Microservices: A Performance Tester's Dream or Nightmare?

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

Cor-Paul Bezemer
University of Alberta
Edmonton, Canada
bezemer@ualberta.ca

Weiyi Shang
Concordia University
Montreal, Quebec
shang@encs.concordia.ca

Dušan Okanović
University of Stuttgart
Stuttgart, Germany
okanovic@iste.uni-stuttgart.de

André van Hoorn
University of Stuttgart
Stuttgart, Germany
van.hoorn@iste.uni-stuttgart.de

ABSTRACT

In recent years, there has been a shift in software development towards microservice-based architectures, which consist of small services that focus on one particular functionality. Many companies are migrating their applications to such architectures to reap the benefits of microservices, such as increased flexibility, scalability and a smaller granularity of the offered functionality by a service.

On the one hand, the benefits of microservices for functional testing are often praised, as the focus on one functionality and their smaller granularity allow for more targeted and more convenient testing. On the other hand, using microservices has their consequences (both positive and negative) on other types of testing, such as performance testing. Performance testing is traditionally done by establishing the baseline performance of a software version, which is then used to compare the performance testing results of later software versions. However, as we show in this paper, establishing such a baseline performance is challenging in microservice applications.

In this paper, we discuss the benefits and challenges of microservices from a performance tester's point of view. Through a series of experiments on the TeaStore application, we demonstrate how microservices affect the performance testing process, and we demonstrate that it is not straightforward to achieve reliable performance testing results for a microservice application.

KEYWORDS

Microservices, DevOps, Performance, Regression testing

ACM Reference Format:
Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. 2020. Microservices: A Performance Tester's Dream or Nightmare?. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20), April 20–24, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3358960.3379124

1 INTRODUCTION

Microservices [24, 32, 38] are a popular trend in software architecture in recent years for building large-scale distributed systems. Microservices employ architectural design principles leading to explicit domain-based bounded contexts and loose coupling [40], exploit modern cloud-based technologies including containerization and self-healing [17], and are suitable for modern software engineering paradigms such as DevOps [5] including agile development methods and continuous delivery.

Similar to other software systems, product quality [23] plays an important role for microservices and microservice-oriented architectures. An important non-functional quality attribute is performance, which describes a system's properties with respect to timeliness and resource usage, including aspects such as scalability and elasticity [22]. Timeliness may seem to be achievable more easily by cloud features such as auto-scaling. However, resource usage becomes extremely relevant because inefficient architectures and implementations lead to high costs as a part of pay-per-use charging models for cloud resources.

The performance of a system can be assessed through several performance engineering techniques, such as performance testing [8, 25]. Performance testing is already considered challenging in traditional systems [30]. Even worse, the architectural, technological, and organizational changes that are induced by microservices have an impact on performance engineering practices as well [21]. While some of these changes may facilitate performance testing, others may pose considerable challenges.

In this paper, we discuss whether microservices are a performance tester's dream or nightmare. In particular, we run a series of experiments on the TeaStore [51], a reference microservice application, to demonstrate the challenges that come with performance testing microservices. Our experiments address the following research questions:

RQ1: How stable are the execution environments of microservices across repeated runs of the experiments? Our experiments demonstrate that the execution environments of microservices are not stable across experiment runs, even when the total number of provisioned instances of a microservice is kept the same.

ICPE '20, April 20–24, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-6991-6/20/04.
https://doi.org/10.1145/3358960.3379124

RQ2: How stable are the performance testing results across repeated runs of the experiments? Our experiments show that although the CPU busy time may not be significantly different between scenarios, there often exist statistically significant differences in response time. Such differences may have a direct negative impact on the user-perceived performance, and make analyzing the performance test results of microservices more challenging.

RQ3: How well can performance regressions in microservices be detected? It is possible to detect performance regressions in microservices applications. However, one needs to ensure that enough repetitions of the performance tests are done to deal with the variance in the deployment of the services and the environments in which they run.

Our results show that performance testing microservices is not straightforward and comes with additional challenges compared to performance testing 'traditional' software. Hence, future research is necessary to investigate how to best tackle these additional challenges. We provide a peer-reviewed replication package [16] to assist other researchers in the reproduction and extension of our case study.

In the rest of this paper, first we discuss related work (Section 2), the characteristics of microservices (Section 3) and the ideal conditions for performance testing (Section 4). Section 5 discusses the benefits of microservices for performance testing. Section 6 presents our experimental setup and results. Section 7 reflects on our experiments and discusses the challenges of performance testing microservices and promising future research directions. Section 8 discusses the threats to the validity of our experiments. We conclude the paper in Section 9.

2 RELATED WORK

In this section, we discuss the prior research that is related to this paper. We split the related work into works on performance testing in cloud environments in general, works on testing of microservices, and works in the field of performance regression testing.

2.1 Performance testing in cloud environments

The results of performance tests in the cloud have been studied extensively over the past years [4, 27, 28, 31, 43, 48].

One of the first works in this area is from Leitner and Cito [31], which lays the ground by assessing the performance variation and predictability of public clouds, such as Amazon Web Services (AWS). Afterwards, Laaber et al. [28] study the use of these public cloud environments to conduct performance microbenchmarking. Study results show that the variability of microbenchmarking results differs substantially between benchmarks and instance types from the cloud providers. However, executing a test on the same instance can still detect slowdowns with the help of statistical analysis. On the other hand, Arif et al. [4] compare the performance testing results from virtual and physical environments, where findings show that the performance testing results may not be consistent across different environments.

In order to address the variation and instability of cloud environments when conducting performance tests, follow-up research by Scheuner and Leitner [43] presents a new execution methodology. The approach combines micro- and application benchmarks and is particularly designed for cloud benchmarking environments. In addition, Scheuner and Leitner [27] examine the problem from a different angle, i.e., the quality of performance microbenchmarks and propose a quality metric that can be used to assess them. Costa et al. [12] proposed a tool to further improve the quality of microbenchmarks by detecting bad practices in such benchmarks. By acknowledging the variation and instability of cloud environments, He et al. [20] design an approach that estimates a sufficient duration of performance tests that are conducted in a cloud environment.

It can be noted that the focus of these works was not on the performance testing of microservice-based applications. Although they demonstrate promising results of using cloud environments for performance testing, microservice-based architectures may have a negative impact on the performance testing process due to variation and instability of cloud environments.

2.2 Microservices

Due to the wide practical adoption of microservice architectures, there exists a body of research discussing visions on the testing of microservices. This research, however, often focuses on functional tests, e.g., [38]. Nonetheless, performance is one of the major aspects of consideration when adopting microservices. Heinrich et al. [21] argued that traditional performance modeling and monitoring approaches in most cases cannot be reused for microservices. Knoche [26] presented a simulation-based approach for transforming monolithic applications into microservice-oriented applications while preserving their performance. Aderaldo et al. [1] discussed several reference applications for microservices, and their advantages and disadvantages in terms of being used as a benchmark application. Dragoni et al. [14] argued that network overhead will be the major performance challenge for microservice-oriented applications. Jamshidi et al. [24] provided an overview of the evolution of microservices, and described the ten technological 'waves' that steered this evolution. To our knowledge, our paper is the first to highlight challenges that arise in the context of microservices performance testing.

2.3 Performance regression detection

A performance regression is an unintended change in the performance behavior across software versions. Therefore detection of performance regressions is an important task in performance engineering. Performance regression testing can be conducted on several levels — from a code-level microbenchmarking to a system-level load testing. Regressions are detected by looking at (the evolution of) performance metrics, from system-level metrics, e.g., resource utilization, over architectural metrics, e.g., calls between components, to end-user metrics, e.g., response times [44, 54]. A key challenge is to find a set of suitable performance metrics based on which to reliably detect real regressions [44]. Several algorithms can be used to detect regressions in the data, including simple statistical tests, time-series analysis, and machine learning [11, 19]. However, none of the existing research takes into consideration the associated challenges of microservice architectures. To the best of our knowledge, this paper is the first work that studies performance regression detection in the context of microservices.

3 TRAITS OF MICROSERVICES

While there is no precise definition of a microservice architecture, there are particular traits that are common in such architectures [24, 32, 38]. Below, we summarize the most important ones while keeping performance testing in mind.

(T1) Self-containment. Each microservice is technically self-contained, i.e., it has its own deployment unit, runs in its own process context, and has its own data storage.

(T2) Loosely coupled, platform-independent interfaces. Microservices expose their functionality only using well-defined, platform-independent interfaces (e.g., REST). The interfaces are loosely coupled, in particular, there are no components like client libraries that would establish build-time dependencies between different microservices.

(T3) Independent development, build, and deployment. The strict technical isolation and self-containedness allow each microservice to be built and deployed independently of other microservices. Usually, microservices are developed by different teams in separate code repositories, and the services are deployed into production per the team's own decision. Therefore, microservices are considered a premier architecture for DevOps [5].

(T4) Containers and Container Orchestration. The deployment units in microservice architectures are usually *container images* that are run on a container-based virtualization platform such as *Docker*.[1] These images contain the microservice itself, the required runtime libraries, and even parts of the underlying operating system. In order to efficiently manage the potentially large number of container instances in a microservice architecture, container orchestration solutions such as *Kubernetes*[2] have emerged. These tools automatically deploy containers on a cluster of hosts, provide auto-scaling and load balancing functionalities, and even provide self-healing capabilities in case of node failure.

(T5) Cloud-native. Many well-known microservice installations (e.g., at Netflix and Amazon) are deployed in the cloud. Due to the loose coupling and independent deployment, microservices allow for high scalability, and the cloud provides the resources to actually run the required instances.

4 REQUIREMENTS FOR PERFORMANCE TESTING

The performance of an application is impacted by many factors. As a result, setting up and conducting a performance test is challenging, as the tester needs to control as much as possible for each of these impacting factors. Many existing performance testing techniques assume that (most of) the impacting factors are controlled. Therefore, the ideal performance test and the environment in which it executes have the following requirements:[3]

(R1) A stable testing environment which is representative of the production environment. Conducting a performance test in the production environment may have a negative impact on the users of the production environment. Hence, performance tests are often conducted in test (or staging) environments. However, it is challenging to predict the production performance from a performance test on a smaller, less powerful testing environment. For example, there is a discrepancy between performance test results from a virtual and physical environment [4]. Hence, the testing environment should ideally be equal to the production environment. In addition, a heterogeneous testing environment makes the analysis of performance testing results more challenging [18]. Hence, ideally we can run tests in a homogeneous testing environment.

(R2) A representative operational profile (including workload characteristics and system state) for the performance test. A performance test must execute a workload that is realistic and representative of the actual usage of the application [25]. One possible way to create such a workload is using workload characterization and extraction [9, 49]. This workload should be preceded by the initialization of the application in a representative state (e.g., initializing the database contents). It is important that this initialization step (i.e., the ramp-up time) is not included when analyzing the test results.

(R3) Access to all components of the system. Performance is tested at several levels (e.g., at the unit level [47] and integration level [30]). Hence, it is important that the complete application (including all its components) is available to thoroughly test its performance.

(R4) Easy access to stable performance metrics. Performance metrics (e.g., performance counters) should be easily accessible [33]. In addition, the metrics that are monitored and analyzed during the test should yield the same measurements each time the test is conducted under the same circumstances. If not, the tester cannot decide whether the variance in measurements is because of changes in the environment, or a performance degradation of the application.

(R5) Infinite time. In many cases, a longer performance test yields more information about an application. As a result, a performance test may take hours or even days to complete.

5 A PERFORMANCE TESTER'S DREAM

Microservices have several characteristics that benefit performance testing when compared to 'traditional' software. In this section, we give an overview of the benefits of microservices for performance testers, based on the previously discussed requirements and traits.

Benefit 1: Containerization. Microservices are typically deployed inside lightweight (e.g., Docker) containers (trait T4). A major benefit of such containers is the ease with which tests can be deployed, executed, and reproduced. Because containers help with setting up a consistent test environment, containers can be a useful tool for performance testers.

Benefit 2: Granularity. Microservices are self-contained and do not specify any build dependencies (trait T1). Therefore, by design every microservice has to encapsulate a well-defined set of functionality. Additionally, dependencies to other microservices are managed via loosely coupled, platform-independent interfaces, which are usually implemented as REST APIs (trait T2). This prevents a microservice from depending on implementation details of other microservices.

[1] https://www.docker.com
[2] https://https://kubernetes.io
[3] Note that unfortunately, often these requirements are in conflict with reality.

In traditional systems it is often unclear what constitutes a component for component-level testing. The clear encapsulation and small size of microservices makes them an ideal abstraction for component-level testing. As microservices access dependent microservices using REST APIs, any dependencies can be easily mocked using frameworks such as Wiremock.[4] In theory, no integration tests are required as all interactions between microservices can be mocked based on their API documentation.

Benefit 3: Easy access to metrics. In traditional architectures, collecting data about software execution usually requires instrumentation of the software code or libraries. In microservice architectures, services can be developed using different platforms (trait T3) and it is not always possible to develop one monitoring solution to support this.

However, monitoring solutions for microservices are integrated seamlessly, with very little configuration, and without any knowledge of implementation technology. They take the advantage of the loose coupling between microservices (trait T2) and work on the API level. Tools are able to continuously discover microservices, and scale with the system under test, taking the advantage of trait T5. Loose coupling using APIs also allows for easier full stack monitoring, i.e., collecting data from various system levels, e.g., from the supporting infrastructure, containers, and the application itself. If the application is deployed to multiple cloud providers, monitoring tools allow for collecting and aggregating the data from all of them.

We can conclude that microservice architectures provide more support for fulfillment of requirements R3 (access to all components) and R4 (easy access to metrics), in comparison to traditional architectures.

Benefit 4: Integration with DevOps. Performance testing is typically conducted in a late stage of the release cycle, i.e., after the system is built and deployed in the field or in a dedicated performance testing environment. The complexity and the cost of performance testing makes it challenging to fit into a typical fast release cycle of DevOps. The adoption of microservices brings the opportunity of bridging the gap between performance testing and DevOps. The smaller scope of a microservice itself significantly reduces the complexity and cost of the performance testing (trait T3). Hence, performance testing would not become the bottleneck of a release pipeline. In addition, the simpler development history of a microservice makes it easier for practitioners to identify the introduction of a performance issue and fix it in a timely manner.

6 CASE STUDY

Despite the benefits that microservices have for performance testing, there also exist characteristics that make performance testing challenging. In this section, we demonstrate these challenges through experiments on the TeaStore application. In particular, we conduct two experiments. In the first experiment, we investigate how stable the execution environments and performance measurements of microservice-based applications are. In the second experiment, we investigate how well the performance measurements from microservice-based applications can be used to detect performance regressions.

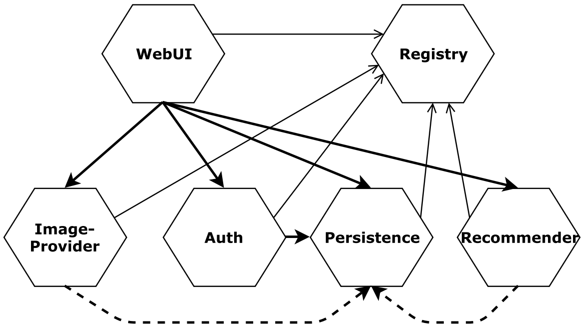


Figure 1: TeaStore Architecture (Source [51]).

6.1 Experimental setup

For our experiments, we deployed the TeaStore (version 1.3.0), a reference microservice application on a Google Kubernetes Engine (GKE 1.11.6-gke.2) cluster [51]. The cluster has two node pools: the first node pool consists of two n1-standard-2 VMs (2 vCPUs, 7.5 GB memory) and contains the Kubernetes control plane services. The second cluster consists of 20 VMs with 1 vCPU and 6.5 GB memory each, resulting in a total of 20 vCPUs and 130 GB memory. As shown in Figure 1, the TeaStore consists of five services and a service registry. We do not consider the registry as part of the system under test and therefore deploy it in the node pool containing the Kubernetes control plane. We deploy eight instances of each of the five services, resulting in a total of 40 service instances. Kubernetes uses two parameters to control the resource usage of containers, resource requests and resource limits. Resource requests are used to guide scheduling decisions, for every node, the sum of resource requests can not exceed the node capacity. For our case study, we initially assign resource requests of 420 m(illi) CPU (about half a core) and 1 GB memory for each service instance. Resource limits manage the actual resource consumption of service instances by providing an upper limit for them. We assign resource limits of 2000 m CPU (about 2 cores) and 3 GB memory. This configuration is in line with the best-practices recommended by Google [13]. As a workload we use the browse profile of the TeaStore, in which users login, browse categories and products, add items to their shopping cart and finally logout. We measure for three different load-levels: 700 requests per second, 800 requests per second and 900 requests per second. Every measurement run consists of 20 minutes warm-up phase and five minutes of measurements. Based on this default setup, we measure the performance of the TeaStore in the following scenarios in the first experiment:

- **Default** This scenario uses the exact specification described above and represents the starting point of our experimentation.
- **Balanced** During our experiments, we noticed that the services differ in their CPU usage. Hence, for this scenario, we enforce co-location of the WebUI service (the most CPU-intensive service) with the Recommender service (the least CPU-intensive service) by adjusting their resource requests (WebUI: 620m CPU; Recommender: 220m CPU).

[4] http://wiremock.org/

Scenario	#Nodes	Cores/Node	Memory/Node
Default	20	1	6.5 GB
Balanced	20	1	6.5 GB
LargeVMs	5	4	26 GB
Autoscaling	5	4	26 GB
Regression (baseline)	5	4	26 GB
Regression	5	4	26 GB

Table 1: Cluster size in the different scenarios.

- **LargeVMs** To investigate the influence of the node size, this scenario replaces the 20 small VMs with five larger VMs with 4 vCPUs and 26 GB memory each. This configuration uses 20 vCPUs and 130 GB memory in total, which is the same total amount as in the default scenario.
- **Autoscaling** Many microservice applications rely on autoscaling instead of static deployments. Hence, in this scenario, we use the Kubernetes Horizontal Pod Autoscaler (HPA) instead of a fixed number of service instances. The HPA for every service is configured to target a CPU utilization of 0.8 and is allowed to use up to 8 pods.

In the second experiment, we investigate how well we can detect performance regressions in the TeaStore. Therefore, in the second experiment we injected a performance regression and compared it to the measurements for the same configuration without the performance regression. However, at this time GKE 1.11.6-gke.2 was no longer available on the Google Cloud Platform and from initial testing it seemed that this version change was impacting the application performance. Therefore, we also measured the baseline scenario again with the new GKE version.

- **RegressionBaseline** Identical setup to the LargeVMs scenario, but uses GKE version 1.12.8-gke.10 instead of GKE 1.11.6-gke.2.
- **10% Regression** To investigate the impact of performance regression of a service on the overall system performance, we injected a performance regression in the form of an active delay of 4 ms (≈ 10% of the base response time of the full system) in the WebUI service.
- **30% Regression** Identical setup to the 10% regression scenario, but with a delay of 12 ms instead of 4 ms (≈ 30% of the base response time of the full system instead of ≈ 10%).

Table 1 shows the cluster and node size for each of the experimental scenarios. To produce stable and reproducible results, we repeated every scenario ten times resulting in a total of 180 measurement runs of 25 minutes each (6 scenarios * 3 load-levels * 10 repetitions). Additionally, to avoid human error during the experiments, we fully automate the cluster provisioning using Terraform, the deployment using .YAML files and the load generation using the load driver that was provided by von Kistowski et al. [50]. In order to assist other researchers in the reproduction and extension of this case study, we provide a peer-reviewed replication package [16] containing: (i) the scripts to reproduce our measurements, (ii) the measurement data obtained in this study and (iii) the code used to replicate the following analysis and figures. In the remainder of this section we describe the motivation, approach and findings for each of the research questions that we address through our experiments.

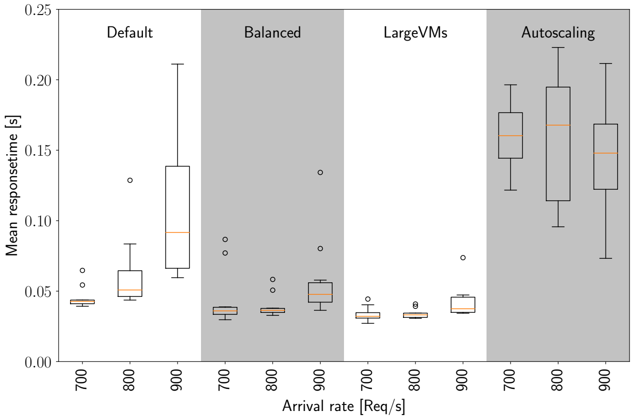


Figure 2: Mean response time for four scenarios and three load-levels each (all distributions consist of ten elements, one for each repetition of the scenario).

6.2 RQ1: How stable are the execution environments of microservices across repeated runs of the experiments?

6.2.1 Motivation: As discussed in Section 4, one of the most important requirements for performance testing is having a stable testing environment. In this research question, we study whether the environment in which microservices execute is typically stable.

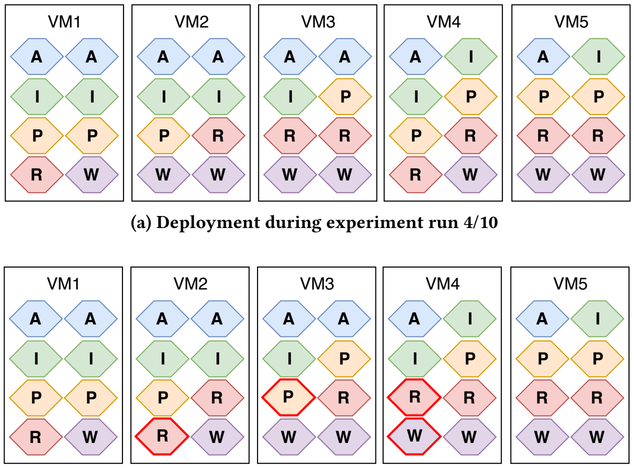
6.2.2 Approach: First, we study how microservices are deployed across virtual machines in different runs of the experiments in which the number of instances of each microservice is fixed. Second, we study the deployment of microservices across virtual machines in the autoscaling scenario.

6.2.3 Findings: **The non-deterministic behaviour of the autoscaler results in different numbers of provisioned microservice instances when scaling the same load.** Table 2 shows the total number of instances of each microservice that were provisioned during the runs of the autoscaling scenarios for 700, 800 and 900 requests per second. For all microservices (except the WebUI microservice) the number of provisioned instances varied across the runs of an experiment. In some cases, such as for the Auth microservice in the 700 requests per second experiment, the difference in the number of instances was as large as 4 instances (three instances in runs 8 and 10 versus seven in run 6). These differences in the number of provisioned instances are also reflected by the large variation in mean response time across the runs of the Autoscaling experiments in Figure 2.

Even when fixing the number of provisioned instances of a microservices, their deployment across VMs differs. Figure 3 shows the deployments from two consecutive runs of the LargeVMs scenarios with 700 requests per second. Although the total number of provisioned instances of a microservice is constant

Load	Service	Experiment run									
		1	2	3	4	5	6	7	8	9	10
	Auth	4	5	4	4	4	7	4	3	4	3
	WebUI	8	8	8	8	8	8	8	8	8	8
700	Recom.	2	2	1	1	1	1	1	1	1	1
	Persist.	8	8	7	6	7	5	6	6	6	6
	Image	4	4	4	4	4	5	3	3	4	4
	Auth	5	6	4	4	4	4	4	4	4	4
	WebUI	8	8	8	8	8	8	8	8	8	8
800	Recom	1	3	1	1	1	2	1	1	1	1
	Persist.	7	8	7	7	7	7	7	7	7	7
	Image	4	5	4	5	4	4	3	4	4	4
	Auth	5	5	5	5	5	5	4	5	5	3
	WebUI	8	8	8	8	8	8	8	8	8	8
900	Recom.	2	2	2	2	2	2	2	2	2	2
	Persist.	8	8	8	8	8	7	7	8	8	7
	Image	5	5	5	5	5	5	5	4	5	4

Table 2: Number of provisioned service instances after twenty minutes of warmup across ten experiment repetitions in the Autoscaling scenario.



(a) Deployment during experiment run 4/10

(b) Deployment during experiment run 5/10

Figure 3: Deployment from two repetitions of LargeVMs scenario with 700 requests/second. The differences in deployment are indicated by thick red borders (A = authentication service, I = ImageProvider service, P = Persistence service, R = Recommender service, W = WebUI service).

across the runs (i.e., all microservices have eight instances), their deployment differs. We observe that the VM2, VM3, and VM4 virtual machines host a different set of microservices across the two runs.

> *The execution environments of microservices are not stable, even when the total number of provisioned instances is kept the same. Hence, it is difficult to ensure that a microservice always executes under the same circumstances.*

6.3 RQ2: How stable are the performance testing results across repeated runs of the experiments?

6.3.1 Motivation: The findings of RQ1 illustrate the possible different deployments in microservices, which may be transparent and can bring uncertainty to the stakeholders of the systems that are deployed on top. However, although they are different, the deployments do not necessarily cause actual differences in performance. Hence, the differences in deployments could have a minimal impact on the stakeholders. In this research question, we study the stability of the performance measurements that were made during our experiments across different experimental runs.

6.3.2 Approach: We compare the distributions of the measured response times to measure the differences in user-perceived performance across all experimental scenarios and workloads. To decide whether the differences between two distributions are significant, we perform a Mann-Whitney U test [36] between every pair of scenarios for a particular workload. For example, we test whether the difference between the performance measurements for the Default and Balanced scenarios is significant when the workload is 700 requests per second. We chose the Mann-Whitney U test since it does not have requirements on the distribution of the data. The null hypothesis of the Mann-Whitney U test is that the two input distributions are identical. If the test computes a p-value that is smaller than 0.05, we consider the two input distributions to be significantly different. Otherwise, the difference between the two input distributions is not significant.

In addition, when two input distributions are found to be significantly different, we compute Cliff's Delta d [34] to quantify the difference. We used the following thresholds for interpreting d [41]:

$$\text{Effect size} = \begin{cases} \mathit{negligible}(N), & \text{if } |d| \leq 0.147. \\ \mathit{small}(S), & \text{if } 0.147 < |d| \leq 0.33. \\ \mathit{medium}(M), & \text{if } 0.33 < |d| \leq 0.474. \\ \mathit{large}(L), & \text{if } 0.474 < |d| \leq 1. \end{cases}$$

6.3.3 Findings: **There exist statistically significant differences between the performance testing results from different scenarios.** Table 3 shows that all pairs of scenarios across our three workloads have statistically significant differences. In particular, all of the differences have medium or large effect sizes. We further examined the performance testing results and found that, with a higher workload, the performance, e.g., CPU usage and response times, often has a larger variance (see Figure 2). Such larger variance illustrates the uncertainty in such execution environments of microservices when the workload is high, which may require more resources from the execution environment. Hence, under a higher workload, performance test results of microservices become less reliable, requiring even more repetitions to improve the test result confidence.

Table 3: Comparing response time and total CPU time between different scenarios under the same workload (all distributions consist of ten elements, one for each repetition of the scenario).

		Response time			CPU		
700/sec		p-value	Eff. size		p-value	Eff. size	
Default	Balanced	0.01	0.60	(L)	0.06	—	
Balanced	LargeVMs	0.04	0.46	(M)	0.15	—	
Default	LargeVMs	0.00	0.82	(L)	0.00	0.76	(L)
		Response time			CPU		
800/sec		p-value	Eff. size		p-value	Eff. size	
Default	Balanced	0.00	0.76	(L)	0.15	—	
Balanced	LargeVMs	0.02	0.56	(L)	0.00	0.80	(L)
Default	LargeVMs	0.00	1.00	(L)	0.00	0.98	(L)
		Response time			CPU		
900/sec		p-value	Eff. size		p-value	Eff. size	
Default	Balanced	0.00	0.80	(L)	0.43	—	
Balanced	LargeVMs	0.02	0.56	(L)	0.00	0.98	(L)
Default	LargeVMs	0.00	1.00	(L)	0.00	1.00	(L)

The total CPU busy time may not be statistically significantly different between scenarios. Although there exist statistically significantly different response times between scenarios, the differences between CPU busy time do not necessarily have to be. Table 3 and Figure 4 show that the differences in the CPU busy time of the Default and Balanced scenarios are never (in our experiments) significant, regardless of the workload. Hence, our observed different response times may not be fully due to the lack of computational resources, but rather due to the deployment of the microservices and the environment in which they execute.

Autoscaling is optimized for utilization, leading to higher response times on the system. We found that, by enabling autoscaling, the number of engaged nodes is low. In particular, we find that in all our runs, all of the services except for the WebUI do not provision all eight allowed service instances. For example, Table 2 shows that the *Recommender* service sometimes only provisions one service instance out of the total eight instances that are available. This autoscaling strategy is optimized for utilization. In particular, we find that the total CPU busy time in the autoscaling scenario is not statistically different from other scenarios, while the response time of the system is considerably higher (see Figure 2). On the one hand, this strategy eases the management and lowers the cost of the users of the cloud services while it does not consider the impact on the end users, leading to sub-optimal performance.

> *Although the CPU busy time may not be significantly different between scenarios, there often exist statistically significant differences in response time. Such differences may have a direct negative impact on the user-perceived performance, and makes the analysis of performance test results of microservices more challenging.*

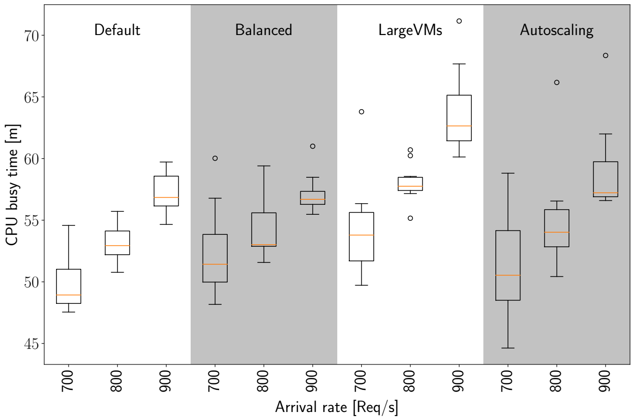


Figure 4: CPU busy time for four scenarios and three load-levels each (N=10).

6.4 RQ3: How well can performance regressions in microservices be detected?

6.4.1 Motivation: A common use case of performance testing is to detect performance regression, i.e., unintended changes of a software system's performance-related behavior between two or more versions [2, 6, 18]. Typical symptoms for performance regressions include increased response times and resource usage, or decreased throughput. The root causes of regressions can be many-fold. Examples include the introduction of software performance anti-patterns, such as chatty services, software bottlenecks and memory leaks. Typical metrics for evaluating the quality of regression techniques include false/true positive/negative rates, as well as precision and recall. In this research question, we study to what degree the regression detection quality is impacted by the underlying variance in the infrastructure and performance testing results that was demonstrated in the previous research questions.

6.4.2 Approach: To determine if performance regressions can be detected with a single execution of a performance test, we compared the response time distributions of single measurement runs. Algorithm 1 demonstrates how we defined the 'golden standard' for the runs that are selected for comparison. In summary, we did a pairwise comparison of all possible pairs of runs of the Regression-Baseline, and the 10% Regression and 30% Regression scenarios at the same load level.

First, each baseline measurement is paired with every other baseline measurement at the same load level and labeled as False, which indicates that there was no regression. Next, each baseline measurement is compared to every measurement of the selected regression scenario at the same load level and labeled as True, which indicates that there was a performance regression. We then employ three regression detection approaches to investigate how well they can identify regressions using only data from single experiment runs.

We applied three tests for comparing distributions to investigate whether the performance regression could be detected. We used the Wilcoxon signed rank [52], the two-sampled Kolmogorov-Smirnov [46] and the Anderson-Darling [3] tests to compare the

Algorithm 1 Defining the golden standard for deciding whether one of the two experiment runs is from a scenario in which a regression occurred.

```
function getIsRegressionPairs($regrBaseline$, $regr$)
    $isPairRegression$ = []
    for $load$ in [700, 800, 900] do
        for $i$ = 1; $i \leq 10$; $i$++ do
            for $j$ = 1; $j \leq 10$; $j$++ do
                if $i < j$ then
                    $isPairRegression$.append(
                                  [$regrBaseline[load,i]$,
                                  $regrBaseline[load, j]$,
                                  False])
                end if
                $isPairRegression$.append(
                                  [$regrBaseline[load,i]$,
                                  $regr[load,j]$,
                                  True])
            end for
        end for
    end for

    return $isPairRegression$
end function
```

distributions. With these statistical tests, two distributions are considered different if the calculated test statistic exceeds a certain threshold. To analyse the impact of different threshold values on the regression detection accuracy, we calculate a Receiver Operating Curve (ROC) curve based on the test statistics. Figure 6 shows the ROC curve.

Next, we investigated if the performance regression could be accurately detected if we collect the data from ten experiment runs rather than a single one. We compared the ten measurement runs of the RegressionBaseline scenario to the ten measurement runs of the 10% and 30% regression scenarios. We applied the same test as in RQ2 (the Mann-Whitney U test) for comparing the distributions to investigate whether the performance regression could be detected. We assume that we can only detect the performance regression if the distributions of the responses times before and after injecting the regression are considered significantly different. Table 4 shows the results of the performance regression analysis for ten experiment runs.

6.4.3 Findings: **The results for the non-regression scenarios also apply to the regression scenario.** The results of RQ1 showed that the deployments across VMs differ for the LargeVMs scenario. The results for the Regression scenario show the same effect. Figure 5 shows the response times for the Regression scenario. It can be observed that, according to the findings in RQ2, the response times —and their variance— increase with increasing workloads.

Using only a single experiment run results in flaky performance tests. Figure 6 shows the ROC curve of the detection accuracy of the three tests that we use for regression detection. As shown by Figure 6, the area under the ROC curve (AUC) is between 0.86 and 0.90 for the 10% Regression scenario and between 0.96 and 0.98 for the 30% Regression scenario. While these AUCs are fairly high (a perfect regression detection would have an AUC of 1), the regression detection mechanisms suffer from flakiness. In particular, even the best-performing regression detection approach has a false-positive rate of 0.2 for the relatively small regression of 10%, meaning that the outcome of the performance test is wrong approximately one out of five times even for these 30 minute long tests. Test flakiness is a serious problem for regression testing, as the non-deterministic output of the tests reduces the applicability of such tests in automated testing processes, such as those necessary for continuous integration [35]. Table 4 shows that the regressions can be detected reliably when increasing the number of test runs to ten. However, this corresponds to 5h of measurements. The required number of test runs is system-dependent and may require to be empirically defined (which is costly for performance tests).

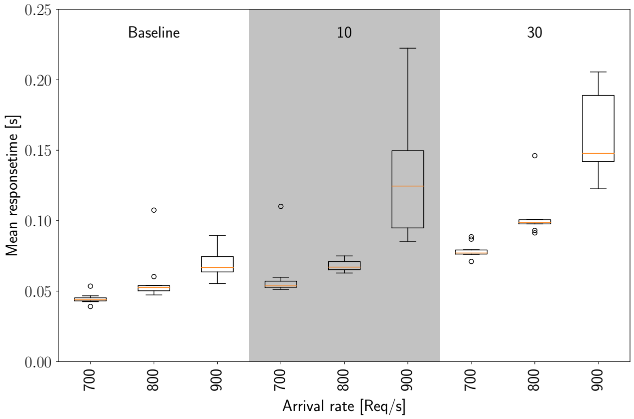


Figure 5: Mean response time for the regression baseline, 10% regression and 30% regression scenarios at three load-levels each (N=10).

Table 4: Comparing the distributions of the response time and total CPU time between different scenarios with regression between the LargeVMs and 10% Regression scenario (all distributions consist of ten elements, one for each repetition of the scenario—hereafter identified as N=10).

Load [Req/s]	Response time			CPU Utilization		
	p-value	Eff. size		p-value	Eff. size	
700	0.00	1.00	(L)	0.00	1.00	(L)
800	0.00	1.00	(L)	0.00	1.00	(L)
900	0.00	1.00	(L)	0.00	1.00	(L)

> *It is possible to detect performance regressions in microservices applications. However, one needs to ensure that enough repetitions of the performance tests are done to deal with the variance in the deployment of the services and the environments in which they run.*

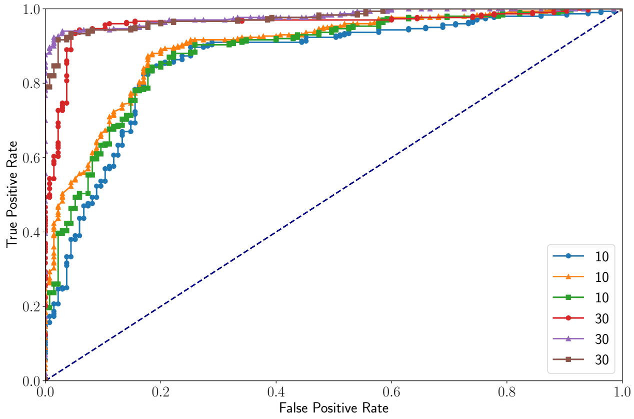


Figure 6: ROC curve showing the detection accuracy for the 10% and 30% regression.

7 SOLVING/ADDRESSING THE PERFORMANCE TESTER'S NIGHTMARE

Based on the results presented in Section 6, we discuss challenges that exist in the performance testing of microservice-based applications in Section 7.1 and propose how to address the challenges in Section 7.2.

7.1 Nightmares

Nightmare 1: Stability of the environment: Although it is one of the most important requirements for successful performance testing (R1), it is very difficult to ensure that a microservice always executes under the same circumstances. Our analysis (RQ1) shows that the number of provisioned services, as well as their deployment over the virtual machines, can differ between experiment runs. Hence, the *execution environment of microservices cannot be expected to be stable*, even with a constant number of provisioned instances. Such instability makes it very difficult to decide whether performance issues stem from the underlying execution environment or the application itself.

Cloud providers may also make changes in their infrastructure, both hardware and software, causing the instability of the environment, as we encountered in our experiment. Namely, the Kubernetes was updated from the version *GKE 1.11.6-gke.2* in the first experiments, to version *1.12.8-gke.10* for regression experiments, which was beyond our control. Beside software updates, there might be changes in the hardware. This can later cause deploying virtual machines on different hardware between experiments.

Nightmare 2: Reproducibility and repeatability of the experiments. The *repeated experiment runs may not result in the same performance measurements* when testing microservice-based applications. Causes for this, as shown previously (see Nightmare 1) can come from the unstable environment, but also from microservices themselves. Microservice architectures are inherently dynamic and contain dynamic elements, e.g., scaling mechanisms, load balancers, and circuit breakers. Scaling mechanisms take care that there are always enough instances to serve the incoming workload. When a component crash occurs, orchestration frameworks simply start a new microservice instance, while the traffic is directed to currently running instances. Thanks to these mechanisms, microservice-based applications can survive such incidents and be resilient but leaves traditional performance testing techniques challenges to cope with. Although statistical analysis can be leveraged to ease addressing such challenges, our results from RQ2 highlight the challenges when facing a rather small magnitude of regression (see Figure 6).

Nightmare 3: Detecting small changes. The variation in test results presents an even bigger issue when *performing regression testing and trying to detect small changes.* Figure 2 shows that with a higher load, i.e., 900 req/s arrival rate, the variation of mean response time becomes much larger than being under a lower load. Such high variation makes it necessary to execute the same test a larger number of repetitions to achieve a successful regression detection with statistical significance. However, the requirement of a larger number of repetitions would require more resources and may pose an additional delay in detecting the regression in practice.

7.2 Research directions

Direction 1: Variation reduction in executing performance tests. As explained in Section 7.1, there may exist a large variation in the execution environment and performance of microservice-based applications. Hence, it is important to reduce this variation as much as possible. A straightforward approach is to start many instances of a microservice and take aggregate performance measurements across these instances (e.g., using the median). Other possibilities include running performance tests in a more controlled environment or randomizing test order. Future studies should investigate how well such approaches function for microservice-based applications. In addition, future studies are necessary to investigate how successful existing approaches for mitigating the effect of changing/unstable environments on performance tests, such as those by Foo et al. [18], can be applied to microservices.

Direction 2: Studying the stability of (new) performance metrics. As we show in Section 7.1, it is difficult to achieve repeatable performance test results with microservice-based applications, as the application's environment and deployment configuration may be ever-changing and the results could be considerably different due to a plethora of factors. Therefore, future research should (1) investigate the applicability of analyzing traditional performance metrics during performance testing in a microservice-oriented environment (e.g., in terms of their stability), and (2) search for new metrics that can be used to assess the performance of such an environment. For example, Kubernetes is often used in microservice-based applications and offers access to several microservice and container-level metrics [42]. Such container-level metrics could be less susceptible to variation than response-time based metrics. Future research should investigate whether such metrics can be beneficial for better analyzing performance test results of microservice-based applications.

Direction 3: Creating a benchmark environment for microservice-oriented performance engineering research. One of the main challenges in evaluating software performance engineering research is to deploy a testing environment and application and to test it using a representative workload. For traditional performance engineering research, several benchmark applications exist, e.g., RUBiS [10] and the DaCapo [7] benchmarks.

To ease the evaluation of performance engineering approaches and to allow comparison between these approaches, a benchmark environment that takes into account the properties of microservice-based applications is crucial [45]. Microservice-oriented applications, e.g., the Sock Shop[5] or Tea Store[6] applications, can be considered as a starting point for building such benchmark applications, together with the CASPA platform [15], which can provide workload and monitoring.

8 THREATS TO VALIDITY

In this section, we discuss the construct, external and internal threats to the validity of our study [53]. Construct validity considers how well the measurements relate to the research question, external validity considers the generalizability of the results and internal validity considers the causality between the experiment settings and the outcomes, i.e., how well the experimental conditions are controlled.

8.1 Construct

In our experiments, we used limited sets of configurations of the TeaStore (Default, Balanced, LargeVMs, Autoscaling), load levels (700, 800, and 900 requests per second), types and intensities of injected regressions (active delay with 10 % and 30 % of the base response time) and the studied performance metrics (response time and CPU usage). This may have an impact on the observations regarding the research questions. Selecting every possible combination of configurations is infeasible. We have selected representative configurations that provide a good coverage of the overall system behavior and performance. Regarding the metrics, we have selected those that are commonly used in load test (regression) experiments. Future studies should investigate further how our findings apply to other configurations, load levels, regressions and performance metrics.

8.2 External

For our experiments, we have used a single system under test — namely the TeaStore [51] — and a single cloud environment —namely GKE. The reason for conducting the experiments was to illustrate the challenges that users experience when conducting performance tests for microservice-based applications in the cloud. We chose the TeaStore as it is one of the few available reference microservice applications for performance testing [51], and GKE because it is one of the popular engines for deploying microservices. We are convinced that the overall conclusions can be transferred to other systems and platforms, but the quantitative results will certainly not be transferable. Our peer-reviewed replication package [16] allows to conduct follow-up experiments by us and other researchers in the future. Future studies should investigate whether our findings apply to other microservice-based applications and cloud environments.

8.3 Internal

Performance experiments in the cloud always impose a high degree of uncertainty due to the performance variability of cloud resources. This performance variability has been discussed in a number of existing studies [29, 37]. To mitigate this threat, we followed recommended practices for conducting and reporting cloud experiments [39]. In addition, the uncertainty in cloud performance is a major reason for the challenges and nightmares that are reported in this paper.

9 CONCLUSION AND RESEARCH VISION

Microservices are a popular trend in software development for building scalable, large-scale software systems. While some of the characteristics of microservices make the lives of performance testers easier, others turn their (performance testing) lives into a nightmare.

In this paper we laid out the most important benefits and challenges that arise when performance testing microservices. Through several experiments on the TeaStore, a reference microservice application, we demonstrated the challenges that come with performance testing microservices.

The most important findings of our paper are:

- It is difficult to ensure that a microservice always executes under the same circumstances. Even when the total number of provisioned instances is kept constant, the execution environments of microservices can be different across repeated runs of a task (Section 6.2).
- The analysis of performance test results of microservices is challenging, since the user-perceived performance may be worse while this is not necessarily shown by all performance metrics (Section 6.3).
- It is possible to detect performance regressions in microservices, however, one needs to ensure that enough repetitions of the performance tests are done (Section 6.4).

In summary, we show that performance testing of microservice-based applications is challenging, and requires additional care when setting up test environments or conducting tests in comparison to 'traditional' software applications. This paper motivates the need for several research directions (in Section 7.2). In particular, future studies should investigate how to reduce the variation in performance test execution, and propose new, more stable performance metrics that can be used reliably in microservice-based applications.

ACKNOWLEDGMENTS

This research was partly funded by the German Federal Ministry of Education and Research (grant no. 01IS17010, ContinuITy) and the European Union's Horizon 2020 research and innovation programme (grant no. 825040, RADON), and supported by the Google Cloud Platform research credits program. The work was conducted by the SPEC RG DevOps Performance Working Group.[7] In addition, we would like to thank Holger Knoche for his feedback on early versions of this paper.

[5] https://microservices-demo.github.io/

[6] https://github.com/DescartesResearch/TeaStore

[7] https://research.spec.org/devopswg

REFERENCES

[1] Carlos M. Aderaldo, Nabor C. Mendonça, Claus Pahl, and Pooyan Jamshidi. 2017. Benchmark requirements for Microservices Architecture Research. In *Proc. 1st IEEE/ACM International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering, (ECASE@ICSE '17)*. IEEE, 8–13.

[2] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 2016. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report. In *International Conference on Mining Software Repositories (MSR)*. ACM, 1–12.

[3] T. W. Anderson and D. A. Darling. 1952. Asymptotic Theory of Certain 'Goodness of Fit' Criteria Based on Stochastic Processes. *Ann. Math. Statist.* 23, 2 (06 1952), 193–212.

[4] Muhammad Moiz Arif, Weiyi Shang, and Emad Shihab. 2018. Empirical study on the discrepancy between performance testing results from virtual and physical environments. *Empirical Software Engineering* 23, 3 (2018), 1490–1518.

[5] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective.* Addison-Wesley.

[6] Cor-Paul Bezemer, Elric Milon, Andy Zaidman, and Johan Pouwelse. 2014. Detecting and Analyzing I/O Performance Regressions. *Journal of Software: Evolution and Process (JSEP)* 26, 12 (2014), 1193–1212.

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.* 41, 10 (2006), 169–190.

[8] André B. Bondi. 2014. *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice.* Addison-Wesley Professional.

[9] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. 2016. Workload Characterization: A Survey Revisited. *Comput. Surveys* 48, 3 (2016), 1–43.

[10] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. 2002. Performance and Scalability of EJB Applications. *SIGPLAN Not.* 37, 11 (2002), 246–261.

[11] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2017. Analytics-driven Load Testing: An Industrial Experience Report on Load Testing of Large-scale Systems. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 243–252.

[12] D. E. Damasceno Costa, C. Bezemer, P. Leitner, and A. Andrzejak. 2019. What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering* (2019), 1–14.

[13] Sandeep Dinesh. 2018. Kubernetes best practices: Resource requests and limits. https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits.

[14] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, Cham, 195–216.

[15] Thomas F. Düllmann, Robert Heinrich, André van Hoorn, Teerat Pitakrat, Jürgen Walter, and Felix Willnecker. 2017. CASPA: A Platform for Comparability of Architecture-Based Software Performance Engineering Approaches. In *Proc. IEEE International Conference on Software Architecture (ICSA 2017) Workshops*. IEEE, 294–297.

[16] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. 2019. Microservices: A Performance Tester's Dream or Nightmare? - Replication package. https://doi.org/10.5281/zenodo.3582707.

[17] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. 2016. Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing* 3, 5 (2016), 10–14.

[18] King Chun Foo, Zhen Ming (Jack) Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2015. An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments. In *Proc. of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE, 159–168.

[19] R. Gao, Z. M. Jiang, C. Barna, and M. Litoiu. 2016. A Framework to Evaluate the Effectiveness of Different Load Testing Analysis Techniques. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 22–32.

[20] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. 2019. A Statistics-based Performance Testing Methodology for Cloud Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 188–199.

[21] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance Engineering for Microservices: Research Challenges and Directions. In *Companion 8th ACM/SPEC on International Conference on Performance Engineering (ICPE 2017)*. ACM, 223–226.

[22] Nikolas Roman Herbst, Samuel Kounev, and Ralf H. Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proc. 10th International Conference on Autonomic Computing (ICAC'13)*. 23–27.

[23] International Organization for Standardization (ISO). 2005. ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE).

[24] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.

[25] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1091–1118.

[26] Holger Knoche. 2016. Sustaining Runtime Performance While Incrementally Modernizing Transactional Monolithic Software Towards Microservices. In *Proc. 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16)*. ACM, 121–124.

[27] Christoph Laaber and Philipp Leitner. 2018. An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment. In *Proc. Proc. 15th International Conference on Mining Software Repositories (MSR '18)*. IEEE.

[28] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software Microbenchmarking in the Cloud. How Bad is It Really? *Empirical Softw. Engg.* 24, 4 (Aug. 2019), 2469–2508.

[29] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software microbenchmarking in the cloud. How bad is it really? *Empirical Software Engineering* 24, 4 (2019), 2469–2508.

[30] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proc. 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, 373–384.

[31] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos – A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.* 16, 3, Article 15 (2016), 1–23 pages.

[32] James Lewis and Martin Fowler. 2014. Microservices. Retrieved June 3, 2018 from https://martinfowler.com/articles/microservices.html

[33] David J. Lilja. 2005. *Measuring Computer Performance: A Practitioner's Guide.* Cambridge University Press.

[34] Jeffrey D. Long, Du Feng, and Norman Cliff. 2003. *Ordinal Analysis of Behavioral Data.* John Wiley & Sons, Inc.

[35] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 643–653.

[36] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.* 18, 1 (03 1947), 50–60.

[37] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming performance variability. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 409–425.

[38] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media, Inc., Sebastopol, California, USA.

[39] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, P. TÅŕma, and A. Iosup. 2019. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. *IEEE Transactions on Software Engineering* (2019), 1–1.

[40] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. 2018. Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective. *IEEE Software* 35, 3 (2018), 36–43.

[41] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices. In *Annual meeting of the Southern Association for Institutional Research.*

[42] Jean-Mathieu Saponaro. 2016. Monitoring Kubernetes performance metrics. Retrieved June 6, 2018 from https://www.datadoghq.com/blog/monitoring-kubernetes-performance-metrics/

[43] Joel Scheuner and Philipp Leitner. 2018. A Cloud Benchmark Suite Combining Micro and Applications Benchmarks. In *Companion 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. 161–166.

[44] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2015. Automated Detection of Performance Regressions Using Regression Models on

Clustered Performance Counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 15–26.

[45] S. Elliot Sim, Steve Easterbrook, and Richard C. Holt. 2003. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. 74–83.

[46] Nikolai V Smirnov. 1939. Estimate of deviation between empirical distribution functions in two independent samples. *Bulletin Moscow University* 2, 2 (1939), 3–16.

[47] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. 2017. Unit Testing Performance in Java Projects: Are We There Yet?. In *Proc. of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, 401–412.

[48] Alexandru Uta and Harry Obaseki. 2018. A Performance Study of Big Data Workloads in Cloud Datacenters with Network Variability. In *Companion 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. 113–118.

[49] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. 2018. WESSBAS: Extraction of Probabilistic Workload Specifications for Load Testing and Performance Prediction—A Model-Driven Approach for Session-Based Application Systems. *Softw. and Syst. Modeling* 17, 2 (2018), 443–477.

[50] Jóakim von Kistowski, Maximilian Deffner, and Samuel Kounev. 2018. Run-time Prediction of Power Consumption for Component Deployments. In *Proceedings of the 15th IEEE International Conference on Autonomic Computing (ICAC 2018)*.

[51] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)*.

[52] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.

[53] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering*. Springer.

[54] Pengcheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. 2013. vPerfGuard: An Automated Model-driven Framework for Application Performance Diagnosis in Consolidated Cloud Environments. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*. ACM, New York, NY, USA, 271–282.