Microservices Integrated Performance and Reliability Testing

Matteo Camilli
matteo.camilli@unibz.it
Free University of Bozen-Bolzano
Bolzano, Italy

Antonio Guerriero
antonio.guerriero@unina.it
Università di Napoli Federico II
Napoli, Italy

Andrea Janes
andrea.janes@unibz.it
Free University of Bozen-Bolzano
Bolzano, Italy

Barbara Russo
barbara.russo@unibz.it
Free University of Bozen-Bolzano
Bolzano, Italy

Stefano Russo
stefano.russo@unina.it
Università di Napoli Federico II
Napoli, Italy

ABSTRACT

Continuous quality assurance for extra-functional properties of modern software systems is today a big challenge as their complexity is constantly increasing to satisfy market demands. This is the case of microservice systems. They provide high control on scale of operation by means of fine-grained service decomposition, but this demands for careful consideration of the relations between performance of individual microservices and service failures.

In this work, we propose MIPaRT, a novel methodology and platform to automatically test microservice operations for performance and reliability in combination. The proposed platform can be integrated into a DevOps cycle to support continuous testing and monitoring by the automatic (1) generation and execution of performance-reliability ex-vivo testing sessions, (2) collection of monitoring data, (3) computation of performance and reliability metrics, and (4) integrated visualization of the results.

We apply our approach by operating the platform on an open source benchmark. Results show that our integrated approach can provide additional insights on performance and reliability behaviour of microservices as well as their mutual relationships.

CCS CONCEPTS

• **Software and its engineering** → *Software performance*; *Software reliability*; *Software verification and validation.*

KEYWORDS

Microservices systems, reliability testing, performance testing

ACM Reference Format:
Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. 2022. Microservices Integrated Performance and Reliability Testing. In *3rd ACM/IEEE International Conference on Automation of Software Test, May 21–22, 2022, Pennsylvania, PA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

AST 2022, May 21–22, 2022, Pennsylvania, PA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The microservices architectural style is widely used by leading companies like Netflix, Amazon, Google, Microsoft to develop large-scale service-based systems composed of loosely coupled services, running in their process, and communicating via lightweight mechanisms, such as RESTful APIs [1]. Microservices systems are usually engineered using DevOps, a set of practices for which operations become part of the development and infrastructure moves into the code (Infrastructure-as-a-code) [2]. To this aim, operations specialists become part of the development teams and system administrators and corporate IT groups are able to write the code that maintains the infrastructure. DevOps aims at reducing the time between committing a change to and the change being deployed to production, while ensuring high quality [3, 4]. In this work, we focus on two fundamental qualities for microservices systems: *performance* and *reliability*, and on their inter-relationship.

Continuous testing and *monitoring* represent two key DevOps practices. Testing provides engineers with a quality feedback at decision gates, to establish if a release candidate is ready for production. To assess whether it meets a desired quality, tests are performed in production, or in a staging environment with realistic users' behaviour and workload intensity [5, 6]. Monitoring is essential in DevOps to collect usage data (how the users interact with the system) and raw measurements data (how the system performs). Such data is typically used by Quality Assurance (QA) teams to drive the testing sessions and ultimately support release decisions [7].

Figure 1 illustrates the main high-level activities to carry out integrated performance and reliability testing of microservices in DevOps cycles. Such an iterative process provides the opportunity to learn from the history of recent executions, due to the availability of online monitoring tools, such as OpenAPM[1]. Historical data can be used to improve the knowledge on the expected workload intensity and the behaviour of the various actors (which we refer to as *behaviour mix*). This information is prone to change dynamically. On the one hand, *evolutionary* changes (e.g., a new release due to new pieces of functionality) can cause changes in the behaviour of the actors or addition of new actors. On the other hand, *operational* changes (e.g., new actors and different behaviour mix) can unveil issues and therefore trigger a new Dev cycle. In this perpetual loop, tracking changes allows proper testing activities to be carried out. Indeed, the behaviour of actors in operation can be very different from the one conceived by the testers before the release.

[1] https://openapm.io/

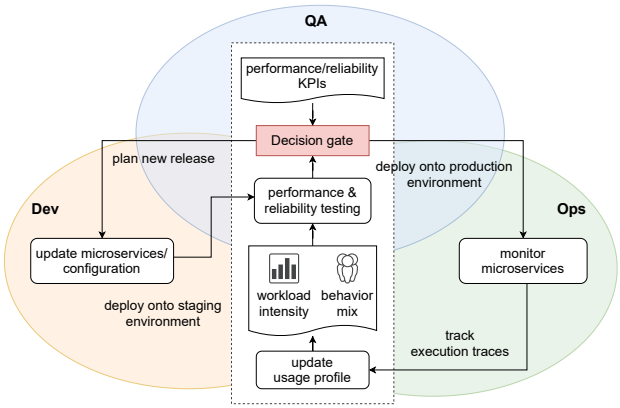


Figure 1: Continuous performance & reliability testing integrated into DevOps cycles.

For instance, some of the actors can change the way they usually interact with the microservices due to unforeseen events, like the circumstances induced by the pandemic. Thus, a central concern in performance and reliability testing is the rigorous characterization of the actors and how their behaviour may lead to technical failures. While performance testing [8, 9] and reliability testing [10, 11] of microservices systems have been studied as separate problems, there exists a striking lack of integrated testing approaches able to assess the two quality attributes and derive mutual relationships.

This paper presents MIPaRT (Microservices Integrated Performance and Reliability Testing), a novel methodology and support platform to automatically execute ex-vivo testing sessions for continuous integrated performance and reliability analysis of microservice systems. In a DevOps process, MIPaRT leverages usage and system data from past Ops phases to automate the generation and execution of performance and reliability tests at a decision gate. It then computes and visualizes Key Performance Indicators (KPIs) of the services exposed to its actors by the system. This way, MIPaRT offers both coarse-grained and fine-grained means to aid engineers to pinpoint problems (e.g. bottlenecks, faulty microservices).

This work aims at answering the following research questions in the context of microservices systems engineering:

RQ1: Does the integrated performance-reliability testing provide advantages compared to the verification of the two qualities in isolation?

RQ2: Can MIPaRT detect existing relations between performance and reliability issues?

To answer these questions, MIPaRT is evaluated through controlled experiments with the Train Ticket microservices system benchmark [12], reproducing two scenarios of evolutionary and operational changes typically occurring in DevOps. The evaluation shows MIPaRT features to support finding performance and reliability issues and their relationships.

The remainder of the paper is as follows. In Sec. 2 we discuss related work. In Sec. 3 we introduce the Train Ticket benchmark used as system under test (SUT) in the evaluation. In Sec. 4 we describe MIPaRT. In Sec. 5 we present the evaluation and we answer to the research questions. In Sec. 6 we discuss threats to validity. Finally, in Sec. 7 we report concluding remarks.

2 RELATED WORK

Related work is examined in the following mainly with reference to performance and reliability quality factors of microservice systems.

Scalability and performance are among the quality attributes of microservices that pose most of the testing challenges [13]. The IT industry considers performance testing as the main pain, demanding for further research efforts [14, 15]. Generating appropriate tests to reveal issues for these qualities is challenging, as the input space is usually large and hard to explore for all microservices. Therefore, trying all possible input value combinations in test generation is impractical and in many cases even infeasible. For instance, only a few input values can detect issues in performance [16], and finding those values is mostly a manual, intellectually intensive and laborious activity [17].

Research and practice generally focus on the separate assessment of each quality attribute. Recent literature proposes ML-techniques [18–20] and symbolic execution methods [21, 22] to detect appropriate input values that can discover performance or reliability issues separately. Their application can be expensive in a DevOps environment, and sometimes they are not more accurate than simple random input generation [23]. Using them for reliability and performance in combination may be even more expensive. In addition, methods based on symbolic execution do not scale to systems with a large set of input data, since the number of paths to search grows exponentially with respect to the input size [24]. Most of these methods are suited for white-box settings, where engineers have access to the source code (e.g. [22]).

We propose here a black-box approach that does not require code availability and is able to automatically detect performance and reliability issues at the level of individual microservice request types. It exploits API specifications to determine valid and invalid request input, building a DTMC model, randomly sampling the partitioned input space, and finally computing and visualizing performance and reliability in combination. The approach generates and executes tests ex-vivo in a non-intrusive way, without instrumenting a specific framework to isolate and test individual microservices, as instead proposed in [25]. The approach can be integrated into the staging environment of a DevOps pipeline, allowing for fast feedback on the discovered issues.

Performance testing. Approaches for performance testing typically follow one of the two strategies: testing to fulfill single user experience (e.g., [26]) or to satisfy a scalability requirement (e.g., [8, 27]). Single-user performance tests evaluate the performance of an application under the load of one user. The purpose of such tests is to understand the control flow of user requests and identify segments that consume the major part of the response time. Such an approach is typically adopted in usability engineering [26]. Scalability testing verifies the target system's ability to meet the performance requirements under demanding load situations [28]. Recent approaches and tools measure performance degra dation of microservices against a benchmark configuration of the SUT that fulfills the given scalability requirement [8, 29, 30]. The approach has been further adopted to automate the detection of performance violations and suggest optimal configurations of microservice decomposition with respect to monolith architectures [31]. The approach has also been used to associate performance degradation

Table 1: Examples of requests for the Train Ticket benchmark.

Request	Core microservice	Relative path from /api/v1	Method	Arguments (with type and constraints)
createUser	adminUserService	/adminuserservice/users	POST	documentNum: numeric (alphanumeric, required), documentType: numeric (positive integer, required), email: string (alphanumeric), gender: numeric, password: string (alphanumeric and special characters, required), userName: string (alphabetic characters), authorization: string (alphanumeric, required)
login	authService	/users/login	POST	username: string (alphanumeric, required), username: string (alphanumeric and special characters, required)
searchTicket	travelService	/travelservice/trips/left	POST	startingPlace: string (alphabetic characters), endPlace: string (alphabetic characters), departureTime: string (date-time format)
booking	preserveService	/preserveservice/preserve	POST	accountId: string (alphanumeric, required), contactsId: string (alphanumeric, required), tripId: string (alphanumeric, required), seatType: string (integer $\geq$ 0), date: string (date-time format), from: string (alphabetic), to: string (alphabetic), assurance: string (integer $\geq$ 0), foodType: numeric (integer $\geq$ 0), foodName: string (alphabetic), foodPrice: numeric (float $\geq$ 0), stationName: string (alphabetic), storeName: string (alphabetic), authorization: string (alphanumeric, required)
getAssuranceTypes	assuranceService	/assuranceservice/assurances/types	GET	authorization: string (alphanumeric, required)
pay	insidePayService	/inside_pay_service/inside_payment	POST	orderId: string (alphanumeric, required), tripId: string (alphanumeric, required), authorization: string (alphanumeric, required)

with the prediction of security attacks [9]. Finally, *chaos engineering* is an emerging approach in industry to evaluate large scale systems by running in-vivo experiments [32]. These experiments carry out performance and scalability tests to identify availability issues that might occur in production (e.g., in Netflix [32, 33]).

Reliability testing. Automated reliability testing of microservices is of paramount importance in DevOps, [11]. Several techniques and tools have been proposed to this aim. DevOpRET is a technique to estimate reliability at the acceptance testing stage in DevOps cycles. Heorhiadi *et al.* propose Gremlin [34], a framework for resilience testing to assess the ability of a microservice system to recover from failures. Jindal *et al.* introduce Terminus [35] to estimate the capacity of a microservice, defined as the maximum number of successfully processed user requests per second, on different deployment configurations via load tests, and fitting a regression model to the acquired performance data. The goal is to define the appropriate resources for each microservice, so that the whole system achieves the best perforan minimizing their overall consumption. Pietrantuono *et al.* developed MART [10] and its enhancement EMART [36] as testing techniques for reliability assessment of microservice-based system, starting from the definition of an *operational profile*, namely of the expected usage in operation.

3 BENCHMARK MICROSERVICES SYSTEM

We describe and evaluate MIPaRT with reference to a benchmark microservice system used in software engineering research, called Train Ticket[2]. This containerized train ticket booking application runs onto 41 microservices implemented by using a modern technology stack, as described in [37]. The benchmark has been selected according to a number of criteria, including: (*i*) usage of well-established microservice architectural patterns; (*ii*) possibility of using automated deployment practices in software containers; (*iii*) support for different deployment configuration options.

Table 1 lists a number of services available to Train Ticket users. Users access the system through a web interface that basically allows tickets to be searched, reserved, bought, and refunded. For each service, the business logic involves a number of microservices; among them, a core RESTful microservice is identified, listed in the second column in Table 1; the remaining columns list the core microservice relative path, the arguments to carry out a user request, and constraints on their values, as per the documentation. For instance, a *guest user* can search (travelservice) a train from a source city to a destination at desired date and time. A *registered user* can log in (login) and then book a ticket (preserveService), specifying the passenger, the class of the seat, and the assurance type (assuranceService). Upon successful booking, the user is required to pay (insidePayService). A user can also change a ticket (subject to time limitations) or ask for refund. An *administrator user* can register new users (adminUserService) and add, delete, or change the information of trains.

[2]Train Ticket is an open source project. Sources and documentation are available at https://github.com/FudanSELab/train-ticket.

4 MIPART

4.1 Overview

As anticipated in Sec. 1, changes to a deployed system (namely, to its microservices) typically occur frequently, in short DevOps cycles. Failures, as well as performance degradation, may occur due to changes in the users behaviour (e.g., most invoked services) and in the workload (e.g., number of concurrent users). Both new releases (evolutionary) and usage profile (operational) changes are tracked and monitored in a DevOps process. This offers the opportunity to QA engineers to update the knowledge on the operating conditions, carry out proper testing activities, take decision at quality gates, and provide feedback to the other teams. MIPaRT performs automated performance and reliability testing sessions as part of the QA stage of a DevOps cycle, triggered by evolutionary or operational changes (e.g., a new release of a microservice, or a change in the workload).

The methodology follows three stages shown in Fig. 2:

i) definition of the *operating conditions* (based on the usage data collected from Ops), composed of workload intensity and behaviour of the actors (Sec. 4.2);
ii) execution of ex-vivo testing sessions, loading the SUT with the specified workloads (Sec. 4.3);
iii) integrated analysis, fed by raw measurements, to compute and visualize performance and reliability estimates (Sec. 4.4).

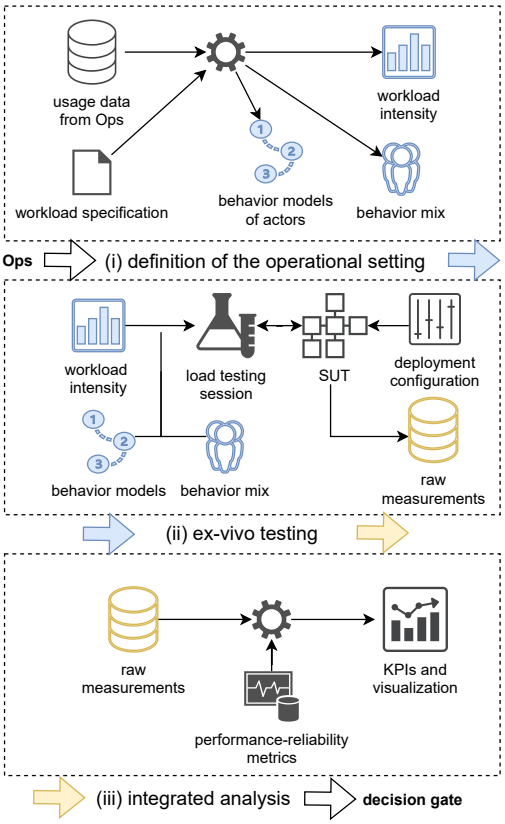


Figure 2: The three stages of MIPaRT.

4.2 Definition of the operating conditions

The first stage consists in defining the operating conditions to be reproduced for testing the system. Such a definition extends the one introduced in [9] and it includes the following elements:

- the *workload specification* that describes allowed requests that a user can invoke on the SUT together with details on the way to generate the requests to each operation (i.e., relative paths, parameters, and constraints, as shown in Table 1);
- a set of *behavioural models*, each providing a stochastic representation of user sessions in terms of (valid and invalid) requests generated according to the workload specification;
- a *workload intensity* value: the expected number of concurrent users, likely to access the system in operation.
- a *behaviour mix*, namely a distribution of frequencies of behavioural models, representing their occurrence probability within the defined workload intensity.

A user interacts with the system according to a given behavioural model. The model is generated by combining the information extracted from the documentation (i.e., the workload specification) and the frequency of requests issued by different actors extracted from the usage data. For example, a possible actor for Train Ticket is the *guest* who searches for tickets without logging in, while the *buyer* is a logged-in actor who searches and then reserves a ticket. The buyer may perform the following sequence of requests: visit the home, login, search ticket, book a ticket, and then pay.

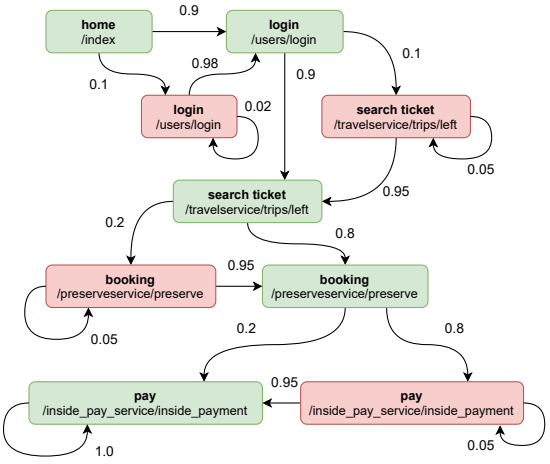


Figure 3: A DTMC behavioural model for the *buyer* actor.

In MIPaRT, we propose a behavioural model that provides a probabilistic representation of user sessions in terms of a Discrete Time Markov Chain (DTMC) [38]. Here, we extend the modeling approach introduced in [39] by additionally considering the input space in the construction of the Markov chain. Thus, *the DTMC is the main building block of our integrated approach.* It is the model that drives the testing activity and then the integrated reliability and performance assessment. Essentially, the nodes of the DTMC model represent the requests that can be issued to the system by providing either a *valid* or *invalid* input values, according to the API specification. Thus, the input space for each request is partitioned into valid and invalid classes, henceforth referred to as *request classes*. The transitions (i.e., weighted edges) in the DTMC specify the probability of moving from a given request class to the next one. Figure 3 shows an example of DTMC for the buyer actor. Green nodes model valid requests, whereas red nodes model invalid ones. For instance, from the valid request `login`, a buyer can move to the valid request `searchticket` with probability 0.9 and to the invalid request `searchticket` with probability 0.1. Based on the API relative path associated with each DTMC node, we can also determine the core microservice in charge of handling the requests (second column in Table 1). For instance, in Fig. 3, the request `search_ticket` maps to `travelService`. The DTMCs are used to drive the generation of instances of synthetic users (i.e., actors) for the testing sessions. The behaviour mix defines the percentage of concurrent users to be sampled for each actor. For instance, for a workload intensity of N concurrent users, the following behaviour mix:

$$(\textit{guest}\text{: } 0.5;\ \textit{buyer}\text{: } 0.3;\ \textit{refund\_claimer}\text{: } 0.2) \tag{1}$$

is used to emulate a scenario where 50% of the N users are guests, 30% of them carry out a reservation, and 20% request refunding.

The operating conditions (behavioural models, behaviour mix, and workload intensity) are typically extracted automatically from the usage data collected during the Ops stages of a DevOps cycle and raw sessions are automatically recorded in session logs and then analyzed to extract the workload intensity and DTMCs using *clustering* algorithms [11, 39]. In this case, a cluster represents an

actor and is a set of sessions represented by similar DTMCs. Thus, to automatically generate the operating conditions, we first need the following data in a session log: "session identifier", "request start time", "request end time", "request relative path"and combinations of "valid" and "invalid values" for the arguments of each request. Once the DTMCs are generated, the frequency associated with DTMC is computed as frequencies of sessions in clusters over all sessions. Thus, the frequencies defines the empirical categorical distribution for workload intensity.

4.3 Ex-vivo testing

In this stage, joint performance/reliability tests are performed ex-vivo in the operational environment[3]. The SUT is deployed at the beginning of each test session (and un-deployed at the end), then loaded with synthetically generated users that replicate the operating conditions of interest. The sessions are generated and then orchestrated according to the following factors defined by the tester:

- the DTMC behavioural models of the users;
- the behaviour mix categorical distribution;
- a set Λ of workload intensity values;
- a set of deployment configurations C (e.g., memory, CPU, and replicas per each microservice).

For each pair $\langle \lambda, c \rangle \in \Lambda \times C$, the SUT is deployed by using the configuration c. Thus, the testing session starts and generates the workload intensity λ. Each actor instance is drawn with a probability of the actor's behaviour mix. Given an actor instance, the testing process automatically samples requests as well as inputs according to the corresponding DTMC. Namely, each input is generated by drawing from one of the two classes according to the current node and outgoing transition probability. For instance, according to Fig. 3, a *buyer* instance from the state `login`, can either perform a search with a valid input (with probability 0.9) or an invalid one (probability 0.1). An invalid search request can be issued, for example, by inserting special symbols in the argument `startingPlace`, or by using a wrong date-time format for the `departureTime` argument. Between each request the process applies a pseudo-random *think time* using an exponential distribution (with average inter-arrival time between 1 and 5 seconds) to represent realistic user behaviour.

During all the testing sessions, we collect raw measurement data, that are then used in the integrated performance and reliability analysis and visualization as described in the following.

4.4 Performance-reliability analysis

4.4.1 Metrics. The analysis starts by estimating performance and reliability during the observation period T (i.e., duration of a test session) for each request class p (e.g., $\texttt{login}_{\texttt{valid}}$).

For each class p, we define the *Performance estimator*, $\hat{P}(p)$, as the normalized distance from the average response time $\mu(p)$ to a performance *threshold* $L(p)$:

$$\hat{P}(p) = \begin{cases} \frac{L(p)-\mu(p)}{L(p)} & \mu(p) < L(p) \\ 0 & otherwise \end{cases} \tag{2}$$

[3] According to [40], ex-vivo testing "indicates any type of software testing performed in-house using information extracted from the field".

The lower the value, the worse is performance. It is worth noting that the parametric threshold $L(p)$ in Eq. 2 can be set for any class p. There are essentially two ways known in literature to set this threshold: according to a user-based experience [26] or a scalability requirement [8]. The former approach follows usability engineering practices for web-based applications. In this case, $L(p)$ can be set to 1 *sec* if we want to represent the limit for the user's flow of thought to stay uninterrupted, or 10 *sec* for keeping the user's attention focused. According to the latter approach and existing literature [9, 30], $L(p)$ can be empirically derived as a *scalability threshold*: $L(p) = \mu_0(p) + 3 \cdot \sigma_0(p)$, with $\mu_0(p)$ and $\sigma_0(p)$ average and standard deviation of the response time for the request class p, measured during a testing session carried out under ideal operating conditions, like a small number of users and full availability of system resources. We further define *Performance Degradation* (PD) as $1-\hat{P}(p)$, so that the higher its value, the worse is the performance.

We then define the *Reliability estimator*, $\hat{R}(p)$, as the ratio of non-failing requests in T, according to the *Nelson–Aalen* non-parametric estimator [36, 41]:

$$\hat{R}(p) = 1 - \frac{F(p)}{N(p)} \tag{3}$$

with $N(p)$ total number of issued requests in p, and $F(p)$ number of failed requests in p, so that the lower the value, the worse is reliability. Then we define the ratio of *Failed Requests* (FR) as $1-\hat{R}(p)$, so that higher values correspond to worse reliability. In our work, detect a failure or success of a request on the HTTP status code. Specifically, every status code other than `2xx` (success) is considered as a failed request. In our experiments, we empirically observed recurring issues that we grouped into the two default categories reported above: *server errors* (`500` and `502` response codes) and *connection errors* (codes `503` and `504`).

To investigate issues associated with performance and reliability at finer level, MIPaRT provides engineers with additional metrics for each request class p:

- *Request Ratio* (RR): ratio of requests in class p over of all the requests of the test session.
- *Connection Errors ratio* (CE): requests that return a connection error out of all the failed requests in p over of all the requests of the test session.
- *Server Errors ratio* (SE): requests that return a server error out of all the failed requests in p over of all the requests of the test session.

4.4.2 Visualization. To detect performance and reliability issues of a request class p, MIPaRT compares the values of the performance and reliability estimators, $\hat{P}(p)$ and $\hat{R}(p)$, in the so-called *criticality plot* shown in Fig. 4. Each class p yields a point in the coordinate space $(\hat{P}, \hat{R})$, which, for demonstration purpose, we have divided in three areas corresponding to high, medium, and low criticality levels, according the the Euclidean distance from the most critical point ($\hat{P} = 0, \hat{R} = 0$). In Figure 4, all requests in the class $\texttt{pay}_{\texttt{valid}}$ have no issue (i.e., $\hat{P}(p) = \hat{R}(p) = 1$), while invalid requests ($\texttt{pay}_{\texttt{invalid}}$) yield reliability issues (i.e., $\hat{R}(p) = 0$).

MIPaRT visualizes the five metrics RR, CR, SE, FR, and PD in a radar plot for each request class p as illustrated in Fig. 5. Based on the area identified by the radar coordinates, engineers can quantify

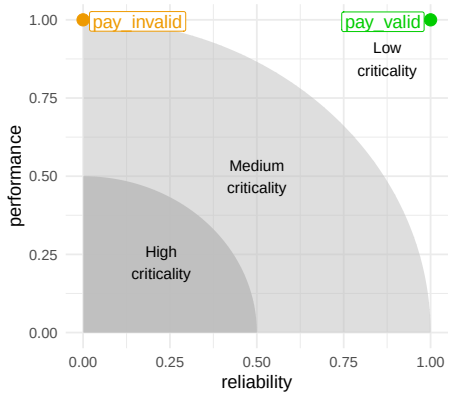
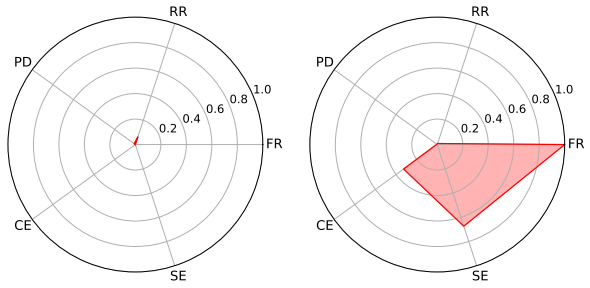


Figure 4: Criticality plot for performance-reliability analysis of request classes.



(a) $\text{pay}_{\texttt{valid}}$ **class** **(b)** $\text{pay}_{\texttt{invalid}}$ **class**

Figure 5: Sample radar plots for requests to the pay **service.**

and compare performance and reliability of each request class detected as high critical in criticality plot. Using Table 1, engineers can further trace these results back to the core microservices and prioritise their maintenance activities to these microservices. Figure 5a shows the radar plot for the $\text{pay}_{\texttt{valid}}$ class. In this case, the *request ratio* (RR) is low and all the other indicators are zero, meaning proper service operation under a relatively small workload intensity. Figure 5b shows another example for the $\text{pay}_{\texttt{invalid}}$ class. Even though the request ratio is very small (RR close to zero, though not null) the class exhibits severe issues according to the percentage of *failed requests* out of the total number of requests issued to *p* (FR axis). More than 60% of the failed requests are *server errors* (SE axis), while almost 40% of the failed requests are *connection errors* (CE axis). Performance related indices show instead good results. Indeed, the *performance degradation* (PD axis) is close to 0. These results suggest engineers investigating the presence of software defects causing fast failures in the management of invalid requests to the pay service.

4.5 Integrated platform

MIPaRT is fully automated and requires the following inputs: the RESTful API specification, the target operating conditions, and the performance threshold for each class of requests.

The software platform supporting the MIPaRT methodology is implemented using Python3 and our in-house developed tool PPTAM [29, 42][4]. The platform integrates and orchestrates multiple modules that collectively realize the main stages of the approach. The operational conditions sampled from Ops, are defined in a declarative manner through the BenchFlow domain-specific language [43]. By using the language, the tester essentially declares: the DTMC behavioural models of the actors, the behaviour mix, the set of workload intensities, and one or more deployment configurations. We make use of Docker (https://www.docker.com) to deploy/undeploy the microservices of the Train Ticket benchmark onto an in-house testing environment composed of two virtualized computing units: the *driver unit* (running the testing sessions), and the *SUT unit* (running the SUT). Once the SUT is deployed onto its unit, the orchestrator spawns one or more testing sessions according to the BenchFlow declaration. The framework Locust (http://locust.io) is used to generate the workload intensity according to the behavioural models and the behaviour mix. The classes can be automatically generated by using EvoMaster (www.evomaster.org) provided that the RESTful API includes a schema in OpenAPI/Swagger format (https://swagger.io/specification/). Raw measurements are collected during each test session to compute the performance and reliability estimators as well as the additional indices per each individual class. At the end of the sessions, the tester visualizes the criticality plot and the radars in a interactive notebook implemented using Apache Zeppelin (https://zeppelin.apache.org/).

5 EVALUATION

In this section we discuss our experience in using MIPaRT on different versions of the Train Ticket benchmark. We first introduce two realistic high-level scenarios in Sec. 5.1, where testers may benefit from MIPaRT. We then present the design of controlled experiments in Sec. 5.2 and, finally, we present and discuss the results in Sec. 5.3 and Sec. 5.4, respectively.

5.1 Scenarios

The following two scenarios exemplify how MIPaRT help engineers detect performance and reliability issues of a microservice system.

Scenario 1 (evolutionary change). This scenario emulates the modification or addition of microservices of the target system, due to changes in requirements or user preferences. Evolutionary changes or, more in general, maintenance to individual microservices may alter reliability/performance of the exposed functions. This is, for instance, the case of login or the search of new tickets, two important functions for Train Ticket. For this reason, a new QA phase is required since it may trigger additional development and new release cycles. We reproduced this scenario, starting from version v1 of Train Ticket, and then introducing a version v2, to be assessed in a QA phase after a new Dev phase. The two versions of Train Ticket adopt alternative implementations of the microservices `travelService`, `adminUserService`, and `authService`.

Scenario 2 (operational change). This scenario emulates an unexpected increase of the amount of concurrent users (workload intensity), or unforeseen changes of their behaviour possibly caused

[4]Open source software publicly available at https://github.com/pptam/pptam-tool.

by the release of a new piece of functionality or even by external factors. For example, in a pandemic situation an unexpected number of users may cancel issued tickets and ask for refund. This situation yields different interaction patterns and the microservices involved in issuing vouchers may exhibit performance and reliability issues. We reproduced this scenario by injecting changes in the way the system is used during an Ops phase. Specifically, we assume that due to the pandemic, users are "less prone" to buy tickets than usual, and increasingly prone to ask for refund. This causes changes in the DTMCs as well as their behaviour mix.

5.2 Experiments

We designed a set of controlled experiments, operating MIPaRT with the benchmark system under defined operating conditions and collected raw measurements to carry out the integrated analysis. The machines used as ex-vivo testing infrastructure have the following characteristics:

- MIPaRT node: 4 GB RAM, 1 CPU at 2.6 GHz;
- SUT node: 16 GB RAM, 8 CPUs at 2.6 GHz;
- both nodes: magnetic disks with 15,000 rpm, 10 Gbit/s network connections.

We deployed two versions of Train Ticket and varied the operating conditions, reproducing the two DevOps scenarios described in Section 5.1. Overall, we identified four actors for the two scenarios:

- *guest*: search for travel options without logging in;
- *buyer*: search for travel options, log in and buy tickets;
- *renouncing*: log in and cancel a reservation;
- *refund_claimer*: log in and claim refunding of the cost of a previously issued ticket.

The behaviour of each of these actors is described by a DTMC model that drives the generation of the requests in each class. In all testing sessions, we set performance threshold $L(p) = 10\ sec$ for all classes p (i.e., limit for keeping the user's attention focused in web applications) according to user-based experience practice [26] as described in Section 4.4.

5.3 Results

In this section, we illustrates the results for the two considered scenarios (evolutionary and operational change).

5.3.1 Scenario 1 (evolutionary change). In this scenario, we have three actors, (guest, buyer, and renouncing) operating the two Train Ticket versions `v1` and `v2` with the following behaviour mix:

$$(guest : 0.5; buyer : 0.4; renouncing : 0.1) \quad (4)$$

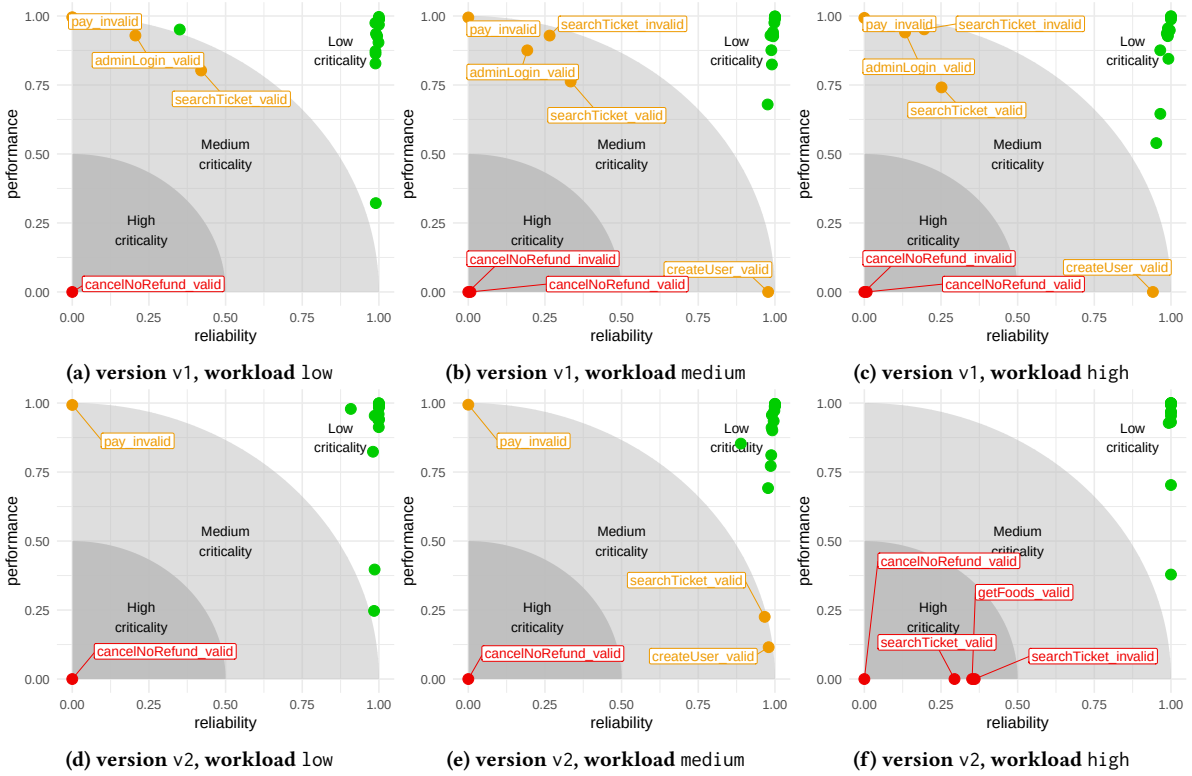
under three workload intensities: `low` (150 users); `medium` (200 users), and `high` (250 users). We execute a testing session for each triplet (version, behaviour mix, workload intensity), for a total of 6 sessions. Each testing session lasts 20 minutes in which we sample more than $15k$ requests according to the three DTMCs and their mix. The criticality plots in Fig. 6 show the results for each test of the two versions `v1`, `v2` (labels of input classes in the low criticality region are not shown for the sake of readability). The plots for version `v1` indicate an increase of the number of problematic request classes with the increase of the workload intensity. For some of the classes, such a criticality affects both performance and reliability (e.g., $\texttt{cancelNoRefund}_{\texttt{invalid}}$). In version `v2`, we can observe a general decrease in the number of problematic classes. Nevertheless, the high workload yields a larger number of high-critical classes than in `v1`. These classes exhibit a substantial performance degradation and an unreliable behaviour with high workload as shown in Fig. 6f. The comparison also shows that requests `searchTicket` in version `v2` are problematic with high workload both for valid and invalid inputs. The problem refers to a drop in performance in version `v2`, although none of the two implementations appears to solve the reliability issues of the service (see Fig. 6c and Fig. 6f). According to Table 1, the core microservice in charge of handling such request classes is `travelService`. With this information, engineers might prioritize its maintenance in a future Dev phase.

By expanding the analysis to all five metrics of MIPaRT, we can have a better understanding of the type of failures the services experiences for Scenario 1. Table 2 shows a summary of the relevant issues per request class and test session according to the following eight cases[5]:

- *Performance and Reliability issues (Perf&Rel)*: when FR > 0.05, PD > 0.1, CE > 0, and SE > 0;
- *Performance issues (Perf)*: when PD > 0.5;
- *Reliability issues (Rel)*: when FR > 0.5 and SE > 0;
- *Connection and Server errors (Conn&Serv)*: when SE > 0 and CE > 0;
- *Connection errors (Conn)*: when only CE > 0;
- *Server errors (Serv)*: when only SE > 0;
- *No criticality (ok)*: if there is at least one request for the considered input class, and none of previous cases applies;
- *No requests (noReq)*: zero requests issued to that class.

The cases have been conceived to indicate which of the issues may originate from both performance and reliability problems. For instance, the case $Perf\&Rel$ detects those classes for which reliability and performance are not satisfactory (although they may be not highly problematic as for the cases $Perf$ or Rel), but whose requests come back with both connection (e.g., load or transmission problems) and server errors (e.g., code problems). This case is subtle and may not be easily caught by a tester as FR and PD of these requests may not be as high as for the cases $Perf$ or Rel. For instance, although $\texttt{cancelNoRefund}_{\texttt{valid}}$ is always in the origin of the criticality plot, it is related only to *Performance issues*. This depends on a high response time (always over the threshold), related to CEs occurring for each request. Moreover, we observe that the changes introduced in `v2` improve reliability for the class $\texttt{createUser}_{\texttt{valid}}$ (mapping to `adminUserService`) and $\texttt{login}_{\texttt{valid}}$ (mapping to `authService`). The valid requests issued to `searchTicket` (a core microservice for Train Ticket) deserve specific attention: even though both performance and reliability of $\texttt{searchTicket}_{valid}$ degrade as the workload increases, for version `v1` yields a medium criticality level for all workload intensities. For version `v2` the criticality level increases even more with the workload intensity. Under workloads `low` and `medium`, the reliability is close to 1.0, whereas the PD increases from 0.60 to 0.78. Under `high` workload, we observe a drop in reliability, bringing the class to the highest criticality level. In Scenario 1, Table 2 helps identify those

[5]The thresholds have been chosen only to exemplify our approach on Train Ticket and are not intended to be generally valid for other systems.



(a) **version** v1, **workload** low (b) **version** v1, **workload** medium (c) **version** v1, **workload** high

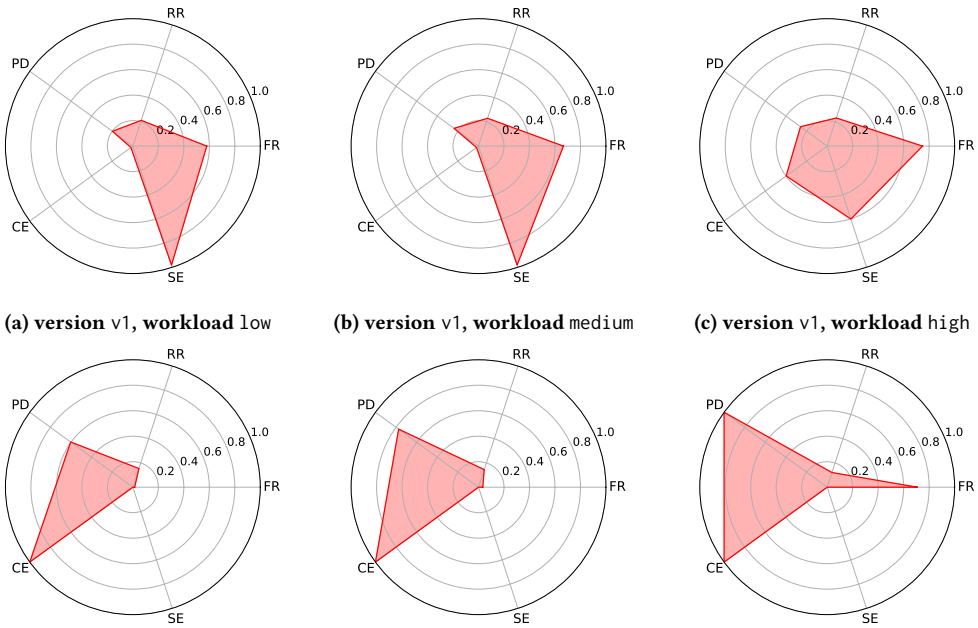
(d) **version** v2, **workload** low (e) **version** v2, **workload** medium (f) **version** v2, **workload** high

Figure 6: Scenario 1 – Criticality plot before (version v1) and after a change (v2) for various workload intensities.

Table 2: Scenario 1 – Summary of the detected issues per classes of user input.

user requests (input class)	version v1 workload low	version v1 workload medium	version v1 workload high	version v2 workload low	version v2 workload medium	version v2 workload high
$login_{valid}$	Conn&Serv	Conn	Conn&Serv	ok	Conn	ok
$login_{invalid}$	Conn	Conn	Conn	ok	Conn	ok
$searchTicket_{valid}$	Perf&Rel	Perf&Rel	Perf&Rel	Perf	Perf	Perf
$searchTicket_{invalid}$	Rel	Rel	Rel	Serv	Perf&Rel	Perf&Rel
pay_{valid}	ok	ok	ok	ok	ok	Perf
$pay_{invalid}$	Rel	Rel	Rel	Rel	Rel	noReq
$booking_{valid}$	Conn&Serv	Conn&Serv	Conn&Serv	Conn&Serv	Serv	Serv
$booking_{invalid}$	ok	Serv	Conn&Serv	Serv	Serv	ok
$cancelNoRefund_{valid}$	Perf	Perf	Perf	Perf	Perf	Perf
$cancelNoRefund_{invalid}$	noReq	Perf	Perf	noReq	noReq	noReq
$createUser_{valid}$	Perf	Perf	Perf&Rel	Perf	Perf	ok
$navigatetoClientLogin_{valid}$	ok	ok	Conn	ok	ok	ok
$adminLogin_{valid}$	Rel	Perf&Rel	Rel	Conn	Conn	ok
$getAssuranceTypes_{valid}$	Conn	Conn	Conn	ok	Conn	ok
$getFoods_{valid}$	Conn&Serv	Conn	Conn	Conn	Conn	Perf
$home_{valid}$	Conn	Conn	Conn	ok	ok	ok
$selectContact_{valid}$	Conn	Conn	Conn	Conn	Conn	ok
$selectOrder_{valid}$	Conn	Conn	Conn	ok	Conn	ok

request classes that deserve more attention and be further analysed through the MIPaRT radar plots. For example, the analysis in Table 2 reports *Perf&Rel* issues for the request class $searchTicket_{valid}$ for all loads in version $v1$, whereas *Perf* problems for all load of version $v2$. The radar plots for such class are illustrated in Figure 7. The radars of the class in version $v1$ show that under low and medium workload intensity, the requests show a low failure rate (FR) all due to server errors, whereas some connection errors occur



(a) version v1**, workload** low **(b) version** v1**, workload** medium **(c) version** v1**, workload** high

(d) version v2**, workload** low **(e) version** v2**, workload** medium **(f) version** v2**, workload** high

Figure 7: Scenario 1 – Radar plots for the request class $\texttt{searchTicket}_{valid}$ **per version and workload intensity.**

at high workload intensity. The areas in the radars also quantifies the evolution of such change. With version *v*2, performance degrades from 0.60 to 1 with the increase of the workload and the few errors are only associated to the connection. This degradation causes the accumulation of pending requests into request queue of the microservice `travelService`. This in turn causes saturation of the resources that makes the `travelService` temporarily unable to handle the requests, triggering a visible manifestation in terms of reliability drop.

5.3.2 Scenario 2 (operational change). The experiments considering this second scenario aim at showing MIPaRT ability to quantify the impact of operational changes onto performance/reliability exposed by the system in production. We considered two operational conditions, testing the same version of Train Ticket v2 under low workload intensity, adopting the two following behaviour mix:

$$(guest : 0.5; buyer : 0.4; renouncing : 0.1; refund\_claimer : 0.0) \quad (5)$$

$$(guest : 0.3; buyer : 0.4; renouncing : 0.0; refund\_claimer : 0.3) \quad (6)$$

Figure 8 shows the effect of the operational change in terms of number of requests to request classes. For instance, that the amount of requests issued to obtain a refund voucher increases after the change (i.e., behaviour mix in Eq. 6). We also observe that the occurrences of the other classes decrease, except for $\texttt{login}_{invalid}$, $\texttt{pay}_{valid}$, $\texttt{pay}_{invalid}$, and $\texttt{searchTicket}_{valid}$. Figure 9 shows the criticality plot for the two testing sessions. The red data points represent the results obtained with the behaviour mix in Eq. 5 (pre-change), whereas the blue data points represent the results obtained with the behaviour mix in Eq. 6 (post-change). The plot highlights the effects of the operational change on performance and reliability associated to input partitions. We can make for instance the following observations. According to Fig. 8, the voucher is never requested before the operational change: both performance and reliability issues associated with $\texttt{getVoucher}_{valid}$ requests occur after the operational change and they are detected and made visible by MIPaRT. The reliability issues associated with requests $\texttt{pay}_{invalid}$ are detected also after the operational change. This is consistent with Fig. 8 that shows a comparable number of occurrences of this request class.

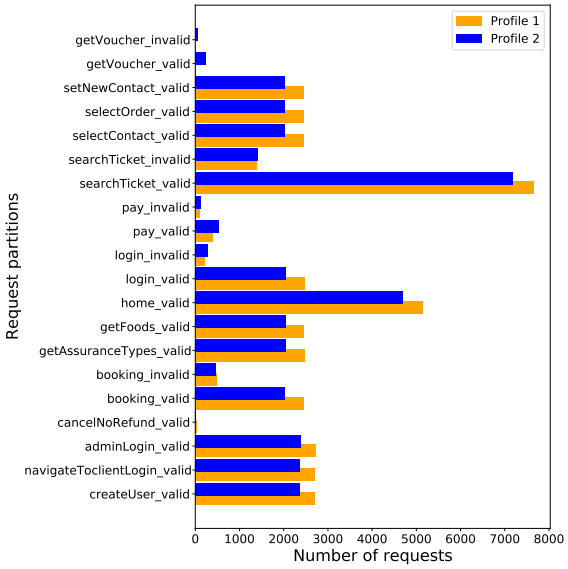


Figure 8: Scenario 2 – Request count of input classes.

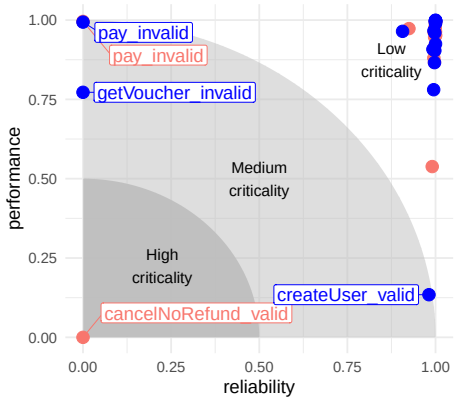


Figure 9: Scenario 2 – Criticality plot showing the effects of operational changes.

5.4 Discussion of the results

From the application of MIPaRT and its visualization layer (Scenario 1), we can provide the QA team with insights on performance, reliability, and performance and reliability jointly. Thus, MIPaRT elevates the attention on those request classes, that developers may consider more critical according to the analysis performed with MIPaRT and decide their Dev activities of the future DevOps cycles. In particular, developers are able to detect issues and then prioritize activities on those microservices whose requests may not evidently show issues of performance or reliability only. MIPaRT captures variation of such issues under operational changes (Scenario 2).

RQ1 Summary

Does the integrated performance-reliability testing provide advantages compared to the verification of the two qualities in isolation?
The integrated testing approach yields additional insights compared to performing performance or reliability testing in isolation. The future Dev activities can be prioritized accounting for both extra-functional qualities. Furthermore, MIPaRT quantifies the joint effect on both performance and reliability caused by evolutionary/operational changes.

With our platform, developers can also have further insights on the nature of the issues of between performance and reliability (Scenario 1, table summary and radar plots) and how such issues can be affected by a specific operational change (Scenario 2).

RQ2 Summary

Can MIPaRT detect existing relations between performance and reliability issues?
According to our controlled experiments, MIPaRT is able to detect some relations between the two extra-functional qualities. In particular, by using the radar plot visualization we have been able to characterize reliability issues either as fast failures possibly caused by implementation defects or saturation of resources caused by performance issues.

6 THREATS TO VALIDITY

External validity threats of this work concern the replication of our experience to other systems or settings. We addressed them selecting a representative benchmark in MSA research and adopting a common technology stack in microservices systems, as described in [37]. We also built an ex-vivo testing environment using a modern infrastructure supporting continuous deployment. As described in [11], this represents a common setting for DevOps practices.

Threats to *internal* validity could limit the extent to which the results obtained in the evaluation support our claims. We mitigated these threats through a careful design of the two scenarios of interest (i.e., evolutionary and operational changes) as well as the controlled experiments.

Construct validity threats concerns possible misinterpretation of what our measures reflect in our controlled experiments. Thus, we mitigated these threats by assessing the metrics used in MIPaRT. Reliability has been measured by using the Nelson-Aalen non-parametric estimator that represents a de-facto standard in software reliability engineering [36]. Performance has been measured based on well-established practices according to the usability engineering guideline presented in [26].

Conclusion validity threats concern the possibility of obtaining results by chance since the testing sessions were guided by stochastic sampling. We addressed them by sampling a large number of requests. Each test session lasted around 20 minutes, during which we sampled more than $15k$ requests.

7 CONCLUSIONS

Performance and reliability are two fundamental quality factors for microservices systems, typically analyzed separately. We believe that in such contexts where software releases for the service building blocks (microservices) are very frequent, and where service usage patterns may change often too, QA teams would benefit from the availability of techniques and tools to rapidly and jointly investigate performance and reliability issues which may arise due to evolutionary and operational changes. To this aim, we have proposed MIPaRT, a framework for microservices systems ex-vivo testing for joint analysis of performance and reliability aspects. MIPaRT builds on techniques for service usage, workload modeling, and for integrated performance-reliability testing. It provides QA engineers with a platform to automatically generate test cases, orchestrate testing sessions, computing relevant metrics and visualize results, for continuous assessment of microservices systems. This comes at the cost of the availability of microservices specifications and of service usage and system monitoring data, which are typically readily available in RESTful microservice systems development and operation contexts. We described controlled experiments in operating MIPaRT with an open microservices systems benchmark, showing its benefits in integrated performance and reliability testing.

REFERENCES

[1] J. Lewis and M. Fowler. Microservices - a definition of this new architectural term. Available at: http://martinfowler.com/articles/microservices.html, 2014. URL https://martinfowler.com/articles/microservices.html.
[2] M. Loukides. *What is DevOps?* O'Reilly Media, Inc., 2012.
[3] L. J. Bass, I. M. Weber, and L. Zhu. *DevOps - A Software Architect's Perspective.* SEI series in software engineering. Addison-Wesley, 2015.

[4] P. Abrahamsson, G. Botterweck, H. Ghanbari, M. G. Jaatun, P. Kettunen, T. J. Mikkonen, A. Mjeda, J. Münch, A. N. Duc, B. Russo, and X. Wang. Towards a secure DevOps approach for cyber-physical systems: An industrial perspective. *International Journal of Systems and Software Security and Protection*, 11(2):38–57, 2020.

[5] J. A. Morales, H. Yasar, and A. Volkman. Implementing DevOps practices in highly regulated environments. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, XP '18. ACM, 2018.

[6] K. C. Bourne. Chapter 7 - change control management. In K. C. Bourne, editor, *Application Administrators Handbook*, pages 96–111. Morgan Kaufmann, Boston, 2014.

[7] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[8] A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn. 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)*, volume 10469 of *Lecture Notes in Computer Science*, pages 159–174. Springer, 2018.

[9] A. Avritzer, V. Ferme, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Menasché, and V. Rufino. Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. *Journal of Systems and Software*, 165(110564):1–16, 2020.

[10] R. Pietrantuono, S. Russo, and A. Guerriero. Run-time reliability estimation of microservice architectures. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–35. IEEE, 2018.

[11] A. Bertolino, G. De Angelis, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo. DevOpRET: Continuous reliability testing in DevOps. *Journal of Software: Evolution and Process*, 2020;e2298:1–17, 2020.

[12] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2021.

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

[14] J. Soldani, D.A. Tamburri, and W.-J. Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[15] E. Casalicchio and V. Perciballi. Auto-scaling of containers: The impact of relative and absolute metrics. In *Proc. FAS*W@SASO/ICCAC*, pages 207–214. IEEE, 2017.

[16] T. Ahmad, A. Ashraf, D. Truscan, and I. Porres. Exploratory performance testing using reinforcement learning. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 156–163. IEEE, 2019.

[17] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 270–281. ACM, 2015.

[18] M. H. Moghadam. Machine learning-assisted performance testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 1187–1189. ACM, 2019.

[19] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, page 156–166. IEEE, 2012.

[20] Sunghun Kim, E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008. doi: 10.1109/TSE.2007.70773.

[21] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39, 2019.

[22] B. Chen, . Liu, and W. Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, page 49–60. ACM, 2016.

[23] A. Sedaghatbaf, M. H. Moghadam, and M. Saadatmand. Automated performance testing based on active deep learning. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 11–19. IEEE, 2021.

[24] J. Koo, C. Saumya, M. Kulkarni, and S. Bagchi. Pyse: Automatic worst-case test generation by reinforcement learning. In *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 136–147. IEEE, 2019.

[25] L. Gazzola, M. Goldstein, L. Mariani, I. Segall, and L. Ussi. Automatic ex-vivo regression testing of microservices. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, AST '20, page 11–20. ACM, 2020.

[26] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[27] M. Andreolini, M. Colajanni, and P. Valente. Design and testing of scalable web-based systems with performance constraints. In *2005 Workshop on Techniques, Methodologies and Tools for Performance Evaluation of Complex Systems (FIRB-PERF'05)*, pages 15–25, 2005.

[28] C.-P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker. How is performance addressed in DevOps? In V. Apte, A. Di Marco, M. Litoiu, and J. Merseguer, editors, *2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*, pages 45–50. ACM, 2019.

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

[30] M. Camilli and B. Russo. Modeling performance of microservices systems with growth theory. *Empirical Software Engineering*, 27(39):1–44, 2022.

[31] M. Camilli, C. Colarusso, B. Russo, and E. Zimeo. Domain metric driven decomposition of data-intensive applications. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*, pages 189–196. IEEE, 2020.

[32] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[33] D. Kesim, A. van Hoorn, S. Frank, and M. Häussler. Identifying and prioritizing chaos experiments by using established risk analysis techniques. In M. Vieira, H. Madeira, N. Antunes, and Z. Zheng, editors, *31st IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 229–240. IEEE, 2020.

[34] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66. IEEE, 2016.

[35] A. Jindal, V. Podolskiy, and M. Gerndt. Performance modeling for cloud microservice applications. In *2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 25–32. ACM, 2019.

[36] R. Pietrantuono, S. Russo, and A. Guerriero. Testing microservice architectures for operational reliability. *Software Testing, Verification and Reliability*, 30(2): e1725, 2020.

[37] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018*, pages 323–324. ACM, 2018.

[38] J. R. Norris. *Markov chains*. Number 2 in Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge university press, 1997.

[39] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar. WESSBAS: Extraction of probabilistic workload specifications for load testing and performance prediction–a model-driven approach for session-based application systems. *Software & Systems Modeling*, 17(2):443–477, 2018.

[40] A. Bertolino, P. Braione, G. De Angelis, L. Gazzola, F. Kifetew, L. Mariani, M. Orrù, M. Pezzè, R. Pietrantuono, S. Russo, and P. Tonella. A survey of field-based testing techniques. *ACM Computing Surveys*, 54(5):92:1–92:39, 2021.

[41] W. Nelson. Theory and applications of hazard plotting for censored failure data. *Technometrics*, 42(1):12–25, 2000.

[42] A. Avritzer, M. Camilli, A. Janes, B. Russo, J. Jahic, A. van Hoorn, R. Britto, and C. Trubiani. PPTAM$^{\lambda}$: What, Where, and How of Cross-domain Scalability Assessment. In *18th IEEE International Conference on Software Architecture Companion ICSA-C*, pages 62–69. IEEE, 2021.

[43] V. Ferme and C. Pautasso. 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 '18, page 261–272. ACM, 2018.