Autoscaler Evaluation and Configuration: A Practitioner's Guideline (Author Preprint)

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

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

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

Autoscalers are indispensable parts of modern cloud deployments and determine the service quality and cost of a cloud application in dynamic workloads. The configuration of an autoscaler strongly influences its performance and is also one of the biggest challenges and showstoppers for the practical applicability of many research autoscalers. Many proposed cloud experiment methodologies can only be partially applied in practice, and many autoscaling papers use custom evaluation methods and metrics. This paper presents a practical guideline for obtaining meaningful and interpretable results on autoscaler performance with reasonable overhead. We provide step-by-step instructions for defining realistic usage behaviors and traffic patterns. We divide the analysis of autoscaler performance into a qualitative antipattern-based analysis and a quantitative analysis. To demonstrate the applicability of our guideline, we conduct several experiments with a microservice of our industry partner in a realistic test environment.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Computer systems organization** → **Cloud computing**; • **Applied computing** → *Service-oriented architectures.*

KEYWORDS

Autoscaling; evaluation; methodology; antipatterns; guideline

1 INTRODUCTION

Efficiency and reliability are two central characteristics of a successful cloud application. In order to ensure these characteristics even under fluctuating load, autoscalers adapt supplied resources dynamically. An increasing number of enterprises are transforming their applications into modern microservice applications and deploying them in a cloud environment [8]. As a result, more and more companies are coming into direct contact with the issues of software performance and autoscaling [5]. The choice of a suitable production-ready autoscaler, and especially the configuration, is by no means trivial. Our previous work found that the configuration of autoscalers is complex and influences their performance significantly [26]. Furthermore, the configuration might differ from service to service, and due to regular updates, continuous reconfiguration and fine-tuning are necessary.

Configuration tuning requires (i) a standardized experiment design and (ii) metrics and methods for evaluating autoscaler performance. In the literature, there is no widely-used evaluation methodology; many papers even evaluate their algorithms only by simulation [24]. For practitioners, it is crucial to get reliable and explainable information about the performance of an autoscaler configuration with reasonable effort, i.e., with not too many, too long measurements. Therefore, measurement methodologies designed for scientists have only limited applicability because of the enormous effort involved.

This paper presents a comprehensive guideline suited for practitioners using state-of-the-art best practices. Specific requirements are understandability, from the design of the experiments to the result analysis, as well as reasonable complexity and effort. We present step-by-step instructions on usage behavior definition and traffic pattern selection for autoscaling experiment design. The analysis of autoscaler performance is divided into two parts. In the first part, the qualitative analysis, we present seven autoscaling antipatterns that indicate misconfigurations. We also explain possible causes and appropriate reconfigurations. In the second part, the quantitative analysis, we present metrics that can be used to compare an autoscaler under test against another configuration or baseline. In all parts of the guideline, we illustrate our statements with experiments using a business service of our industry partner that we evaluate in a realistic test environment.

ICPE '23, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0068-2/23/04…$15.00
https://doi.org/10.1145/3578244.3583721

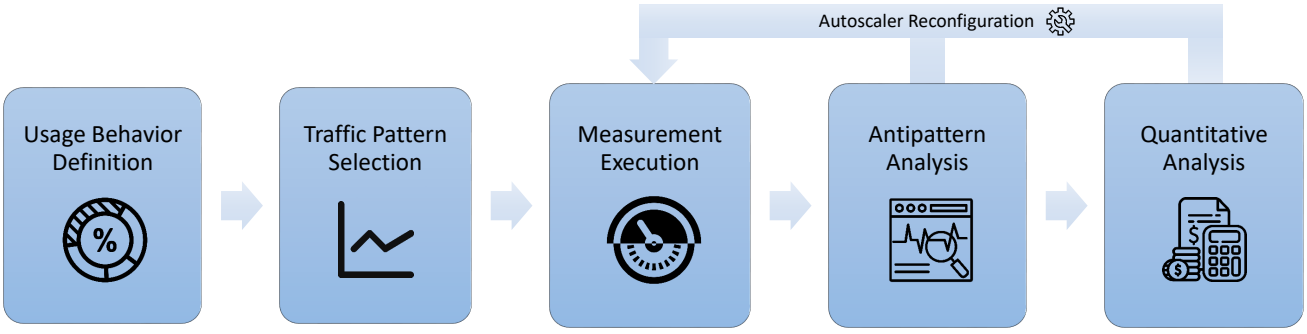


Figure 1: Autoscaler evaluation and reconfiguration process.

The goal of this paper is to give a methodology, primarily for practitioners, to get a meaningful evaluation of their autoscaling configuration with reasonable overhead. Moreover, it should motivate researchers to do structured, standardized evaluations for their proposed autoscalers. In summary, the contribution of this paper is twofold:

- We provide a step-by-step methodology for autoscaler evaluation and configuration especially tailored to the needs of practitioners
- We illustrate the applicability of the guideline by performing a set of experiments with a business microservice application in a realistic environment

The remainder of this paper is structured as follows: Section 2 presents our test application and setup used as a running example in the guideline. Section 3 contains the guideline with four major steps: usage behavior definition, traffic pattern selection, antipattern analysis, and quantitative analysis. We discuss open problems in the areas of autoscaler evaluation and configuration in Section 4. In Section 5, we classify our work into research areas and distinguish ourselves from existing work, while conclusions are drawn in Section 6.

2 RUNNING EXAMPLE

We use a running example throughout this guideline to illustrate our proposed methodology. As our test application, we use a business service from our industry partner DATEV eG with the symbolic name *vacation*. It is responsible for operations connected to an employee's absence and holiday times. From a technical point of view, the service offers four endpoints from which three cause requests to a connected PostgreSQL database, and one retrieves information from a local information cache. The test application uses Java and Spring[1] as implementation technologies.

The test service is deployed in an environment that mirrors our industry partner's technology stack. We use a KubeCF[2] cluster, an open-source and Kubernetes native distribution of the platform-as-a-service environment CloudFoundry for deployment and scaling. The autoscaler under test provides the same functionality as the CloudFoundry App Autoscaler,[3] a simple reactive autoscaler using one scaling metric (e.g., CPU, memory), which is evaluated in fixed periods (e.g., every 60 seconds). The autoscaler uses an upper and lower threshold for the scaling metric to determine its decisions. Whenever the scaling metric is above (below) the upper (lower) threshold, the autoscaler adds (removes) one instance. We use the HTTP Load Generator[4] to send user requests. The load generator and the PostgreSQL database are deployed on dedicated machines outside the KubeCF cluster to avoid interference with the test service.

3 AUTOSCALER EVALUATION AND CONFIGURATION GUIDELINE

In this section, we describe our autoscaler evaluation and configuration guideline. Figure 1 visualizes our proposed process. First, it is important to design an autoscaling experiment that will produce meaningful results. Therefore, it is crucial to define a realistic usage behavior, i.e., which endpoints of the test application users would request. Afterward, the traffic pattern, i.e., the request rates over time, has to be selected. After this step, the workload for the autoscaling experiment is defined, and the measurement can be executed. The measurement results are analyzed in two steps: a qualitative analysis based on antipatterns and a quantitative analysis. Depending on the evaluation outcome, the autoscaler might be reconfigured, and measurements in the same setup are executed to test the reconfiguration effects. The process might be repeated until a proper configuration has been found. In the following, we describe each proposed step in more detail.

3.1 Usage Behavior Definition

First, we need to ensure that the workload (mix of requests) represents realistic traffic that the microservice would receive when deployed in production. This means that the workload needs to contain requests to each endpoint/API implemented by our microservice. Two endpoints of a microservice can induce vastly different loads; for example, getting a single item from an inventory service is much faster than getting all items that meet a criterion. However,

[1] https://spring.io/
[2] https://kubecf.io/
[3] https://github.com/cloudfoundry/app-autoscaler-release
[4] https://github.com/joakimkistowski/HTTP-Load-Generator

Endpoint	Request Type	Path Variables	Request Parameters	Input data in Body	Workload Share
/resultdata	GET	No	Yes	No	20%
/absences	GET	Yes	Yes	No	40%
/absences	POST	Yes	No	Yes	20%
/holidays	GET	No	Yes	No	20%

Table 1: Endpoints, parameterization, and workload share for the running example.

in practice, the endpoint retrieving all items that match a criterion will also be accessed far less often than the get single item endpoint. Therefore, we need to ensure a realistic distribution of requests between the endpoints of our microservice. Depending on the application domain, one might define more than one request distribution to emulate special situations. For example, a company that updates its requirements on user account passwords might temporarily get unusually many requests to a change password endpoint. If such situations are expected to impact the scaling behavior greatly, they might be considered in the autoscaler evaluation and configuration process.

Other factors that influence the load induced by a request are the parameterization and method body of the request (payload). For example, the execution time of a resize image endpoint depends on the size of the image. Therefore, requests during an autoscaling experiment need to have realistic input values for our microservice. Any requests that result in reads from a database, such as the retrieval of product information, should have varying parameters to avoid unrealistic caching effects. Further, the size of the test dataset in the database should be realistic, as a SELECT from a table with hundreds of items performs much faster than with millions of items in the table.

Step 1: Determine endpoint frequency. Assume an equal distribution of requests among the endpoints of the microservice unless domain knowledge or (if applicable) monitoring data from production suggests that a specific endpoint might be called more or less frequently. For example, we might model a 3 to 1 ratio between product pages and purchase requests.

Step 2: Identify relevant request parameters. Identify the key parameters influencing the performance of each microservice endpoint based on knowledge about the application. For example, for an image recoloring endpoint, the size of the image will be impactful, while the target color scheme is unlikely to affect execution time.

Step 3: Determine request parameterization. Provide best-effort estimations for the values of the selected key parameters or derive from production monitoring data. Then, select a small set of representative inputs that will be used in the load test. For example, we might assume that small and medium images are the most common for an uploaded profile picture and use two images with sizes 200x200 and 400x400.

Running Example

Table 1 shows the four endpoints implemented in the vacation service from the running example.

Step 1: Determine endpoint frequency. The vacation service is not yet deployed in production at our industry partner, so we can not use monitoring data to determine the access frequency of the endpoints. Based on our domain knowledge, we can expect that absence times are queried more often than the other functionalities of the service. Consequently, we estimate that these requests should occur twice as much as the others.

Step 2: Identify relevant request parameters. After talking to one of the engineers building the service, we determine that none of the parameters should significantly influence the performance of the service as they have a near-constant size and do not impact the control flow within the service.

Step 3: Determine request parameterization. Most endpoints have path variables, request parameters, or input data in the request body that are required to form valid requests. While we did not identify any of these parameters as particularly performance-relevant, we still want to provide varying inputs to prevent excessive caching. Therefore, we decide to define a set of values for the input parameters for each endpoint and then randomly pick for every request a value from this set.

3.2 Traffic Pattern Selection

After modeling realistic user behavior, we need to select a traffic pattern, that is, how the number of requests per second changes over time. Selecting a realistic traffic pattern is important since the same autoscaling configuration can be suitable for a certain traffic pattern and unsuitable for a different traffic pattern. Therefore, we must select a realistic traffic pattern for our autoscaling experiment. For most human-related traffic, there is a repeated 24h cycle. However, using a representative 24h cycle during a load test is very time-intensive — as it takes 24h per experiment. Therefore, we want to select the most relevant time periods from the daily cycle, e.g., when the request rate grows and decreases the fastest, as well as any significant bursts contained in a typical 24h cycle. The test environment in which the autoscaling experiments run often has fewer resources than the production environment. Therefore, we need to rescale the traffic intensity so that it does not overload the test environment. The more the test environment mirrors the production environment, the better the autoscaler evaluation results can be transferred to the latter. Hence, it is recommended to choose a test environment with a similar hardware and software stack as present in production. Our guideline recommends the following three steps for traffic pattern selection:

Step 1: Select traffic patterns of interest. If the service is already deployed in production, we can analyze a 24h cycle and determine the segments during which we expect to be most challenging for the autoscaler. Usually, this means looking at the sharpest changes (both up and down) in the traffic intensity. If the service is not in production yet, synthetic workload segments that cover common autoscaling scenarios, e.g., rising/decreasing intensities or

bursts, can be used. In Section 3.3, we use four scenarios with synthetic traffic patterns to find autoscaler misconfigurations. In some use cases, one might also use open-source traces (if available) from the same domain if similar workloads are expected in production.

Step 2: Estimate maximum load in a test environment. First, deploy the maximum number of instances that the autoscaler will be able to scale up to in the test environment. Then conduct a load test with a linearly increasing load pattern without autoscaling activated. Repeat this with increasing numbers of requests per second until the load driver starts to report timeouts or errors. The highest load the system could handle without timeouts/errors will be the estimated maximum load the test environment can process. This experiment can be repeated multiple times to validate the estimation and assess performance variability.

Step 3: Rescale the traffic patterns to the maximum load the test environment can process. To do so, first, determine the ratio between the maximum number of requests per second the test environment can handle and the maximum number of requests per second in the traffic pattern. Then, multiply every request per second value in the traffic pattern with that ratio.

Running Example

As mentioned earlier, we do not have access to the production data of our test service. Therefore, we can not use monitoring data to derive traffic patterns. Instead, we make use of a 24h trace from IBM [2] (shown in Figure 2a), as this trace is from a similar domain as our vacation service.

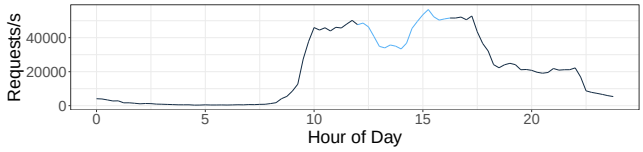
Step 1: Select traffic patterns of interest. As we are not interested in the low load periods overnight and want to pick an interesting sub-trace for the autoscaling test, we select the 4-hour span around the lunch break (12pm-4pm). As highlighted in Figure 2a, it has both a decreasing and increasing load.

Step 2: Estimate maximum load in a test environment. We perform a load test with linearly increasing load intensity to estimate the maximum load in our test environment. We deploy five instances of our test service and then start to stress these instances with a slowly, linearly increasing workload. We observe that at around 220 requests per second (req/s), the response time starts to rise exponentially. Consequently, we estimate the maximum load to be around 220 req/s.

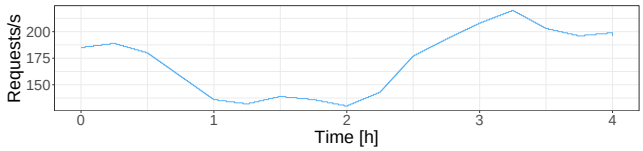
Step 3: Rescale the traffic patterns to the maximum load that the test environment can process. To rescale the trace, we multiply each value in the trace with the ratio of the desired and actual maximum values. In our case, this ratio R evolves as:

$$R = desiredMax/realMax = 220/56574 = 0.00388\,.$$

We now multiply every value in our trace with this ratio. In order to make this trace usable for a load generator, we have to rescale the time axis as well, as most load generators require arrival rates for every second. The trace above has a 15-minute difference between each value. To address this, we oversample the trace above and use linear interpolation to calculate intermediate values. Figure 2b shows the resulting trace.



(a) IBM trace (black) and the selected traffic pattern (blue)



(b) Final trace for the autoscaler evaluation

Figure 2: Traffic pattern selection for the running example.

3.3 Antipattern Analysis

After defining the usage behavior and the traffic pattern selection, the scaling experiment can be executed. It is important to ensure that the initial system state is always the same at the start of the experiment. This means that a warmup may have to be performed, and database states must be reset. We distinguish two methods for the experiment analysis: qualitative analysis and quantitative analysis. In this section, we focus on the first one.

In the qualitative analysis, the visual impression of the scaling behavior plays the most important role, and the methodology is based on seven typical antipatterns that can be discovered. These antipatterns are signs of autoscaler misconfigurations. To evaluate the performance and scaling behavior of an autoscaler, we check whether it can handle four simple load scenarios. If antipatterns occur, they can be recognized clearly in those rather simple settings. In general, the antipattern analysis has two main use cases. First, we can run dedicated autoscaling test runs with simple traffic patterns based on the four defined scenarios. The advantage is that antipatterns are nicely visible, and the results are comprehensive. Second, antipattern analysis can be used to analyze the autoscaler performance while running a complex traffic pattern and focusing only on smaller parts of the experiment. These smaller parts are then similar to the four basic scenarios. The scenarios, antipatterns, and potential reasons for each antipattern are summarized in Table 2. Note that the potential reasons in this table primarily apply to horizontal threshold-based autoscaling, which is commonly used in the area of microservice applications [24, 26]. In the following, we describe every scenario and the linked antipatterns in more detail and name options for autoscaler reconfiguration.

Scenario A: Increasing/Decreasing Load. The first scenario tests whether the autoscaler can sense and react to varying loads. Therefore, any increasing or decreasing load can be used in this scenario to see if the autoscaler makes upscaling or downscaling decisions. The expectation is that when violations of the service level objectives (SLOs) occur, upscaling will occur. For the downscaling test, the concrete expectation depends on the use case. However, an autoscaler should remove resources in case of very low or no load up to a configured minimum.

Scenario	Expected Behavior	Antipattern	Potential Reasons
A: Increasing/Decreasing Load	Service instances are being added/removed	*(1) No reaction:* Neither up- nor downscaling is performed	- Wrong scaling metric - Bad scaling thresholds - Technical issues
B: Constant Load	No scaling after a certain time, SLOs are met	*(2) Wrong steady state provisioning:* No scaling actions, but SLO violations	- High upscaling threshold - Wrong scaling metric - Slow service implementation - Slow dependencies
		(3) Jitter: No steady state is reached, repeated scaling	- Small desired metric range - Short scaling period
C: First increasing, then constant load	Upscaling phase, then potential corrections, followed by steady state	*(4) Rapid upscaling:* Too many instances added, large underutilization	- Low upscaling threshold - Short scaling period - Conservative upscaling policy
		(5) Slow upscaling: Not enough instances added, SLO violations occur	- High upscaling threshold - Long scaling period - Upscaling limits enforced
D: First decreasing, then constant load	Downscaling phase, then steady state	*(6) Rapid downscaling:* Too many instances removed, SLO violations occur	- High downscaling threshold - Aggressive downscaling policy
		(7) Slow downscaling: Instances hesitantly removed, high costs	- Low downscaling threshold - Conservative downscaling policy - Long scaling period

Table 2: Autoscaling antipatterns.

The first antipattern *No reaction* captures the behavior when the autoscaler does not react to varying load and SLO violations. The possible causes of this misbehavior can be divided into two groups: misconfiguration of the autoscaler or misconfiguration of the environment. In the first group, either the scaling metric or the scaling policy, e.g., up- and downscaling thresholds, may be misconfigured. If no suitable scaling metric is chosen, the autoscaler cannot infer overload or underload from the measurements and, therefore, cannot make any decisions. If the scaling metric is not the problem but the autoscaler does not react, another reason could be that the desired metric range, i.e., the interval between the lower and upper threshold, is too large. Depending on the cause, possible reconfigurations are the selection of another scaling metric or new threshold settings.

The second group of possible reasons is technical problems in the cloud setup. For example, the autoscaler may not receive valid values as input. This can be especially the case when metrics are obtained from third-party monitoring frameworks. It has to be ensured that all required metrics are available to the autoscaler at regular intervals. Log data from the autoscaler or the monitoring framework should be used to verify that all components are working correctly. Even if this is probably the simplest failure, a test is worthwhile in practice.

Scenario B: Constant Load. The second scenario covers the opposite of the first one. With the help of constant load, we test the stability of the system and our autoscaling mechanism. The expected behavior is that after a certain settling period, the resources provided remain constant, and all SLOs are fulfilled.

The second antipattern *Wrong steady state provisioning* describes the case where the provided resources are stable, but SLO violations occur. Again, there are two groups of errors: misconfiguration of the autoscaler or errors in the environment. The first group is similar to Antipattern 1. A possible cause can be a high upscaling threshold, which leads to the fact that despite SLO violations, upscaling decisions are not made. Another possible cause is an unsuitable scaling metric, which prevents the autoscaler from detecting overload or underload. The second group of causes does not concern the autoscaler itself but the test application and its environment. This is the case when classical scaling/performance metrics (e.g., CPU, memory) are all in reasonable ranges, but SLO violations, usually high response times., are still observed. This can be due to a poorly performing test application or slow dependencies (e.g., databases). In this case, not the autoscaler but the test service and its dependencies must be optimized. If this is not possible, the definition of the SLOs should be refined.

In the same scenario, it can also happen that no steady state is reached at all, i.e., even after a long time, up- and downscalings take place. In this case, we talk about the antipattern *Jitter*, which has also been discussed in recent papers [12, 28]. A misconfiguration of the autoscaler most likely causes the misbehavior. For example, the allowed range of the scaling metric can be too small. Imagine a theoretical example with a CPU-based autoscaler with an upscaling threshold of 80% and a downscaling threshold of 50%. If we now consider one service instance with 85% CPU utilization, our autoscaler would start a second instance. After the second instance has started, both instances would theoretically have a CPU utilization of less than 50%, assuming the same load level and perfect distribution. Consequently, this would again lead to a downscaling

decision in the next interval. Another reason for jitter could be a short scaling period, where too many decisions are made in a short time and could cancel each other out. Possible reconfigurations include adjusting the thresholds and the period and introducing cooldown times.

Scenario C: Increasing and Constant Load. In this scenario, the upscaling speed and convergence are tested. The test service is stressed with an initially increasing and later constant load. The expected behavior of the autoscaler is that an upscaling phase is initiated when SLO violations occur. When the load increase ends, the autoscaler should keep a stable resource allocation in the shortest possible time without violating any SLOs.

The fourth antipattern *Rapid upscaling* describes the behavior when the autoscaler starts too many instances in the upscaling phase. This leads to the fact that in the constant load phase afterward, many instances must be shut down again, and unnecessarily high costs are created. To counteract this behavior, a reconfiguration of the upscaling policy is necessary. One reason could be that the autoscaler makes upscaling decisions too early, i.e., the threshold at which upscaling is initiated is too low. A second cause could be that the autoscaler makes upscaling decisions too quickly, i.e., the interval between two scaling decisions is too short. A third cause could be that during a scaling decision, too many new instances are started at once, i.e., the upscaling policy is too conservative. Depending on the identified cause, increasing the upscaling threshold or the scaling period is recommended. As an alternative, introducing a limit for the number of instances that can be started simultaneously might be helpful.

Slow upscaling, the opposite of the previous antipattern, is dangerous because it causes unnecessary SLO violations. The reasons for this are the opposite of those for rapid upscaling. One reason could be a high upscaling threshold that causes upscaling actions to be triggered lately. Furthermore, due to a high scaling period, the frequency of upscaling decisions can be too low to react appropriately to rapidly increasing load. Last but not least, upscaling limitations can also be the reason that not enough instances can be created at the same time.

Scenario D: Decreasing and Constant Load. The scenario for testing the downscaling behavior is analogous to the upscaling tests. A traffic pattern with initially decreasing and then constant load is used to stress the test application. The desired behavior of the autoscaler is a downscaling phase followed by a transition to the steady state. Optimally, no SLOs should be violated during the whole process.

The sixth antipattern *Rapid downscaling* describes the behavior when the autoscaler removes too many instances during the downscaling phase, and SLO violations occur. Specifically, this means that the application goes from an underloaded state to an overloaded state due to clumsy scaling. This is one of the most dangerous and unnecessary misbehaviors because it creates unnecessary SLO violations and costs, as seen in our running example later on. The reason for this could be a high downscaling threshold, which causes the downscaling to start too early. Another reason could be a too-aggressive downscaling policy, i.e., too many instances are removed simultaneously. Note that the scaling period is a more unlikely root cause here than in the upscaling scenario, as a service removal usually takes less time than a service start. Besides lowering the downscaling threshold, potential reconfigurations could introduce downscaling limits and cooldown times.

Our last antipattern *Slow downscaling* might be the most acceptable misbehavior depending on the use case. It describes the situation when the autoscaler removes resources unexpectedly hesitantly, and thus the system is heavily underloaded. This leads to higher costs but not to SLO violations. The reasons are opposite to the previous antipattern. Thus, choosing a higher downscaling threshold could force more downscaling decisions. By choosing a more aggressive downscaling policy, one could remove more instances concurrently and thus increase the impact of a scaling decision. Likewise, one could lower the scaling period or cooldown times to increase the frequency of downscaling decisions. As already mentioned, a conservative downscaling behavior may be desirable, especially as it prevents SLO violations in case of unexpected load increases.

Running Example

In the following, we illustrate one antipattern per scenario with the vacation service in our realistic test environment. Figure 3 visualizes the deployed number of instances, the traffic pattern, and the measured response times for all scenarios.

Scenario A. We stress two instances of the vacation service with a constant load of 10 req/s first and then increase the load directly to 200 req/s. The service should answer with an average response time smaller than 4 seconds. We use a memory-based autoscaler with an upscaling threshold of 80% and a downscaling threshold of 25%. During the whole experiment, the autoscaler performs no up- and downscaling. The measured response times are shown in Figure 3. We see that the service has massive performance problems with the increased load. The response time rises to 15 seconds, the configured HTTP timeout.

The memory utilization in Figure 4 shows why the autoscaler did not come into action. It stays within the desired range (between 25% and 80%) during the whole experiment. We conclude that memory utilization is the wrong scaling metric for this service for multiple reasons. First, the metric does not reflect overutilization clearly. We see a small increase of about 9% in memory utilization when the load level changes and the response time rises. However, we see that there is still enough memory capacity available even for the high load intensity. Second, we see that memory utilization never decreases. This is problematic for our threshold-based downscaling strategy. Therefore, a promising reconfiguration is to try another scaling metric (e.g., CPU) for this service.

Scenario B. We stress our test service with a constant load of 60 req/s. The service should answer requests with an average response time smaller than 2 seconds. We use a CPU-based autoscaler with an upscaling threshold of 80% and a downscaling threshold of 25%. The autoscaler deploys one service instance during the whole experiment and performs no up- and downscaling. The measured response times are shown in Figure 3. We see that the measured response time is above our target of 2 seconds. The CPU utilization varies between 31% and 45% during the experiment and is, therefore, in the desired range. It indicates that the CPU is not a bottleneck in this case. In the next step, we check for other bottlenecks (e.g., memory, network). We find out that none of them appears to be a

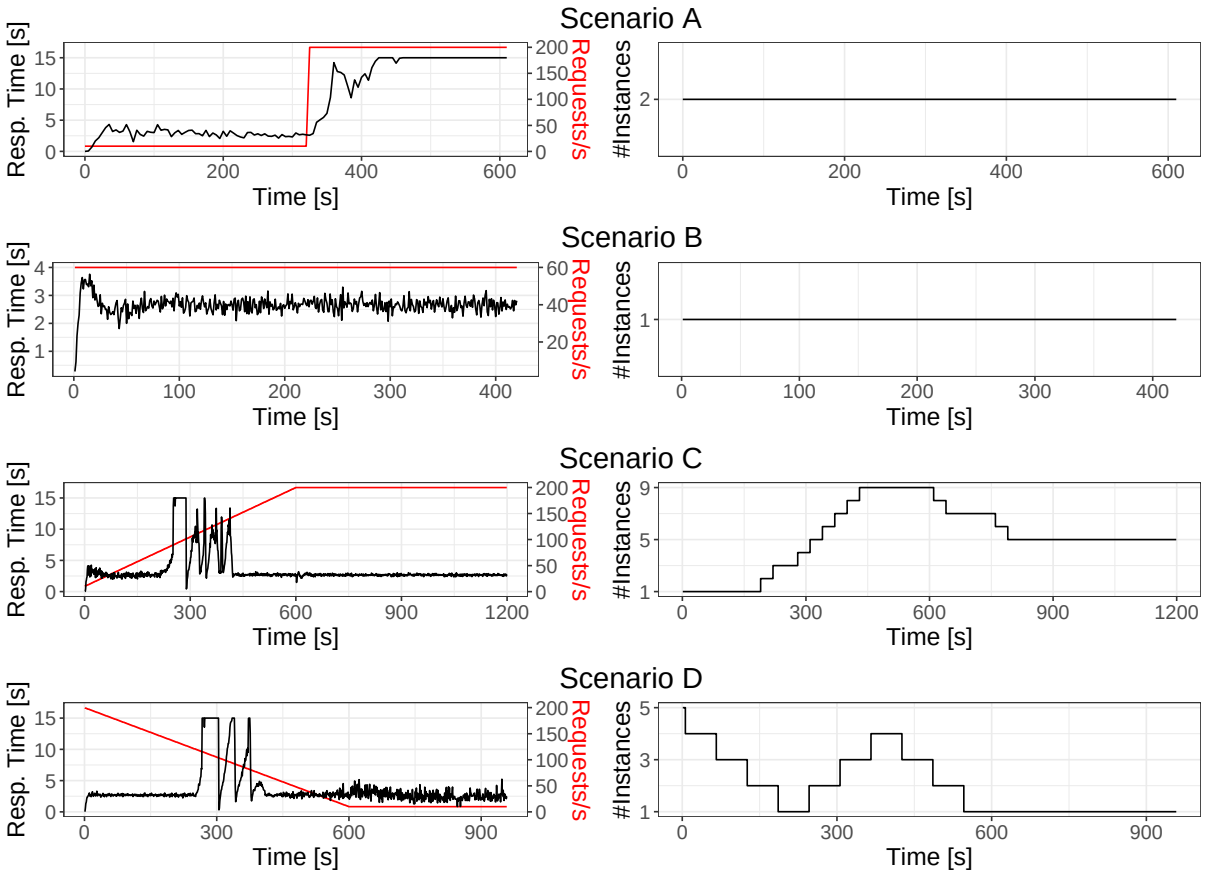


Figure 3: Response times, traffic patterns, and deployed instances for the antipattern analysis.

bottleneck. Consequently, we conclude that external connections, the test service implementation, or the SLO definition are the root cause. This is also confirmed by our results for other experiments in Figure 3, where we observe that the response time does not fall under the 2-second mark, even for low loads. As a result, we cannot derive any statements about the autoscaler configuration, as the autoscaler is not part of the problem here.

Scenario C. We stress our test service with a linearly increasing load first and a constant load of 200 req/s afterward. We use a CPU-based autoscaler with a scaling period of 30 seconds, an upscaling threshold of 80%, and a downscaling threshold of 25%. The number of deployed instances over time and the load intensity are shown in Figure 3. We see that, starting at 190 seconds, the autoscaler begins to deploy one instance in nearly every scaling period and increases the instance count from 1 to 9. With nine deployed instances, the system is in an overprovisioned state. This can be seen in two characteristics in the graph. First, shortly after the maximum load intensity has been reached, the autoscaler directly scales down, which means that the average CPU utilization is under 25%, and the system is clearly underutilized. Second, the autoscaler performs multiple consecutive downscaling actions until the steady state of 5 instances is reached. In our case, the rapid upscaling behavior comes from the low scaling period of 30 seconds. The average readiness time of the vacation service is between 40 and 45 seconds. The autoscaler tends to add more instances than required because it makes new decisions before the old ones are executed.

Scenario D. We stress our test service with a linear decreasing load first and a constant load of 10 req/s afterward. We use a CPU-based autoscaler with a scaling period of 60 seconds, an upscaling threshold of 80%, and an aggressive downscaling threshold of 50% to save costs. The number of deployed instances over time and the load intensity are shown in Figure 3. It is shown that the autoscaler starts to remove one instance per minute until the minimum number of 1 is reached. At this point in time, the system is in an overloaded state, as shown by the response time graph. Consequently, the autoscaler has to revert its latest decisions and scales up at three intervals in a row. After that, the system is underutilized again, and downscaling takes place until a steady state is reached. We see that rapid downscaling behavior can be dangerous for various reasons. First, performance problems evolve, and the service quality drops. Second, unnecessary costs might be created because of upscaling decisions to correct wrong decisions. We conclude that a lower downscaling threshold is needed to achieve a safe downscaling behavior.

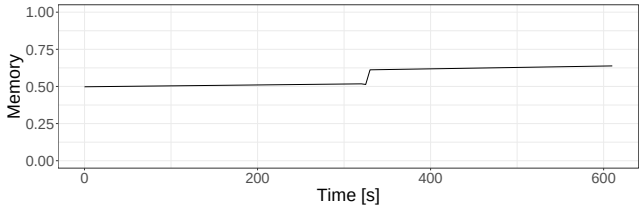


Figure 4: Memory utilization in Scenario A.

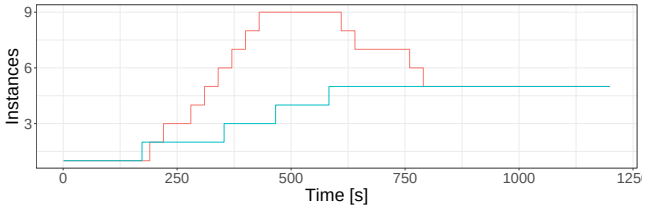


Figure 5: Comparison between test autoscaler (red) and baseline (blue) for Scenario C.

3.4 Quantitative Analysis

Qualitative analysis using antipatterns primarily aims to check whether an autoscaler meets all essential requirements. When using complex workloads, it is more challenging to evaluate the quality of an autoscaler only visually. Quantitative analysis metrics are therefore necessary. In the following section, we present the most common metrics for this purpose and start with metrics that can be used to compare two autoscalers or configurations.

SLO violations: One goal of every autoscaler is to enable a specific level of service quality despite varying workloads. This quality level is defined by service level objectives (SLOs). In the field of web services, those are normally defined by upper bounds of response times, which product requirements might give. A quality measure of the autoscaler is the ratio of failed requests, which is defined by the number of slow/failed requests divided by the total number of sent requests — the lower this ratio, the better the performance of the autoscaler. Note that this metric considers the load intensity (total sent requests) but not the temporal evolvement.

Costs: Preventing SLO violations is only one side of the medal. The autoscaler has to achieve this goal while keeping the costs as low as possible. Different approaches can be used to quantify the costs of a service deployment. In general, we multiply the temporal difference between two scaling decisions with the number of instances. This is equal to calculating the integral of the deployed instances over time. This abstract cost measure can be turned easily into more interpretable units by multiplying a monetary measure, e.g., dollars per instance second.

Combining costs and SLO violations: The goals of minimizing costs and SLO violations might be weighted differently depending on the use case. For example, a core backend service with many dependencies should have no SLO violations (e.g., high response times), so cost minimization might be a secondary goal here. On the other hand, a rarely used legacy service should consume as few resources as possible, and high response times in some cases are acceptable. In order to combine costs and SLO violations in one metric while taking the weight of the goals into account, we can use a simple scaling performance metric P_w [26]. P_w is defined as:

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

Hereby, V is the SLO violation ratio, C is the total costs, $1/C_{max}$ is a normalization factor, and w is the weight for our goals. The normalization factor is needed to transform the costs to a value between 0 and 1. Similar to the costs and SLO violations, the scaling performance metric P_w is a lower-is-better metric.

Consider the following example, where we compare two configurations of an autoscaler. Config A achieves an SLO violation ratio of 0.2 while having normalized costs of 0.6. Config B has a lower SLO violation ratio of 0.1 while having higher normalized costs of 0.8 compared to Config A. If we now assign equal weights to SLO violation prevention and cost minimization, Config A would be better with a $P_{0.5}$ value of 0.4 (Config B: 0.45). If we put a higher importance on the SLO violation prevention by setting w to 0.7, then Config B is better with a $P_{0.7}$ value of 0.31 (Config A: 0.32).

Metrics for comparing the autoscaler against a baseline: In the following, we consider metrics where the performance and temporal behavior of the autoscaler are compared against a baseline. This baseline is often referred to as the demand or desired curve in the literature, and several methodologies exist to determine this baseline [4, 14]. BUNGEE [10] uses dedicated benchmarking runs to determine the maximum load for different numbers of instances. All methods have in common that they are only an estimate of the actual demand. The actual demand might depend on many factors depending on the test application and environment, e.g., user behavior, software and hardware technology stack, and co-locations. Once the baseline is determined, several elasticity metrics [10] can be calculated, e.g., underprovisioning accuracy and timeshare. The first is the average amount of resources that are underprovisioned during the experiment time. The latter is the total time spent in an underprovisioned state normalized by the experiment time. Analogous measures exist for overprovisioning.

Metrics for quantifying the transient behavior: The following metrics do not directly quantify the autoscaler performance but might still be relevant for tuning the autoscaler configuration. We know that new instances do not start in zero time, and the performance of a freshly started instance differs from the performance of an instance that has already handled some minutes of workload. Similar behavior is known as the cold start phenomenon in the serverless computing domain [18]. These facts lead to a transient phase after a scaling decision, where the system behaves differently for a certain load level than it would in a steady state. To analyze this transient phase, one might consider classical performance metrics, e.g., the response time while up- and downscaling or the start time of a new container. Furthermore, the number of scaling events, i.e., how often the autoscaler performed up- or downscaling, might be an interesting metric, e.g., to quantify jitter. The insights we get from this analysis can influence the autoscaler configuration. For example, if our service has a start time of 40 seconds, it makes no

Metric	Value
SLO Violations	0.150
Total Costs	6264
Normalized Costs	0.58
$P_{0.5}$	0.365
$P_{0.7}$	0.279
Underprovisioning accuracy	0.014
Underprovisioning timeshare	0.014
Overprovisioning accuracy	1.543
Overprovisioning timeshare	0.475

Table 3: Autoscaling metrics for Scenario C.

sense to have a scaling period of 30 seconds, as the scaling decision from the previous interval may not have been executed yet.

Running Example

We reuse our example from Scenario C (Antipattern 4) and evaluate the performance of our test autoscaler. We perform dedicated runs according to the BUNGEE methodology to determine the baseline for the comparison. We stress a different number of instances with different load levels to obtain an estimation of the maximum load that can be processed without violating our SLOs. The result is a mapping function that outputs the number of desired instances for a certain request rate. For further information on the BUNGEE methodology, refer to the original paper [10]. Figure 5 shows the behavior of the baseline compared to the test autoscaler. From the visual impression, we see the test autoscaler scales up rapidly and therefore tends to overprovision resources. Additionally, we see a small period around the 200-second mark where the autoscaler fails to provide enough resources.

Table 3 reports all autoscaling metrics for the experiment. The normalized costs were calculated by setting C_{max} to 10800 instance seconds, which would be the costs if nine instances (maximum instances deployed by autoscaler, see Figure 5) had been deployed during the whole experiment. We used a response time limit of 4 seconds for the SLO violations, while the service responds within 2-3 seconds under low load. We see that the overprovisioning metrics are much higher than the underprovisioning metrics. If we take the visual impression into account and our knowledge from the antipattern analysis, we see that the test autoscaler tends to a rapid upscaling behavior. The accuracy and timing metrics presented in this section help verify this and allow for quantitative comparison, e.g., to evaluate a reconfigured version of the autoscaler.

4 OPEN CHALLENGES

This paper presents guidelines that cover both the design and evaluation of autoscaling experiments. However, there are still open challenges in the areas of autoscaling evaluation and configuration and limitations to this work which we discuss in the following.

Challenge 1: Automating autoscaling experiments. Generally speaking, automatability is crucial for adopting new processes and standards into practice. Automated autoscaler evaluation and configuration require a solution dependent on the targeted cloud environment. Hence, we could not make general statements about the measurement execution, and this is why it is the only step from Figure 1 that is not covered within this guideline. In general, these experiments require significant manual efforts. This motivates the development of a comprehensive framework for autoscaling experiments based on benchmarking best practices.

Challenge 2: Automating autoscaler configuration tuning. In Section 3.3, we give hints on which reconfigurations can be applied based on detected antipatterns. The reconfiguration step and the choice of refined parameter values require manual effort. Automation is desirable here, similar to the execution of autoscaling experiments. In general, the problem of automatic configuration can be approached from two directions. On the one hand, it can be seen as a continuous process with feedback, like in reinforcement learning. On the other hand, it can be seen as a classical optimization process where methods like genetic algorithms could be used to explore the configuration space. Either way, both approaches require expert knowledge and possibly many repeated experiments. As discussed in the next challenge, this raises the question of what costs one is willing to invest in configuration tuning.

Challenge 3: Assessing costs and benefits of autoscaler tuning. Besides the absence of a universal automation solution for autoscaler evaluation and configuration, another challenge is the trade-off between the costs and benefits of autoscaling tuning. Each test run requires preparation, execution, and post-processing time and consumes significant resources depending on the use case. The question arises of when to stop the tuning process. Last but not least, there is not one optimal autoscaling configuration for a service. This is due to the different characteristics of real-world workloads, e.g., bursts and unexpected user behavior. It is, therefore, essential to define a realistic usage behavior and traffic pattern but also to keep in mind that, in reality, things can get more complex.

Challenge 4: Tuning ML-based autoscalers. In recent years, autoscalers based on machine, deep, or reinforcement learning have become increasingly popular [24]. While both the antipatterns and our quantitative metrics are applicable to these types of autoscalers, the possible reasons for misconfigurations and available reconfiguration options might be different. Our work assumes that upscaling and downscaling behavior is threshold-based, or at least a policy is directly adjustable. For ML-based autoscalers, this may not be the case. Here, the mapping between discovered antipatterns or bad results to suitable reconfiguration is much more challenging. The amount and type of training data play an enormous role. This challenge goes along with the challenge of explainability from our previous study [26]. For processes with limited explainability, (automated) reconfigurations are much harder.

Challenge 5: Coordinating multi-service autoscaling. Our guideline focuses on the autoscaler evaluation and configuration concerning a single service. This is also a common focus of many other papers [24]. A modern application typically consists of multiple services with complex inter-service communication. Therefore, the reconfiguration of the autoscaling policy of a single service can impact the autoscaling behavior of multiple other services. A common example is bottleneck shifting: Consider a scenario with two services, X and Y, where X sends requests to Y. Assume that we detect a rapid upscaling behavior for service Y and fix it by reconfiguration. Service X could now get problems because Y does not scale up as fast as before. This is just one example where we see

that it is crucial to maintain the global view despite service-specific configurations.

5 RELATED WORK

In this section, we discuss recent contributions from the areas of measurement and benchmarking, configuration, and evaluation of autoscalers.

5.1 Measurement Principles and Autoscaling Benchmarking

In their papers, Papadopoulos et al. [20] and Schwartzkopf et al. [23] proposed eight and seven principles for reproducible experiments in cloud computing, respectively. For supercomputing environments, Hoefler and Belli [11] suggested 12 principles, and Frachtenberg and Feitleson [9] discuss 32 pitfalls as well as measurement principles. In contrast to the aforementioned articles, Leznik et al. [17] and Vitek and Kalibera [27] elaborate shortcomings of state-of-the-art experiments. In terms of evaluating or benchmarking autoscalers, different approaches were proposed. Papadopoulos et al. [19] introduced PEAS that evaluates autoscalers based on simulations. Another tool for measuring the performance of autoscalers in IaaS environments was introduced by Jindal et al. [14]. Further tools that assess the performance of autoscalers automatically are BeCloud [4] and BUNGEE [10]. In this paper, we propose an autoscaling experiment methodology and evaluation process that is usable in practice. For this purpose, it is essential to get meaningful and interpretable results with a reasonable effort. This is the main contrast point to methodologies with a purely scientific background. Nevertheless, we use best practices from generic measurement guidelines and tailor them to the autoscaling use case while preserving the applicability in practice.

5.2 Autoscaler Configuration

Finding the most suitable configuration for an autoscaler remains a trial-and-error task, as the type and meaning of configuration parameters needed by various approaches are manifold. For instance, rule-based autoscalers require many critical manual settings (at least up- and/or downscaling thresholds) that influence the performance of the autoscaler massively [1]. More sophisticated approaches require inputs like manually created models [3] or costs of reconfiguration [21]. Especially hybrid scalers [13, 25] often require manual parameter or offline tuning. Although reinforcement learning-based approaches [15, 22, 29] help to reduce the configuration overhead, most reinforcement learners assume a static application and have problems with changes introduced by updates [7]. In summary, autoscaler configuration is non-trivial in practice, as every service might need different settings, and one must understand the configuration parameters' meaning and influence on the autoscalers decision logic. In this guideline, we show seven common antipatterns which can be used to discover autoscaler misconfigurations. Moreover, we link possible reasons and reconfigurations to each antipattern.

5.3 Autoscaler Evaluation

Although there are different benchmarks for quality assessment and comparison of autoscalers, most of the proposed autoscalers were evaluated in custom experiments. Some approaches were only simulated on up to four different workloads [6, 13, 16, 25, 29] or built a custom testbed with up to two different workloads [3, 15, 21, 22], while using their own quality measures to quantify the performance of their autoscalers. By using simulation or own metrics or custom testbeds, the results can be poorly generalized and thus transferred into practice. In this guideline, we propose to split the result analysis into two parts. While the qualitative analysis tests whether the autoscaler fulfills essential requirements using easily reproducible test setups, the quantitative analysis provides an in-depth look at the autoscaler performance using well-defined metrics.

6 CONCLUSION

Autoscalers are needed for any cloud application that is exposed to dynamic workloads. The configuration of an autoscaler has a massive impact on its performance. It is not trivial for practitioners to measure the performance of an autoscaler and derive a proper configuration. This paper presented a comprehensive methodology and guidelines specifically for practitioners. It allows obtaining meaningful insights about autoscaler performance with reasonable overhead.

We propose step-by-step instructions for the design of autoscaling experiments, including usage behavior definition and traffic pattern selection. The evaluation of the experiment results is divided into two parts. In the antipattern analysis, the expected behavior is compared with the actual behavior of the autoscaler in simple scenarios. We link every scenario and antipattern to possible reasons and reconfigurations. In the quantitative analysis, we define a set of metrics to compare one autoscaler against another autoscaler or a baseline. Experiments with a microservice from our industry partner illustrate all steps within this guideline. Furthermore, we identify open challenges in the areas of autoscaler evaluation and configuration. All in all, this paper aims to serve as an entry point to these areas for practitioners. Moreover, it gives researchers an idea of how a simple yet meaningful evaluation of an autoscaler might be done.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Marta Beltrán. 2016. BECloud: A new approach to analyse elasticity enablers of cloud services. *Future Generation Computer Systems* 64 (2016), 39–49.

[5] 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.

[6] 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.

[7] 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.

[8] European Statistical Office (Eurostat). 2021. *Cloud computing - statistics on the use by enterprises*. Retrieved October 13, 2022 from https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises

[9] Eitan Frachtenberg and Dror G. Feitelson. 2005. Pitfalls in Parallel Job Scheduling Evaluation. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*.

[10] Nikolas Roman Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. 2015. BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 46–56.

[11] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA.

[12] Chunmao Jiang and Peng Wu. 2021. A Fine-Grained Horizontal Scaling Method for Container-Based Cloud. *Scientific Programming* 2021 (11 2021), 1–10.

[13] 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.

[14] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2018. Autoscaling performance measurement tool. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 91–92.

[15] 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.

[16] Reihaneh Khorsand, Mostafa Ghobaei-Arani, and Mohammadreza Ramezanpour. 2018. FAHP approach for autonomic resource provisioning of multitier applications in cloud computing environments. *Software: Practice and Experience* 48, 12 (2018), 2147–2173.

[17] Mark Leznik, Johannes Grohmann, Nina Kliche, André Bauer, Daniel Seybold, Simon Eismann, Samuel Kounev, and Jörg Domaschka. 2022. Same, Same, but Dissimilar: Exploring Measurements for Workload Time-Series Similarity *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 89–96.

[18] Garrett McGrath and Paul R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 405–410.

[19] Alessandro Vittorio Papadopoulos, Ahmed Ali-Eldin, Karl-Erik Årzén, Johan Tordsson, and Erik Elmroth. 2016. PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 1, 4 (2016), 1–31.

[20] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim Von Kistowski, Ahmed Ali-Eldin, Cristina Abad, José Nelson Amaral, Petr Tuma, and Alexandru Iosup. 2019. Methodological principles for reproducible performance evaluation in cloud computing. *IEEE Transactions on Software Engineering* (2019).

[21] 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.

[22] 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.

[23] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. 2012. The Seven Deadly Sins of Cloud Computing Research. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*. USENIX Association, Boston, MA.

[24] 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.

[25] 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.

[26] 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 on International Conference on Performance Engineering* (Beijing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 105–115.

[27] Jan Vitek and Tomas Kalibera. 2012. R3: Repeatability, Reproducibility and Rigor. *SIGPLAN Not.* 47, 4a (2012).

[28] Thomas Wang, Simone Ferlin, and Marco Chiesa. 2021. Predicting CPU Usage for Proactive Autoscaling. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 31–38.

[29] 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).