Why Is It Not Solved Yet? Challenges for Production-Ready Autoscaling

Martin Straesser
University of Würzburg
Würzburg, Germany
martin.straesser@uni-wuerzburg.de

Johannes Grohmann
University of Würzburg
Würzburg, Germany
johannes.grohmann@uni-wuerzburg.de

Jóakim von Kistowski
DATEV eG
Nürnberg, Germany
joakim.vonkistowski@datev.de

Simon Eismann
University of Würzburg
Würzburg, Germany
simon.eismann@uni-wuerzburg.de

André Bauer
University of Würzburg
Würzburg, Germany
andre.bauer@uni-wuerzburg.de

Samuel Kounev
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

ABSTRACT

Autoscaling is a task of major importance in the cloud computing domain as it directly affects both operating costs and customer experience. Although there has been active research in this area for over ten years now, there is still a significant gap between the proposed methods in the literature and the deployed autoscalers in practice. Hence, many research autoscalers do not find their way into production deployments. This paper describes six core challenges that arise in production systems that are still not solved by most research autoscalers. We illustrate these problems through experiments in a realistic cloud environment with a real-world multi-service business application and show that commonly used autoscalers have various shortcomings. In addition, we analyze the behavior of overloaded services and show that these can be problematic for existing autoscalers. Generally, we analyze that these challenges are only insufficiently addressed in the literature and conclude that future scaling approaches should focus on the needs of production systems.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Software performance**.

KEYWORDS

Autoscaling, cloud computing, microservices

ACM Reference Format:
Martin Straesser, Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. 2022. Why Is It Not Solved Yet? Challenges for Production-Ready Autoscaling. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3489525.3511680

ICPE '22, April 9–13, 2022, Bejing, China

ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511680

1 INTRODUCTION

Autoscaling has been a vital research topic since the beginning of the cloud computing era [6] and has high relevance in several subdomains, such as serverless computing [1] or fog computing [49]. In general, scaling refers to the task of dynamically provisioning computing resources under varying load. Scaling has to be automated in modern cloud environments with highly dynamic and complex workloads [28]. Moreover, scaling has a major impact on the business value of cloud software as it affects both operating costs and customer experience. An optimal autoscaler is able to minimize costs as well as violations of service level objectives (SLO).

Autoscaling solutions in the industry, for example, offered in public clouds, like Google Cloud Engine [20], Microsoft Azure [9], or Amazon Web Services [42], are often relatively simple and rely on user-defined scaling rules (e.g., CPU utilization thresholds). The default autoscaling behavior for Kubernetes, a widely-used container orchestrator, is also based on a simple scaling rule assuming a linear relationship between the supplied resources and the target metric [8]. In a recent survey on autoscaling of web applications [47], over 100 autoscalers proposed in the literature were evaluated. These autoscalers are usually more complex, employing mechanisms based on concepts from queueing theory, fuzzy methods, control theory, reinforcement learning, and more. In general, we observe a big difference between state-of-the-art autoscaling in research papers and production systems. Consequently, the question of why the majority of research autoscalers are not deployed in practice arises.

To address this question, we state six core challenges that autoscalers face in modern production systems covering conceptual, technical, and non-functional requirements. These challenges are illustrated through experiments with a real-world multi-service business application (based on Java and Spring microservices) running different realistic workloads. A cluster with a representative hardware and software technology stack is used for the deployment. We evaluate the performance of different scaling strategies, including reactive, proactive, and hybrid scalers and different scaling metrics. Moreover, we report on the performance behavior of overloaded services and, based on our findings, outline arising problems for autoscalers.

We then analyze how these challenges are addressed in the literature and show that current research autoscalers rely on assumptions

that prevent them from being successfully deployed in practice. Among other things, we see that most research autoscalers limit themselves to either reactive or proactive scaling, which we consider insufficient for workloads with limited predictability. Many autoscalers rely only on platform-level metrics (such as CPU, memory), which are not always suitable to reflect the application state and, hence, may lead to wrong autoscaling decisions. Finally, most research autoscalers rely on various configuration parameters and are typically evaluated in simulation environments or using synthetic workloads only.

The goal of this paper is to serve as a reference for novel autoscaling approaches and how they may find a way into being adopted in industrial settings as well as to motivate further research addressing the challenges in real-life production systems.

Summarizing, the contribution of this paper is twofold:

- We discuss autoscaling challenges in production systems in detail and illustrate these challenges with experiments with different autoscalers in a representative test setting.
- We highlight that these challenges are currently given insufficient attention in the literature and pinpoint common assumptions of research autoscalers that prevent their usage in production systems.

The remainder of this paper is structured as follows: In Section 2, we conduct multiple experiments to evaluate the performance of different scaling strategies in our test cluster. Section 3 discusses our findings and describes challenges for autoscaling in industrial settings. In Section 4, we summarize related work in cloud service autoscaling and derive common assumptions and limitations of the proposed approaches. Finally, we conclude the paper in Section 5.

2 EXPERIMENTAL STUDY

In this section, we describe our experiment setup for investigating different autoscalers. Sections 2.1 to 2.3 describe preliminaries such as the test application and environment, while Sections 2.4 to 2.6 describe our results. Further discussion is presented in Section 3.

2.1 Application Under Test

For the experiments in this paper, we use a test application that comprises a representative subset of some business services of our industry partner in production. An overview of the application, which consists of a `gateway` service, an `eureka` instance, and two services with their own databases, is shown in Figure 1. Every user request is first processed by the `gateway` service, which verifies whether the request is valid. A request is considered valid if special HTTP headers are present and the call refers to a registered URI. The `gateway` service checks if these conditions are met. It then either rewrites some headers and forwards the request to the business services `service1` and `service2` or rejects the request.

Microservice `service1` offers five endpoints overall, from which three generate SELECT or INSERT requests to a connected PostgreSQL database, one generates a request to `service2`, and one retrieves information from a local information cache. In contrast to this, `service2` offers only one endpoint, which causes a SELECT call to another PostgreSQL database. At startup, each service registers itself at the `eureka` service instance. All services are implemented using Java and Spring[1], a widely used framework for backend development.

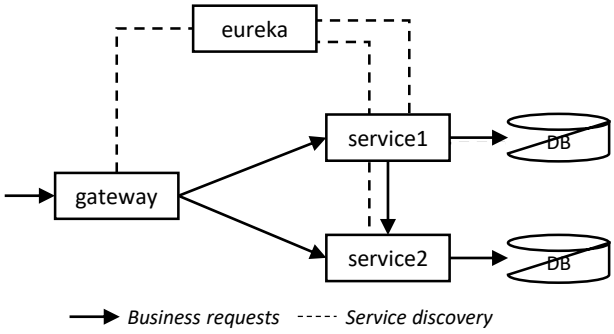


Figure 1: Application Under Test

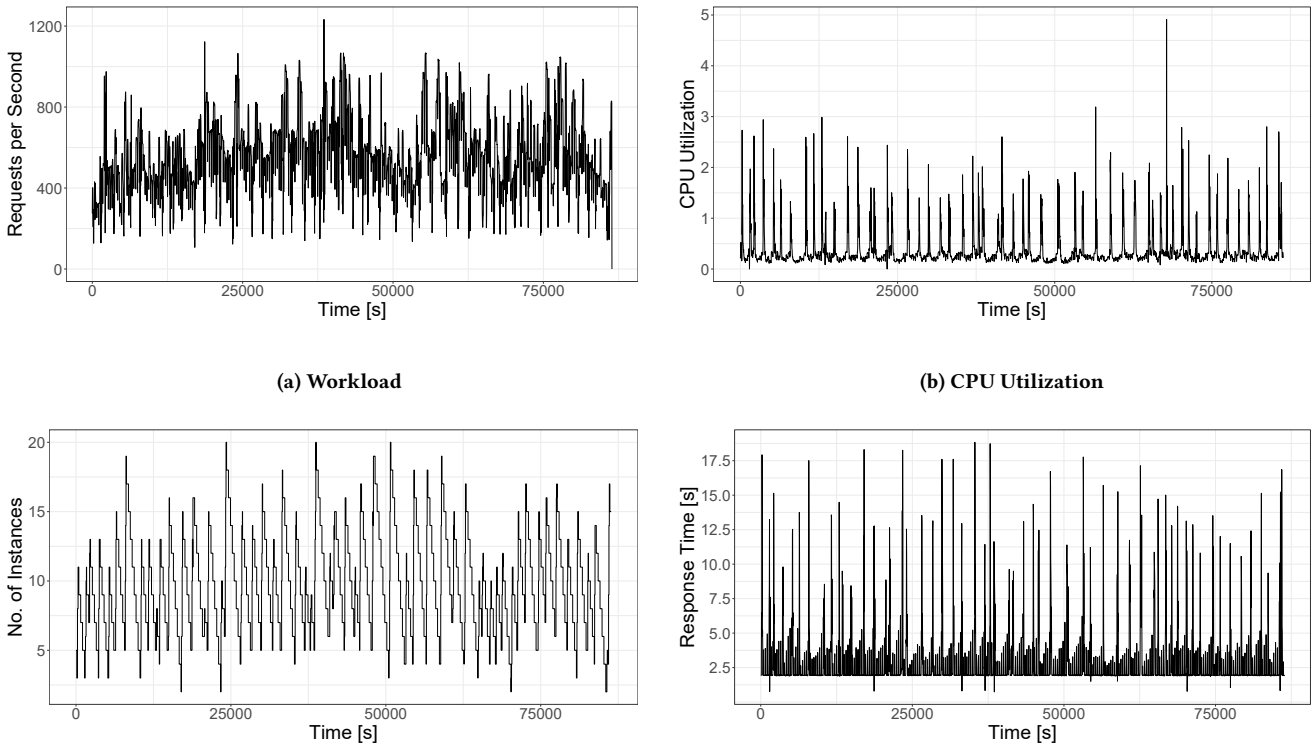
2.2 Technical Setup

The test application is deployed in a `kubecf` [22] cluster. `kubecf` is an open-source distribution of the platform-as-a-service environment Cloud Foundry[2]. The `kubecf` components run on top of a Kubernetes cluster and are deployed using Kubernetes pods. Business applications to be hosted in a `kubecf` cluster are built and deployed in so-called Diego cells, which offer an isolated execution environment. For each application, a memory limit has to be set. The maximum CPU usage of an application is then derived from the memory setting. For each service of our test application, we use a memory limit of 1024MB. We deploy nine Diego cells in total, each with a capacity of 40GB. For more information on `kubecf` and its application resource management refer to the technical documentation [22]. In general, this technology stack mirrors the production setup of our industry partner.

Our cluster consists of one physical master node and six physical worker nodes. Five worker nodes are HPE ProLiant DL360 Gen9 servers with Intel(R) Xeon(R) E5-2650 CPU and 16 GiB DDR4 RAM, and one worker node is an HPE ProLiant DL20 Gen9 server with Intel(R) Xeon(R) E3-1230 CPU and 16 GiB DDR4 RAM. We use a Prometheus v2.27.1 server for monitoring, which scrapes both metrics from the `kubecf` platform (such as the number of currently deployed instances and their CPU, memory, and disk usage) and the application instances once every 30 seconds. The Spring services expose their metrics using Spring Boot Actuator[3].

For load generation, we use three Apache JMeter[4] v5.4.1 instances, which generate the six different request types supported by the test application (see Section 2.1). Request parameters are sampled from a uniform distribution. The implemented autoscalers query metrics from the Prometheus server and send scaling requests to the master node. The autoscalers, Prometheus server, load generators, and PostgreSQL databases run on dedicated servers outside the `kubecf` cluster and are not scaled within the experiments.

[1] https://spring.io/projects/spring-framework
[2] https://www.cloudfoundry.org/
[3] https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html
[4] https://jmeter.apache.org/



(a) Workload **(b) CPU Utilization**

(c) Deployed Instances **(d) Response Time**

Figure 2: Workload and Measurement Results for Reactive CPU Scaling

2.3 Scaling Rules and Quality Measures

In our experiments, we limit ourselves to horizontal scaling, which is a common focus in autoscaling research for microservices [47]. Although we evaluate different scaling metrics, we use one generic scheme for scaling. Independently from the scaling metric, we use the generic and widely used Kubernetes default scaling rule to calculate the instances of service s to be deployed in the next scaling interval [8]:

$$n_{t+1}(s) = \left\lceil n_t(s) \cdot \frac{m_t(s)}{m^*(s)} \right\rceil . \tag{1}$$

Hereby, n_{t+1} is the number of deployed instances in the next interval, while n_t is the number of currently deployed instances. The measured value of the scaling metric averaged over all instances of the service is denoted as m_t and the desired metric value is denoted as m^*. We limit the upscaling to 5 instances per one minute for stability reasons. The downscaling is limited to 2 instances per 5 minutes. These rules represent company policies deduced from the production environment of our industry partner. The scaling mechanism is triggered once every 30 seconds, which means every time new measurement values are available (see Section 2.2).

To rate the quality of the different scaling strategies, we consider both costs and SLO violations. We define T as the set of all measurement intervals in the experiment and S as the set of all services that need to be scaled. For our experiments, we do not include the `eureka` service for scaling. Hence, S consists of `service1`, `service2`, and `gateway`. At the beginning of the experiments, five instances of each service are deployed. The total costs C are defined as the sum of all deployed instances n_t of all services in all measurement intervals:

$$C = \sum_{s \in S} \sum_{t \in T} n_t(s) . \tag{2}$$

For the SLO violation metric V, we consider the ratio of failed requests r_t and total requests R_t for each measurement interval t. A request is considered failed if its response time is above 20 seconds or if an unhealthy response code is returned. We sum this ratio up for all intervals t and then divide the sum by $|T|$, which is the number of measurement intervals in the experiment:

$$V = \frac{1}{|T|} \cdot \sum_{t \in T} \frac{r_t}{R_t} . \tag{3}$$

As stated above, the performance of an autoscaler should take both costs and SLO violations into account. Depending on the use case, they can be weighted differently. This is why we introduce a scaling performance metric P_w, which includes an adjustable weight w for the desired costs and SLO violation ratio:

$$P_w = w \cdot V + (1 - w) \cdot \frac{C}{C_{max}} . \quad (4)$$

Thereby, $1/C_{max}$ is a normalization factor that maps the costs to a scale between 0 and 1. We choose:

$$C_{max} = |S| \cdot |T| \cdot n_{max} ,$$

where n_{max} is the highest number of deployed instances for one service. Hence, C_{max} would be the costs if n_{max} instances of all services would be deployed the whole time. In our experiments, the observed maximum number of instances of one service was 20, and we set n_{max} accordingly. All C, V, and P_w can be considered as lower-is-better metrics.

2.4 Threshold-based CPU Autoscaling

In Experiment 1, we evaluate threshold-based CPU autoscaling, as it can be configured in public cloud environments. We use the Kubernetes default scaling rule (1) to calculate the number of instances. As the desired CPU utilization, we use a value of 80 percent. In the first part, we use a simple reactive scaling strategy, which is later compared to proactive and hybrid strategies. We use a typical workload from the production system of our industry partner and evaluate the scaling behavior over 24 hours. We repeat the measurement three times for each scaling strategy to validate our results.

Figure 2a shows the workload used in this experiment. It varies between 300 and 700 requests per second, with significant outliers in both directions. In our setup, mostly `service1` and `gateway` are bottlenecks, while `service2` shows a consistent performance with few instances. Figure 2b shows the average CPU utilization, while Figure 2c shows the number of deployed instances of `service1` over time measured in one run. We see that the number of instances deployed by the reactive autoscaler rises sharply in times of high load. The upscaling period ends with a short time where the peak number of instances is deployed. After that, a long and consistent downscaling period follows. Figure 2d further shows the average response times in each measurement interval. We see that, congruent to the CPU utilization, the response times rise from their normal value range of 2.5 to 4 seconds to a peak value of about 19 seconds. This congruence shows that CPU utilization can be used to detect overloaded services. The reactive scaler responds only to high CPU utilization, and it has limited capabilities to prevent SLO violations, as the response time rises exponentially in these cases.

We compare a proactive and a hybrid autoscaler to this baseline. The proactive autoscaler uses the time series of the total CPU utilization, that is, the sum of all CPU usages of all instances of one service, and it predicts the value one minute in the future. This forecast horizon is long enough to start new instances, as the average readiness time of the evaluated microservices is about 35 seconds. The forecast value is then divided by our desired metric value of 80 percent and yields the number of instances to be deployed. As a time series forecaster, we use a non-seasonal ARIMA model [36] as it is commonly used in many forecasting scenarios. The hyperparameters p, d, and q are optimized based on a grid search and the time series conducted in the experiments with the reactive autoscaler. The hybrid autoscaler uses both the reactive and proactive approach and deploys the rounded up mean number of instances calculated by both strategies. It is, therefore, a simple combination of both approaches. The problem of weighting reactive and proactive scaling is further discussed in Section 3.

Table 1: Quality Metrics for CPU Autoscaling

Strategy	C	V	$P_{1/2}$	$P_{2/3}$
Reactive	46626	0.205	0.237	0.226
Proactive	41465	0.213	0.226	0.222
Hybrid	44818	0.203	0.231	0.222

Table 1 shows the average costs and SLO violation scores of all three scaling strategies. We see that the proactive autoscaler incurs about 11.1% lower costs compared to the reactive autoscaler, while causing 3.9% more SLO violations. Compared to the reactive approach, the hybrid autoscaler lowers both the costs (-3.9%) and SLO violations (-0.9%). Which scaling strategy performs best depends on the desired costs-to-SLO-violation ratio as shown by the $P_{1/2}$ and $P_{2/3}$ scores. If we weigh both goals equally, the proactive autoscaler would perform better. The hybrid autoscaler is the better choice if we put more weight on reducing SLO violations, which is desired for production-grade customer-oriented business applications.

2.5 Exploring Alternative Scaling Metrics

In Experiment 2, we use application-level metrics for scaling instead of platform-level metrics like CPU utilization. Overall, we collect 73 metrics per service, which can be divided into three groups. The first group is the platform metrics queried from `kubecf`, for example, the CPU, memory, and disk usage. The second group is application metrics exported by the Spring microservices, for example, different JVM and logging metrics. The last group is automatically created metrics from the Prometheus monitoring server, such as the scrape duration, which is the time, Prometheus needs to query application metrics from a service instance. In the following, we pick metrics from all these three categories and evaluate how a simple reactive autoscaler performs with these input values.

A suitable scaling metric should have a preferably simple relationship to the application state and performance; also, it needs to depict overloads as a minimum requirement. From Experiment 1, we know that CPU utilization fulfills this requirement in our case. In search of alternative scaling metrics, we analyzed the results from the previous section and selected those metrics from each category that fulfill the stated requirements best. The first metric selected for further evaluation is the thread ratio θ. We define θ as the ratio of the total number of threads of a service instance divided by the number of running threads. A value of 1 would mean that all created threads are running. The higher this value, the more threads are in a waiting or blocked state in proportion to the number of running threads. This means that requests are potentially queueing and waiting for processing time. For the autoscaler, we use the average value of θ of all instances of one service and use formula (1) to calculate the number of instances. The desired metric value is set to 7 based on the experiment data from Section 2.4.

This metric works only for the business services `service1` and `service2`. The `gateway` service works with a nearly constant number of threads. For this service, we choose the application metric

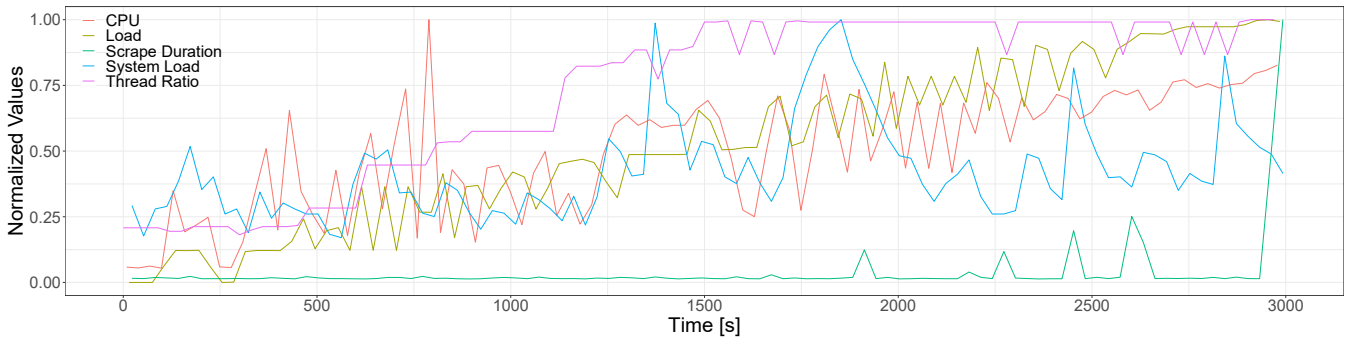


Figure 3: Behavior of Scaling Metrics with Increasing Load

`system_load_average_1m`, which is exposed by Spring Boot Actuator. This metric shows the sum of the number of runnable entities queued to available processors and the number of runnable entities running on the available processors averaged over the last minute [35]. Similar to the thread ratio, this metric depicts some kind of queue length. We use 4.5 as the desired metric value for scaling.

From the group of automatically created metrics, we use the scrape duration exported by the Prometheus server. This metric mainly captures the response time of a service instance to its endpoint for metric exposition. Hence, in contrast to conventional response time measurements, we only use the response time of one request to an endpoint per 30 seconds, which is not relevant for the user. Consequently, this metric is available without any additional overhead and does not require an external tracing engine or similar. It can be rather compared to a health check, whose response time is interpreted as an independent metric. The desired value for scaling is set to three seconds.

To show the relationship of these metrics, we performed a short preliminary test. We deployed a single instance of `service1` and stressed it with an increasing load intensity. The test ended when the first request timed out. Figure 3 shows the temporal courses of the different scaling metrics with increasing load. As all metrics have different value ranges, we normalized the values by the observed maximum. In general, we see that all metrics are sensitive to increasing load while having their own characteristics. The CPU utilization follows the load course best while the system load has higher variations. The thread ratio reaches its maximum value prior to most other metrics, which means that, at this time, the maximum number of total threads and running threads has been reached. Consequently, it could be interpreted as an early warning of potentially evolving overload. In contrast to this, the scrape duration does not change much for different load levels and rises exponentially when the service reaches its maximum throughput.

In the following, we evaluate how reactive autoscalers with these different input metrics perform in our environment. We use the 3-hour workload from the production system of our industry partner shown in Figure 4 for evaluation. This workload consists of a peak, followed by a phase of a rather low workload (lunch break) and an increasing load at the end. We compare three different scaling strategies. The first autoscaler is the same CPU-based autoscaler used in Experiment 1. The second autoscaler, the application-specific autoscaler, uses the thread ratio for scaling `service1` and `service2` and the system load for scaling the `gateway` service. In rare cases when application metrics are not reported for one minute or longer, the application-specific autoscaler temporarily falls back to CPU utilization for scaling (see Section 2.6). The third autoscaler uses the scrape duration as the scaling metric. We performed three repetitions per scaling strategy similar to the previous experiment to validate our results.

Table 2 shows the average costs and SLO violation metrics for all scaling strategies. We see that the application-specific autoscaler caused the fewest SLO violations but also the highest costs. The CPU-based autoscaler has lower costs (-27%) but a higher number of SLO violations (+15.1%) compared to the application-specific strategy. The autoscaler with the scrape duration as its scaling metric performs best in this case as it has the lowest costs and only a few more SLO violations compared to the application-specific autoscaler. This is also underlined by the $P_{1/2}$ and $P_{2/3}$ scores. In general, we see that all strategies have their pros and cons and the unconventional strategies achieve comparable results to the CPU-based autoscaler.

2.6 Issues Found With Overloaded Services

In this section, we focus on four phenomena that we observed during the experiments from the previous sections and discuss how they affect autoscalers.

Scaling metrics might be delayed, invalid, or unavailable. In Experiment 2, we use the scrape duration as a scaling metric. In general, varying scrape duration is rather a problem than an opportunity. In our experiments, the scrape duration varies between 0.2 and 10 seconds. The highest value is thereby a third of the monitoring interval. Such delays can be problematic for autoscalers for various reasons. First, these effects mainly appear when a service is overloaded, that is, exactly when a scaling action is necessary. The delays can increase the reaction time of reactive autoscalers. Nevertheless, also for proactive autoscalers, delayed metrics can be problematic, especially in cases when time series forecasting is used, as many time series forecasters rely on or work best with equidistant time

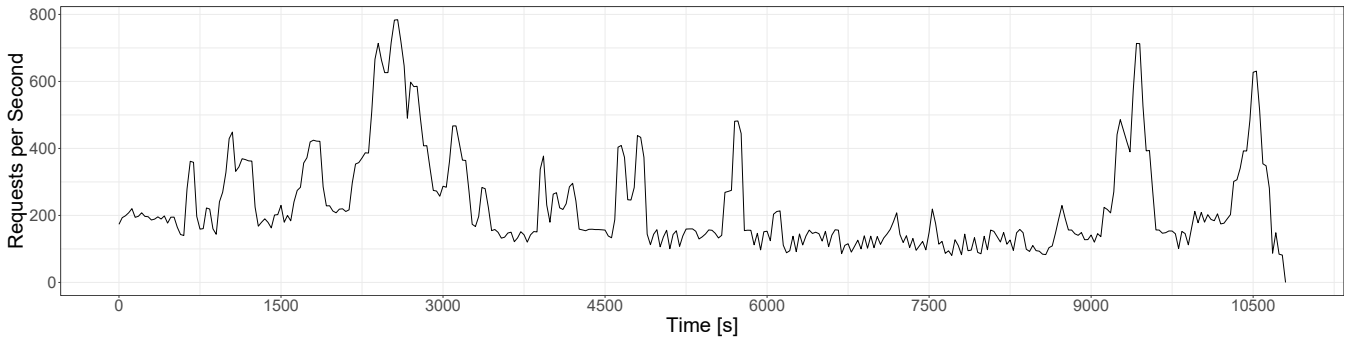


Figure 4: Workload for Experiment 2

series [13]. In addition to delayed values, we observed that scaling metrics can be unavailable or invalid, for example, when the CPU utilization is zero. These failures appear more often in our experiments with application metrics than with platform metrics. This is why our application-specific autoscaler from Experiment 2 temporally falls back to platform metrics for scaling if application metrics are not reported for one minute or longer.

Response time measurements can be misleading. As a special case for misleading measurements, we analyze response time measurements because they play an essential role in many state-of-the-art autoscalers (see Section 4). In our experiments, we observe that the response time cannot always be used to characterize the application state. In Figure 2d, we see that the response time rises exponentially when service instances are overloaded and the phases of slightly increasing response times are short. Moreover, we notice a clear lower bound near 2.5 seconds, which equals the minimum response time of business applications in our setting. This time is independent of the provisioned resources and hence cannot be lowered further by supplying more resources. However, a few measurements report response times below this boundary. This is the case when `service1` is overloaded and returns an unhealthy response code. This shows that low response times can be misleading and might be misinterpreted by autoscalers. We conclude that the response time cannot be used to quantify the application state without limitations as it has lower and upper boundaries.

Dependencies between services affect scaling metrics. As shown in Figure 1, our test application consists of several services that depend on each other. We especially see that every user request has to pass the `gateway` service first. As a result, the informative value of scaling metrics varies. We observe that whenever the `gateway` service is overloaded, the processing time of a request increases. This results in the fact that the arrival rate and consequently also the resource demand of `service1` and `service2` is lowered. Especially the simple reactive CPU autoscaler sometimes performs a downscaling action on these cases. This evolves as problematic when the `gateway` service returns to normal operation and processing time, as the number of forwarded requests to `service1` and `service2` increases significantly in a short time interval. This leads to the fact that the bottleneck moves from the frontend to the backend. Similar observations for the response time have been made in the literature [27]. These dependencies and the potentially limited informative value of scaling metrics are problematic for many autoscalers as services are mostly treated as independent entities.

Table 2: Quality Measures for Different Scaling Metrics

Strategy	C	V	$P_{1/2}$	$P_{2/3}$
CPU-Based	3833	0.206	0.236	0.226
App-Specific	5250	0.179	0.272	0.241
Scrape Duration	3822	0.187	0.226	0.213

Health monitoring causes restarts of overloaded services. Especially in environments with orchestration frameworks, like Kubernetes clusters or our test environment based on `kubecf`, a health monitoring unit is used. For many microservices, it is common to check the application's health by sending HTTP requests to dedicated endpoints. If these requests fail or the response time is too high, the respective service instance might be restarted. This phenomenon occurs mainly for overloaded services. In our experiments, these restarts occur up to 486 times in one repetition of Experiment 1 and up to 15 times in one repetition of Experiment 2. Most autoscalers are not aware of such restarts, although they influence the application performance significantly. During the restart, fewer instances are processing the workload than assumed by the autoscaler. In addition, the performance of recently started instances differs from the performance of instances that are longer in operation. These and more effects are discussed further in the next section.

3 DISCUSSION

We divide our discussion into two parts. Section 3.1 discusses challenges for production-ready autoscalers, while Section 3.2 touches upon the limitations of this study.

3.1 Challenges for Production-Ready Autoscalers

This section summarizes our findings from the conducted experiments and states six challenges that are likely to arise in production systems and, therefore, should be addressed by production-ready autoscalers.

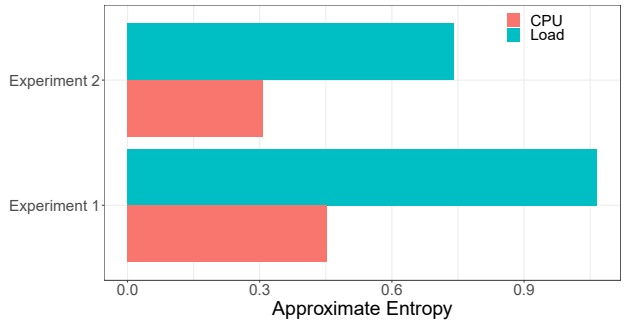


Figure 5: Entropy Values for CPU and Load Time Series

Challenge 1: Balancing proactive and reactive scaling. In an optimal setting, we would always provide resources proactively, which means that we would know the workload and resource demand in advance and know how to match this demand in a cost-efficient manner. In many customer-oriented business applications, the workload and its associated resource demands contain seasonal patterns (e.g., daily, weekly, or monthly cycles). For seasonal time series, a bunch of forecasting approaches is available [34]. However, this is only one side of the medal. Real-world workloads are often more complex, contain bursts, batch jobs, or anomalies, and differ from one service to another. Hence, workloads cannot always be reduced to strict seasonality and trend components.

Another problem for predictive scaling approaches is choosing the right metric to forecast. In research and industry, both CPU forecasting [33, 37, 42] and workload forecasting [5, 11, 50] is used. For further investigation, we analyze the load intensity and total CPU utilization time series from the experiments in Section 2 and calculate their approximate entropy (ApEn) [38]. ApEn is an algorithm for determining the regularity of a time series based on the existence of patterns [18]. As a measure of entropy, ApEn quantifies the information content of a time series [43]. The lower the result value, the less information is captured by the time series. We calculate ApEn[5] for the time series measured in the experiments from Sections 2.4 and 2.5 and report the average values for both the load and CPU utilization time series in Figure 5. We see that the CPU utilization has a lower information content compared to the number of arriving requests. Concerning the numeric entropy values and the visual impression from Figure 2b, we see that pure forecasting of CPU utilization is not sufficient to predict the future resource demand accurately, and workload forecasting may model usage patterns better.

In general, we see that forecasting realistic workloads is challenging, and it can only achieve limited accuracy in the presence of unexpected anomalies. Consequently, a reactive scaling component, able to act in case of unexpected SLO violations, should be part of every production-ready autoscaler. This necessitates hybrid scaling, which also showed promising results in Experiment 1. However, when having both a reactive and proactive component in operation, conflict situations occur where the outputs of both components are different and have to be aggregated into a single decision [5]. This raises several other research questions like how to aggregate proactive and reactive scaling decisions, especially in conflict situations.

Challenge 2: Combining application-specific and generic platform metrics. Modern cloud services expose many metrics that describe the application state and health. In our setup with Spring Boot Actuator, we retrieve 62 different application metrics from every service. Not all of them are indeed meaningful for scaling tasks. However, Experiment 2 showed that simple reactive scalers based on unconventional metrics like the scrape duration can achieve competitive results compared to traditional CPU-based scaling. In general, the question of the best metric for scaling is congruent to the question of what metric correlates most with the KPIs and has to be stated for nearly every single service. While CPU, memory, and disk usage are easily interpretable and mostly available metrics, custom measures could be advantageous in cases when the performance profile of a service is not clearly CPU- or memory-dominated. Moreover, hardware metrics are limited by design, and the resource demand of a service cannot be derived in all cases [10]. Overall, it is not trivial for practitioners to determine the best scaling metrics for individual services. Furthermore, when considering both application and platform metrics for scaling, researchers and experts in the industry face the conflict of developing autoscalers that are generic and general-purpose, on the one hand, but perform specifically tailored to their cloud application on the other hand. This also has implications for the configurability (Challenge 3) and explainability (Challenge 6) of autoscalers.

Challenge 3: Keeping configuration overhead as small as possible. One of the major points of criticism for many approaches in the autoscaling domain is the aspect of configurability, for example, determining suitable scaling thresholds, cooldown times, model parameters, or similar. In the experiments in this paper, we used fixed CPU utilization thresholds of 80 percent. The thresholds for the custom metrics have been chosen based on the results of Experiment 1. As stated earlier, the configuration of an autoscaler is hard, and the complexity is increased further when the configuration differs from one service to another. Often, many configuration values depend on the desired costs-to-SLO-violations ratio, that is, how conservative the autoscaler should act. However, other factors influence the configuration, such as expected application start and shutdown times or the anticipated kind of workload. Although the DevOps principle stands for stronger coupling of the development and the operation of cloud software, application developers are often not concerned with autoscaling or performance of the application [15]. Furthermore, autoscaling might be outsourced to the cloud provider. In these cases, the person responsible for autoscaling has only limited insight into the functionalities and performance characteristics of the services they should scale. The problem is aggravated by frequent code updates. This might lead to changed performance properties and the need to adapt models or thresholds. In such situations, it is not feasible to rely on manual reconfiguration. Therefore, production-ready autoscalers should minimize configuration overhead and rely on self-optimization instead.

[5]The algorithm needs input values for the parameters m (template length) and r (noise filtering). For our calculations, we choose $m = 2$ and $r = 0.25\sigma$, where σ is the standard deviation of the time series, according to the recommendations in the literature [18].

Challenge 4: Scaling metrics might be unavailable, incomplete, inaccurate, or delayed in production systems. Trusting the input metrics is crucial for any autoscaler. A lot of advances have been achieved in the field of continuously observing cloud systems. However, especially during high loads, several effects can influence the validity of measurements in cloud environments. Application metrics are particularly prone to delay or failed measurements, as they have to be queried directly from the application instances. In our experiments, we saw that the scrape duration during high loads increased by more than nine seconds, and some instances do not even report valid values when overloaded. Platform metrics are less prone to failures; however, especially in large clusters, measurements could be erroneous due to the fact that metrics have to be collected from many nodes distributed all over the cluster.

Response time measurements play a special role in the design and operation of many state-of-the-art autoscalers (see Section 4). However, the reliable acquisition of response times, especially in high load scenarios, is non-trivial. These rely either on measurements by the application itself or have to be sampled using external tracing frameworks. Moreover, by design, the response time always reflects a past state of the application, as it is measured shortly before or after the request left the system. Therefore, a delay of, for example, 10 seconds in response time measurement is only observable after these 10 seconds have passed. Moreover, one has to consider that the response time should not be the only criteria to rate the application's health and quality. In the presence of some errors, the response time might be even lowered, for example, when a service responds faster because of an internal error as discussed in Section 2.6.

Another critical problem when interpreting scaling metrics is the dependency between different cloud services. A typical property of microservice architectures is that multiple services are involved in processing a single user request. Consequently, errors and high response times might propagate across several services, although they have enough resources assigned to them [27]. In our experiments, we saw similar effects. Whenever the `gateway` service was overloaded, not all requests have been forwarded to the backend services `service1` and `service2`. This resulted in a temporary reduction of the CPU utilization, and the simple reactive CPU autoscaler reduced the number of instances of the backend services. The backend services then experienced degradations when the `gateway` pursued working. All in all, we see that several factors can influence the reliability of scaling metrics.

Challenge 5: Combining autoscaling, load balancing, and resilience mechanisms. As discussed in Section 2.6, the autoscaler is not the only mechanism that controls service instances in modern cloud environments. We showed that the health monitoring unit might restart overloaded services as they fail to send heartbeats. An autoscaler can profit from knowledge about such restarts, as they influence the application performance. First, the number of available instances is reduced during the restart, and some requests might be dropped. Second, after the restart, the performance of the newly deployed instance often differs from those instances which are running for a longer time. However, the detection of such restarts remains a technical challenge. In addition to this, two conceptual challenges arise. First, in production systems, there are technical or implicit SLOs present, which are enforced by third parties and the autoscaler might not be aware of, like the mentioned restarts caused by failed heartbeats. Second, there is a strong interdependence of autoscaling, load balancing, and resilience mechanisms in general, for example, health monitoring from a platform view or circuit breakers from an application view. Load balancing is the task of how much load to forward to a specific instance of a service. Autoscaling is the task of determining how much resources are needed to process a given load. Resilience mechanisms and health monitoring are tasks to ensure that the application itself works fine and possibly end malfunctions. These three cloud management activities have to work together to fulfill their goal to keep the application quality as high as possible. Consequently, autoscalers need to interoperate with appropriate load balancing and health management mechanisms, and vice versa.

Challenge 6: Keeping autoscalers explainable. As stated earlier, scaling plays an outstanding role in the operation of cloud services. It directly influences both customer experience and operating costs. Obviously, in modern complex cloud environments, scaling tasks have to be handled automatically. Moreover, many years of research have shown that an autoscaler that addresses all of the previously discussed challenges needs a considerable amount of complexity. Nevertheless, an additional requirement for autoscalers in production systems is the transparency of its decisions. This is not only useful for debugging purposes but also necessary to increase trust and potentially propose further enhancements. This explainability does not mean that all calculations must be comprehensible for everyone; it is rather the requirement for an autoscaler to state reasons for its actions. Potentially, log entries like *scaling up because metric x has/will have value y which is considered too high* may provide real benefits. However, this presents a challenge, especially for many black-box or machine-learning-based approaches. We argue that explainability, together with deployability and configurability (see Challenge 3), are key requirements for production-ready autoscalers in order to increase the trust in the resulting decisions.

3.2 Study Limitations

In this section, we discuss the limitations of our experimental study. We limited ourselves to horizontal scaling problems and assumed the usage of homogeneous instances; that is, all replicas of one service have the same resource limitations. This is a common primary focus for autoscaling in the microservice domain [47], and we argue that most of our observations can be transferred to vertical scaling as well. Our results have been produced in one specific technology stack consisting of the hardware described in Section 2.2, `kubecf` as a PaaS software, and Java and Spring as implementation technologies for the evaluated microservices. Consequently, our results cannot be generally transferred to other environments. However, we claim that our setup is still representative as it mirrors the production setup of our industry partner and consists of several isolated machines and state-of-the-art cloud software. Moreover, we validated the measured results by performing multiple repetitions.

Considering the scaling logic, we worked with the generic formula (1), which is the default relationship used for scaling in Kubernetes. This scaling rule assumes a linear relationship between the scaling metric and the number of instances, which is only an approximation in most cases. We generally keep the scaling logic

constant and straightforward for different experiments and scaling metrics. The used thresholds are set arbitrarily and have not been optimized further. We used ARIMA as one representative state-of-the-art general purpose time series forecaster for evaluation in the proactive scaling domain. Consequently, we analyzed only a small subset of possible scaling approaches. We argue that our results are still meaningful for mainly two reasons. First, the scaling logic in production-grade autoscalers is also kept simple. Our proactive scaler is designed similar to the AWS EC2 predictive autoscaler [42], which combines metric forecasting and threshold-based scaling as well. Second, many of the problems stated in Section 3.1, such as invalid measurements, appear in production systems independently from the used scaling logic.

4 RELATED WORK

This section summarizes related works from the autoscaling domain and analyzes how the stated challenges are addressed. We identify shortcomings that limit the applicability of research autoscalers in practice. This section is structured according to the challenges proposed in Section 3.1. A summary is given at the end of this section.

Challenge 1. Singh et al. [47] summarize research approaches for autoscaling of web applications in cloud environments. Less than 15%, in total 15 out of 104, of the analyzed papers in that survey combine reactive and proactive scaling. As stated earlier, we argue that hybrid scaling should be used in production systems to combine predictive power and stabilizing actions in case of unseen load spikes. Ali-Eldin et al. [5] analyze different combinations of reactive and proactive scalers and conclude that reactive scalers should be involved in upscaling decisions while downscaling should be initiated by proactive components only. This principle has also been adapted by other approaches [29, 31]. These approaches require a proactive scaler that is not too conservative, meaning it should regularly trigger downscaling actions. Otherwise, the goal of cost-efficiency is not reached. The decision of whether to use proactive or reactive scaling logic is often solved by using user-defined thresholds [29, 48, 50] or other user-specified parameters [11]. Most hybrid scalers rely on the reactive component only in case of SLO violations [4]. Bauer et al. [12] use so-called trust thresholds that take the accuracy of the predictive model into account and possibly omit reactive decisions to resolve scaling conflicts. Hence, we state that many state-of-the-art hybrid autoscalers often rely on additional configuration parameters to balance reactive and proactive scaling. This stands in conflict with the goal of keeping configuration overheads as small as possible.

Challenge 2. As stated earlier, platform metrics are not always the best scaling metrics available. Many papers use platform metrics such as CPU utilization and memory metrics as their only input for scaling [2, 26, 45, 49]; some are even purely CPU-focused [25, 33, 46]. The number of incoming requests is the most used application-level metric used by more than half of the approaches analyzed by Singh et al. [47]. Other custom scaling metrics used in literature are limited to the number of active connections [16] or the number of active sessions [17]. Based on our results from Section 2.5, we argue that there is much potential in scaling based on other application metrics and on the combination of platform and application metrics for autoscaling. Many papers in the state-of-the-art literature only combine the arrival rate and platform metrics. As a possible future research direction, middleware metrics, such as JVM measures, could come into focus. They are currently more established in other areas of performance engineering, e.g., software performance optimizations [41].

Challenge 3. The aspect of configurability is rarely addressed in the autoscaling domain. Kalyvianaki et al. [33] provide a resource provisioning scheme based on Kalman filters while explicitly claiming low configuration overhead. The type and meaning of configuration parameters needed by various approaches are manifold. Threshold- or rule-based autoscalers require many critical manual settings that influence the performance of the autoscaler massively [3]. Most autoscalers require at least up- and downscaling thresholds. Some require other inputs like manually created models [12] or costs of reconfiguration [39]. As stated above and by Jiang et al. [30] especially hybrid scalers often require manual parameter or offline tuning. Many of these settings cannot be determined in advance by application developers or require extensive load testing to be set appropriately. An additional challenge is keeping configurations up to date especially with respect to frequent application updates, which are more probable in DevOps contexts. Reinforcement learning as used by various recent approaches [32, 40, 51] offers one way to reduce configuration overhead. However, it comes with the difficulties of defining a suitable reward function and needing lots of training data. Moreover, most reinforcement learners assume a static application and have problems with changes introduced by updates [21].

Challenge 4. To the best of our knowledge, there is currently no autoscaling approach explicitly concerning the problem of inaccurate or delayed metrics. A major point of criticism for research autoscalers is that many of them are evaluated in simulation environments only. According to the survey by Singh et al. [47], 30 autoscaling approaches are evaluated using simulation only, and an even greater subset is using synthetic workloads. In general, this leaves the question unanswered, how well these autoscalers perform in production environments and leads to the fact that delayed, inaccurate, or incomplete measurements cannot be considered.

Many autoscaling approaches use response time measurements for an internal model evaluation or, in cases when reinforcement learning is used, to calculate the reward. For example, Aslanpour et al. [7] require low and high response time thresholds for down- and upscaling, respectively. We argue that there are two problems when relying heavily on response time measurements. First, as described in Section 3.1, response time measurements can be erroneous or delayed in critical scenarios. As a special case, if errors occur in the application, the response time alone might not be suited to characterize an overloaded service. In these cases, other metrics might depict the application state better. Second, the response time has by design lower and upper bounds. The lower bound is given by the minimal execution time of business requests, which cannot be further reduced by provisioning more resources for the respective service (cf. Figure 2d). The upper bound in interactive applications is given by request timeouts. This is why the response time cannot be used to derive scaling decisions without limitations.

In Section 3.1, we further discussed that inter-service dependencies might be influencing factors for scaling metrics. Most research

autoscaler do not consider dependencies to other services. Some approaches explicitly target multi-tier web applications [14, 50, 52, 53]. For example, Sharma et al. [44] investigate multi-tier applications and estimate response times with a multi-stage queueing network. However, most of these approaches assume static applications, and their performance concerning modern microservice applications has not been evaluated.

Challenge 5. Only a few existing approaches target the dependency between autoscaling and other orchestration tasks, like load balancing and health monitoring, explicitly. Chen et al. [16] propose an autoscaling mechanism and also evaluate load dispatching algorithms in parallel. Dezhabad and Sharifan [19] connect a scaling and a load balancing unit for the provisioning of firewall applications. Gandhi et al. [23] connect the question of server provisioning and traffic routing in a multi-tier data center. These studies indicate that load balancing and other orchestration mechanisms have to work together to achieve a good application service quality.

Challenge 6. Concerning the aspect of explainability in the autoscaling domain, Ghanbari et al. [24] state that model-based autoscaling, e.g., queueing network, are hard to understand, while rule-based scaling is, in general, better in terms of explainability. However, the authors state that, also for rule-based scaling, complexity can be high when a large set of rules is used. Many research autoscaler achieve some kind of explainability by reducing the complexity of the scaling problem through the use of many configuration parameters. We argue that explainability gets a concern, especially for autoscalers that fulfill the goal of low configuration overheads, as well as for machine-learning-based or black-box autoscalers.

In summary, we see that no approach tackles all of the stated challenges, and there are some common assumptions made by research papers that might be violated in production systems. For example, most research autoscalers rely on continuous, error-free, in-time monitoring data or are evaluated in simulation environments only. This stands in conflict with Challenge 4 and our observations from Section 2.6. Moreover, many autoscalers assume that CPU or platform metrics are the best scaling metrics. As shown in Section 2.5 and captured in Challenge 2, application metrics can also be used for scaling and can achieve at least comparable results. Another limitation of current approaches is that many configuration parameters are required, which influence the scaling behavior significantly. This stands in conflict with Challenge 3, as these parameters are hard to determine by practitioners and might be service-specific. Finally, autoscalers are often evaluated as standalone mechanisms, i.e., not as a part of an orchestration or cloud management framework. As discussed in Challenge 5, the interference of autoscaling, load balancing, and resilience mechanisms might be non-neglectable in production environments.

5 CONCLUSION

Although autoscaling is an established area of research in the performance engineering and cloud computing community, many research autoscalers do not find their way into production deployments. This paper states six core challenges for autoscalers in production systems and performs experiments in a realistic setting to illustrate these challenges. We have seen that hybrid autoscalers should be preferred over purely reactive or proactive scalers, especially when the workload is irregular and complex. We show that autoscalers have not to be focused on CPU or platform metrics only, as application metrics exposed by modern microservices can also be advantageous for scaling. However, custom metrics always introduce a configuration cost, which must not be underestimated for practical applicability. Moreover, we analyze effects in connection with overloaded services, which show that scaling metrics might be delayed or inaccurate in production systems and motivate a stronger interaction between autoscaling, load balancing, and resilience mechanisms.

We analyze state-of-the-art research autoscalers and summarize how they address these challenges. We pinpoint common assumptions that are not always given in production environments. We came up with the fact that most autoscalers focus on either reactive or proactive scaling. Moreover, many research autoscalers rely on various configuration parameters, which heavily influence their performance and scaling behavior. We argue that these parameters are hard to choose or must be tuned offline. This is especially difficult in DevOps contexts with frequently updated applications, and we conclude that lower configuration overhead should be one focus of future autoscaling approaches. Another weakness is that many approaches focus much on algorithmic details and are evaluated with synthetic workloads or in simulation environments only, making the applicability in practice questionable.

All in all, we deduce that the properties of production environments should be considered more in future autoscaling papers to increase the success of research autoscalers in practice. Close cooperation between industry and research is needed in this domain. For example, more real-world workload traces would help researchers to conduct realistic evaluations of their approaches. We conclude that production-ready autoscaling is not solved yet, and both researchers and practitioners should combine their individual strengths to face this problem.

REFERENCES

[1] Cristina Abad, Ian T. Foster, Nikolas Herbst, and Alexandru Iosup. 2021. Serverless Computing (Dagstuhl Seminar 21201). In *Dagstuhl Reports*, Vol. 11. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[2] Auday Al-Dulaimy, Javid Taheri, Andreas Kassler, M. Reza Hoseiny Farahabady, Shuiguang Deng, and Albert Zomaya. 2020. MULTISCALER: A Multi-Loop Auto-Scaling Approach for Cloud-Based Applications. *IEEE Transactions on Cloud Computing* (2020).

[3] Fahd Al-Haidari, Mohammed H. Sqalli, and Khaled Salah. 2013. Impact of CPU Utilization Thresholds and Scaling Size on Autoscaling Cloud Resources. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 2. 256–261.

[4] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. 2012. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd workshop on Scientific Cloud Computing*. 31–40.

[5] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. 2012. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*. 204–212.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. 2009. *Above the clouds: A berkeley view of cloud computing.* Technical Report. Technical Report UCB/EECS-2009-28, EECS Department, University of California.

[7] Mohammad Sadegh Aslanpour, Mostafa Ghobaei-Arani, and Adel Nadjaran Toosi. 2017. Auto-scaling web applications in clouds: A cost-aware approach. *Journal of Network and Computer Applications* 95 (2017), 26–41.

[8] The Kubernetes Authors. 2021. *Horizontal Pod Autoscaler.* https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[9] Microsoft Azure. 2021. *How To Scale Cloud Services.* https://docs.microsoft.com/de-de/azure/cloud-services/cloud-services-how-to-scale-portal.

[10] André Bauer, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. 2018. On the Value of Service Demand Estimation for Auto-Scaling. In *Proceedings of 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB 2018) (Lecture Notes in Computer Science, Vol. 10740).* Springer, Cham, 142–156.

[11] André Bauer, Nikolas Herbst, Simon Spinner, Ahmed Ali-Eldin, and Samuel Kounev. 2019. Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2019), 800–813.

[12] André Bauer, Veronika Lesch, Laurens Versluis, Alexey Ilyushkin, Nikolas Herbst, and Samuel Kounev. 2019. Chamulteon: Coordinated Auto-Scaling of Micro-Services. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS).* 2015–2025.

[13] André Bauer, Marwin Züfle, Nikolas Herbst, Samuel Kounev, and Valentin Curtef. 2020. Telescope: An automatic feature extraction and transformation approach for time series forecasting on a level-playing field. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).* IEEE, 1902–1905.

[14] Marta Beltrán. 2015. Automatic provisioning of multi-tier applications in cloud computing environments. *The Journal of Supercomputing* 71, 6 (2015), 2221–2250.

[15] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Monica Villavicencio, Jürgen Walter, and Felix Willnecker. 2019. How is Performance Addressed in DevOps?. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India) *(ICPE '19).* Association for Computing Machinery, New York, NY, USA, 45–50.

[16] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. 2008. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI'08).* USENIX Association, USA, 337–350.

[17] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, and Alla Segal. 2009. Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. In *2009 IEEE International Conference on e-Business Engineering.* 281–286.

[18] Alfonso Delgado-Bonal and Alexander Marshak. 2019. Approximate Entropy and Sample Entropy: A Comprehensive Tutorial. *Entropy* 21, 6 (2019).

[19] Naghmeh Dezhabad and Saeed Sharifian. 2018. Learning-based dynamic scalable load-balanced firewall as a service in network function-virtualized cloud computing environments. *The Journal of Supercomputing* 74, 7 (2018), 3329–3358.

[20] Google Cloud Docs. 2021. *Autosclaing groups of instances.* https://cloud.google.com/compute/docs/autoscaler.

[21] Xavier Dutreilh, Aurélien Moreau, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. 2010. From Data Center Resource Allocation to Control Theory and Back. In *2010 IEEE 3rd International Conference on Cloud Computing.* 410–417.

[22] CloudFoundry Foundation. 2021. *KubeCF: A Kubernetes Native Distribution of Cloud Foundry.* https://kubecf.io/.

[23] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4, Article 14 (Nov. 2012).

[24] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. 2011. Exploring alternative approaches to implement an elasticity policy. In *2011 IEEE 4th International Conference on Cloud Computing.* IEEE, 716–723.

[25] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. 2010. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *2010 International Conference on Network and Service Management.* 9–16.

[26] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. 2019. Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19).* Association for Computing Machinery, New York, NY, USA, 149–162.

[27] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. 2021. SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Virtual Event, France) *(ICPE '21).* Association for Computing Machinery, New York, NY, USA, 165–176.

[28] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *10th International Conference on Autonomic Computing (ICAC 13).* USENIX Association, San Jose, CA, 23–27.

[29] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. 2011. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems* 27, 6 (2011), 871–879.

[30] Jing Jiang, Jie Lu, Guangquan Zhang, and Guodong Long. 2013. Optimal cloud resource auto-scaling for web applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing.* IEEE, 58–65.

[31] Bibal Benifa J.V. and Dejey Dharma. 2018. HAS: Hybrid auto-scaler for resource scaling in cloud environment. *J. Parallel and Distrib. Comput.* 120 (2018), 1–15.

[32] Bibal Benifa J.V. and Dejey Dharma. 2019. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications* 24, 4 (2019), 1348–1363.

[33] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. 2009. Self-Adaptive and Self-Configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters. In *Proceedings of the 6th International Conference on Autonomic Computing* (Barcelona, Spain) *(ICAC '09).* Association for Computing Machinery, New York, NY, USA, 117–126.

[34] G. Mahalakshmi, S. Sridevi, and S. Rajaram. 2016. A survey on forecasting of time series data. In *2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE'16).* 1–8.

[35] Micrometer Metrics. 2021. *Micrometer GitHub Repository and Documentation.* https://github.com/micrometer-metrics/micrometer.

[36] Paul Newbold. 1983. ARIMA model building and the time series analysis approach to forecasting. *Journal of forecasting* 2, 1 (1983), 23–35.

[37] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *10th International Conference on Autonomic Computing (ICAC 13).* USENIX Association, San Jose, CA, 69–82.

[38] Steven M Pincus, Igor M Gladstone, and Richard A Ehrenkranz. 1991. A regularity statistic for medical data analysis. *Journal of clinical monitoring* 7, 4 (1991), 335–345.

[39] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *2011 IEEE 4th International Conference on Cloud Computing.* 500–507.

[40] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20).* Association for Computing Machinery, New York, NY, USA, Article 16.

[41] Semih Sahin, Wenqi Cao, Qi Zhang, and Ling Liu. 2016. JVM Configuration Management and Its Performance Impact for Big Data Applications. In *2016 IEEE International Congress on Big Data (BigData Congress).* 410–417.

[42] Amazon Web Services. 2021. *Predictive Scaling for EC2.* https://aws.amazon.com/en/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/.

[43] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[44] Upendra Sharma, Prashant Shenoy, and Donald F. Towsley. 2012. Provisioning Multi-Tier Cloud Applications Using Statistical Bounds on Sojourn Time. In *Proceedings of the 9th International Conference on Autonomic Computing* (San Jose, California, USA) *(ICAC '12).* Association for Computing Machinery, New York, NY, USA, 43–52.

[45] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11).* Association for Computing Machinery, New York, NY, USA, Article 5.

[46] Bradley Simmons, Hamoun Ghanbari, Marin Litoiu, and Gabriel Iszlai. 2011. Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree. In *2011 7th International Conference on Network and Service Management.* 1–5.

[47] Parminder Singh, Pooja Gupta, Kiran Jyoti, and Anand Nayyar. 2019. Research on auto-scaling of web applications in cloud: survey, trends and future directions. *Scalable Computing: Practice and Experience* 20, 2 (2019), 399–432.

[48] Parminder Singh, Avinash Kaur, Pooja Gupta, Sukhpal Singh Gill, and Kiran Jyoti. 2021. RHAS: robust hybrid auto-scaling for web applications in cloud computing. *Cluster Computing* 24, 2 (2021), 717–737.

[49] Fan-Hsun Tseng, Ming-Shiun Tsai, Chia-Wei Tseng, Yao-Tsung Yang, Chien-Chang Liu, and Li-Der Chou. 2018. A Lightweight Autoscaling Mechanism for Fog Computing in Industrial Applications. *IEEE Transactions on Industrial Informatics* 14, 10 (2018), 4529–4537.

[50] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. 2008. Agile Dynamic Provisioning of Multi-Tier Internet Applications. *ACM Trans. Auton. Adapt. Syst.* 3, 1, Article 1 (March 2008).

[51] Yi Wei, Daniel Kudenko, Shijun Liu, Li Pan, Lei Wu, and Xiangxu Meng. 2019. A reinforcement learning based auto-scaling approach for SaaS providers in dynamic cloud environment. *Mathematical Problems in Engineering* 2019 (2019).

[52] Song Wu, Binji Li, Xinhou Wang, and Hai Jin. 2016. HybridScaler: Handling Bursting Workload for Multi-tier Web Applications in Cloud. In *2016 15th International Symposium on Parallel and Distributed Computing (ISPDC).* 141–148.

[53] Pengcheng Xiong, Zhikui Wang, Simon Malkowski, Qingyang Wang, Deepal Jayasinghe, and Calton Pu. 2011. Economical and Robust Provisioning of N-Tier Cloud Workloads: A Multi-level Control Approach. In *2011 31st International Conference on Distributed Computing Systems.* 571–580.