Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests

Alberto Avritzer[b], Vincenzo Ferme[f], Andrea Janes[a], Barbara Russo[a], André van Hoorn[d], Henning Schulz[c], Daniel Menasché[e], Vilc Rufino[e]

[a]*Free University of Bozen-Bolzano, Bolzano, Italy*
[b]*EsulabSolutions, Inc., Princeton, NJ, USA*
[c]*Novatec Consulting GmbH, Leinfelden-Echterdingen, Germany*
[d]*University of Stuttgart, Stuttgart, Germany*
[e]*Federal University of Rio de Janeiro (UFRJ), Rio de Janeiro, Brazil*
[f]*Kiratech S.p.A., Paradiso (Lugano), Switzerland*

Abstract

Microservices have emerged as an architectural style for developing distributed applications. Assessing the performance of architecture deployment configurations — e.g., with respect to 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 microservice configuration alternatives. The approach provides a Domain-based metric for each alternative that can, for instance, be applied to make informed decisions about the selection of alternatives and to conduct production monitoring regarding performance-related system properties, e.g., anomaly detection.

We have evaluated our approach using extensive experiments in a large bare metal host environment and a virtualized environment. First, 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 experiments presented in this paper, we observed that the evaluated Domain-based metric is a non-increasing function of the number of CPU resources for one of the environments under study. In a subsequent series of experiments, we investigate the application of the approach to assess the impact of security attacks on the performance of architecture deployment configurations.

1. Introduction

Background. The microservices architectural style (Newman, 2015) 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 (Newman, 2015). 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 (Francesco et al., 2017).

Currently, several configuration alternatives are possible for microservices deployment, for example, serverless microservices using functions (e.g., Amazon Lambda[1]), container-based deployment (e.g., Docker[2]), virtual machines per host, and several hosts. Of course, depending on the microservice granularity, a combination of these mechanisms could be used. The available architecture alternatives and their parameters imply a large space of architecture deployment configurations (Taylor et al., 2009) to choose from.

Challenges. Microservices are supposed to be independent of 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 lookup, and at the different hardware and software layers that are shared among the microservices. Ueda et al. (2016) 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 architecture deployment alternatives. In addition, the impact of the expected production workloads on the performance of specific microservice deployment configurations needs to be taken into account. The alternatives for microservice architecture deployment considered in this paper are memory allocation, CPU fraction used, and the number of (Docker) container replicas assigned to each microservice.

Goals. In this paper, we introduce a quantitative approach for the performance assessment of microservice deployment alternatives. The approach uses automated performance testing results to quantitatively assess each architecture deployment configuration in terms of a Domain-based metric introduced in this paper. For performance testing, we focus on load tests based

[1]`https://aws.amazon.com/lambda/`
[2]`https://www.docker.com/`

Accepted manuscript, Journal of Systems and Software, `https://doi.org/10.1016/j.jss.2020.110564`.

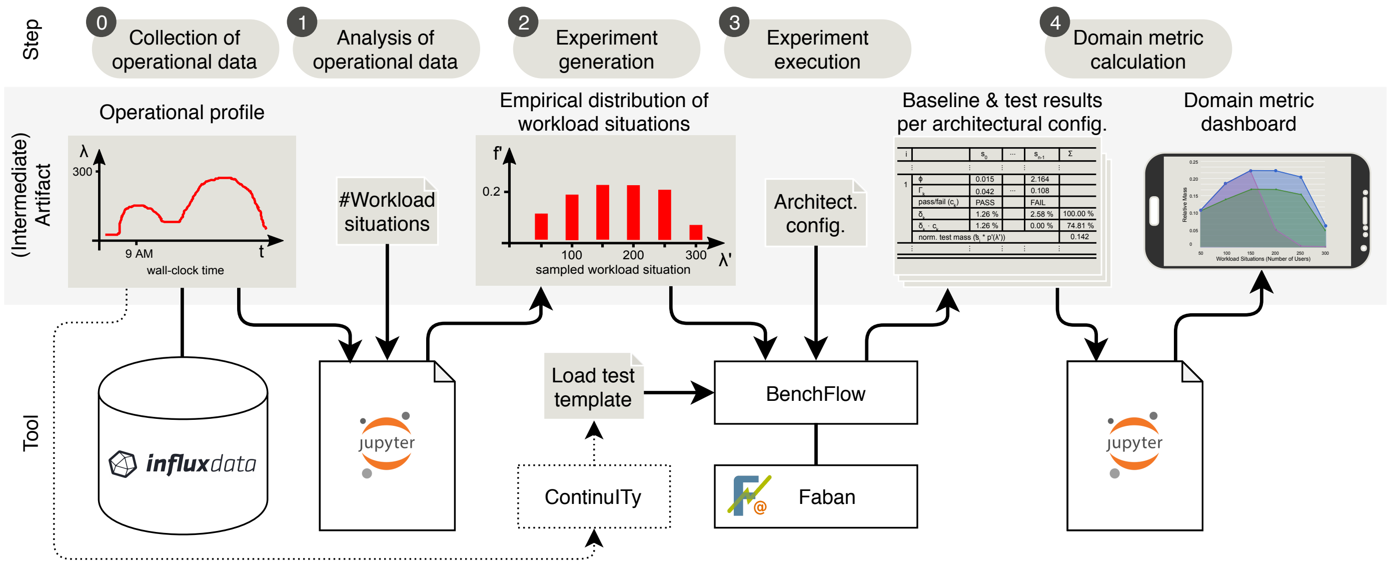


Figure 1: Overview of methodology steps and framework architecture.

on operational workload situations (Jiang and Hassan, 2015; Vögele et al., 2018), e.g., arrival rates or the concurrent number of users.

Methodology. The proposed methodology steps and framework architecture for scalability assessment are illustrated in Figure 1. The steps are briefly introduced in the following and are detailed in Section 3. We combine analysis of operational profile data with performance testing results to generate a domain metric dashboard. The dashboard illustrates system scalability with respect to operational profile distribution in production, i.e., empirical distribution of workload situations, and performance results in the load test environment. Operational profile data is used to estimate the probability of occurrence of each workload situation in production. Scalability requirements are used to assess each architecture deployment configuration. The resulting quantitative assessment is a metric value between 0–1 that assesses the fitness of a certain architecture alternative to perform under a defined workload situation.

Experiments. We evaluate and apply the proposed approach in two types of experiments.

First, we have computed the introduced Domain-based metric for twelve different configurations based on two different memory allocations, two different CPU allocations, and three different values for the number of microservice replicas. The experiments were executed in two data center environments. We have identified, for each environment, the architecture deployment configuration that produced the best value for the Domain-based metric. It is very significant that in both environments, increasing the number of replicas for the service being evaluated or the fraction of CPU allocation did not guarantee better performance, as assessed by the Domain-based metric. Additional experiments in one of the environments were executed as a basis for our subsequent investigation of the Domain-based metric's sensitivity concerning the configured scalability requirement.

Second, we have evaluated the impact of security attacks on the Domain-based metric for scenarios with and without security attacks.

Contributions. This paper's key contributions are as follows:

- *Methodology for scalability assessment:* A new quantitative approach and framework for the assessment of microservice architecture configuration alternatives.
- *Experimental validation:* The experimental validation of the proposed approach for scalability assessment with and without attacks.

Prior art. Our approach for the Domain-based metric evaluation is based on the input domain partition testing strategy (Weyuker and Jeng, 1991) and domain-based load testing (Avritzer and Weyuker, 1995). 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 (Avritzer and Weyuker, 1995). This paper is an extension of our previous works on the proposed approach (Avritzer et al., 2018) and the framework implementation (Avritzer et al., 2019). The relation to the previous works is as follows. First, this paper is a self-contained and revised presentation of the previously separated materials. It includes an extended sensitivity analysis of the scalability assessment scenario. Moreover, this paper presents an additional application and experiment study for assessing the performance under attacks using the proposed approach and metric (see Table 1).

Paper outline. The remainder of this paper is organized as follows. Section 2 contains a summary of the reviewed literature on microservice architecture challenges and performance

	Signatures for security	Fully automated	Customer-affecting metrics	Server-side metrics
Partition testing (Avritzer and Weyuker, 1995)				×
Performance signatures (Avritzer et al., 2010)	×			×
Assessment of architecture alternatives (Avritzer et al., 2018)			×	
PPTAM framework (Avritzer et al., 2019)			×	
This paper	×	×	×	

Table 1: The contributions of this paper in comparison with the previous work of the authors.

assessment. Section 3 contains an overview of the proposed approach and infrastructure for performance assessment of microservice architectures. Sections 4–7 describe the experimental design, and report and discuss the experimental results obtained by applying the proposed approach. Section 8 presents our conclusions and suggestions for future research. A reproducibility package is provided online (Avritzer et al., 2020).

2. Related Work

In this section, we present a summary of the previous work by the authors, the reviewed literature on microservice architecture challenges, the performance assessment of microservice architectures, and intrusion detection tools. In each case, we relate the reviewed literature to this paper's contributions.

2.1. Previous Work of the Authors

Table 1 provides an overview of the relationship between this paper's contributions and the contributions by our previous related publications, as detailed hereinafter.

The current paper is an extension of a previous work published at the 12th European Conference on Software Architecture (ECSA 2018) (Avritzer et al., 2018) and also includes the contents of a two-page tool paper (Avritzer et al., 2019). The paper at ECSA 2018 introduced the quantitative approach based on the Domain-based metric to evaluate different microservice architecture deployment alternatives and compared them in bare-metal and virtual environments. In the ECSA 2018 paper, we extended previous work (Avritzer and Weyuker, 1995; Weyuker and Avritzer, 2002) to define a new methodology for the assessment of microservice deployment alternatives — also referred to as *(architecture) configurations*. In (Weyuker and Avritzer, 2002), we introduced a metric to assess software scalability. This metric uses the requirement definition, high-level architecture modeling, and system measurement results to assess the system architecture's ability to meet performance requirements as a function of workload increases. In (Avritzer and Weyuker, 1995), 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 system under test (SUT). The Markovian approximation is used to estimate the steady-state probability of occurrence of each test case. In this way, the 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 (Avritzer and Weyuker, 1995).

The tool paper (Avritzer et al., 2019) proposed a fully automated framework to implement the approach introduced in the ECSA 2018 paper. The approach enables experimental replication in different contexts, and result visualization using a dashboard (e.g., mobile device).

The current paper conveys all previous work and further includes: (*i*) new experiments that incorporate security attacks using the Mirai tool (Barker, 2016), (*ii*) new experiments in the virtualized environment (UNIBZ) using a different operational profile and additional CPU and memory resources, (*iii*) a sensitivity analysis of the performance threshold in the experiments with and without attacks in the UNIBZ environment. For a description of the UNIBZ environment please refer to Section 4.3.

2.2. Microservice Architectural Challenges

Alshuqayran et al. (2016) 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 (2016) 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. In this paper, we address some of these concerns by: (*i*) presenting a methodology for scalability assessment that can be integrated into tool automation, and, (*ii*) conducting experiments to validate the proposed methodology.

2.3. *Performance Assessment of Microservice Architectures*

Casalicchio and Perciballi (2017) address the problem of selecting more appropriate performance metrics to activate autoscaling actions. Specifically, they investigate the use of relative and absolute metrics and propose a new autoscaling algorithm that is able to reduce the response time by a factor between 0.66 and 0.5, when compared to the actual Kubernetes' horizontal auto-scaling algorithm. In this paper, we introduce a new Domain-based metric that captures: (*i*) scalability testing results in the SUT using several architecture deployment configurations, (*ii*) expected production usage derived from operational data analysis. Therefore, the metric represents the system's ability to satisfy scalability requirements for the evaluated workload situations and architecture deployment configurations.

2.4. *Intrusion Detection System Tools*

A comprehensive survey of intrusion detection systems (IDS) was presented by Milenkoski et al. (2015), where IDS tools were classified by: (*i*) monitored platform (host-based, network-based, or hybrid), (*ii*) attack detection method (misuse-based, anomaly-based, hybrid), and, (*iii*) deployment architecture (non-distributed and distributed). In the anomaly-based IDSs, a baseline profile of normal operations is developed and deviations from the baseline profile are identified as intrusions using performance signatures.

Avritzer et al. (2010) proposed an architecture for intrusion detection systems using off-the-shelf IDSs complemented by performance signatures. The authors have shown that the performance signature of well-behaved systems and of several types of security attacks could be identified in terms of certain performance metrics, such as CPU and memory percentage or number of active threads. In this paper, we evaluate the impact of security intrusions on the Domain-based metric.

3. Methodology and Framework

Our methodology is designed to support the automated assessment of architecture deployment configurations. For each configuration, this results in a measure—the so-called Domain-based metric—that quantifies the configuration's ability to satisfy scalability requirements under a given operational profile. We define workload situation as an abstract concept to represent the output of operational data analysis per application domain. Specifically, the number of concurrent users is the focus of this paper. However, in other application domains, such as banking, transaction rate could be used.

For a complex system under test, such as the microservice architecture under study in this paper, it would be difficult to assess resource saturation, as several types of services are executed that demand work from several software and hardware resources, virtualization engines, load balancers, CPU, memory, I/O, and network. Therefore, the importance of the proposed methodology resides in its ability to assess system scalability for different architecture configurations in complex microservice architectures.

This section describes the approach (Section 3.1), applications (Section 3.2), and the tooling infrastructure (Section 3.3). For the sake of the reader, Table 2 summarizes the notations used in the approach.

Table 2: Table of notation

Variable	Description
Λ	set of workload situations
λ	workload situation, $\lambda \in \Lambda$
$p(\lambda)$	probability of occurrence of λ
$f(\lambda)$	frequency of occurrence of λ
s_j	j-th service
λ_0	baseline workload situation
$\Gamma_j(\lambda_0)$	pass/fail threshold for s_j $\Gamma_j(\lambda_0) = x(\lambda_0)_j + 3 \times \sigma(\lambda_0)_j$
$x(\lambda_0)_j$	mean response time of s_j for λ_0
$\sigma(\lambda_0)_j$	standard deviation of s_j for λ_0
$x(\lambda)_j$	mean response time of s_j for λ
α	architecture deployment configuration
δ_j	fraction of calls to service s_j
$\hat{s}(\lambda)$	fraction of successful calls to all services
$D(\alpha, \mathcal{S})$	Domain-based metric $D(\alpha, \mathcal{S}) = \sum_i p(\lambda_i)\hat{s}(\lambda_i)$

3.1. *Computation of the Domain-based Metric*

The approach introduced in this paper and illustrated in Figure 1 consists of the following steps:

1. *Collection of operational data*, i.e., data on normal system usage (e.g., HTTP requests) in a given time window are collected,

2. *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,

3. *Experiment generation*, i.e., the automated generation of the load test cases for the architecture deployment configurations under evaluation,

4. *Baseline computation, quantitative definition of the scalability requirements*, 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,

5. *Experiment execution*, i.e., the execution of load test cases for the architecture deployment configurations specified in the experiment generation step, and the computation of the Domain-based metric.

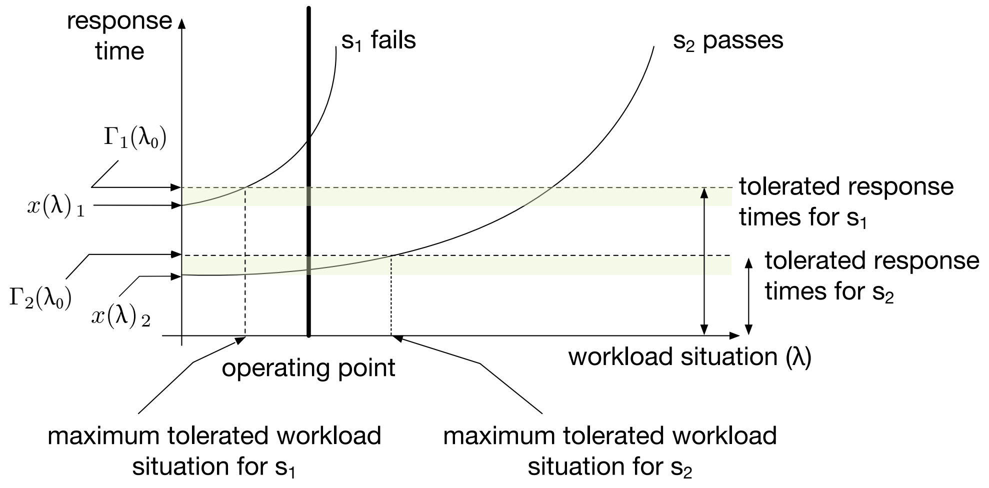

response time
s1 fails
s2 passes
Γ1(λ0)
x(λ)1
tolerated response times for s1
Γ2(λ0)
x(λ)2
tolerated response times for s2
operating point
workload situation (λ)
maximum tolerated workload situation for s1
maximum tolerated workload situation for s2

Figure 2: System scalability illustrating response time as a function of workload situations, for two services: one that passes the scalability test, and the other that fails, given the considered operating point. Note that $x(\lambda)_1$ and $x(\lambda)_2$ refer to the mean response times of services 1 and 2 under baseline load for one specific architecture deployment configuration. $\Gamma(\lambda_0)$ for $i = 1, 2$ are the baseline measurements for the corresponding services.

In this section, we illustrate the approach with a running example, which is based on the experiments reported in Sections 4 and 5. The operational profile is taken from publicly available information about a video streaming service, as described in Section 4.5.

Step 1—Analysis of Operational Data.
The operational profile describes for each workload situation $\lambda \in \Lambda$, its probability of occurrence $p(\lambda)$, which is estimated by the relative frequency of occurrence $f(\lambda)$ in the SUT. The operational profile is used to answer two questions: (*i*) How do we identify the workload situations to test? (*ii*) How representative are the selected tests with respect to production usage?

To illustrate this step, we first analyze the operational data to create the operational profile as frequency of occurrence of the workload situations found in the system at a certain time t. The operational profile is the output of Step 1 in Figure 1. To reduce the number of tests to execute, the workload situations are aggregated in bins $\lambda_1, ..., \lambda_k$ and the final operational profile represents the frequency of occurrence of such bin values, $f(\lambda_1), ..., f(\lambda_k)$. The test suite coverage criterion is based on the values of such frequencies.

Step 2—Experiment Generation. In this step, we define the experiment settings for the test cases. The elements of this step are:

1. **The load test sequence** is obtained by selecting the workload situations defined in Step 1 (i.e., the bins of the operational profile of the SUT),
2. **The load test specification** consists of a workload situation of the load test sequence and a choice of an architecture deployment configuration,
3. **The baseline requirement** defines, for each service s_j, the test pass/fail criteria based on a performance metric. We make no specific assumption about such a performance metric. An example used in this work is the average response time of a service for a reference architecture deployment configuration,
4. **A test case** consists of a set of experiments performed according to a load test specification and evaluated against a baseline requirement. In this work, we performed 60 experiments for each test case.

Step 3—Baseline Computation, Quantitative Definition of the Scalability Requirements. In this step, we describe the approach that we used to calculate the fraction of correctly executed services $\hat{s}_i$ for test case i. Initially, a test case is run to identify the baseline requirement. The test is performed according to a load test specification defined by a deployment configuration with high resources $\alpha_0 \in A$ and a low workload $\lambda_0 \in \Lambda$. For such a test case, the average response time, $x(\lambda_0)_j$, and the standard deviation, $\sigma(\lambda_0)_j$ for each service s_j, under the baseline workload λ_0 are measured. The *scalability requirement* for service s_j is then defined as $\Gamma_j(\lambda_0) = x(\lambda_0)_j + 3\times\sigma(\lambda_0)_j$. This is an innovative approach for the definition of scalability requirements that employs a measured baseline performance to automatically identify a tolerance for scalability degradation under load.

Table 3 shows the baseline requirements $\Gamma_j(\lambda_0)$ we measured for the services s_j of the SUT used in our study. As the scalability requirement is the same for all test cases, the values in Table 3 are used as scalability requirements for all load test specifications.

Figure 2 illustrates our approach for two services with response time as metric and Γ_0 as the baseline requirement and for a workload situation referred to as the *operating point*. Response time is averaged over the experiments of the test case. The curves in Figure 2 represent the average response time of the two services under increasing workload situations (λ). At the operating point, service s_1 fails and service s_2 succeeds as the average response time over the given experiments of s_1 $(x(\lambda)_1)$ exceeds the baseline threshold $\Gamma_1(\lambda_0)$ before the operating point whereas $x(\lambda)_1$ does not. In general, service s_1 and s_2 fail at a given operating point once their average response time exceeds $\Gamma_1(\lambda_0)$ and $\Gamma_2(\lambda_0)$, respectively. The workload situation for which this occurs is indicated as the *maximum tolerated workload situation* and corresponds to the *maximum tolerated response time*.

Step 4a—Experiment Execution (Pass/Fail Assessment). In this step, each service s_j is tested under a certain load test specification, i.e., workload $\lambda \in \Lambda$ and configuration $\alpha \in A$. In the following, we drop the configuration α to simplify the notation, as these computations are repeated for each workload situation and configuration. Each test case executes the set of n services $\{s_1, \dots, s_n\}$ and produces $\{\delta_1, \dots, \delta_n\}$, each of which is the fraction of service executions that were assessed as successful by comparison with the scalability requirement. Each service s_j is marked as pass for workload λ and configuration α, if $x(\lambda)_j < \Gamma_j(\lambda_0)$. 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. Therefore, the fraction $\hat{s}$ of successful executions of all services can be evaluated as:

$$\hat{s}(\lambda) = \sum_{j=1}^{n} \delta_j c_j \tag{1}$$

Equation 1 assumes that the execution of the services j is mutually exclusive, which is accurate for the microservice architecture considered in this paper. However, for a more general applicability of the approach it would be more accurate to develop a queuing network model to compute $\hat{s}(\lambda)$. The reader is referred to Denning and Buzen (1978) for a detailed review of the required assumptions and limitations for the applicability of queuing network models.

Table 4 illustrates the pass/fail estimation for one load test specification where $\lambda = 100$. For this test case, the fraction of correctly executed services was evaluated as $\hat{s} = 74.81\%$ (Eq. 1).

Step 4b—Experiment Execution (Computation of Domain-based Metric). Finally, the total Domain-based metric $D(\alpha, S)$ for the configuration α, with respect to a test suite S defined by workload situations λ_1,...,λ_z can be evaluated as:

$$D(\alpha, S) = \sum_{i=1}^{z} p(\lambda_i)\hat{s}(\lambda_i) \tag{2}$$

where $p(\lambda_i)$ is the probability of occurrence corresponding to workload situation λ_i (as in Step 1). The Domain-based metric per workload situation is $D(\alpha, S, i) = p(\lambda_i)\hat{s}(\lambda_i)$. The resulting quantitative assessment is a measure between 0–1 that can be used to assess the performance of different architecture deployment configurations. For the running example, as illustrated in Figure 1, $D(\alpha, S)$ was evaluated as 0.615 and the contribution of the test case reported at the end of Step 4a was 0.142, i.e., $0.19 \times 74.81\%$ where 0.19 is the frequency of occurrence of the workload situation $\lambda = 100$ in the operational profile (Figure 3).

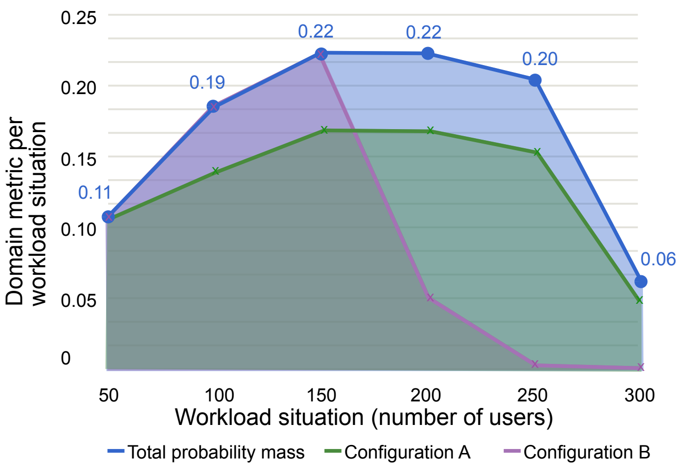


Figure 3: Plots of Domain-based metric per workload situation for different architecture deployment configurations of the test environment

The contribution to the Domain-based metric of workload situations in a load test sequence can be displayed in plots. Such a plot depicts the degree to which a given system deployment configuration satisfies the fail/pass criterion. In the example shown in Figure 3, the plots at the workload situations of the load test sequence show gaps between the total probability mass (outer polygon, light blue) and the obtained measurements (inner polygons) for two architecture deployment configurations. These gaps represent the impact of the measured performance degradation on the Domain-based metric.

3.2. Applications

In this work, we perform a set of case studies to illustrate and validate the approach introduced in this paper. Our first case study was presented in our previous work (Avritzer et al., 2018) and concerned the comparison between load testing in a bare-metal environment (HPI) and in a virtual environment (UNIBZ). Such analysis is described in Section 5.1. We have then performed a new case study to assess the sensitivity of the domain metric with respect to the scalability requirement, by scaling the threshold by a factor between 0.2 and 50 times the original value as described in Section 6. The last case study, which is presented in Section 5.2, compares the system under test (SUT) normal behavior with its behavior under security attacks.

3.3. Framework Architecture

Figure 1 includes details on the tool ecosystem used to implement the proposed approach. It comprises the following major components:

(*i*) An analysis component that gathers the operational profile from a production system and computes the baseline probability of finding the system at a given state (workload situation).

(*ii*) An infrastructure that generates and executes load test experiments with different architecture deployment configurations to collect system performance against a baseline (fail/pass criteria).

Table 3: Scalability requirements based on baseline requirements (in seconds)

s_j	createOrder	basket	getCatalogue	getItem	login	...
$x(\lambda_0)_j$	0.018	0.008	0.011	0.012	0.033	...
$\sigma(\lambda_0)_j$	0.008	0.003	0.002	0.009	0.025	...
$\Gamma_j(\lambda_0)$	0.042	0.017	0.017	0.039	0.108	...

Table 4: Pass/fail based on scalability requirements (in seconds) for workload specification λ

s_j	createOrder	basket	...	login	...
$\Gamma_j(\lambda_0)$	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 %	...

(*iii*) A graphical user interface that calculates and visualizes the current performance metric in a report and on a smartphone user interface (UI).

Each of these components is operated by a series of tools that we developed or integrated. For gathering the operational profile, we utilize Application Performance Monitoring (APM) (Heger et al., 2017) tools, e.g., an open-source tool from the OpenAPM initiative.[3] These tools commonly utilize time-series databases, e.g., InfluxData, for storing the monitored operational data. Our tool (packaged as a Jupiter Notebook[4]) connects to an InfluxData, retrieves the raw operational data, and generates the empirical distribution of the workload situations (best test masses) defined in the script. The infrastructure for load testing uses the open-source BenchFlow tool (Ferme and Pautasso, 2018) to automate the deployment of the defined experiments (building on container-based virtualization using Docker) for the defined workload situations and system configurations. The Faban load testing framework is used to run the experiments, to collect the performance measures, and to automate the analysis of the testing results. Load test specifications are either manually defined or extracted automatically (e.g., using ContinuITy (Schulz et al., 2019)). R scripts (as a Jupyter Notebook and Rshiny[5] application) are used to evaluate the performance of the test against a baseline, compute the performance for each test at each state of the system, generate the plots of total distribution mass and the previously obtained Domain-based metric curves, and compute the sensitivity analysis and plots. React-native, a Javascript framework for building native mobile apps, is used to create the UI[6].

Our PPTAM tool including a demo is publicly available.[7]

4. Experiment Design

In our evaluation, we show how to use our approach as illustrated in Figure 1 and Section 3.1 to compute the Domain-based metric and assess the system scalability and performance during regular behavior and under attack. The evaluation considers: (*i*) a given operational environment, (*ii*) a specific SUT, and, (*iii*) its architecture deployment configuration alternatives. We use the operational profile from Step 1 in Section 3.1 and apply it to the SUT Sock Shop microservices demo in the two different environments as described in Section 4.1. We execute the experiments generated by our Step 2 and compare the results against individual baselines as per Step 3.

In designing a load testing methodology, approaches are needed to help cope with the test space explosion. In our methodology, we cope with the state space explosion in the following ways:

- We aggregate the number of workload situations to be measured by combining neighboring workload situations. In the following, to simplify notation, we use concurrent number of users to represent workload situations.
- We create a small number of architecture deployment configurations to be tested by focusing on CPU, memory and the number of instances of one microservice, namely *cart*. This was done because *cart* was the mostly used microservice.
- We generate pass/fail results, for each architecture deployment configuration for the constant number of workload situations identified. Therefore, for each architecture deployment configuration, we run a small number of tests.

The remainder of this section describes the precise details of our experiment design. The experimental settings and the associated results are included in our reproducibility package (Avritzer et al., 2020).

4.1. System Under Test

As system under test (SUT), we utilize the Sock Shop microservices demo (most recent version as per March 28, 2018[8])

[3]`https://openapm.io/`
[4]`https://jupyter.org/`
[5]`https://shiny.rstudio.com`
[6]`https://facebook.github.io/react-native`
[7]`https://github.com/pptam`
[8]`https://microservices-demo.github.io/`

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 users' shopping carts. For the 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 (Aderaldo et al., 2017). The main criteria used for selecting Sock Shop as the SUT were: (*i*) the usage of well-known microservice architectural patterns, (*ii*) the automated deployment in containers, and, (*iii*) the support for different deployment options.

4.2. Load Testing Tool

We use BenchFlow (Ferme and Pautasso, 2018) as the load testing tool. BenchFlow is an open-source framework[9] that automates the end-to-end process of executing performance testing. BenchFlow reuses and integrates state-of-the-art technologies, such as Docker[10], Faban[11], and Apache Spark[12]. BenchFlow reliably executes load tests, automatically collects performance data, and computes performance metrics and statistics. BenchFlow is also used to validate the reliability of the obtained results.

BenchFlow users define their performance intent by relying on a declarative domain-specific language (DSL) for goal-driven load tests. Declarative templates are provided for expressing tests' requirements such as: the test goals, the test types, the metrics of interest, the test stop conditions (e.g., maximum test execution time), and the parameters to vary during the test execution. The BenchFlow framework implements strategies and processes that are driven by the user's input specification. In addition, during test execution, BenchFlow monitors the real-time state of the SUT.

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[13] hypervisor at the Free University of Bozen-Bolzano (UNIBZ).

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 (UNIBZ) 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 an EMC VNC 5400 series network attached storage solution[14] 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. Definition and Execution of Performance Tests

By relying on BenchFlow's DSL (Ferme and Pautasso, 2018), users can specify performance tests in a declarative manner. In our case, we defined a load test exploring different system configurations, as presented in Section 4.6. BenchFlow supports a wide range of variables to be automatically explored during configuration tests, e.g.: (*i*) number of concurrent 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. The environment is deployed as Docker containers, with each container implementing a distinct microservice. The containers run on top of the Docker engine, executing as a daemon process on a hypervisor.

Figure 4a depicts the deployment at HPI, showing the Docker Swarm including the Docker manager node, and the Docker worker node that includes the Docker containers. These containers execute the microservice components. For the HPI deployment, the Docker swarm is deployed on a bare metal machine. The same deployment is used at UNIBZ. However, for the UNIBZ deployments, there is an additional virtualization layer, as shown in Figure 4b. There, the bare metal server runs VMWare ESXi, which runs two virtual machines in exactly the same configuration as at HPI.

4.5. Operational Profiles of Workload Situations

In this study, we use the operational profile of two production systems: a video streaming application, as shown in Figure 5a, and Wikipedia, as shown in Figure 5b. These operational profiles are built using the frequency of occurrence of workload situations, which is specified in terms of the number of concurrent users. To be able to compare the results derived from two different operational profiles, their maximum workload intensity levels have been scaled to the same maximum number of users (300).

All experiments have been performed using the video streaming application operational profile and were replicated using the Wikipedia operational profile, as presented in Section 6. The scaled operational profiles were used in two steps: (*i*) the generation of the workload situations by BenchFlow, and, (*ii*) for the computation of the Domain-based metric.

[9]`https://github.com/benchflow`

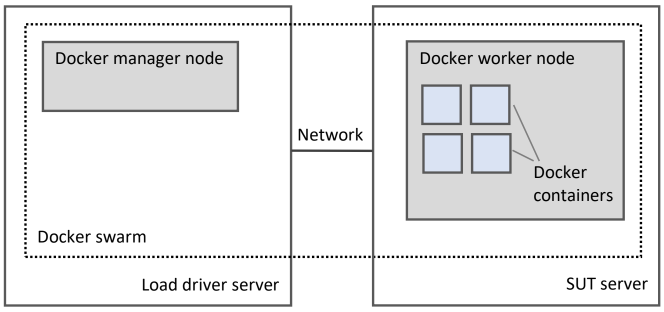
[10]`http://docker.com`

[11]`http://faban.org`

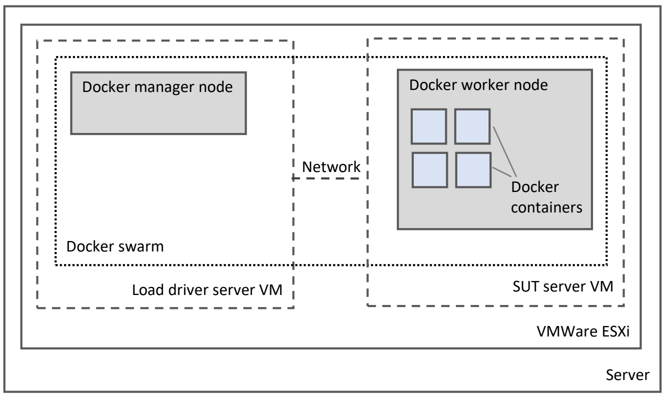
[12]`http://spark.apache.org`

[13]`https://www.vmware.com/products/esxi-and-esx.html`

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



(a) Architectural overview of the setup at HPI: two bare metal machines run Docker and are part of a Docker swarm; within this swarm, one machine acts as load driver, the other deploys the SUT and performs the tests.



(b) Architectural overview of the setup at UNIBZ: one bar metal machine runs VMWare ESXi, which runs two virtual machines that are configured exactly as in HPI.

Figure 4: Overview of the testing infrastructures

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. This approach represents the types of users that are likely to use Sock Shop in the operational environment that is modeled in this paper. We model the following behavior mix (Vögele et al., 2018): three types of users with the respective relative frequency and a maximum allowed 5% deviation for the defined frequency distribution:

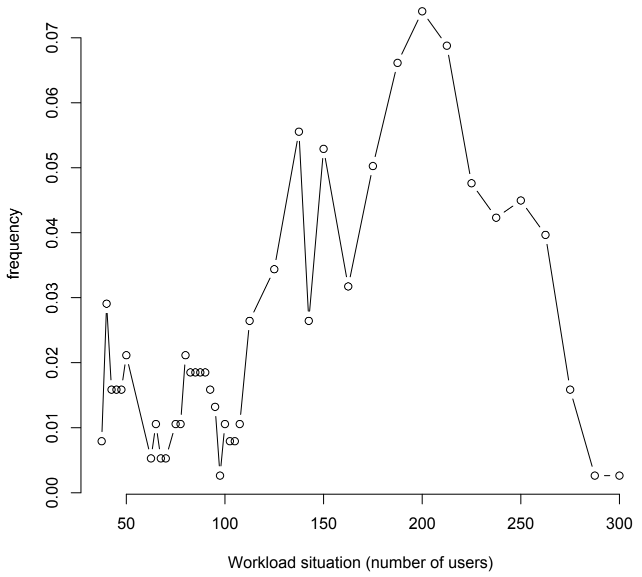
- *visitor* (40%): visits the home page, views the catalog and the details of some products.
- *buyer* (30%): visits the home page, logs in, views the catalog and some details, adds a product to the cart, visits the cart, and creates an order.
- *order visitor* (30%): visits the home page, logs in, and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

The summary of all requests sent to Sock Shop, and their occurrence number, per user type, are provided in Table 5. We have defined a workload intensity function (Vögele et al., 2018) that included a 1 minute ramp-up, and 30 minutes of steady-state execution, to ensure that the system reaches the steady state during the test execution. At the end of the test run, the performance data is collected. 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 think time.

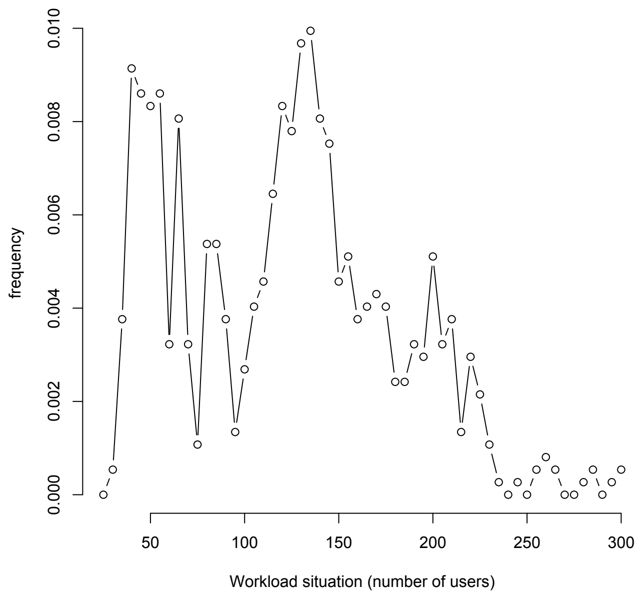
4.6. Architecture Deployment Configurations

We deployed the SUT using different architecture deployment configurations, as specified by the experiment generation step of the methodology introduced in Section 3. The parameters that were varied over the different deployment configurations were the amount of available RAM, the CPU share, and the replicas for the *cart* service.

We targeted the *cart* service, as most of the requests issued by the designed workload, as described in Section 4.5, targeted the *cart* service. The different configurations we explored are reported in the table of Figure 6c. RAM configurations were selected from the set {0.5 GB, 1 GB}, CPU shares were selected from the set $\{0.25, 0.5\}$, and the number of replicas was selected from the set $\{1, 2, 4\}$. The remaining resources of the server on which we deployed the SUT were shared among all the other services that are 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 behavior in the Docker engine. We further replicated the experiments with architecture configurations of increased memory (8 and 16 GB). In total, we executed 264 experiments with different configurations.



(a) Video streaming application



(b) Wikipedia

Figure 5: The operational profiles used in the study. Workload situations are scaled to an equal maximum value (300).

4.7. Baseline Requirement

To assess the impact of the workload situations and of the architecture deployment configurations on the response time, we defined a baseline response time Γ_j, as defined in Step 3 in Section 3.1, and we evaluated the fraction of the services that experienced a response time higher than the considered baseline as defined in Step 4a in Section 3.1.

The baseline requirement is measured through an experiment run with the workload situation $\lambda_0 = 2$ (i.e., two concurrent users) and the highest memory resources allocated for the experiments with normal workload. Thus, for the experiments in Section 5.1 and 5.2, we used the architecture deployment configuration of 4 GB memory, 1 CPU share, and 1 replica for the *cart* service. To discuss the choice of such a baseline requirement, we further performed a sensitivity analysis, which is presented in Section 6.

By relying on the operational data presented in Section 3, we identified the following aggregated workload situations as load test specification $\{50, 100, 150, 200, 250, 300\}$ as described in Step 2 of Section 3.2. In addition, 19 services of Sock Shop, $\mathcal{S} = \{s_1, s_2, \ldots, s_{19}\}$ were configured in the SUT. Each concurrent user calls multiple services while accessing the system. For instance, one service may be called by a user to insert an item into the shopping cart, to proceed to checkout, and to confirm the order. We have observed that most of the requests issued by such workload situations target the *cart* microservice.

4.8. Attacks

The attacks reported in this paper were conducted in the controlled UNIBZ lab environment that was described in Section 4.3, and used a modified version of the Mirai malware. Antonakakis et al. (2017) describe the Mirai botnet that was used to create a DDoS attack in 2016. This attack harnessed the power of insecure IoT (Internet of Things) devices. The authors have also presented an analysis of the Mirai timeline covering seven months, which included up to 600k security intrusions. Kambourakis et al. (2017) presented a review of Mirai and its mutations and alerted to the risks posed by large botnets formed by using compromised IoT devices.

Mirai is composed of two components (Barker, 2016):

- The "command and control" (CNC) server, written in Go, provides the admin interface that is used to perform attacks, to store the list of available bots, to parse, format, and build shell commands, and to send the commands to the appropriate bots.
- One or more bots, written in C, are used to perform the actual attacks and infect new devices. To find and infect new devices, a bot performs a brute force scan over a range of IP addresses. Once it finds an IP address, it performs a port scan against it. If the bot is able to successfully connect to an IP address and port, it tries to authenticate using frequently used credentials (e.g., admin/admin, support/support, or admin/12345). If the bot is able to authenticate, it tries to enable the system's shell, to access it, and to report back to the CNC server the just discovered new bot, i.e., the IP address, the port, and the authentication credentials (Barker, 2016).

A bot is able to perform attacks using different protocols (Barker, 2016): it performs denial-of-service attacks using the transport protocols UDP or TCP, flooding target devices with

Table 5: Summary of requests, their numbers of occurrence by user types (V=visitor, B=buyer, O=order visitor), and actual overall workload relative frequency (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%

packets, or sending malformed packets; it is also able to attack using HTTP, sending HTTP GET or POST requests containing cookies and random data. As long as the connection is held, i.e., a valid response is returned, the bot continually floods the target device with HTTP requests, with the goal to render the target devices inoperable, or to consume excessive amounts of resources on routers, servers, and intrusion prevention systems/intrusion detection systems devices. From the publicly available Mirai source code[15], we extracted the bot to conduct Mirai attacks to a given target, for a configurable time. We removed the scanner part, which searches for new devices to infect. In a test with attacks, we start the load test in the same way as without attacks but also run a parallel process, which waits for 3 minutes and then starts one Mirai bot with the following parameters:

- duration of attack: long attacks of 20 minutes (1200 seconds) and short attacks of 5 minutes (300 seconds);
- protocol used: HTTP;
- IP address to attack: the IP address of the SUT, i.e., the machine with Sock Shop installed;
- number of threads: 256.

The attacks were conducted from the load driver server machine, which is configured with 4 GB RAM, 1 core at 2600MHz and is connected to the SUT server machine through a 1000 MBit network.

After the attack, the test continues as before for the remaining time, i.e., for 7 minutes, resulting in a total testing time of 30 minutes for the tests with attack and without attack.

[15]`https://github.com/queupe/Mirai-Source-Code`

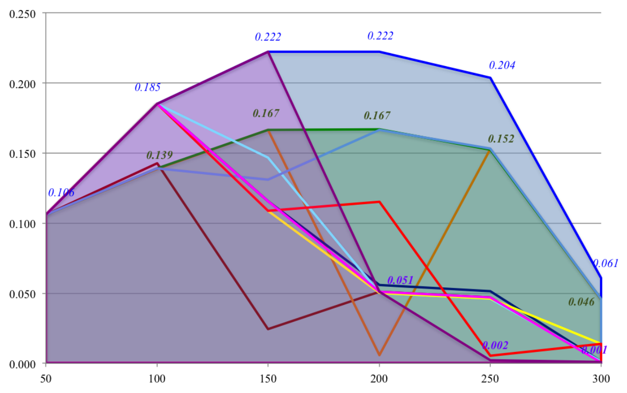
5. Experiments

In this section, we analyze and discuss the results of our sets of experiments on Sock Shop that uses the video streaming operational profile. This set of experiments compares the two testing infrastructures, as described in Section 5.1, and evaluates the effects on the Domain-based metric of the attacks on Sock Shop in the virtual environment, as presented in Section 5.2.

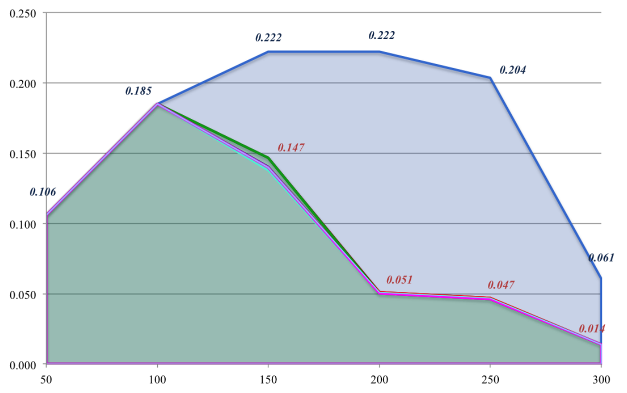
5.1. Domain-based Metrics — Bare Metal vs. Virtual Environment

Figure 6 shows the Domain-based metric computed for each architecture deployment alternative, for the defined workload situations, and for the two environments: HPI (Figure 6a), and UNIBZ (Figure 6b). The total Domain-based metric for each investigated architecture deployment configuration is shown in the table contained in Figure 6c. The outer plot in the figure represents the *theoretical maximum* (i.e., the total probability mass of the given operational profile as described in Step 1 in Section 3.2). The theoretical maximum is reached if all tests pass and it corresponds to the set of frequency values of the operational profile at the workload situations. The other lines in the figure represent the Domain-based metric computed for an investigated architecture deployment configuration. None of the analyzed architecture deployments reached the theoretical maximum, because of the scalability assessment failures identified.

For the HPI environment, the configuration with 1 GB of RAM, 0.5 CPU share, and four *cart* replicas did not exhibit scalability assessment failures for up to 150 concurrent users. However, when the number of concurrent users was increased to values greater than 150, we observed a significant decrease



(a) HPI



(b) UNIBZ

Configuration (α)			Domain-based metric	
RAM	CPU	# Replicas	HPI	UNIBZ
0.5 GB	0.25	1	0.615	0.541
1 GB	**0.25**	**1**	**0.776**	0.539
1 GB	0.5	1	0.536	0.541
0.5 GB	0.5	1	0.515	0.548
0.5 GB	0.5	2	0.510	0.541
1 GB	0.25	2	0.741	0.548
1 GB	0.5	2	0.534	0.541
0.5 GB	***0.5***	***4***	0.505	***0.550***
1 GB	0.25	4	0.372	0.543
1 GB	0.5	4	0.567	0.543

(c) D(α,S) per configuration

Figure 6: Domain-based metric per workload situations ((a) and (b)) and in total (c) in the two environments (HPI, UNIBZ) for the operational profile of the Video streaming application over different configurations α. Blue area = total probability mass, green area = best configuration, violet area = best configuration for low loads.

in the value of the Domain-based metric, as shown by the violet area in Figure 6a. The best observed value for the Domain-based metric, $D(\alpha, S) \approx 0.78$, was achieved for the configuration with 1 GB of RAM, 0.25 CPU share, and one *cart* replica. The addition of *cart* replicas resulted in performance degradation as assessed by the Domain-based metric, as illustrated in the table of Figure 6c. In contrast, the worst observed value for the Domain-based metric, $D(\alpha, S) \approx 0.37$, was achieved for the configuration with 1 GB of RAM, 0.25 CPU share, and four *cart* replicas. This is an interesting result with significant implications for the assessment of architecture 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 UNIBZ experiments show a significant performance degradation as assessed by the Domain-based metric when compared to the HPI experiment. In addition, most of the experiment results are within a narrow Domain-based metric range as can be seen from Figure 6b, where most of the lines overlap. Figure 6b does not show scalability assessment failures for up to 100 concurrent users. Further increases in the number of concurrent users causes the Domain-based metric to decrease with a similar rate, for all architecture deployment configurations. The configuration with 0.5 GB of RAM, 0.5 CPU share, and four *cart* replicas showed the best value for the Domain-based metric, $D(\alpha, S) \approx 0.55$. The worst value achieved for the Domain-based metric employed an architecture deployment configuration with 1 GB of RAM, 0.25 CPU share, and one *cart* replica.

Discussion These results show that determining the best deployment configuration for a system requires the systematic application of engineering approaches for quantitative performance assessment. We have found that adding more CPU power or increasing the number of Docker container replicas may not result in system performance improvement in the bare-metal environment (HPI). In the virtual environment (UNIBZ), the Domain-based 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 greater than 150.

The difference in the Domain-based metric assessment between the HPI and UNIBZ environments, for the same architecture deployment configurations, as shown in the table of Figure 6c, seems to indicate that additional architecture factors, such as VMware Hypervisor overhead, I/O bandwidth, may be impacting system performance.

These findings support the recommendation that practitioners can benefit from the application of the methodology proposed in this paper, by evaluating the expected operational profile and deployment alternatives in their own context. Moreover, these findings suggest that bottleneck analysis and careful performance engineering activities should be executed before additional resources are added to the architecture deployment configuration.

5.2. *Performance Under Attacks*

In this section, we evaluate the performance degradation of the SUT when it is under attack. In our experiments, we executed security intrusions in parallel to the normal workload, and computed the Domain-based metric for each of the considered setups. The attacks were launched with the Mirai botnet. BenchFlow, Mirai, and the SUT were executed on different servers. Table 6 compares the total Domain-based metric computed for the different deployment configurations for "short attacks", "long attacks", and "no attack" tests. Rows in the table show that certain security attacks, such as the one used in our experiments, may influence system performance and therefore impact the Domain-based metric – the effect being more prominent under the "long attack" (20 minutes attack). While in the "no attack" test the Domain-based metric is assessed as $D(\gamma, \mathcal{L}; \mathcal{S}) \simeq 0.55$ under all system configurations, for "short attack" and "long attack", the Domain-based metric varied in a broader range. The worst Domain-based metric assessment is for the 20-minute attack with the deployment configuration of 1 GB RAM, 0.25 CPU and 1 replica, where the Domain-based metric was assessed as $D(\gamma, \mathcal{L}; \mathcal{S}) \simeq 0.407$.

Table 6: Domain-based metric per configuration with and without attacks of different duration. The best configurations with respect to the Domain-based metric are highlighted.

Configuration (α)			Domain-based metric		
RAM	CPU	# Replicas	Attack duration		
			5m	20m	no attack
0.5 GB	0.25	1	0.516	0.541	0.541
0.5 GB	0.5	1	0.517	0.498	0.548
0.5 GB	0.5	2	0.460	0.512	0.541
0.5 GB	**0.5**	**4**	0.514	0.540	**0.550**
1 GB	0.25	1	0.541	0.407	0.539
1 GB	0.25	2	0.517	0.512	0.548
1 GB	**0.25**	**4**	0.540	**0.543**	0.543
1 GB	0.5	1	0.499	0.407	0.541
1 GB	0.5	2	0.473	0.495	0.541
1 GB	**0.5**	**4**	**0.543**	0.467	0.543

Discussion The obtained results indicate that developers of sophisticated algorithms for intrusion detection of performance-impacting attacks may use customer-affecting metrics (e.g., response times) as one of the approaches used to detect intrusions. Specifically, in large mission-critical systems, where performance metrics are baselined and tracked, detecting deviations from the performance baseline could be implemented as a simple add-on to performance tracking.

6. Experiments and Analysis' Variations with Increased Resources

The assessment of the Domain-based metric in virtual environment experiments with limited resources has shown small variability with respect to the architecture deployment configurations evaluated, as shown in Figure 6b. To understand how much the environment influences the experiments, we performed new sets of experiments with increased environmental resources: we changed the configuration of the VMware ESXi-based virtual machine for the SUT to 8 CPUs with 1 core and 16 GB RAM. The variations discussed in the following sections are performed within this new environment.

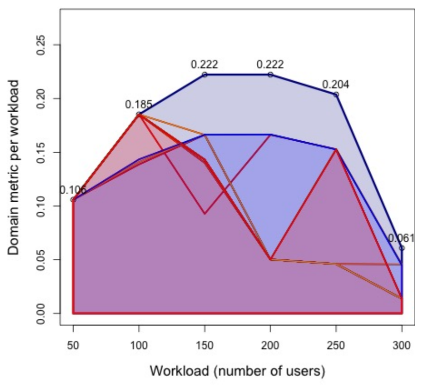
Increased memory. We replicated the experiments in the table of Figure 6c by including new configurations with increased memory sizes, either 8 or 16 GB was used, as shown in the table of Figure 7b. By comparing Figure 6a and the table of Figure 6c with Figure 7a and the table of Figure 7b, respectively, we see that the added memory had little impact on scalability assessment, as the Domain-based metric only improved by about 0.030. Nonetheless, more variation of the Domain-based metric among different configurations can be observed, when comparing Figure 6a and Figure 7a.

Under Attack. In order to obtain the results of Section 5.2, we selected a 20-minutes long security attack. This was done to facilitate the analysis of the impact on the Domain-based metric, as shown in Table 6. In the experiment where additional resources were added to the testing environment, and the security attack was run for 20 minutes, we were able to observe performance degradation of the SUT as reflected by the Domain-based metric, as shown in Figure 8. In particular, the blue area in Figure 8 and first row in Table 6, which represents the Domain-based metric obtained for the best architecture deployment configuration, and the red area in Figure 8, which represents the Domain-based metric obtained best architecture deployment configuration for low loads, were impacted by the security attack.

A different operational profile. To assess the impact of different operational profiles on the proposed methodology, we have repeated the analysis using a different operational profile.

For this analysis, we replicated our study on the operational profile extracted from a Wikimedia dump of accesses to Wikipedia pages during the whole month of July 2016. The scaled loads extracted from such a database are illustrated in Figure 5b. The scalability assessment results obtained when using this new operational profile are shown in Figure 9, where the performance impact of security attacks can be seen in every architecture deployment configuration. The theoretical Domain-based metric maximum is reached up to the workload situations of 100 concurrent users, with and without attacks.

Sensitivity Analysis. To discuss the choice of the baseline threshold, described in Section 3, we performed a sensitivity analysis by scaling the threshold vector by a factor t (scale) between 0.2 and 50, including the case $t = 1$ for the original threshold. Choosing a threshold lower than the original one may result in a lower Domain-based metric as more services would in principle fail. Vice-versa, choosing a threshold greater than the original one may result in a higher Domain-based metric as fewer services would fail. In the latter case, we can study the increase of the Domain-based metric over the scaled thresholds. This will give us a sense of the goodness of the choice of the original threshold. Such analysis is performed for all experiments with and without attack (Figure 8a) and with more resources (Figure 7a). Table 7 summarizes the major findings

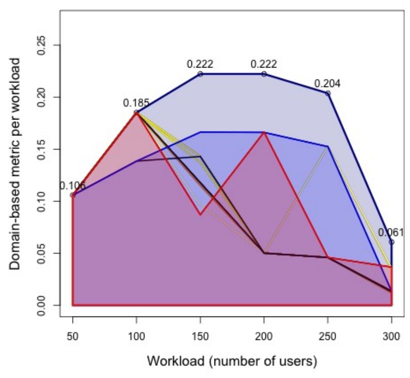


(a) More memory: 8 and 18 GB

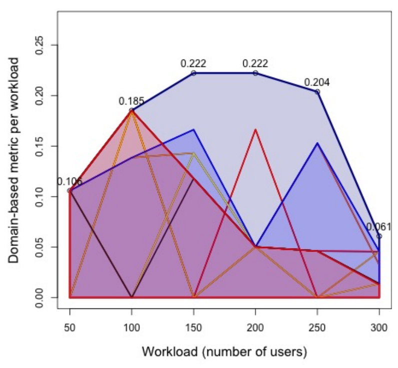
Configuration (α)			D(α, S)
Memory	CPU	# Replicas	More resources
8	**0.25**	**4**	**0.779**
16	0.25	2	0.716
8	0.25	2	0.662
16	0.25	4	0.651
8	0.25	1	0.599
16	0.5	4	0.544
8	0.5	2	0.544
16	0.5	4	0.544
8	0.5	1	0.544
16	0.5	1	0.541
8	0.5	2	0.541
16	0.25	1	0.521

(b) D(α, S) over configurations

Figure 7: Domain-based metric over configurations, per workload situation (a) and total (b) — configurations with 8 or 16 GB of memory — Video streaming profile



(a) No attack

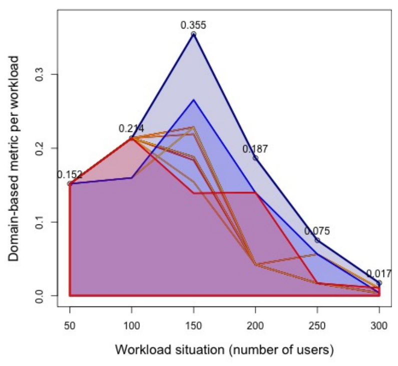


(b) Attack

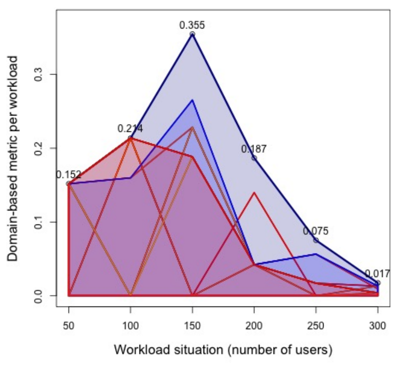
Configuration (α)			D(α,S)	
Memory	CPU	# Replicas	No Attack	Attack
0.5	0.25	2	**0.743**	0.457
1	0.25	2	0.665	**0.66**
0.5	0.25	4	0.627	0.526
1	0.25	4	0.626	0.445
0.5	0.25	1	0.544	0.483
1	0.5	4	0.542	0.249
0.5	0.5	4	0.519	0.519
1	0.5	2	0.519	0.518
0.5	0.5	2	0.519	0.518
1	0.5	1	0.519	0.355
0.5	0.5	1	0.515	0.357
1	0.25	1	0.496	0.355

(c) D(α,S) over configurations

Figure 8: Domain-based metric over configurations per workload situation ((a) and (b)) and total (c), without and with Mirai attack — Video streaming profile



(a) No attack



(b) Attack

Configuration (α)			D(α,S)	
Memory	CPU	Replicas	No Attack	Attack
0.5	0.25	2	**0.777**	0.505
1	0.25	2	0.693	**0.689**
0.5	0.25	4	0.672	0.474
0.5	0.25	1	0.657	0.596
1	0.5	4	0.657	0.26
1	0.25	4	0.629	0.475
1	0.5	2	0.617	0.617
0.5	0.5	2	0.617	0.617
0.5	0.5	4	0.617	0.617
1	0.5	1	0.617	0.443
0.5	0.5	1	0.612	0.443
1	0.25	1	0.603	0.404

(c) D(α,S) over configurations

Figure 9: Domain-based metric over configurations per workload situation ((a) and (b)) and total (c), without and with Mirai attack — Wikipedia

in the three cases. Figure 10 and Figure 11 illustrate the sensitivity analysis in two cases: for the architecture deployment configuration that achieves the best Domain-based metric for low loads, and the overall architecture deployment configuration that achieves the best Domain-based metric.

Figure 10 illustrates the analysis for the configuration that achieved the best value for the Domain-based metric, for low loads, which corresponds to the red area in Figure 8a. In Figure 10, the top six plots show the values of the Domain-based metric over the scale t, for each of the six workload situations (50–300).

Each plot reports both the value of t and the corresponding threshold value on the x-axis and the Domain-based metric for the given threshold on the y-axis. The (blue) dashed horizontal line represents the value of the operational profile for the given workload situation, whereas the (red) dashed vertical line indicates the value of the original threshold, $t = 1$ (i.e., 0.43). In the legend of the plot, we report the workload situation (load), the value of the scale t at which the Domain-based metric reaches its maximum within the interval $[0.2, 50]$, the final gap in percentage between such maximum, and the theoretical maximum of the horizontal dashed line, which is the value obtained from the operational profile.

The polygon and the table at the bottom of Figure 10 illustrate the Domain-based metric over workload situations, under the original threshold (i.e., $t = 1$), for the configuration that achieved the best value of the Domain-based metric of the SUT's normal behavior, as shown in Figure 8a.

The plots show that the theoretical maximum, i.e., the value obtained from the operational profile, is reached within the scale range, for workload situations (loads) up to 150 concurrent users. When the workload situation (load) is increased, the gap between the Domain-based metric and the theoretical maximum is greater than 11%. When using those thresholds, the value of the Domain-based metric can increase (e.g., for the workload situations of 150 and 250). However, there is no unique scale factor within the chosen range that could be adopted for all workload situations.

In addition, with a small increment (at most $t = 1.76$) of the original threshold, in four out of six cases the Domain-based metric value is constant for up to $t = 20$, which indicates that even increasing the original threshold by a scale of 20, the number of failing services (and the Domain-based metric) do not change for the majority of the loads.

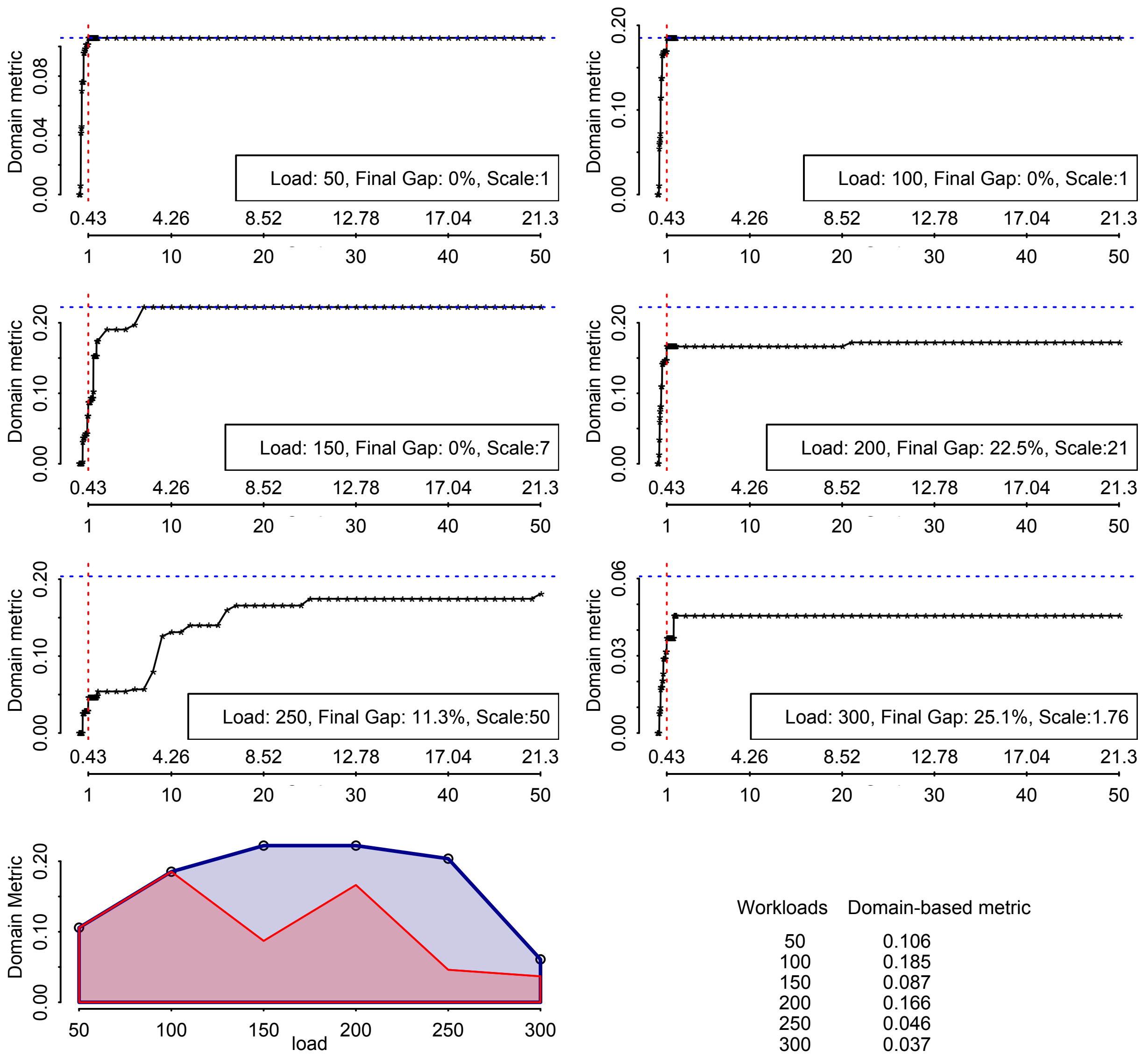
Figure 11 reports the sensitivity analysis for the overall architecture deployment configuration that achieved the best Domain-based metric, as illustrated by the blue area in Figure 8a. Similarly, no unique scale factor within the chosen range can be adopted for all workload situations. Again, with a small increment of the original threshold, in five out of six cases the Domain-based metric value is constant up to $t = 19$ scale.

Table 7 illustrates the sensitivity analysis experimental results with attack, without attack, and with the use of additional resources. The table shows the configuration that achieved the best Domain-based metric, the maximum number of workload situations at which the theoretical maximum obtained from the operational profile is achieved, within the interval $[0.2, 50]$ of the scale t, and the scale at which such optimum is reached. The table shows that regardless of the state of the system (under or no attack) and more resources for the test, the SUT does not reach the theoretical maximum for loads greater than 150 even when the threshold is 50 times the original one.

Experiment	Best Configuration			Max Load	Max scale
	Memory	CPU	Replicas		
No attack	0.5	0.25	2	150	$t = 7$
Attack	1	0.25	2	150	$t = 9$
More resources	8	0.25	4	150	$t = 9$

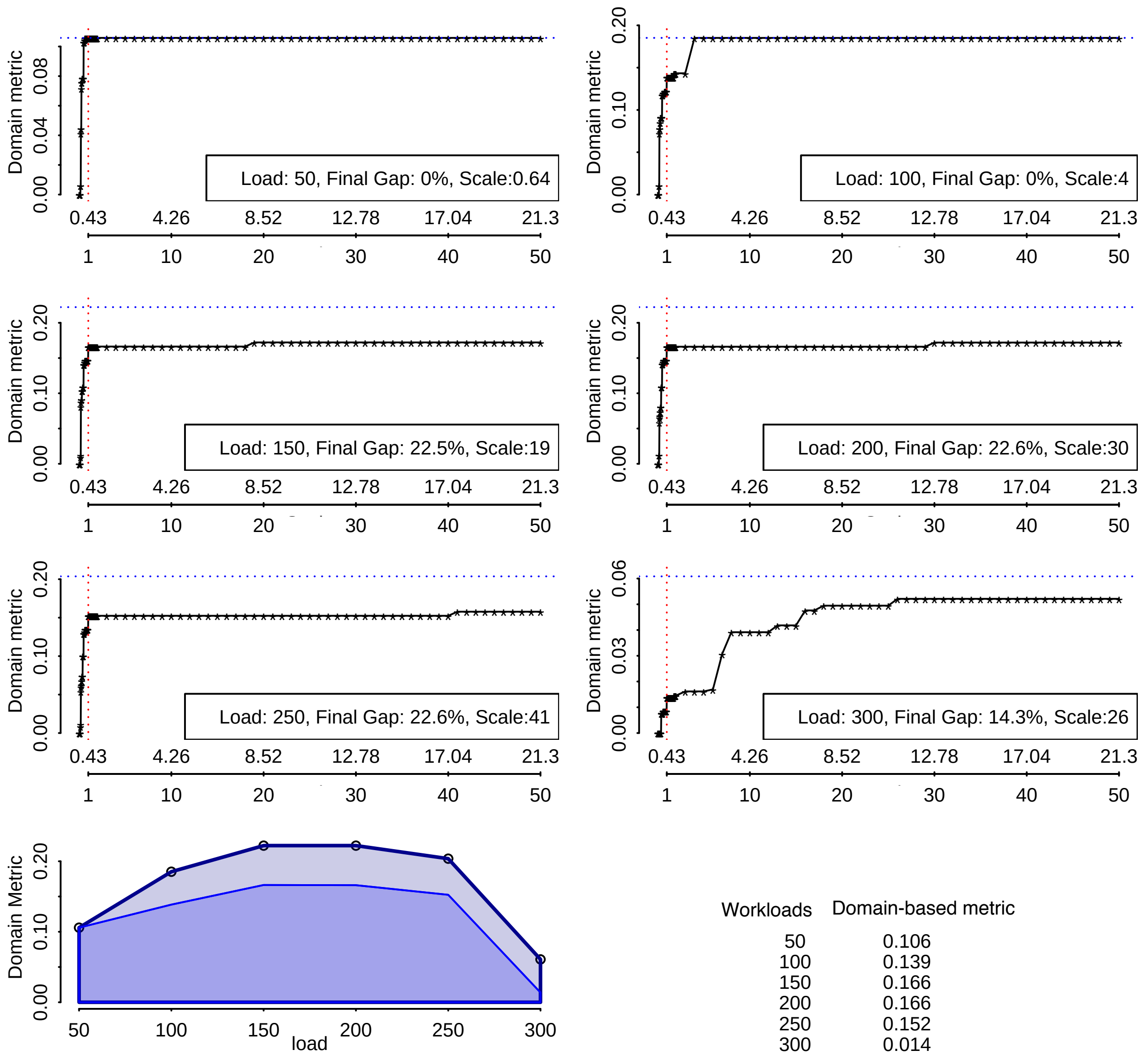
Table 7: Sensitivity analysis for the best configuration of experiment for normal behavior, with attack, and with more resources - operation profile of the streaming video application

Discussion In this section, we have analyzed the sensitivity of the Domain-based metric to the value of the scalability threshold introduced in Step 3 of Section 3.2. Specifically, we were interested in assessing if the selected value of the threshold is sensitive enough to correctly determine the pass/fail criteria for scalability assessment. As it can be seen from Figure 10, for the load levels of 50 and 100 concurrent users, the employed threshold value of 0.43 is sufficient to achieve the fraction of the total probability mass assigned the load levels of 50 and 100. The figure depicts the case of the architecture deployment configuration that achieved the best Domain-based metric, for low workload situations (red polygon in Figure 8a). Therefore, we can observe that for the load levels of 50 and 100, the Domain-based metric scalability is not sensitive to the threshold value, because the system is scalable for these load levels. In contrast, for the load levels of 150, 200, 250 and 300, the scalability metric is sensitive to the threshold multiplication factor, because of the degradation in performance observed for these load levels. For example, for the load level of 150, a scale factor of seven is required to achieve the fraction of the total probability mass of the operational profile associated with 150 concurrent users. For increased load levels, even a threshold scalability factor greater than 50 is required. A similar result holds true for the experiments run under Mirai attack and more resources (shown in Table 7). Figure 11 illustrates the sensitivity analysis for the best configuration (blue polygon in Figure 8a). For this case, the plots show a clear distinction between the workload situations of less than 100 concurrent users, and more than 100 concurrent users. For the workload situation of more than 100 concurrent users, a threshold of over 50 times the original threshold was required to achieve the optimal value of the Domain-based metric. Therefore, we can conclude that using a threshold of $\Gamma = x(\lambda) + 3 \times \sigma(\lambda)$ for the baseline λ (as proposed in Step 3 of Section 3.2) is adequate to detect scalability degradation, as the sensitivity of the Domain-based metric to the threshold value is related to system scalability issues for the larger load level values of 150, 200, 250 and 300.



Workloads	Domain-based metric
50	0.106
100	0.185
150	0.087
200	0.166
250	0.046
300	0.037

Figure 10: Sensitivity analysis for the best configuration for low loads as red polygon in Figure 6a that is illustrated by the plot and the table at the bottom.



Workloads	Domain-based metric
50	0.106
100	0.139
150	0.166
200	0.166
250	0.152
300	0.014

Figure 11: Sensitivity analysis for the best configuration as the dark blue polygon in Figure 6a that is illustrated by the plot and the table at the bottom.

7. Discussion

7.1. Summary of the Analysis

In this section, we summarize the results obtained from the analysis of the experiments and their replications.

Testing infrastructure We have found that adding more CPU power or increasing the number of Docker container replicas in a bare-metal testing infrastructure may not result in system performance improvement. On the other hand, limited resources allocated to a virtual testing infrastructure constraints performance within a narrow range and do not lead to a better performance if the number of users is large. The observed difference in the performance assessment between the bare metal and a virtual testing infrastructure seems to indicate that additional architecture factors may be impacting system performance, such as VMware Hypervisor overhead, I/O bandwidth, etc. Also, in limiting cases measurement noise is common. The methodology results show the importance of executing a detailed performance analysis in such systems.

Having more Docker containers improves the performance of the services regardless of the presence or duration of the attacks. On the other hand, fewer replicas simplify the traceability of the attack. Providing additional resources to the testing infrastructures reduces the need for replicas.

Scalability assessment and detection of security breaches We have shown that the proposed quantitative approach consistently reports the same findings with different operational profiles and more resources allocated to the experiments. We have further demonstrated that using customer-affecting metrics (e.g., response times) to detect intrusions constitutes an efficient and privacy-friendly solution, as those metrics are intrinsically publicly available. Finally, we can conclude that our baseline threshold computed on the statistical distance from the average behavior of a small number of users accessing a relatively large amount of resources for the interaction with the SUT is adequate to detect scalability degradation.

7.2. Threats to Validity

In this work, we have introduced a methodology and a framework for scalability assessment of microservice architecture configurations by leveraging operational profiles and load tests, as illustrated in Figure 1. We have identified the following threats to validity:

Operational profile data analysis. The Domain-based 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-based metric assessments. Several approaches can be used to overcome the lack of accurate operational profile data (Avritzer and Weyuker, 1995), such as: using related systems as a proxy for the SUT, conducting user surveys, and analyzing log data from previous versions of the SUT. In this paper, we opted for the first option and performed the experiments on the SUT based on the operational profile of a video streaming application. This choice might have bound our results to the specific operational profile. Thus, we further replicated the analysis on the Wikipedia operational profile. The results of the two sets of experiments are very similar and support our choice.

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 (Weyuker and Avritzer, 2002), and developing heuristics based on Markovian approximations (Avritzer and Weyuker, 1995). In this work, we used data derived from two different operational profiles, which enriched our studies with different granularities and operational distributions.

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., $3 \times \sigma$) 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. In this work, we have validated our choice by performing a sensitivity analysis on the original threshold.

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. To this extent, we have built a framework called PPTAM that can be easily deployed and redistributed in a production environment (Avritzer et al., 2019). The reproducibility package is also publicly available (Avritzer et al., 2020).

Domain-based metric calculation. The implicit assumption used in the calculation of the Domain-based metric is that the fraction of calls for the service s_j occurs with rate δ_j and can be computed from the application's access log. This assumption can be used to support good heuristics to compute δ_j, for several practical microservice architectures. However, in general, the activation of each of the microservices might not be mutually exclusive. In such cases, the application of queueing network modeling approaches for the computation of the microservices activation rates δ_j from overall input rates and transition probabilities between microservices might be required (Denning and Buzen, 1978).

8. Conclusion

In this paper, we have introduced a new four-step approach for the quantitative assessment of microservice architecture deployment configuration alternatives. Our approach consists of operational profile data analysis, experiment generation, baseline requirements computation, and experiment execution. Our Domain-based metric is computed for each microservice alternative, specified as an architecture deployment 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 testing environment and a virtualized environment. The approach took advantage of the 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. In addition, we were able to fully automatize our approach and provide a framework called PPTAM that can be deployed in any real production environment.

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. We also found that our model is able to capture performance degradation related to intrusions. The approach we proposed is stable under the variation of experimental settings like reference operational profile and baseline threshold to detect service failure.

Acknowledgment

This work has been partly supported by EsulabSolutions, Inc., the German Federal Ministry of Education and Research (grant no. 01IS17010, ContinuITy), the European Union's Horizon 2020 research and innovation programme (grant no. 825040, RADON), 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 for providing the infrastructure.

References

Aderaldo, C.M., Mendona, N.C., Pahl, C., Jamshidi, P., 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 2017), IEEE. pp. 8–13.

Alshuqayran, N., Ali, N., Evans, R., 2016. A systematic mapping study in microservice architecture, in: Proc. IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA 2016), pp. 44–51.

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., et al., 2017. Understanding the Mirai botnet, in: USENIX Security Symposium, pp. 1092–1110.

Avritzer, A., Ferme, V., Janes, A., Russo, B., van Hoorn, A., Schulz, H., Menasché, D., Rufino, V., 2020. Reproducibility package for "Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests". URL: `https://doi.org/10.5281/zenodo.3689500`.

Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., van Hoorn, A., 2018. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing, in: Proceedings of the 12th European Conference on Software Architecture (ECSA), pp. 159–174.

Avritzer, A., Menasché, D.S., Rufino, V., Russo, B., Janes, A., Ferme, V., van Hoorn, A., Schulz, H., 2019. PPTAM: production and performance testing based application monitoring, in: Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 39–40.

Avritzer, A., Tanikella, R., James, K., Cole, R.G., Weyuker, E., 2010. Monitoring for security intrusion using performance signatures, in: Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering (ICPE), ACM. pp. 93–104.

Avritzer, A., Weyuker, E.J., 1995. The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng. 21.

Barker, C., 2016. Mirai (DDoS) source code review. URL: `https://medium.com/@cjbarker/mirai-ddos-source-code-review-57269c4a68f`.

Casalicchio, E., Perciballi, V., 2017. Auto-scaling of containers: The impact of relative and absolute metrics, in: Proc. FAS*W@SASO/ICCAC, pp. 207–214.

Denning, P.J., Buzen, J.P., 1978. The operational analysis of queueing network models. ACM Comput. Surv. 10, 225–261.

Ferme, V., Pautasso, C., 2018. A declarative approach for performance tests execution in continuous software development environments, in: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 261–272.

Francesco, P.D., Malavolta, I., Lago, P., 2017. Research on architecting microservices: Trends, focus, and potential for industrial adoption, in: Proc. 2017 IEEE International Conference on Software Architecture (ICSA 2017), pp. 21–30.

Heger, C., van Hoorn, A., Mann, M., Okanovic, D., 2017. Application performance management: State of the art and challenges for the future, in: Proceedings of the 2017 ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 429–432.

Jiang, Z.M., Hassan, A.E., 2015. A survey on load testing of large-scale software systems. IEEE Trans. Softw. Eng. 41, 1091–1118.

Kambourakis, G., Kolias, C., Stavrou, A., 2017. The Mirai botnet and the IoT zombie armies, in: Proceedings of the Military Communications Conference (MILCOM 2017), IEEE. pp. 267–272.

Milenkoski, A., Vieira, M., Kounev, S., Avritzer, A., Payne, B.D., 2015. Evaluating computer intrusion detection systems: A survey of common practices. ACM Computing Surveys (CSUR) 48, 12.

Newman, S., 2015. Building Microservices. 1st ed., O'Reilly Media, Inc.

Pahl, C., Jamshidi, P., 2016. Microservices: A systematic mapping study, in: Proc. 6th International Conference on Cloud Computing and Services Science (CLOSER 2016), pp. 137–146.

Schulz, H., Okanovic, D., van Hoorn, A., Ferme, V., Pautasso, C., 2019. Behavior-driven load testing using contextual knowledge—approach and experiences, in: Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE), ACM.

Taylor, R.N., Medvidovic, N., Dashofy, E.M., 2009. Software Architecture: Foundations, Theory and Practice. John Wiley & Sons, Inc.

Ueda, T., Nakaike, T., Ohara, M., 2016. Workload characterization for microservices, in: Proc. IISWC, pp. 1–10.

Vögele, C., van Hoorn, A., Schulz, E., Hasselbring, W., Krcmar, H., 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, 443–477.

Weyuker, E.J., Avritzer, A., 2002. A metric for predicting the performance of an application under a growing workload. IBM Syst. J. 41, 45–54.

Weyuker, E.J., Jeng, B., 1991. Analyzing partition testing strategies. IEEE Trans. Softw. Eng. 17, 703–711.