A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing

Alberto Avritzer[1], Vincenzo Ferme[2], Andrea Janes[3], Barbara Russo[3], Henning Schulz[4], and André van Hoorn[2]

[1] EsulabSolutions, Inc., Princeton, NJ, USA
[2] University of Stuttgart, Germany
[3] Free University of Bozen-Bolzano, Italy
[4] NovaTec Consulting GmbH, Leinfelden-Echterdingen, Germany

Abstract. Microservices have emerged as an architectural style for developing distributed applications. Assessing the performance of architectural deployment alternatives is challenging and must be aligned with the system usage in the production environment. In this paper, we introduce an approach for using operational profiles to generate load tests to automatically assess scalability pass/fail criteria of several microservices deployment alternatives. We have evaluated our approach with different architecture deployment alternatives using extensive lab studies in a large bare metal host environment and a virtualized environment. The data presented in this paper supports the need to carefully evaluate the impact of increasing the level of computing resources on performance. Specifically, for the case study presented in this paper, we observed that the evaluated performance metric is a non-increasing function of the number of CPU resources for one of the environments under study.

1 Introduction

The microservices architectural style [13] is an approach for creating software applications as a collection of loosely coupled software components. These components are called microservices, and are supposed to be autonomous, automatically and independently deployable, and cohesive [13]. This architecture lends itself to decentralized deployment, and for continuous integration and deployment by developers. Several large companies (e.g., Amazon and Netflix) are reporting significant success with microservice architectures [9].

Currently, several deployment alternatives are possible for microservices deployment, as for example, serverless microservices using lambdas, container-based deployment (e.g., Docker[5]), virtual machines per host, and several hosts. Of course, depending on the microservice granularity, a combination of these deployment mechanisms could be used. The available deployment alternatives

[5] `https://www.docker.com/`

To appear in
Proceedings of the 12th European Conference on Software Architecture (ECSA 2018)

and their configuration parameters imply a large space of architectural configurations [15] to choose from.

Microservices are supposed to be independent from each other. However, the underlying deployment environment might introduce coupling and impact the overall application performance. Coupling can occur at the load balancer, at the DNS look-up, and at the different hardware and software layers that are shared among the microservices. Ueda et al. [16] report the performance degradation of microservice architectures as compared to an equivalent monolithic deployment model. The authors have analyzed the root cause of performance degradation of microservice deployment alternatives (e.g., due to virtualization associated with Docker) and have proposed performance improvements to overcome such a degradation. Therefore, microservice architects need to focus on the performance implications of architectural deployment alternatives. In addition, the impact of the expected production workloads on the performance of specific microservices deployment configuration needs to be taken into account. The alternatives for microservice architecture deployment considered in this paper are memory allocation, CPU fraction used and number of Docker container replicas assigned to each microservice.

In this paper, we introduce a quantitative approach for the performance assessment of microservice deployment alternatives. The approach uses automated performance testing results and high-level performance modeling to quantitatively assess each architectural configuration in terms of a domain metric introduced in this paper. For performance testing, we focus on load tests based on operational workload situations [10,17], e.g., arrival rates or concurrent number of users.

Our approach for the domain metric evaluation is based on the input domain partition testing strategy [19] and domain-based load testing [5] that was designed for the performance testing of telecommunication systems. In partition testing based on input domains, the input domain is divided into subsets that have equivalent fault-revealing behavior. In domain-based load testing the load testing domain is divided into subsets that have equivalent workload situations [5].

Operational profile data is used to estimate the probability of occurrence of each operational workload situation in production. Each operational situation is reflected by a performance test case that is weighted by its relevance based on the operational profile. Scalability requirements are used to assess each architectural configuration. The resulting quantitative assessment is a measure between 0–1 that assesses the fitness of a certain architecture alternative to perform under a defined workload situation.

The key contributions of this paper are as follows:

- a new quantitative approach for the assessment of microservice deployment alternatives, and
- the experimental validation of the proposed approach.

We have evaluated the introduced domain metric for ten different configurations based on two different memory allocations, two different CPU allocations,

and three different values for the number of Docker container replicas. The experiments were executed in two data center environments. We evaluated for each environment the best performing architectural configuration. It is very significant that in both environments, increasing the number of containers for the service being evaluated, or the fraction of CPU allocation did not guarantee better performance. Therefore, we can conclude that it is very important for practitioners to carefully assess expected operational profiles and deployment alternatives of their applications using quantitative assessment approaches, such as the one introduced in this paper.

The remainder of this paper is organized as follows. Section 2 contains a summary of the reviewed literature on microservice architecture challenges and performance assessment. Section 3 contains an overview of the proposed approach for performance assessment of microservice architectures. Section 4 contains the experimental design, while Section 5 presents the experimental results obtained by applying the proposed approach. Section 6 contains the threats to validity identified in this research. Section 7 presents our conclusions and suggestions for future research. A reproducibility package is provided online [4].

2 Related Work

In this section we present a summary of the reviewed literature on microservice architecture challenges.

2.1 Microservice Architectural Challenges

Alshuqayran et al. [2] present a comprehensive literature review of microservice architectural challenges. The authors focus on the challenges, the architecture descriptions, and their quality attributes. They have found that most of the current research on microservice architecture quality attributes has focused on scalability, reusability, performance, fast agile development, and maintainability. Pahl and Jamshidi [14] present a systematic survey of the existing research on microservices and their application in cloud environments. They have found that microservices research is still immature and there is a need for additional experimental and empirical evaluation of the application of microservices to cloud environments. Their literature survey has also identified the need to develop microservices tool automation.

Francesco et al. [9] present a characterization of microservice architecture research. The authors' focus was on answering research questions about publication trends, research focus, and likelihood for industrial adoption. They reported that research on microservices is in the initial phases concerning architecture methodologies and technology transfer from academia to industry. Most of the research focus seems to be on architecture recovery and analysis. An important finding of this paper is that most of the literature reviewed is related to the design phase, and only one research did address requirements. They have

also found from their literature survey that industrial technology transfer of the architecture methodology is still far-off.

Esposito et al. [7] present design challenges of microservice architectures. The authors have identified security and performance as major challenges resulting from size and complexity. They have proposed to address these challenges by carefully trading-off security and performance.

2.2 Performance Assessment of Microservice Architectures

Kozhirbayev and Sinnott [11] present the performance assessment of microservice architectures in a cloud environment using several container technologies. The authors have reported on the experimental design and on the performance benchmarks that were used for this performance assessment. Casalicchio and Perciballi [6] analyze the impact of using relative and absolute metrics to assess the performance of autoscaling containers. They have concluded that for CPU-dominated workloads, the use of absolute metrics can lead to better scaling decisions.

McGrath and Brenner [12] present an approach for the design of a performance-oriented serverless computing platform. The authors have evaluated the performance of their approach using measurements derived from a prototype. They have also discussed how to achieve increased throughput using their approach.

3 Approach

In this paper, we extend our previous approaches [5,18] to define a new methodology for the assessment of microservice deployment alternatives — also referred to as *(architectural) configurations*. Our methodology enables an automatic assessment of scalability criteria for architectural configurations and their comparison. For each configuration, this results in a measure—the so-called domain-metric—that quantifies the configuration's ability to satisfy scalability requirements under a given operational profile.

3.1 Summary of Previous Work

In [18], we introduced a metric to evaluate software architecture alternatives with system workload growth. This metric uses the requirement definition, high-level architecture modeling, and system measurement results to assess the system architecture's ability to meet architecture requirements as a function of workload increases.

In [5], we introduced an approach for the assessment of telecommunication systems using Markovian approximations. This approach uses operational data and a resource-based Markov state definition to derive an efficient test suite that is then used as the basis for the domain-based reliability assessment of the software under test (SUT). The Markovian approximation is used to estimate the steady-state probability of occurrence of each test case. In this way, the

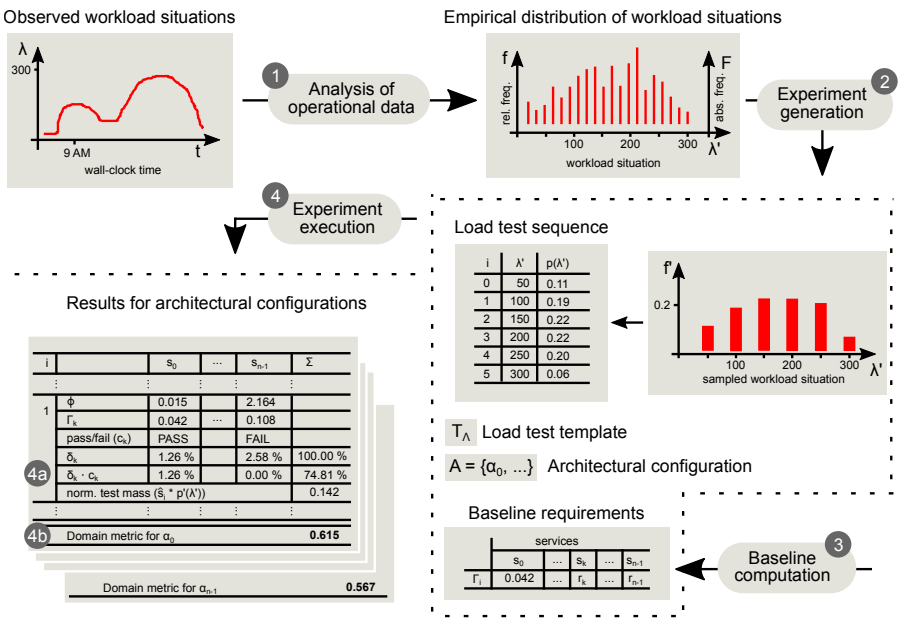

Fig. 1: Overview of the approach

test suite can be effectively reduced to focus on the performance test cases that are most-likely to represent production usage. In domain-based load testing, the input domain is the workload, e.g., in terms of the arrival rate or the concurrent number of users. The total workload is divided into subsets that are related to the probability of occurrence of each workload situation [5]. Therefore, we make an implicit assumption that the testing domain can be divided into fault-revealing subsets. Each subset is then mapped to a load test.

3.2 Computation of the Domain-based Evaluation Metric

The approach introduced in this paper and illustrated in Figure 1 consists of the following steps: 1. *Analysis of operational data*, i.e., the quantitative estimation of the probability of occurrence of a certain workload situation (e.g., number of concurrent users) based on the analysis of the operational data, 2. *Experiment generation*, i.e., the automated generation of the load test cases for the deployment configurations under evaluation, 3. *Baseline computation*, i.e., the quantitative definition of the scalability requirements that consist of the expected pass/fail criteria for the load tests, e.g., based on a specified threshold of the system response time for the expected workload, 4. *Experiment execution*, i.e., the execution of load test cases for the architectural configurations specified in the experiment generation step, and the computation of the domain-based microservice architecture evaluation metric.

In this section, we illustrate the approach with a running example, which is based on the SUT and the experiments from the evaluation in this paper (Sections 4 and 5). The operational profile is taken from publicly available information about a video streaming service.

Step 1—Analysis of Operational Data – This step relies on operational data that includes the workload situations Λ observed over time, i.e., each point in time t_k is assigned a workload situation $\lambda_i \in \Lambda$. We make no specific assumptions about what metric is used to represent the workload situation. Example metrics include the number of concurrent users or arrival rates of requests. We use this operational profile data to estimate for each workload situation $\lambda_i \in \Lambda$ its probability of occurrence $p(\lambda_i)$ estimated by the relative frequency of occurrence $f(\lambda_i)$. The probability of occurrence will be used to weigh each test case execution result. This is called probability of occurrence because it provides a test coverage function for each test case with respect to the total operational profile probability distribution, which we denote by *total probability mass*.

To illustrate this step, we first analyze the operational profile data to create a user profile of the frequency of occurrence (i.e., state frequency) of the number of concurrent users (i.e., workload situation) found in the system at a certain time t. The graph of the state frequency distribution is shown in Figure 1 (result of Step 1). Then, for each workload situation λ_i, we set $p(\lambda_i)$ to the corresponding state frequency. The test suite coverage criteria is based on the values of $p(\lambda_i)$. Then, we use the operational profile obtained from the video streaming service as a proxy for the operational profile of the system being evaluated. We have scaled the number of concurrent users from the operational profile to 0–300.

Step 2—Experiment Generation – This step generates the load test suite to analyze the architectural configurations. The four elements of this step are the load test sequence, the load test template, the architectural deployment configurations, and the baseline requirements (see result of Step 2 in Figure 1). The load test sequence is obtained by sampling the empirical distribution of workload situations f into a so-called aggregated mass of workload situations f' that are representative for the neighboring workload situations. A value $f'(\lambda')$ represents the aggregated probability of neighboring workload situations of λ in f. The reason for having this aggregated mass based on sampling is that it would not be feasible to execute load tests for every single workload situation due to the huge combinatorial space of configurations. The load test template T_Λ is a load test specification that is parameterized by a workload situation $\lambda_i \in \Lambda$. An instance of this load test template will be executed for each element of the cross product of the set of load test sequences and the architectural configurations. The baseline requirement Γ_i defines for each service s_j provided by the SUT the criteria of a passed/failed test based on a performance measure Φ. We denote the concrete measurements for Φ for a service s_j under a workload situation λ as $x(\lambda)_j$. We make no specific assumption about the performance measure. An example used in this section is the average response time of a service.

Step 3—Baseline Computation, Quantitative Definition of the Scalability Requirements – We now describe the approach that can be used to

Table 1: Scalability requirements based on baseline measurements (in seconds)

s_j	createOrder	basket	getCatalogue	getItem	login	...
$x(\lambda')_j$	0.018	0.008	0.011	0.012	0.033	...
$\sigma(\lambda')_j$	0.008	0.003	0.002	0.009	0.025	...
Γ_j	0.042	0.017	0.017	0.039	0.108	...

calculate the fraction of correctly executed services $\hat{s}_i$ for test case i. Initially, a baseline performance test is run for each configuration $\alpha_k \in A$, similarly to the approach used in our previous work [18]. Such baseline is chosen with a starting workload situation $\beta \in \Lambda$. Then, the average response time, $x(\beta)_j$, and the standard deviation, $\sigma(\beta)_j$ for each service s_j, under the baseline workload β are measured.

Using the baseline performance measured for each service s_j, the scalability requirement is defined as $\Gamma_j = x(\beta)_j + 3 \times \sigma(\beta)_j$. Table 1 illustrates the measured results for the baseline measurement workload β, for all services s_j. This is an innovative approach for scalability requirements definition that employs the measured no-load baseline performance to automatically define a specific tolerance for scalability degradation under load.

The proposed approach for setting scalability requirements using a normal distribution follows an existing approach [3], where the normal distribution was shown to be a good approximation for the distribution of a stream of concurrent transactions.

Step 4a—Experiment Execution (Pass/Fail Assessment) – Next, each service s_j is tested under a certain workload $\lambda_i \in \Lambda$ and configuration $\alpha_k \in A$. Each test case execution produces a metric between 0–1 that represents the fraction of the service executions that was assessed as successful by comparison with the scalability requirement.

Each service s_j will be marked as pass for workload λ_i and configuration α_k, if $x(\lambda_i)_j < \Gamma_j$. In this case, $c_j = 1$ will be set to denote that service s_j has passed the test, otherwise $c_j = 0$ will be set.

In the following, we drop λ and the configuration α to simplify the notation, as these computations are repeated for each workload situation and configuration. In addition, as each test case i executes the set of n services $\{s_0, \ldots, s_{n-1}\}$ with activation rates $\{\delta_0, \ldots, \delta_{n-1}\}$, the fraction $\hat{s}_i$ of correctly executed calls to all services can be evaluated as:

$$\hat{s}_i = \sum_{j=0}^{n-1} \delta_j c_j \tag{1}$$

The activation rate δ_j denotes the fraction of calls to the service s_j over the overall number of calls to all services. Table 2 illustrates the pass/fail estimation for one workload situation λ. For this test case, the fraction of correctly executed services was evaluated as $\hat{s}_i = 74.81\%$ (Figure 1).

Table 2: Pass/fail based on scalability requirements (in seconds) for workload situation λ

s_j	createOrder	basket	...	login	...
Γ_j	0.042	0.017	...	0.108	...
$x(\lambda)_j$	0.015	0.009	...	2.164	...
Pass/fail	pass	pass	...	fail	...
δ_j	1.26 %	1.26 %	...	2.58 %	...

Step 4b—Experiment Execution (Computation of Domain-based Metric) – Finally, the domain-based architecture evaluation metric for the configuration α, with respect to a test suite S, $D(\alpha, S)$ can be evaluated as:

$$D(\alpha, S) = \sum_{i=0}^{z} p(\lambda_i)\hat{s}_i \tag{2}$$

where $p(\lambda_i)$ is the frequency of occurrence corresponding to workload situation λ_i (as in Step 1). For the running example, as illustrated in Figure 1, $D(\alpha, S)$ would be evaluated as 0.615. The contribution of the test case case depicted in Figure 1 is 0.142 ($0.19 \times 74.81\%$). The resulting quantitative assessment is a measure between 0–1 that can be used to assess the performance of different architectural deployment configurations.

4 Experiment Design

In our evaluation, we show how to use our approach as illustrated in Figure 1 to assess the scalability of an environment for a specific target system and its architectural alternatives by utilizing the domain metric. We use the operational profile from Step 1 (Section 3) and apply it to the Sock Shop microservices demo in two different environments. We execute the experiments generated by our Step 2 and compare the results against individual baselines as per Step 3. In doing so, we cannot only show the usage of our approach but also reveal interesting insights on scalability of microservice applications and its adoption in practice.

The remainder of this section describes the precise details of our experiment design. A reproducibility package is provided online [4].

4.1 System Under Test

As system under test (SUT), we utilize the most recent version of the Sock Shop microservices demo (as per March 28, 2018[6]) built by Weaveworks. It represents a sample e-commerce website that sells socks, implemented using 12 microservices, one of which is named *cart*, handling the user's shopping cart. For the

[6] `https://microservices-demo.github.io/`

implementation, various technologies were used, e.g., Java, .NET, Node.js and Go. The Sock Shop has been found to be a representative microservice application regarding several aspects [1]. For our research, the usage of well-known microservice architectural patterns, the automated deployment in containers and the support for different deployment options were the main criteria for selecting the Sock Shop as the SUT.

4.2 Load Testing Tool

As the load testing tool, we use BenchFlow [8], that is an open-source framework[7] automating the end-to-end process of executing performance testing. BenchFlow reuses and integrates state of the art technologies, such as Docker[8], Faban[9], and Apache Spark[10] to reliably execute load tests, automatically collect performance data, and compute performance metrics and statistics, as well as to validate the reliability of the obtained results.

BenchFlow users define their performance intent relying on a declarative domain-specific language (DSL) for goal-driven load tests by using provided declarative templates for expressing tests' requirements such as the test goals and test types, metrics of interest, stop conditions (e.g., maximum test execution time) and which parameters to vary during the execution of the test. To satisfy the user's goal, the BenchFlow framework implements strategies and processes to be followed that are driven by the user's input specification and current conditions of the SUT during the execution of those processes.

4.3 Testing Infrastructure

We deployed the load testing tool and the SUT to two different infrastructures. The first one supports containerized deployment to bare metal at the Hasso Plattner Institute (HPI) Future SOC (Service-Oriented Computing) Lab. The second one enables containerized deployment in virtual machines on top of the VMware ESXi[11] bare metal hypervisor at the Free University of Bozen-Bolzano (FUB).

The containerized bare metal machines (HPI) have the following characteristics: **Load driver server** — 32 GB RAM, 24 cores (2 threads each) at 2300 MHz and **SUT server** — 896 GB RAM, 80 cores (2 threads each) at 2300 MHz. Both machines use magnetic disks with 15 000 rpm and are connected using a shared 10 Gbit/s network infrastructure.

The containerized deployment in virtual machines (FUB) has the following characteristics: **Load driver server** — 4 GB RAM, 1 core at 2600MHz and **SUT server** — 8 GB RAM, 4 cores at 2600 MHz with SSDs. Both machines use

[7] `https://github.com/benchflow`
[8] `http://docker.com`
[9] `http://faban.org`
[10] `http://spark.apache.org`
[11] `https://www.vmware.com/products/esxi-and-esx.html`

an EMC VNC 5400 series network attached storage solution[12] and are connected using a shared 10 Gbit/s network infrastructure.

We rely on Docker CE v17.12 for the deployment of the containerized application on both infrastructures.

4.4 Architectural Deployment Configurations

By relying on BenchFlow's DSL [8], users can specify performance tests in a declarative manner. In our case, we defined a load test exploring different system configurations, as presented in Table 4 (on page 12). BenchFlow supports a wide range of variables to be automatically explored during configuration tests, namely: i.) number of simulated users, ii.) amount of RAM/CPU share assigned to each deployed service, iii.) service configurations, through environment variables, iv.) number of replicas for each service.

We rely on BenchFlow's DSL to define all the experiments reported in this section, and on the BenchFlow framework for their automated execution, test execution quality verification, and results retrieval.

Figure 2 depicts an example SUT deployment, showing one Docker container for each of the 11 microservices (i.e., 11 containers) and two Docker containers for the cart service, running on top of a Docker engine, which is deployed as a daemon process on the bare metal server at HPI and hypervisor at FUB. The figure shows that a container can have multiple replicas.

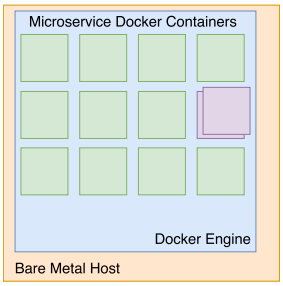

Fig. 2: Docker containers for each microservice (13 containers) running on top of the Docker engine deployed as a daemon process on the bare metal server

4.5 Design of Synthetic User Behavior

Even if we are not focusing on the behavior of an individual user, we need to generate a representative workload on the target system when evaluating its performance. Therefore, we model a synthetic user behavior that is replayed with different numbers of users during the experiments, as per our methodology, representing types of users that could utilize the Sock Shop in reality. We model

[12] `http://www.emc-storage.co.uk/emc-vnx-5400-emc-vnx5400-vnx5400-storage`

Table 3: Summary of requests and its numbers of occurrence in the user types (V=visitor, B=buyer, O=orders visitor) and actual overall workload mix

Label	Path	Method	V	B	O	Mix (%)
home	/index.html	GET	2	3	2	11.85%
login	/login	GET	0	1	1	3.21%
getCatalogue	/catalogue	GET	2	4	2	12.56%
catalogueSize	/catalogue/size?size={}	GET	1	1	0	3.07%
cataloguePage	/catalogue?page={}&size={}	GET	1	1	0	3.07%
catalogue	/category.html	GET	1	1	0	3.07%
getItem	/catalogue/{}	GET	1	5	1	8.42%
getRelated	/catalogue?sort={}&size={}&tags={}	GET	1	2	0	3.78%
showDetails	/detail.html?id={}	GET	1	2	0	3.78%
tags	/tags	GET	1	1	0	3.07%
getCart	/cart	GET	4	9	3	23.34%
addToCart	/cart	POST	0	1	0	0.71%
basket	/basket.html	GET	0	1	0	0.71%
createOrder	/orders	POST	0	1	0	0.71%
getOrders	/orders	GET	0	1	1	3.21%
viewOrdersPage	/customer-orders.html	GET	0	1	1	3.21%
getCustomer	/customers/{}	GET	2	5	1	10.78%
getCard	/card	GET	0	1	0	0.71%
getAddress	/address	GET	0	1	0	0.71%

the following behavior mix [17]: three types of users with the respective relative frequency, and a maximum allowed 5% deviation for the defined frequency distribution:

- *visitor* (40%): visits the home page, views the catalog and the details of some products.
- *buyer* (30%): visits the home page, logs in, views the catalog and some details, adds a product to the cart, visits the cart, and creates an order.
- *order visitor* (30%): visits the home page, logs in, and views the stored orders.

The summary of all requests sent to the Sock Shop and the occurrence of each requests in the user types are provided in Table 3. We set a workload intensity function [17] with 1 minute of ramp-up and 30 minutes of steady state, to ensure the system reaches the steady state and we collect reliable performance data. We have added a negative exponential think time, which is executed between every two requests, with 0, 1, and 5 seconds for minimum, mean and maximum think time respectively and an allowed deviation of 5% from the defined time.

4.6 Experiment Runs

We deployed the SUT using ten different architectural configurations per testing infrastructure. The parameters we vary over the different configurations are the

Table 4: Domain metric $D(\alpha, S)$ per configuration α = (RAM, CPU, # Cart Replicas) in the two environments (HPI, FUB). The configuration with the highest domain metric is highlighted.

RAM	CPU	# Cart Replicas	$D(\alpha, S)$ (HPI)	$D(\alpha, S)$ (FUB)
0.5 GB	0.25	1	0.614 99	0.541 34
1 GB	**0.25**	**1**	**0.77631**	0.538 84
1 GB	0.5	1	0.535 59	0.541 06
0.5 GB	0.5	1	0.515 36	0.547 73
0.5 GB	0.5	2	0.509 95	0.541 11
1 GB	0.25	2	0.740 80	0.547 85
1 GB	0.5	2	0.534 01	0.541 06
0.5 GB	***0.5***	***4***	0.505 31	***0.54939***
1 GB	0.25	4	0.371 62	0.542 72
1 GB	0.5	4	0.567 18	0.542 71

amount of available RAM, the CPU share, and the replicas for the cart service. We target the cart service, as most of the requests issued by the designed workload (see Section 4.5) target the cart service. The different configurations we explore are reported in Table 4. We set the RAM to [0.5 GB, 1 GB], the CPU share to $[0.25, 0.5]$, and the number of replicas to $[1, 2, 4]$.

The remaining resources of the server on which we deploy the SUT are shared among all the other services part of the Sock Shop application and managed by the Docker engine. In order to avoid containers to be "killed" during the execution in case of out-of-memory, we disabled this behaviour on the Docker engine.

By relying on the operational data presented in Section 3, we identified the following number of users interacting with the system, resembling aggregated workload situations for the system: 50, 100, 150, 200, 250, 300.

The baseline experiment (Step 3, Section 3), which we conduct to set a reference point for our methodology, sets the RAM to 4 GB, the CPU share to 1 and the replicas to 1 for the cart service, and measures the performance when 2 users interact with the system.

In total, we executed 122 experiments with different configurations.

5 Empirical Results

In this section, we describe and analyze the results of our experiments that are described in Section 4, as per Step 4 in Section 3. All the experiment results are available online [4].

5.1 Results

Figure 3 shows the test masses for the different investigated architectural configurations in relation to the workload situations Λ (numbers of users). The domain

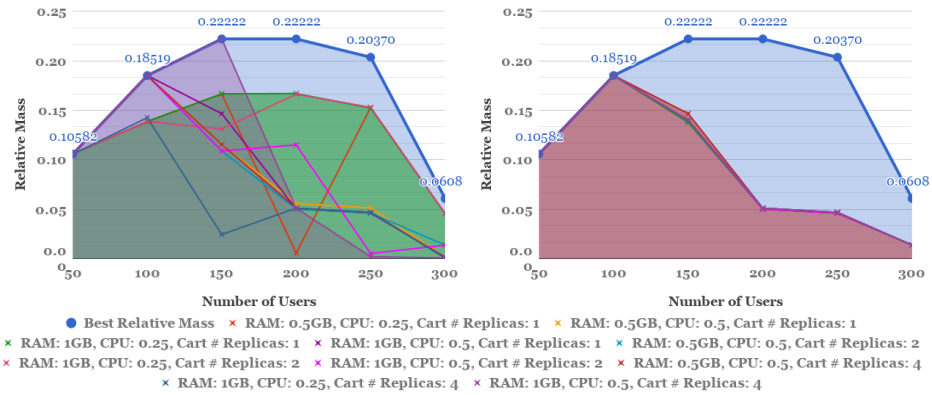

Fig. 3: Relative and best test masses per number of users in the two environments (HPI, FUB)

metrics $D(\alpha_i, S)$ for all configurations $\alpha_i \in A$ are provided in Table 4. The best relative test mass plot represents the theoretical maximum which is reached if all tests pass. It can be seen from Figure 3 that none of the alternatives reached the best relative mass, because of scalability assessment failures identified. For the HPI environment (bare metal), the configuration with 1 GB of RAM, 0.5 CPU share, and four cart replicas not have failures for up to 150 users. However, the relative mass decreases significantly when the number of users is increased. For the FUB environment (bare metal hypervisor), all configurations do not experience failures up to 100 users. After such load, the performance decreases with a similar rate.

For the HPI (bare metal) experiments, the configuration with 1 GB of RAM, 0.25 CPU share and one cart replica has the highest domain metric $D(\alpha, S) \approx 0.78$, followed by the configuration with 1 GB of RAM, 0.25 CPU share, and two cart replicas having a metric value of about 0.74. The worst configuration with $D(\alpha, S) \approx 0.37$ is 1 GB of RAM, 0.25 CPU share, and four cart replicas. This is an interesting result with significant implications to the assessment of architectural deployment alternatives, since adding additional replicas with the same memory and CPU configuration may decrease the application's performance for the HPI environment.

The results for the FUB experiments (VMware ESXi11 bare metal hypervisor), show a significant performance degradation as assessed by the domain metric, when compared to the HPI experiment. In addition, most of the experiment results are within a narrow domain metric range as can be seen from Figure 3 where most of the lines overlap. The configuration with 0.5 GB of RAM, 0.5 CPU share, and four cart replicas obtains the highest domain metric for the FUB experiments, with $D(\alpha, S) \approx 0.54$. The worst domain metric for the FUB experiment is for the 1 GB of RAM, 0.25 CPU share and one cart replica configuration. However, this configuration was assessed as the best configuration among the HPI experiments. The difference in the domain metric assessment between the HPI and FUB environments for the 1 GB of RAM, 0.25 CPU share,

and one cart replica configuration, seems to indicate that additional architecture factors may be impacting system performance, such as VMware Hypervisor overhead, I/O bandwidth, etc. These findings support the recommendation that practitioners have to evaluate the expected operational profile and deployment alternatives in their own context.

5.2 Analysis

Our results show that determining the best deployment configuration for an application requires the systematic application of quantitative performance engineering approaches.

We have found that adding more CPU power or increasing the number of Docker container replicas may not result in system performance improvement. As listed in Table 4, the best configuration at HPI is 1 GB of RAM, 0.25 CPU share and one cart replica, with the domain metric value of 0.77631. In addition, more cart replicas at HPI results in performance degradation. For example, the configuration with 1 GB of RAM, 0.25 CPU share, and four cart replicas, was assessed as $D(\alpha, S) = 0.37162$, while the configuration with 1 GB of RAM, 0.5 CPU share, and one cart replica, was assessed as $D(\alpha, S) = 0.55356$. At FUB, the domain metric oscillates over a narrow range. Scaling beyond 0.5 GB of RAM, 0.5 CPU share, and 4 cart replicas does not lead to a better performance if the number of users is higher than 150. In addition, the choice of the HPI or the FUB deployments have significant impact on the domain metric as shown in Table 4. These findings suggest that bottleneck analysis and careful performance engineering activities should be executed before additional resources are added to the architecture deployment configuration.

6 Threats to Validity

The following threats to validity to our research were identified:

Operational profile data analysis. The domain metric introduced in this paper relies on the careful analysis of production usage operational profile data. Many organizations will not have access to accurate operational profile data, which might impact the accuracy of the domain metric assessments. Several approaches can be used to overcome the lack of accurate operational profile data [5], such as: using related systems as proxy for the SUT, conducting user surveys, and analyzing log data from previous versions of the SUT.

Experiment generation. Experiment generation requires the estimation of each performance test case probability of occurrence, which is based on the operational profile data. When the operational profile data granularity is coarse there is a threat to the accuracy of the estimated operational profile distribution. Some of the suggested approaches to overcome the coarse granularity of the operational profile data are: performing the computation of operational profile data using analytic or simulation models [18], and developing heuristics based on Markovian approximations [5].

Baseline computation. The suggested approach for the quantitative definition of the scalability requirements proposed in this paper consisted of defining the expected pass/fail criteria for system scalability based on a specified percentile (e.g., 3 σ) of the system response. This approach works well if we assume that a baseline performance for each microservice was validated. However, the approach could provide a worst case scalability requirement, if one of the microservices' baseline performance is already exhibiting significant performance degradation.

Experiment execution. The proposed approach for automated execution and analysis of the load test cases needs to be assessed for continuous improvement using a declarative approach and automated deployment.

7 Conclusion

In this paper, we have introduced a new four-step approach for the quantitative assessment of microservice architecture deployment alternatives. Our approach consists of operational profile data analysis, experiment generation, baseline requirements computation, and experiment execution. A domain-based metric is computed for each microservice deployment alternative, specified as an architectural configuration. The metric (0–1) reflects the ability of the deployed configuration to meet performance requirements for the expected production usage load.

We have applied our approach to several deployment configurations in a large bare metal host environment, and a virtualized environment. The approach took advantage of automated deployment of Docker containers using a state-of-the-art load test automation tool.

Our approach contributes to the state of the art by automatically deriving baseline performance requirements in a baseline run and assessing pass/fail criteria for the load tests, using a baseline computation of these requirements.

We have found that in auto-scaling cloud environments, careful performance engineering activities shall be executed before additional resources are added to the architecture deployment configuration, because if the bottleneck resource is located downstream from the place where additional resources are added, increased workload at the bottleneck resource may result in a significant performance degradation.

Acknowledgment

This work has been partly supported by EsulabSolutions, Inc., the German Federal Ministry of Education and Research (grant no. 01IS17010, ContinuITy), German Research Foundation (HO 5721/1-1, DECLARE), the GAUSS national research project, which has been funded by the MIUR under the PRIN 2015 program (Contract 2015KWREMX), and by the Swiss National Science Foundation (project no. 178653). The authors would like to thank the HPI Future SOC Lab (period fall 2017) for providing the infrastructure.

References

1. Aderaldo, C.M., Mendona, N.C., Pahl, C., Jamshidi, P.: Benchmark requirements for microservices architecture research. In: Proc. ECASE@ICSE. pp. 8–13. IEEE
2. Alshuqayran, N., Ali, N., Evans, R.: A systematic mapping study in microservice architecture. In: Proc. SOCA. pp. 44–51 (2016)
3. Avritzer, A., Bondi, A.B., Grottke, M., Trivedi, K.S., Weyuker, E.J.: Performance assurance via software rejuvenation: Monitoring, statistics and algorithms. In: Proc. DSN. pp. 435–444 (2006)
4. Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., van Hoorn, A.: Reprodicibility package for "A quantitative approach for the assessment of microservice architecture deployment alternatives using automated performance testing", `https://doi.org/10.5281/zenodo.1256467`
5. Avritzer, A., Weyuker, E.J.: The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng. 21(9) (1995)
6. Casalicchio, E., Perciballi, V.: Auto-scaling of containers: The impact of relative and absolute metrics. In: Proc. FAS*W@SASO/ICCAC. pp. 207–214 (2017)
7. Esposito, C., Castiglione, A., Choo, K.K.R.: Challenges in delivering software in the cloud as microservices. IEEE Cloud Comp. 3(5), 10–14 (2016)
8. Ferme, V., Pautasso, C.: A declarative approach for performance tests execution in continuous software development environments. In: Proc. ACM/SPEC ICPE. pp. 261–272 (2018)
9. Francesco, P.D., Malavolta, I., Lago, P.: Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: Proc. ICSA. pp. 21–30 (2017)
10. Jiang, Z.M., Hassan, A.E.: A survey on load testing of large-scale software systems. IEEE Trans. Softw. Eng. 41(11), 1091–1118 (2015)
11. Kozhirbayev, Z., Sinnott, R.O.: A performance comparison of container-based technologies for the cloud. Future Gener. Comp. Syst. 68, 175–182 (2017)
12. McGrath, G., Brenner, P.R.: Serverless computing: Design, implementation, and performance. In: Proc. ICDCSW. pp. 405–410 (2017)
13. Newman, S.: Building Microservices. O'Reilly Media, Inc., 1st edn. (2015)
14. Pahl, C., Jamshidi, P.: Microservices: A systematic mapping study. In: Proc. CLOSER. pp. 137–146 (2016)
15. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory and Practice. John Wiley & Sons, Inc. (2009)
16. Ueda, T., Nakaike, T., Ohara, M.: Workload characterization for microservices. In: Proc. IISWC. pp. 1–10 (2016)
17. Vögele, C., van Hoorn, A., Schulz, E., Hasselbring, W., Krcmar, H.: 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), 443–477 (2018)
18. Weyuker, E.J., Avritzer, A.: A metric for predicting the performance of an application under a growing workload. IBM Syst. J. 41(1), 45–54 (2002)
19. Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. IEEE Trans. Softw. Eng. 17(7), 703–711 (1991)