC2 cloud, the quota for the number of VMs is set to 20. In order to have comparable scaling ranges over the experiments, the amount of VMs is limited to 18 (20 VMs minus a load-balancer VM for the experiment and a domain controller VM for the WildFly cluster).

3. FIFA Source: http://ita.ee.lbl.gov/html/contrib/WorldCup.html
4. Wiki Source: https://dumps.wikimedia.org/other/pagecounts-raw/2013/
5. Retailrocket Source: https://www.kaggle.com/retailrocket/ecommerce-dataset
6. Apache CloudStack: https://cloudstack.apache.org/
7. OpenNebula: https://opennebula.org/

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2018.2870389

3.3 Competing Auto-Scalers

For the evaluation, we select five representative auto-scalers that have been published in the literature over the past decade. We identify two groups of auto-scalers differing in the way they treat the workload information. The first group are auto-scalers that build a predictive model based on long-term historical data [18], [19], [20]. The second group consists of auto-scalers that only use recent history to make auto-scaling decisions [21], [22]. The selected methods have been published in the following years: 2008 [23] (with an earlier version published in 2005 [18]), 2009 [21], 2011 [19], 2012 [22], and 2014 [20]. This is a representative set of the development of cloud auto-scalers designs across the past 10 years. We describe each of these in more detail.

REACTIVE: Based on the work of Chieu et al. [21], this auto-scaler realizes a scaling mechanism based on thresholds as provided, e.g., by AWS EC2. Here, the user can set a condition to add new instances in increments or based on the amount of currently running resources when the average utilization is higher than a specified threshold over a specified period. Similarly, the user can set a condition to remove instances. An additional cool-down parameter defines a duration after an action, during which the metrics are not evaluated. This allows to avoid possible oscillations by delaying the next possible action. In our experiments, we set the cool-down parameter to 0 and the condition-true period to 2 minutes for both directions. As thresholds, we use 80% CPU utilization for scaling up and 60% for scaling down while adding/removing the fixed amount of 1 unit.

ADAPT: Ali-Eldin et al. [22] 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. 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 in terms of number of delayed requests and the average number of queued requests, at the cost of some resource over-provisioning.

HIST: Urgaonkar et al. [23] propose a provisioning technique for multi-tier Internet applications. The proposed methodology adopts a queueing model to determine how many resources to allocate in each tier of the application. A predictive technique based on building Histograms of historical request arrival rates is used to determine the amount of resources to provision at an hourly time scale. 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.

REG: Iqbal et al. propose a regression-based auto-scaler (hereafter called Reg) [19]. This auto-scaler has a reactive component for scale-up decisions and a predictive component for scale-down decisions. When the capacity is less than the load, a scale-up decision is taken and new VMs are added to the service in a way similar to Reactive. 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 a new measurement is available. When the current load is lower than the provisioned capacity, a scale-down decision is taken using the regression model. This auto-scaler does not perform on the level of other mechanisms in our experiments due to two factors; first, building a regression model for the full history of measurements for every new monitoring data point is a time consuming task. Second, distant past history becomes less relevant as time proceeds. After contacting the authors, we have modified the algorithm such that the regression model is evaluated only for the past 60 monitoring data points.

CONPAAS: ConPaaS, proposed by Fernandez et al. [20], scales a web application in response to changes in throughput at fixed intervals of 10 minutes. The predictor forecasts the future service demand using standard time series analysis techniques, e.g., Linear Regression, Auto Regressive Moving Averages (ARMA), etc. The code for this auto-scaler is open source hosted by the authors themselves[8].

Each auto-scalers is called every 2 minutes and receives a set of input values and returns the amount of VMs that have to be added or removed. The input consists of the following parameters: (i) the accumulated number of requests during the last 2 minutes, (ii) the estimated service demand per request determined by LibReDE as used in Chameleon, and (iii) the number of currently running VMs. The competing auto-scalers are available online[9]. We do not change default configurations as also used in a simulative evaluation [24].

3.4 Metrics for Comparing Auto-scalers

In order 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 that are endorsed by the Research Group of the Standard Performance Evaluation Corporation (SPEC) [8]. As user-oriented metrics, we report the amount of adaptations, the average amount of VMs, and the average and median response time in combination with the percentage of SLO violations. However, we do not take cost-based metrics into account, as they would directly depend on the applied cost model of a provider and thus could be biased. When using only individual metrics for judging the performance of auto-scalers, the results can be ambiguous. Hence, we use 3 different methods for deriving an overall result from the individual metrics: (i) We calculate the auto-scaling deviation of each auto-scaler from the theoretically optimal auto-scaler. (ii) We perform pairwise comparisons among the auto-scalers. (iii) We compute the gain of using an auto-scaler via an elastic speedup metric. 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 time $t \in [0, T]$, (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 VMs required to meet the SLOs

8. ConPaaS auto-scaler: https://github.com/ema/conpaas/tree/master/conpaas-services/src/conpaas/services/webservers/manager/autoscaling

9. Competing auto-scalers: https://github.com/ahmedaley/Autoscalers [24]

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2018.2870389

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 curve of demanded resource units d over time T is derived by BUNGEE, see Section 3.1. The resource supply s_t is the monitored number of running VMs at time t.

PROVISIONING ACCURACY θ_U AND θ_O: These metrics describe the relative amount of resources that are under-provisioned, respectively, over-provisioned during the measurement interval, i.e., 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. Similarly, 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.

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

WRONG PROVISIONING TIME SHARE τ_U AND τ_O: These metrics capture the time in percentage, in which the system is under-provisioned , respectively over-provisioned, during the experiment interval, i.e., the under-provisioning time share τ_U is the time relative to the measurement duration, in which the system has insufficient resources. Similarly, 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, respectively no over-provisioning, is detected within a measurement.

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

INSTABILITY υ: This last metric describes the time in percentage in which the supply and the demand curves are not changing in the same direction, i.e., the instability υ measures the fraction of time in which the demand change and the supply change have different signs. Values of this metric lie in the interval $[0, 100]$, where 0 is the best value and means that demand and supply are always moving in the same direction.

$$\upsilon[\%] := \frac{100}{T - t_1} \cdot \sum_{t=2}^{T} min(|sgn(\Delta s_t) - sgn(\Delta d_t)|, 1)\Delta t$$

AUTO-SCALING DEVIATION σ: In order to quantify the performance of auto-scalers and rank them, we propose to calculate the deviation of a given auto-scaler compared to the theoretically optimal auto-scaler. For the calculation of the deviation between two auto-scalers, we use the Minkowski distance d_p. Here, the vectors consist of a subset of the aforementioned system- and user-oriented evaluation metrics. We take the provisioning accuracy θ, the wrong provisioning time share τ, the instability υ, and the SLO violations ψ into account. The metrics are specified as percentages. The closer the value of a metric is to zero, the better the auto-scaler performs with respect to the aspect characterized by the respective metric, i.e., the closer the auto-scaling deviation is to zero, the closer the behavior of the auto-scaler to the theoretically optimal auto-scaler.

The first step is to calculate the elasticity metrics. Then, we calculate the overall provisioning accuracy θ and the overall wrong provisioning time share τ. Hereby, we use a weighted sum for both metrics consisting of both components and a penalty factor $0 < \gamma < 1$. This penalty can be set individually (in our case, γ is set to 0.5 to reflect custom preference), with $\gamma > 0.5$ indicating that under-provisioning is worse than over-provisioning, $\gamma = 0.5$ indicating that under- and over-provisioning are equally bad, and $\gamma < 0.5$ indicating that over-provisioning is worse than under-provisioning. This can be expressed as follows:

$$\theta[\%] := \gamma \cdot \theta_U + (1 - \gamma) \cdot \theta_O$$
$$\tau[\%] := \gamma \cdot \tau_U + (1 - \gamma) \cdot \tau_O$$

In the last step, the Minkowski distance d_p between the auto-scaler and the theoretically optimal auto-scaler is calculated. As the theoretically optimal auto-scaler is assumed to know when and how much the demanded resources change, the values for provisioning accuracy θ, wrong provisioning time share τ, instability υ, and the SLO violations ψ are equal to zero. In other words, if an auto-scaler is compared to the theoretically optimal auto-scaler, the L_p-norm can be used as $\|x - 0\|_p = \|x\|_p$ with $x = (\theta, \tau, \upsilon, \psi)$, i.e., in our case the auto-scaling deviation σ between an auto-scaler and the theoretically optimal auto-scaler is defined as follows:

$$\sigma[\%] := \|x\|_4 = \left(\theta^4 + \tau^4 + \upsilon^4 + \psi^4\right)^{\frac{1}{4}}$$

PAIRWISE COMPETITION κ: Another approach for ranking the auto-scalers is to use the pairwise comparison method [25]. Here, for each auto-scaler the value of each metric is pairwise compared with the value of the same metric for all other auto-scalers. As values near to zero are better, the auto-scaler with the lowest value gets one point. If a metric for both auto-scalers is equal, both auto-scalers get each half a point. In addition, we divide the reached score of each auto-scaler by the maximum achievable score. In other words, the pairwise competition κ shows how much of the achievable points each auto-scaler collected.

ELASTIC SPEEDUP ϵ: Besides the auto-scaling deviation and the pairwise competition, we want to introduce a further method that calculates for each auto-scaler its gain based on its scaling behavior compared to the no auto-scaling scenario. This approach allows to rank the auto-scalers by taking only the elasticity metrics into account. In other words, the elasticity metrics $x = (\theta_U, \theta_O, \tau_U, \tau_O, \upsilon)$ of an auto-scaler are compared to the metrics of the no auto-scaling scenario. To this end, the geometrical mean of the ratio between each metric pair is calculated. Mathematically, the elastic speedup κ for an auto-scaler a based on the no auto-scaling scenario b can be formulated as:

$$\epsilon_b := \left(\frac{\theta_{U,b}}{\theta_{U,a}} \cdot \frac{\theta_{O,b}}{\theta_{O,a}} \cdot \frac{\tau_{U,b}}{\tau_{U,a}} \cdot \frac{\tau_{O,b}}{\tau_{O,a}} \cdot \frac{\upsilon_b}{\upsilon_a} \right)^{\frac{1}{5}}$$

This section answers RQ2 from the Section 1 outlining the design of a level playing field for auto-scaler competitions. The results in the following section underline that our evaluation methodology is representative and repeatable with a strong connection of human perceived auto-scaler performance reflected in the metric values and illustrations.

4 Experiment Results

In this section, we benchmark Chameleon and the other auto-scalers. We have conducted 7 different sets of experiments with 6 auto-scalers. Section 4.1 explains how to interpret the results of the measurements. Afterwards, in Section 4.2, the measurements in the private cloud and the public cloud are presented and compared. In Section 4.3, the results of all measurements are combined, illustrated and discussed. Afterwards, we discuss threats to validity and finally, conclude the evaluation with a list of key findings.

4.1 Introduction to the Results

We first introduce the format in which results are presented before discussing the detailed results. To this end, the experiment results for the German Wikipedia trace are shown Figure 4. This diagram shows the Chameleon measurement (top left) compared with the measurements for all competing auto-scalers: Reactive (top right), Adapt (middle left), Hist (middle right), ConPaaS (bottom left) and Reg (bottom right). For each auto-scaler, the x-axis shows the time of the measurement in minutes; the y-axis shows the number of concurrently running VMs. The blue curves represent the supplied VMs of each auto-scaler; the black dashed curves represent the required amount of VMs. The interpretation of the curves are the same as in the single experiment example.

When comparing the scaling behavior of the different auto-scalers for the German Wikipedia trace, a first observation is that the auto-scalers can be grouped into two categories: (i) tendency to over-provision the system (Hist, Reactive, Adapt and Chameleon), (ii) tendency to under-provision the system (ConPaaS and Reg). Furthermore, Reg and ConPaaS have a high rate of oscillations during the measurement. Chameleon is the first auto-scaler that meets the demand surge in the beginning of the experiment and then tends to allocate slightly more VMs than required for the remaining time. Adapt, for instance, has a similar behavior as Chameleon but it allocates more VMs. Reactive allocates almost the right amount of VMs during the increasing and constant load, but is too slow when scaling down. In contrast, Hist roughly meets the demand and holds the amount of allocated VMs for about 30 to 60 minutes after which it drops to the current demand.

To provide a quantitative comparison, the elasticity metrics (see Section 3.4) and user-oriented metrics are computed and listed in Table 1. The first column shows the metrics and the following ones represent the different auto-scalers with the last one corresponding to the no auto-scaling scenario. The rows of the table show the values of different metrics with the best value highlighted in bold. For the elasticity metrics, as well as for the metrics SLO violations, auto-scaling deviation, average response time and median response time it generally holds that the lower the value the better the auto-scaler performs. In contrast, for the pairwise competition and elastic speedup, a higher value is better.

Metric	Cham.	Adapt	Hist	ConPaas	Reg	React.	No AS
θ_U (acc$_U$)	**1.57%**	1.68%	2.37%	14.69%	16.08%	2.10%	25.93%
θ_O (acc$_O$)	11.15%	17.51%	33.55%	15.67%	**4.34%**	28.94%	19.38%
τ_U (ts$_U$)	**5.70%**	9.16%	12.75%	47.41%	51.04%	12.27%	70.48%
τ_O (ts$_O$)	73.25%	80.94%	71.95%	32.07%	**25.24%**	80.77%	26.28%
υ (inst)	5.83%	7.09%	4.75%	12.66%	12.88%	4.97%	**2.94%**
ψ (SLO)	**2.95%**	15.47%	11.86%	59.73%	80.71%	7.15%	85.95%
σ (AS dev.)	**39.49%**	45.24%	42.75%	62.54%	81.71%	46.67%	88.13%
κ (p. comp.)	**77.78%**	50.00%	50.00%	41.67%	41.67%	52.78%	36.11%
ϵ (ES)	**2.30**	1.78	1.51	0.91	1.19	1.56	1.00
#Adapt.	61	66	26	112	102	49	0
Avg. #VMs	10.795	12.343	11.571	11.191	9.3761	10.451	9
Avg. resp. t.	**0.62** s	1.22 s	1.06 s	3.31 s	4.20 s	0.82 s	4.38 s
Med. resp. t.	**0.43** s	0.48 s	0.48 s	5.00 s	5.00 s	0.46 s	5.00 s

TABLE 1: Metric overview for the German Wikipedia trace.

When comparing individual metrics (see Table 2), only the aspect characterized by the respective metric is considered. For instance, in the German Wikipedia scenario, Chameleon has the best values for θ_U, τ_U, and ψ. Reg has the best values for θ_O and τ_O, but the highest ψ compared to all other auto-scalers. Therefore, we evaluate the performance of the auto-scalers based on σ, κ, and ϵ. Chameleon has the best values for each of these aggregate metrics. This is supported by also looking at the average and median response time for which Chameleon also exhibits the best values.

4.2 Auto-Scaling in Private vs. Public IaaS Clouds

In this section, we run additionally the experiments with the FIFA World Cup 1998 trace in the AWS EC 2 and in the DAS-4 environment in order to evaluate the behavior of the different auto-scalers when running in a public cloud. In contrast to our private cloud, in AWS we observe a high performance variation for each VM and a high degree of background noise. The resulting scaling behavior of a subset of the auto-scalers is depicted in Figure 5. The figures are structured as explained in Section 4.1; the left column shows Chameleon and Reactive in the private cloud scenario and the right column shows the same auto-scalers in the public AWS EC2 scenario. While Chameleon scales in a similar manner in both scenarios, a bigger difference is observed for Reactive, as the diagrams in Figure 5 show. In contrast to Chameleon that makes decisions based on the amount of requests and the service demands, the scaling decisions of Reactive are only based on the observed CPU load of

θ_U, θ_O	Accuracy as the relative amount of resource units that are under- (acc$_U$) resp. over-provisioned (acc$_O$) normalized by time.
τ_U, τ_O	Wrong provisioning time share as the relative amount of time in under- (ts$_U$) resp. over-provisioned (ts$_O$) state.
υ	Instability (inst) as the relative amount of time in which demand and supply are not parallel.
ψ	The relative amount of SLO violations.
σ	Deviation from theoretically optimal auto-scaler (AS dev.).
κ	The portion of wins in the pairwise competition (p. comp.).
ϵ	The elastic speedup score (ES) compared to no auto-scaling.

TABLE 2: Metric overview and explanation.

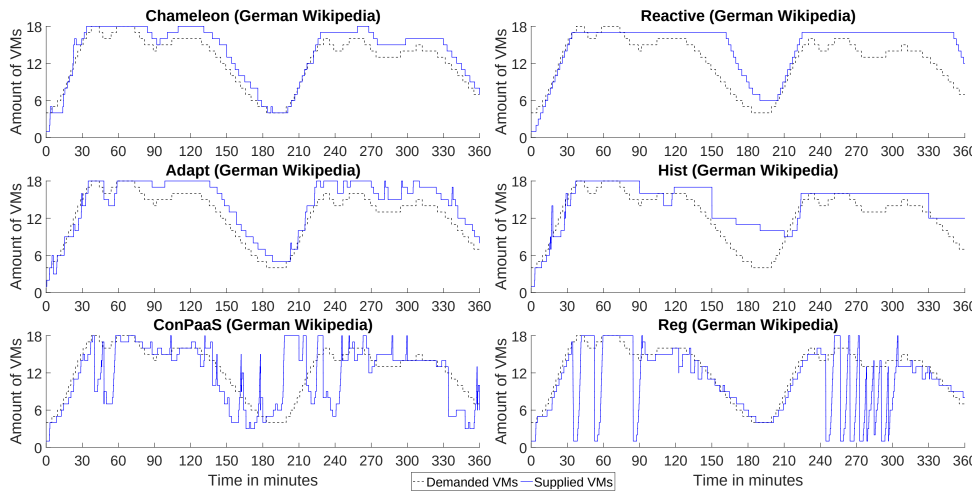


Fig. 4: Comparison of the auto-scalers for the German Wikipedia trace.

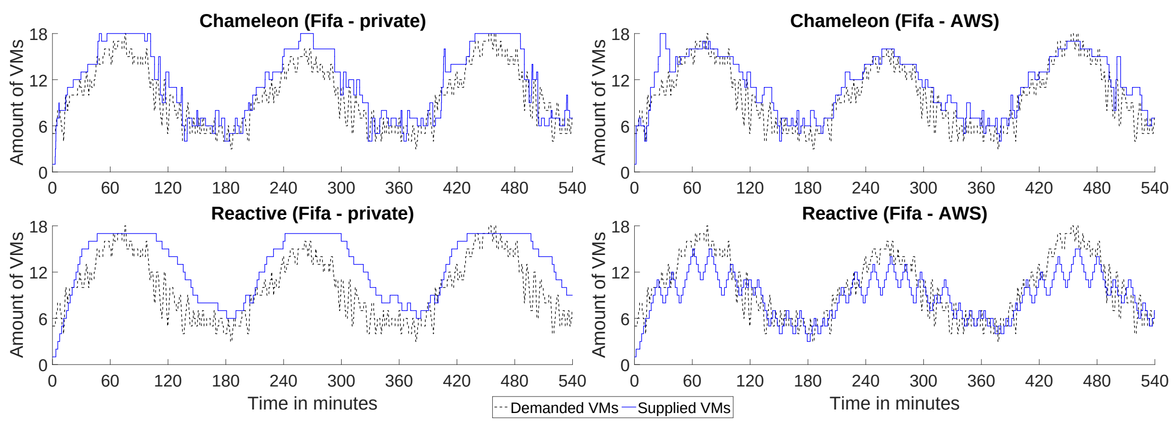


Fig. 5: Comparison of the auto-scalers in both the private cloud scenario and public AWS EC2 cloud.

the system. While the system is in an under-provisioned state, the CPU load drops significantly for a certain time and Reactive scales down as since CPU usage indicates a low load. After the CPU load starts increasing, Reactive tries to scale up the system again. Such drops in the CPU load seem to follow a scheme. Therefore, we assume that AWS EC2 performs migrations in the background for migrating the VMs from overloaded hosts to hosts with less load. After such migrations, the CPU usage drops given that more resources are available on the new host. The associated metrics are listed in Table 3. While in the private scenario, Chameleon achieved the best value for the auto-scaling deviation metric, in the public scenario, Chameleon achieved the best values for the pairwise competition and elastic speedup metrics, in addition to the auto-scaling deviation metric. Reactive has in the private scenario the best score for pairwise competition and elastic speedup, while in the public scenario, it exhibits average performance for the aggregate metrics.

Metric	Chameleon		Reactive		No AS
	private	AWS EC2	private	AWS EC2	
θ_U (acc$_U$)	3.23%	1.64%	**1.56%**	13.36%	14.75%
θ_O (acc$_O$)	21.95%	21.31%	40.20%	**9.19%**	27.41%
τ_U (ts$_U$)	13.09%	11.04%	**5.20%**	57.49%	48.93%
τ_O (ts$_O$)	74.05%	67.92%	87.14%	**25.49%**	44.77%
υ (inst)	16.36%	15.95%	14.68%	19.02%	**12.66%**
ψ (SLO)	8.65%	5.04%	**2.70%**	49.30%	62.14%
σ (AS dev.)	43.88%	**39.81%**	46.76%	54.80%	66.83%
κ (p. comp.)	61.11%	**69.44%**	66.67%	44.44%	44.44%
ϵ (ES)	1.58	**1.93**	**1.93**	1.27	1.00
Avg. resp. t.	0.89 s	0.62 s	**0.60 s**	2.68 s	3.32 s
Med. resp. t.	0.45 s	**0.26 s**	0.45 s	3.23 s	5.00 s

TABLE 3: Metrics of private vs. public AWS EC2 scenarios.

4.3 Overall Evaluation

As the previous subsections show only selected subsets of the experiment results, an overview of all results is presented and discussed here. In our experiment design, each auto-scaler is observed for 18 days on 5 different traces where one day takes 3.2 hours. The experiments took in total over 400 hours, during which about 107 million requests were sent and 5000 adaptations were performed by the auto-scalers. In other words, during one hour about 275.000 requests arrived at the system on average and the system experienced about 13 adaptions on average.

Table 4 shows the average metrics over all traces. Here, each row represents a metric and each column an auto-

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2018.2870389

scaler. As previously, the best values are highlighted in bold. When comparing the auto-scalers based on the auto-scaling deviation, the lowest deviation from the theoretically optimal auto-scaler for all experiments is achieved by Chameleon, followed by Hist, Adapt, Reactive, ConPaaS, and finally, Reg. In a pairwise competition, the most points are collected by Chameleon, followed by Hist, Reactive, Adapt, Reg, and finally, ConPaaS. When taking the elastic speedup into account, the best performance is achieved by Chameleon, followed by Reactive, Adapt, Reg, Hist, and finally, Hist. To summarize, Chameleon achieves the best results for all three aggregate metrics.

Metric	Cham.	Adapt	Hist	ConPaaS	Reg	React.
$\overline{\theta}_U$ (avg. acc$_U$)	**3.63**%	6.45%	4.70%	15.55%	15.69%	6.98%
$\overline{\theta}_O$ (avg. acc$_O$)	17.88%	19.94%	52.64%	25.98%	**10.51**%	34.47%
$\overline{\tau}_U$ (avg. ts$_U$)	**13.32**%	30.43%	22.75%	42.04%	43.71%	25.41%
$\overline{\tau}_O$ (avg. ts$_O$)	65.06%	51.41%	62.35%	41.69%	**33.42**%	62.08%
$\overline{\upsilon}$ (avg. inst.)	13.91%	16.60%	**11.95**%	17.42%	17.02%	12.99%
$\overline{\psi}$ (avg. SLO)	**10.29**%	32.76%	15.59%	44.11%	60.16%	21.96%
$\overline{\sigma}$ (avg. AS dev.)	**39.63**%	46.90%	46.43%	54.03%	63.46%	48.14%
$\overline{\kappa}$ (avg. p. comp.)	**69.44**%	50.00%	58.33%	36.51%	42.46%	55.56%
$\overline{\epsilon}$ (avg. ES)	**2.02**	1.48	1.38	1.10	1.41	1.49

TABLE 4: Average metrics over all experiments.

Trace	Cham.	Adapt	Hist	ConPaaS	Reg	Reactive
Bib	**1.33**	5.33	2.00	5.00	4.33	2.00
Fifa (AWS EC2)	**1.00**	3.00	3.33	5.33	4.33	4.00
Fifa (DAS-4)	**1.33**	2.33	3.00	5.66	5.33	3.00
Fifa (private)	**1.67**	3.67	3.00	5.33	5.67	**1.67**
IBM	**1.00**	4.67	3.33	3.67	3.67	4.67
Wiki	**1.00**	2.67	3.00	5.33	5.33	3.00
Retail	**1.00**	2.33	3.33	6.00	4.33	4.00
Avg. Ranking	**1.19**	3.43	3.00	5.19	4.71	3.19

TABLE 5: Average ranking for each experiment over the three competitions.

To rank the auto-scalers, we compute the average rank of each auto-scaler over all experiment setups. First, we compute the ranks of each auto-scaler for the auto-scaling deviation, the pairwise competition, and the elastic speedup over all experiments. Then, we determine the average rank of each auto-scaler as the arithmetic mean over the three ranks. The ranks are listed in Table 5. Here, each column represents an auto-scaler and each row an experiment. The entries in each cell represent the rank in the associated experiment. As usual, the best values are highlighted in bold. Chameleon has the smallest average rank of 1.19. This means that Chameleon exhibits the best rank on average among all traces and experiments. The second best auto-scaler based on this ranking is Hist, followed by Reactive, Adapt, Reg, and ConPaaS. Except for Chameleon, the auto-scalers in competition show a significant variance in their ranks over the experiments. That is, none of competing auto-scalers, with exception of Chameleon, outperforms the others in all 7 experiment rows.

In order to visualize the scaling behavior of all auto-scalers, Figure 6 shows a spider chart, also known as radar chart, of each auto-scaler. Each spiderweb contains six edges, where the edges represent the elasticity metrics and the SLO violations, and shows the results for each trace in the private cloud scenario. The smaller and thinner the stretched areas are, the better the respective auto-scaler performs. Based on these diagrams, we can conclude that Chameleon tends to slightly over-provision allocating slightly more VMs than required. In contrast, Hist tends to over-provision but has a worse over-provisioning accuracy than Chameleon. Reactive has the worst over-provisioning time share and the second worst over-provisioning accuracy. Adapt tends to a balance of under- and over-provisioning with a slight tendency to over-provision. In the under-provisioned states, Adapt has only a few VMs less than required Reg and ConPaaS have (due to their oscillations) no tendency to either under-provision or over-provision the system. Both auto-scalers exhibit the highest SLO violations. Besides comparing the tendencies, the stability of each auto-scaler over all traces can be investigated. Chameleon exhibits a stable scaling behavior for all five traces as all areas are oriented in one direction and the areas have less deviation to each other. The other auto-scalers have either areas with different orientation or areas with a higher variance than Chameleon. Hence, the other auto-scalers can be seen as less stable than Chameleon.

4.4 Threats to Validity

Although our experimental analysis covered a wide range of different scenarios, the results may not be generalizable to other types of applications, for example, applications that are not interactive or CPU intensive. In principle, for the evaluated competing auto-scalers a comparable behavior has been observed in related works on auto-scaler evaluation for workflows [26] and in simulation [24]: platoons for Hist, some oscillations for ConPaaS and Reg, and an tightly following the demand of the Adapt policy. It is still worth noting that, as discussed in previous studies [24], [26], most of the auto-scalers evaluated in this paper are sensitive to their configuration for a given scenario.

The repeatability of performance related experiments in public cloud environments is limited due to the fact that there is no control given about the placement and co-location of VMs with other workloads running on the cloud. This can cause significant performance variability [27]. To alleviate this problem, we conduct most experiments in a private environment under controlled conditions, however, for completeness, we also include experiments conducted in two different public cloud deployments (DAS-4 and AWS EC2) with a different degree of background load. Furthermore, we conduct long running experiments while not stressing the cloud's APIs much with on average 13 adaptations per hour.

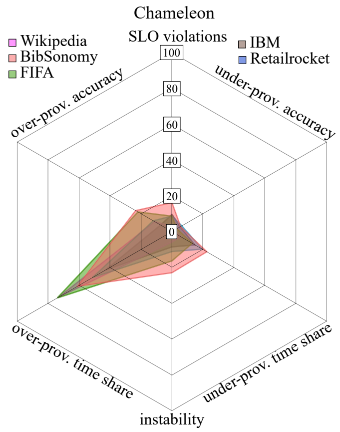
We address the threat of possible bias by using multiple and established sets of metrics that have been officially endorsed by SPEC [8]. In this paper, the detailed elasticity metrics are combined to aggregate metrics in an unweighted manner, treating under-provisioning and over-provisioning as being equally bad. In the case where under-provisioning is considered worse than over-provisioning, the results for Chameleon would further improve due to its tendency to slightly over-provision.

4.5 Summary of Evaluation Findings

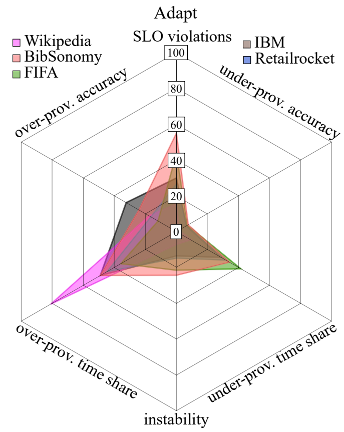
We summarize the main findings of the experimental evaluation as follows:

I: The Chameleon auto-scaler performs best in the evaluated scenarios based on average competition results, under-provisioning time share, under-provisioning accuracy and

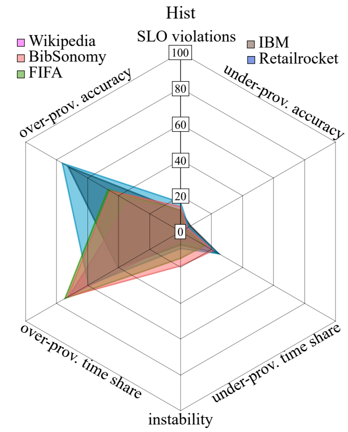
This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TPDS.2018.2870389



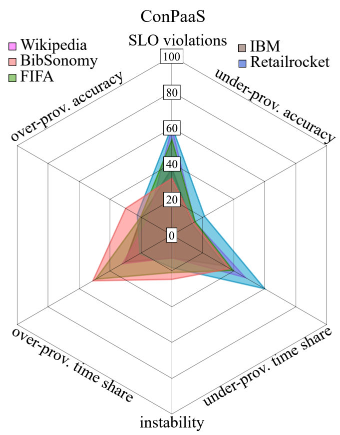
(a) Metric overview Chameleon.



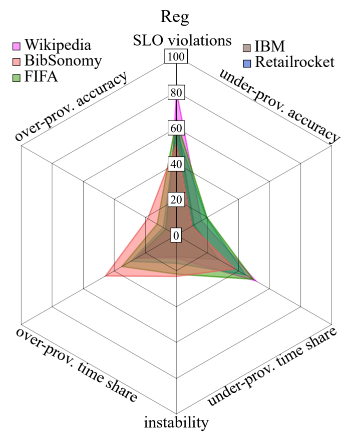
(b) Metric overview Adapt.



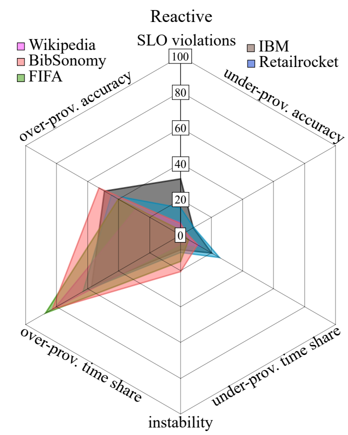
(c) Metric overview Hist.



(d) Metric overview ConPaaS.



(e) Metric overview Reg.



(f) Metric overview Reactive.

Fig. 6: Scaling behavior of all auto-scalers.

SLO violations. Chameleon tends to a reliable slight over-provisioning. **II:** The proposed way of combining proactive and reactive scaling decisions improves the auto-scaling performance. This answers RQ1 from Section 1. **III:** Adapt manages to closely follow the demand with a relatively high number of adaptations. **IV:** Hist and Reactive tend to stronger over-provisioning than others. **V:** ConPaaS and Reg exhibit unstable behavior in some situations and can not be considered as reliable in the covered scenarios. **VI:** Reactive relies on accurate CPU utilization measurements. It shows decreased performance in a public cloud context, where overbooking of virtual resources may cause significant interference with background load. **VII:** In the conducted experiments, Chameleon exhibits consistent scaling behavior, whereas the other auto-scalers show more variance. **VIII:** Among the investigated state-of-the-art auto-scalers, no mechanism outperforms the other ones in all traces, see Table 5.

5 Related Work

The topic of auto-scaling has been a popular research topic in the resource management community over the past decade. There have been multiple recent efforts to survey the state-of-the-art in auto-scaling, for example: (i) G. Galante and L. de Bona [28], (ii) B. Jennings and R. Stadler [29], and (iii) T. Lorido-Botran et al. [1], and recently (iv) C. Qu et al. [30]. These surveys provide an overview on the current state of research on auto-scaling. The survey by Lorido-Botran et. al [1] proposes a classification of auto-scalers into five groups:

Threshold-based rules: These are auto-scalers that react to load changes based on pre-defined threshold rules for scaling a system. As the scaling decisions are triggered by performance metrics and predefined thresholds, they are easy to deploy. Thus, they are popular with commercial cloud providers and clients. Common threshold-based auto-scalers are designed for instance by R. Han et al. [31] or M. Maurer et al. [32].

Queueing theory: Queueing theory is usually used to determine the relationship between incoming and outgoing jobs in a system. Proactive auto-scalers in this category rely on a model of the system for predicting the future resource demand. A simple approach is to model each VM as a queue of requests. State-of-the-art auto-scalers are proposed, e.g., by B. Urgaonkar et al. [23] (Hist) or Q. Zhang et al. [33].

Control theory: Similarly to queueing theory, auto-scalers that use control theory also employ a model of the application. Hence, the performance of such controllers depends on the application model and the controller itself. A new approach in recent research is to combine this type auto-scalers with queueing theory. For example, popular hybrid auto-scalers of this group are from P. Padala et al. [34] or A. Ali-Eldin et al. [22] (Adapt).

Reinforcement learning: Instead of having explicit knowledge or a model of the application, approaches in this category aim to learn the best action for a specific state. The learning is based on a trial-and-error approach converging towards an optimal policy. This allows the controller to be proactive. However, finding the best option for a particular state can take a long time. G. Tesauro et al. [35] or J. Rao et al. [36], for example, propose such auto-scalers. Auto-scalers based on reinforcement learning are highly dependent on the initial extensive training phase and therefore we do not consider them in our evaluation.

Time series analysis: Auto-scalers that use time series analysis are the most common representatives of proactive scaling. They allow to predict the future behavior of sequences of job arrivals. A variety of forecasting methods based on time series analysis exist in the literature. The choice of forecasting technique and its associated parameters influences the forecasting accuracy. Common examples of this type of auto-scalers are proposed by G. Pierre and C. Stratan [37] (ConPaaS) or W. Iqbal et al. [19] (Reg).

The recent survey [30] highlights the importance of combining reactive and proactive auto-scaling mechanisms. Existing previous auto-scalers try to leverage both reactive and proactive methods [19], [22], [23], [38]: For example, Reg [19] limits up-scaling to reactive and down-scaling to proactive decisions. Adapt [22] detects the slope of the recent workload observations and is thus limited to short-term predictions. Hist [23] leverages reactive scaling only in the presence of workload anomalies considered as flash crowds. Biswas et. al. [38] adopt a broker approach for optimizing profit of stakeholders. Thus, proactive and reactive scaling decisions are merged implicitly.

In contrast to the state-of-the-art on hybrid auto-scalers that combine reactive and proactive mechanisms, Chameleon (i) leverages long-term predictions from time series analysis in combination with (ii) predictive models from queueing theory also integrating a (iii) reactive fall-back mechanism. Two explicit reactive and proactive cycles generate scaling decisions covering current and future auto-scaling intervals. The challenge of handling contradicting reactive and proactive events is not covered explicitly in related approaches. Thus, we consider the conflict resolution and scaling decision optimization as part of the Chameleon logic as new.

We identify only two cases of the previous work comparing multiple and different, proactive auto-scaling policies. The work of Padadopoulos et al. [24] establishes theoretical bounds on the worst case performance using simulation. The related experimental evaluation in Ilyushkin et al. [26], compares auto-scaler performance for the different type of workflow applications in one deployment. To the best of our knowledge, currently only the above two works contain a comparably broad set of auto-scalers in their evaluations. Thus, this paper contains the first broad experimental evaluation covering different deployments and traces with a representative set of auto-scaling algorithms.

6 Conclusion

In this paper, we present the hybrid, proactive auto-scaler Chameleon and benchmark it against four other state-of-the-art proactive auto-scalers and one reactive auto-scaler. Chameleon combines forecasting (time series analysis) and service demand estimation (queueing theory), which can optionally be enriched with application knowledge captured a descriptive software performance model to increase the timeliness and accuracy of the auto-scaling reconfigurations. The forecast and the service demand estimation are realized by integrating established open-source tools provided by the research community. In the evaluation, we employ a set of elasticity and user-oriented metrics to benchmark Chameleon in comparison to other auto-scalers. We run experiments in a private CloudStack-based environment, in the public AWS EC2 IaaS cloud and in the OpenNebula-based IaaS Cloud of the Distributed ASCI Supercomputer 4 (DAS-4). For representative and repeatable measurements, the BUNGEE elasticity benchmarking framework is used. We conduct more than 400 hours of experiments. The workload scenarios consist of a CPU intensive Java Enterprise application (endorsed to SPEC SERT^{TM}2) driven by five different real-world load traces. For the other auto-scalers, we observe typical scaling behavior characteristics. Furthermore, the performance of the state-of-the-art auto-scalers depends on the workload characteristics and thus, none of the other auto-scalers outperforms the others for all traces covered. In contrast, Chameleon achieves in all setups and among all traces the best scaling behavior.

We see potential to extend our proposed Chameleon approach towards multi-dimensional and nested auto-scaling. Here, Chameleon would have to scale multiple tiers of a distributed application either horizontally (by adding further nodes) or vertically (by adding resources to existing nodes). Deciding between the two options would require a significant extension to the decision logic that may for example be based on machine learning. Auto-scaling on nested resource layers like virtual machines or containers poses a new challenge on its own. As another direction of extension, we started working on integrating awareness of the applied cost model. In the presence of long accounting intervals per resource, Chameleon could leverage knowledge from the forecast executions to delay or skip scaling events. This would result in a shifted trace-off with decreased elasticity results but lowered costs.

References

[1] 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.

[2] N. Herbst, S. Kounev, A. Weber, and H. Groenda, "BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments," in SEAMS 2015. IEEE Press, 2015, pp. 46–56.

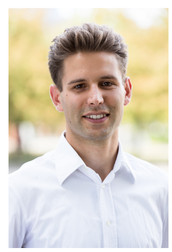
[3] G. E. Box and more, Time Series Analysis: Forecasting and Control. John Wiley & Sons, 2015.

[4] A. M. D. Livera, R. J. Hyndman, and R. D. Snyder, "Forecasting Time Series With Complex Seasonal Patterns Using Exponential Smoothing," Journal of the American Statistical Association, vol. 106, no. 496, pp. 1513–1527, 2011.

[5] 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, July 2016.

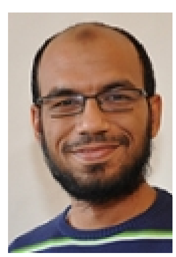
[6] S. Spinner, G. Casale, F. Brosig, and S. Kounev, "Evaluating Approaches to Resource Demand Estimation," Elsevier Performance Evaluation, vol. 92, pp. 51 – 71, October 2015.
[7] H. Bal and more, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," IEEE Computer, vol. 49, no. 5, pp. 54–63, May 2016.
[8] N. Herbst and more, "Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics," CoRR, vol. abs/1604.03470, 2016.
[9] S. Spinner, G. Casale, X. Zhu, and S. Kounev, "LibReDE: A library for Resource Demand Estimation," in ACM/SPEC ICPE 2014. ACM, 2014, pp. 227–228.
[10] S. Spinner, J. Walter, and S. Kounev, "A Reference Architecture for Online Performance Model Extraction in Virtualized Environments," in ACM/SPEC ICPE 2017. ACM, 2016, pp. 57–62.
[11] R. J. Hyndman and A. B. Koehler, "Another Look at Measures of Forecast Accuracy," International Journal of Forecasting, pp. 679–688, 2006.
[12] R. J. Hyndman and Y. Khandakar, "Automatic Time Series Forecasting: The Forecast Package for R," Journal of Statistical Software, vol. 26, no. 3, pp. 1–22, 2008.
[13] X. Wang, K. Smith-Miles, and R. Hyndman, "Rule Induction for Forecasting Method Selection: Meta-learning the Characteristics of Univariate Time Series," Neurocomputing, vol. 72, no. 10 - 12, pp. 2581 – 2594, 2009.
[14] J. Grohmann, N. Herbst, S. Spinner, and S. Kounev, "Self-Tuning Resource Demand Estimation," in IEEE ICAC 2017, July 2017.
[15] M. Arlitt and T. Jin, "A Workload Characterization Study of the 1998 World Cup Web Site," IEEE Network, vol. 14, no. 3, pp. 30–37, 2000.
[16] D. Benz and more, "The social bookmark and publication management system bibsonomy," VLDB, vol. 19, no. 6, pp. 849–875, 2010.
[17] J. R. Bunch and J. E. Hopcroft, "Triangular factorization and inversion by fast matrix multiplication," Mathematics of Computation, vol. 28, no. 125, pp. 231–236, 1974.
[18] B. Urgaonkar et al., "An Analytical Model for Multi-Tier Internet Services and its Applications," in ACM SIGMETRICS, 2005.
[19] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive Resource Provisioning for Read Intensive Multi-tier Applications in the Cloud," Future Generation Computer Systems, vol. 27, no. 6, pp. 871–879, 2011.
[20] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling Web Applications in Heterogeneous Cloud Infrastructures," in IEEE IC2E, 2014.
[21] T. Chieu and more, "Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment," in IEEE ICEBE 2009. IEEE, 2009, pp. 281–286.
[22] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures," in IEEE NOMS 2012. IEEE, 2012, pp. 204–212.
[23] B. Urgaonkar and more, "Agile Dynamic Provisioning of Multi-tier Internet Applications," ACM TAAS, vol. 3, no. 1, p. 1, 2008.
[24] A. Papadopoulos and more, "PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications," ACM ToMPECS, vol. 1, no. 4, pp. 1–31, August 2016.
[25] H. A. David, "Ranking from Unbalanced Paired-Comparison Data," Biometrika, vol. 74, pp. 432–436, 1987.
[26] A. Ilyushkin and more, "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.
[27] A. Iosup, N. Yigitbasi, and D. Epema, "On the Performance Variability of Production Cloud Services," in CCGrid 2011, 2011, pp. 104–113.
[28] G. Galante and L. de Bona, "A Survey on Cloud Computing Elasticity," in IEEE UCC 2012. IEEE, 2012, pp. 263–270.
[29] B. Jennings and R. Stadler, "Resource Management in Clouds: Survey and Research Challenges," Journal of Network and Systems Management, vol. 23, no. 3, pp. 567–619, 2015.
[30] 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, Jul. 2018. [Online]. Available: http://doi.acm.org/10.1145/3148149
[31] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight Resource Scaling for Cloud Applications," in IEEE/ACM CCGrid 2012. IEEE, 2012, pp. 644–651.
[32] M. Maurer, I. Brandic, and R. Sakellariou, "Enacting Slas in Clouds Using Rules," in Euro-Par 2011. Springer, 2011, pp. 455–466.
[33] Q. Zhang, L. Cherkasova, and E. Smirni, "A Regression-based Analytic Model for Dynamic Resource Provisioning of Multi-tier Applications," in IEEE ICAC 2007. IEEE, 2007, pp. 27–27.
[34] P. Padala and more, "Automated control of multiple virtualized resources," in ACM European Conference on Computer Systems. ACM, 2009, pp. 13–26.
[35] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," in IEEE ICAC 2006. IEEE, 2006, pp. 65–73.
[36] J. Rao and more, "VCONF: a Reinforcement Learning Approach to Virtual Machines Auto-configuration," in ACM ICAC 2009. ACM, 2009, pp. 137–146.
[37] G. Pierre and C. Stratan, "ConPaaS: a Platform for Hosting Elastic Cloud Applications," IEEE Internet Computing, vol. 16, no. 5, pp. 88–92, 2012.
[38] A. Biswas, S. Majumdar, B. Nandy, and A. El-Haraki, "A hybrid auto-scaling technique for clouds processing applications with service level agreements," Journal of Cloud Computing, vol. 6, no. 1, p. 29, Dec 2017.

André Bauer is PhD student at the chair of software engineering at the University of Würzburg. He serves as elected newsletter editor of the SPEC Research Group. His research topics include elasticity in cloud computing, auto-scaling and resource management, autonomic and self-aware computing, and forecasting.



Nikolas Herbst is a research group leader at the chair of software engineering at the University of Würzburg. He received a PhD from the University of Würzburg in 2018 and serves as elected vice-chair of the SPEC Research Cloud Group. His research topics include elasticity in cloud computing, auto-scaling and resource management, performance evaluation of virtualized environments, autonomic and self-aware computing.

Simon Spinner is a software engineer at IBM and an associate researcher at the Chair of Software Engineering at the University of Würzburg. He received a PhD in computer science from the University of Würzburg. His research interests include run-time performance and resource management, performance model extraction, performance modeling and analysis, virtualization and Cloud Computing.



Ahmed Ali-Eldin Ahmed Ali-Eldin is a postdoc at Umeå university and UMass, Amherst. He obtained his PhD in 2015 from Umeå University, Sweden.His research interests lies in the intersection of Computer Systems and Performance Modeling.

Samuel Kounev is a professor and chair of software engineering at the University of Würzburg. His research is focused on the engineering of dependable and efficient software systems, systems benchmarking and experimental analysis; as well as autonomic and self-aware computing. He received a PhD in computer science from TU Darmstadt. He is a member of ACM, IEEE, and the German Computer Science Society.