SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2019;e1725.
Published online in Wiley Online Library (wileyonlinelibrary.com). https://doi.org/10.1002/stvr.1725

SPECIAL ISSUE PAPER

Testing microservice architectures for operational reliability

Roberto Pietrantuono, Stefano Russo[*,†] and Antonio Guerriero

[1]*Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione, Università degli Studi di Napoli Federico II, Naples, Italy*

SUMMARY

Microservice architectures (MSA) is an emerging software architectural paradigm for service-oriented applications, well-suited for dynamic contexts requiring loosely coupled independent services, frequent software releases and decentralized governance. A key problem in the engineering of MSA applications is the estimate of their reliability, which is difficult to perform prior to release due frequent releases/service upgrades, dynamic service interactions, and changes in the way customers use the applications. This paper presents an *in vivo* testing method, named EMART, to faithfully assess the reliability of an MSA application in operation. EMART is based on an adaptive sampling strategy, leveraging monitoring data about microservices usage and failure/success of user demands. We present results of evaluation of estimation accuracy, confidence and efficiency, through a set of controlled experiments with publicly available subjects. © 2019 John Wiley & Sons, Ltd.

Received 28 February 2019; Revised 1 September 2019; Accepted 28 October 2019

KEY WORDS: *in vivo* testing; microservice architecture; software reliability

1. INTRODUCTION

Microservice architecture (MSA) is a software architectural style that is gaining popularity in many companies [1]. Netflix, eBay, Amazon, Twitter, PayPal and many other web-based services have evolved to this paradigm recently. MSA shifts traditional service-oriented architectures from a share-as-much-as-you-can philosophy, focused on reuse, to a share-nothing philosophy, emphasizing strong service decoupling. MSA applications are built by architecting a set of services, each providing a well-defined and self-contained business capability and high independence from others. Combined with technologies such as RESTful protocols and containers and agile development practices such as DevOps [2], MSA features lightweight communication and independent and rapid service deployment. These characteristics promote scalability, flexibility, maintainability, prompt reaction to changes and failures and frequent software releases.

We consider the problem of assessing quantitatively the reliability of an MSA application in use. This is a great concern for companies migrating towards MSA. While MSA is expected to favour seamless management of microservices' failures via fault tolerance means, what finally matters is the reliability of the overall MSA actually observed during operations (*operational reliability*). Operational reliability refers to the probability of a system to perform correctly on user demands; it is a user-perceived quality that depends not only on how much reliable a single microservice is, but also on how much it is used. A microservice with low reliability may have small impact on the user perception if it is rarely stimulated. Conversely, a highly reliable yet frequently invoked microservice

*Correspondence to: Stefano Russo, DIETI - Università degli Studi di Napoli Federico II, Via Claudio 21, 80125 Naples, Italy.
[†]E-mail: stefano.russo@unina.it

may determine perceivable MSA unreliability, as the likelihood to observe a failure increases with usage. Decision makers — MSA stakeholders, as well as managers of development, testing and operations — need to be aware of the operational reliability of the MSA. This would drive strategic decisions, for example, about effort allocation to maintenance or re-engineering activities.

So far, research has focused more on performance and maintainability issues than on reliability [1]. As for the latter, the general trend is to exploit microservices' features like loose coupling to increase failure detection, fault tolerance or availability in MSAs (e.g., [3–7]). To the best of our knowledge, the assessment of *operational reliability* of MSAs is not addressed yet. Traditional software reliability assessment techniques, such as operational testing and its derivations (e.g., [8–10]), have limited applicability to MSAs. Indeed, static attempts to gauge reliability are almost useless, as the application and the usage profile change over time because of frequent releases, services upgrades, dynamic service interactions and to how customers use the application.

This article presents Enhanced Microservice Adaptive Reliability Testing (EMART), a method for assessing the reliability of an MSA in operation. EMART extends the *MART* strategy introduced in previous own work, where the idea of using an adaptive testing algorithm to improve the reliability estimation of MSA applications was first presented [11]. EMART's ability to achieve better accuracy and efficiency than operational testing was shown through a case study based on the Netflix MSA stack. Being based on sampling *without replacement* (WOR) — that is, each partition of the input space[‡], from which test cases are drawn, can be selected at most once — *MART* assumes a testing budget smaller than the number of partitions. In past work, we also defined a family of testing algorithms that exploits sampling schemes more efficient than WOR sampling [12]. Building upon this concept, here we present EMART, which generalizes *MART* by removing the upper bound on the number of tests to run. This generalization enables the proposed method to be used also when the tester can afford more tests to the aim of achieving a high level of confidence in the estimate.

EMART implements a testing strategy acting during operations (namely an *in vivo* testing strategy), triggered upon request by a stakeholder who needs an estimate of the MSA operational reliability. It achieves unbiasedness, accuracy and efficiency by three key activities:

1. Monitoring: Field data are gathered about the microservices' usage profile and about failure/success of demands. This provides updated estimates representing the real reliability at the time when the assessment is requested.
2. Testing: Using only passive observations (monitoring) is inadequate for estimates with high accuracy and confidence. Indeed, the application could be not adequately stressed and failures would need much time to be exposed, leading to overestimation of reliability or, conversely, to an excessive number of observations for an acceptable confidence. EMART uses a new testing algorithm based on adaptive statistical sampling, which exploits data gathered during operations to drive the test generation and accelerate the exposure of failures.
3. Estimation: The EMART testing algorithm identifies the most relevant test cases in few steps, by forcing a disproportional selection of test cases with respect to the observed usage profile. In principle, such a type of sampling would yield biased estimates. Therefore, a proper weight-based estimator is adopted at the end of testing in order to counter-balance the selection strategy, ultimately providing an accurate and unbiased estimate with small variance.

EMART is designed starting form the main features and requirements of MSA, such as frequent code updates (hence, evolving failure probability), unstable/unknown usage patterns (evolving usage profile), tight testing budget constraints, loose coupling between services and continuous monitoring. The approach is conceived to cope with these features and to work reasonably well with limited testing budget, by exploiting field data to provide a quick and accurate assessment robust to such continuously changing conditions, by leveraging monitoring facilities usually available in an MSA and by the looser coupling (compared with other architectures) to keep the estimator simple.

[‡]A partition of the input space, also referred to as a *subdomain* in the following, is a subset of inputs believed to have the same chance of exposing a failure. Such a belief can be acquired from requirements specification (e.g., inputs inside/outside a range of admissible values) as well as from the code (e.g., inputs exercising the same code path).

DOI: 10.1002/stvr

The method is experimented on three open subjects available on Github, showing remarkable improvements in terms of effectiveness, efficiency and scalability with respect to operational testing, a technique used as baseline as it is a pillar of the software reliability assessment practice [13].

The rest of the paper is organized as follows. Section 2 surveys the related literature. Section 3 presents EMART. Sections 4 and 5 report the experimental design and results, respectively. Section 6 discusses threats to the validity of results. Section 7 concludes the paper.

2. RELATED WORK

2.1. MSA dependability

A recent study by Di Francesco *et al.* reports that a considerable number of papers are being published on MSA since a few years [1]. A relatively high number of them are on the application of MSA to several industrial domains, witnessing its strategic importance for companies. Besides architectural and design issues, research is targeting dependability concerns and how this new architectural style impacts them. Among the quality attributes of interest, *performance* and *maintainability* are the most investigated ones. *Reliability* is considered in few studies, often in a broad meaning encompassing aspects of resiliency, availability, fault tolerance, robustness and failure/anomaly detection.

Toffetti *et al.* propose an architecture for resilient self-management of microservices in the Cloud. It monitors application and infrastructural properties to provide timely reactions to failures and changing environmental conditions (auto-scaling), minimizing human intervention [14].

Kang *et al.* present the design, implementation and deployment of a microservice-based containerized OpenStack instance [2]. The authors describe a recovery mechanism for microservices to increase availability.

Investigating the adoption of MSA for Internet of Things applications, Butzin *et al.* propose a fault management mechanism based on the circuit-breaker pattern, which prevents a failed service from receiving further requests until its complete recovery, so as to avoid cascading failures [4]. Similarly, the service discovery mechanism proposed by Stubbs *et al.*, based on *Serf* [15], is equipped with monitoring and self-healing capabilities [16].

The use of the *Serf* failure detection ability is also proposed by Kookarinrat and Temtanapat in a decentralized message bus for communication among services [17]. An anomaly detection service is foreseen by Bak *et al.* too, in their MSA for context-based applications [3].

Cardozo *et al.* propose a framework for software service emergence in pervasive environments, where the problem of evaluating reliability under changing environment-dependent services is recognized but left to future work [18].

Testing is a less investigated area for MSAs, with work focusing mainly on resiliency assessment. Heorhiadi *et al.* propose the Gremlin framework for testing the failure-handling capabilities of MSAs by emulating common failures observable in network interactions between microservices [5]. Fault injection is used by Nagarajan *et al.* at Groupon in their automated *Screwdriver* tool to assess resiliency of MSAs [7], and by Meinke *et al.*, who adopt a learning-based testing strategy to evaluate the robustness of MSAs to injected faults [6]. Schermann proposes a formal model for multiphase live testing to test changes or new features in the production environment [19].

The general trend of the related work above is to exploit microservices' features like loose coupling to increase failure detection, fault tolerance or availability in MSAs. Although this impacts the user-perceived probability of failure on demand (namely the *operational reliability*), no study, to the best of our knowledge, dealt with its assessment.

2.2. Operational reliability testing

The problem of providing faithful operational reliability estimates before product release is known to be a big challenge. We are not aware of techniques for operational reliability estimate tailored for MSAs, as the one we propose in this work. Nevertheless, existing techniques could be borrowed.

Operational testing (OT) [10] derives tests for a software product from an estimate of its *operational profile*, a preliminary characterization of how the system is going to be used [20]. Having a

DOI: 10.1002/stvr

faithful profile has always been a big hurdle to OT applicability [21]. This issue is exacerbated in MSAs, where — for example, due to service upgrades — what is observed in real use is likely to be much different from what was expected before release, both for the operational profile and for the failure probability of microservices. Very likely, a pre-release assessment for MSA would be misleading. This suggests to opt for an *in vivo* technique, so as to assess the actual reliability during operations. A further problem of OT is that mimicking the expected usage is often insufficient to expose failures, ending up in inaccurate or highly uncertain estimates. For highly reliable software, an excessive testing effort would be required to reach an acceptable confidence level [22].

More recent operational testing techniques mitigate this problem trying to expose more failures by means of (i) the partitioning of the input space into equivalence classes, which enables to select fewer and more relevant test cases and/or (ii) adaptation, by which the selection of next test cases is driven by the results of previous tests. Chen *et al.* defined *adaptive random testing* [9], which uses adaptation to distribute the next tests across the input domain so as to reduce the expected number of test cases required to detect the first failure, for debugging purposes; as such, adaptive random testing is not meant to estimate reliability. Cai *et al.* proposed *adaptive testing*, where the assignment of the next test to a partition is based on the outcomes of previous tests to reduce the estimate's variance [8, 26]. The profile (assumed known) is defined on partitions, and selection within partitions is done by simple random sampling (SRS) with replacement. Adaptiveness is also exploited in own previous work on reliability testing (not focused on MSA), where *importance sampling* is used to allocate tests towards more failure-prone partitions [23–25].

3. MICROSERVICE ARCHITECTURES RELIABILITY ASSESSMENT METHOD

3.1. Terminology and assumptions

A user accesses an MSA application through a set of *edge* microservices. User *demands* are invocations of edge microservices. These have typically a minimal business logic, basically routing demands to lower layer microservices. Demands may fail, and as we are interested in operational reliability, we observe the outcome of demands at the edge layer, regardless of the failing entity within the MSA. The user-oriented metric of interest is the *probability of failure on a user demand*, which is a discrete measure of the usual reliability concept.[§]

The following assumptions are made, as typical in reliability testing studies [8, 25–27]:

1. The microservice architecture contains k edge microservices.
2. A user demand to (an edge microservice of) the MSA leads to either success or failure, and it is always possible to determine its success or not (*perfect oracle*). As oracle, we use the Hypertext Transfer Protocol status code of the response, distinguishing the following two cases:
    a) Successful demand: the status code is consistent with the input submitted, such as
        - a *2xx* status code (indicating *success*) for a *correct* request (*correct* according to the documentation), or
        - a *4xx* status code (indicating a *client error*) for an *incorrect* request (e.g., a numeric input containing alphabetical characters). These responses are correct replies to incorrect requests, which the API client is required to manage.
    b) Failing demand: (i) the application raises an unexpected, unmanaged, exception, sent to the client, which is reported as *5xx* status code (*server error*) or (ii) the returned status code and message are inconsistent with the input submitted.
3. The execution of a demand is not constrained by previous ones, and its success is independent of the history (a failing demand is always such, independently from past ones). Because of the loosely coupled nature of microservices, this assumption is likely to be easily met in MSAs.

[§]According to the IEEE Recommended Practice on Software Reliability, 'software reliability predictions are a measure of the probability that the software will perform without failure over a specific interval, under specified conditions' [28].

DOI: 10.1002/stvr

4. The input space DS_j of the jth edge microservice can be partitioned into a set of subdomains. The number of subdomains and the partitioning criterion are decided by the tester: example criteria are functional, structural or usage profile-based. The choice does not affect the applicability of the proposed strategy.
5. The overall MSA demand space is the union of subdomains of the input spaces of all edge microservices, $D = \cup_{j=1}^{k} DS_j$. D represents all demands a user can do. To simplify the notation, D is regarded merely as a set of partitions, D: $\{D_1, D_2, \ldots, D_m\}$, where m is the sum of the cardinalities of the partitions of all edge microservices.
6. The MSA operational profile P is described as a probability distribution over the demand space D. Differently from most literature on reliability testing, no assumption is made on the prior knowledge of P. However, we assume the ability to monitor the demands — a commonly available facility in MSA. EMART's adaptive nature makes use of the monitoring data to provide an estimate in line with the *observed* usage profile and failures of services.
7. While EMART is running to estimate the MSA reliability, microservices are not modified, and the normal workload needs not to be suspended.

We define the probability of selecting a failing demand from subdomain D_i as $x_i = f_i \cdot p_i$, where p_i is the probability of selecting a demand from D_i and f_i is the probability that a demand from D_i fails. Under the assumption of independent probabilities of failure, R is defined as [26, 27]

$$R = 1 - \sum_{i=1}^{m} x_i = 1 - \sum_{i=1}^{m} f_i \cdot p_i, \tag{1}$$

where the summation is the probability of failure on a user demand. EMART executes *in vivo* tests to estimate unbiasedly and efficiently the reliability R of an MSA in operation.

3.2. Usage scenarios

EMART is conceived to assess the reliability of an MSA application in two usage scenarios.

In use case UC1, the tester requires an estimate of the current MSA reliability using a constrained testing budget. Let us consider as upper bound on the number of tests that can be performed in operation a value as high as the number of subdomains. In this situation, EMART adopts a without-replacement sampling scheme.[¶]

Use case UC2 corresponds to the situation where higher accuracy and/or stronger confidence in the reliability estimate are required. The tester targets them at the cost of a possibly high number of test cases. In this scenario, without replacement sampling is not applicable, and EMART generalizes the previously proposed *MART* method [11], using a with replacement sampling scheme.

3.3. The EMART method

Figure 1 shows the steps of the EMART process. It includes pre-release activities, to be performed once before release, and *in vivo* activities, to perform the reliability assessment in operation.

3.3.1. Pre-release activities.

Demand space partitioning The demand space D is partitioned in a set of subdomains. To this aim, values of the arguments of methods of edge microservices are grouped in *equivalence classes*. Any partitioning criterion applies; we adopt specification-based partitioning, where equivalence classes are defined based on the input arguments in a method's signature. Consider, for instance, the method `Login(String username, String password)`: values of

[¶]Without replacement sampling schemes are generally expected to be more efficient than their with replacement counterpart, given the same sample size [29].

DOI: 10.1002/stvr

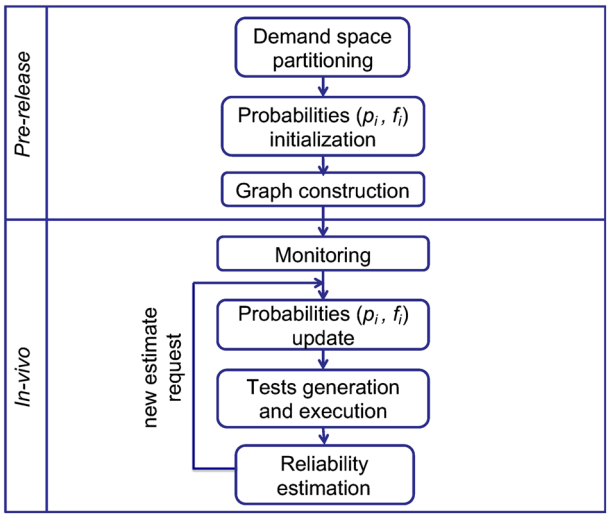

Figure 1. Enhanced Microservice Adaptive Reliability Testing operational reliability assessment steps.

the `username` input can be grouped into five classes according to the string length (in-range, out-of-range) and content (alphanumeric string, string including special characters, or the empty string); for `password`, seven classes are defined, according to the length and content (as for `username`), and to the satisfaction of two application-specific requirements (one upper case letter, one special character). The cartesian product yields 35 combinations. Each of them is referred to as a *test frame* (corresponding to a subdomain).[‖] More sophisticated criteria can exploit further information on the arguments — category-partition testing, with *single*, *error* and *property constraints*, is one such partitioning technique [30].

Initialization. Each test frame is associated with the probabilities p_i and f_i of selection and of failure of a demand from D_i, respectively. Their true value is of course unknown; EMART addresses the estimates $\hat{p}_i$ and $\hat{f}_i$ of the true values. In case the tester has no prior knowledge about expected usage and failure proneness of microservices in operation, all $\hat{p}_i$ and $\hat{f}_i$ are initialized by uniform distributions. EMART then refines the estimates dynamically as more information becomes available from monitoring, using the probabilities update formulas described later. In real cases, the tester may however have some prior knowledge, for example, because of past evidence or as his/her own belief. The partitioning criterion is itself an example of belief of the tester, who judges some classes as more prone to failure than others (e.g., in-boundary and out-of-boundary classes in equivalence classes partitioning). If available, such a belief can be used to assign initial values to p_i and f_i differently from a uniform distribution, to expedite the assessment.

Graph construction. An undirected graph model of the test cases space is constructed, whose nodes represent test frames, and an arc between two nodes represents a dependency between the failure probabilities of the corresponding test frames.

For every pair (i, j) of test frames of a method of an edge microservice, we define a distance d as the *number of differing input classes*. For instance, the distance between $Login(username_1, password_3)$ and $Login(username_2, password_3)$ is $d = 1$. The distance between two frames is assumed to be proportional to the potential differences observed in the control flow paths execution, since different input classes intend, by definition, to capture heterogeneous program's behaviour: the greater the distance (i.e., a bigger number of different input classes), the bigger the chance for two demands taken from the two corresponding frames to execute different control flow paths within the method's code. A *weight* $w_{i,j}$ is associated with the arc (i, j) to capture the belief about the joint failure probability of test frames i and j. Indeed, as demands

[‖]For each method with no input, we count one test frame, so as to include it in the assessment.

DOI: 10.1002/stvr

Table I. Equivalence classes for the example service.

Input	Input class	Description	Short name
`username`	u0	Alphanumerical string in bound (5 chars)	In bound
	u1	Alphanumerical string out of bound	Out of bound
	u2	String with special characters (SC)	SC
	u3	Empty input	Empty
`password`	p0	Alphanumerical string in bound (10 chars)	In bound
	p1	Alphanumerical string out of bound	Out of bound
	p2	String with special characters	SC
	p3	String with one upper case character	Upper Case

Table II. Test frames, example of test cases and usage and failure probability.

Test frames			Test cases		Probabilities	
ID	**Input Class 1**	**Input Class 2**	**Input 1**	**Input 2**	$\hat{p}_i$	$\hat{f}_i$
0	u0	p0	foo	password1	$0.0\overline{6}$	0
1	u0	p1	foo	password100	$0.0\overline{6}$	1
2	u0	p2	foo	passwor*1	$0.0\overline{6}$	0
3	u0	p3	foo	Password1	$0.0\overline{6}$	0
4	u1	p0	foobar	password1	$0.0\overline{6}$	0
5	u1	p1	foobar	password100	$0.0\overline{6}$	1
6	u1	p2	foobar	passwor*1	$0.0\overline{6}$	0
7	u1	p3	foobar	Password1	$0.0\overline{6}$	0
8	u2	p0	f?o*	password1	$0.0\overline{6}$	0
9	u2	p1	f?o*	password100	$0.0\overline{6}$	1
10	u2	p2	f?o*	passwor*1	$0.0\overline{6}$	0
11	u2	p3	f?o*	Password1	$0.0\overline{6}$	0
12	u3	p0	*null*	password1	$0.0\overline{6}$	0
13	u3	p1	*null*	password100	$0.0\overline{6}$	1
14	u3	p2	*null*	passwor*1	$0.0\overline{6}$	0
15	u3	p3	*null*	Password1	$0.0\overline{6}$	0

drawn from two test frames of a method are likely to execute some common code, the failure probability assigned to a test frame affects the belief about the failure probability of another frame, depending on their distance. The weight $w_{i,j}$ expresses the joint probability of failure: $P(i \cap j) = P(i|j) \cdot P(j)$. The conditional failure probability $P(i|j)$ is the probability for test frame i to fail, conditioned on the fact that a failure is observed for frame j. $P(i|j)$ is inversely proportional to the distance: the smaller the distance, the more similar the two frames, and the bigger the conditional probability of failure. We represent this relation by $P(i|j) = P(i) \cdot \frac{1}{d}$ ($d > 0$, as at least one input class differs between two test frames). Weights are computed as: $w_{i,j} = \hat{f}_i \cdot \frac{1}{d} \cdot \hat{f}_j$.

Example. An example of equivalence classes for the method *Login(String username, String password)* is reported in Table I. The 16 test frames — derived as the cartesian product of the four equivalence classes devised for each input — are listed in Table II, which shows also the initial usage and failure probabilities assigned to frames, and a sample test case drawn from a test frame. As for the usage probability, $\hat{p}_i$, we assume, in this example, ignorance of the profile, drawing probabilities from a uniform distribution in [0;1], normalized to sum up to 1. As for the failure probability, $\hat{f}_i$, we assume the tester has some belief about which test frame is expected to fail, and, specifically, (s)he assigns a failure probability 1 whenever the input class of the argument *password* is *out of bound* (namely, class p1). The resulting graph is shown in Figure 2. Because these failing test frames have distance $d = 1$ (i.e., they differ by only one input class), the weights are $w_{i,j} = 1$ if i and j are two failing test frames (linked by an arc) and 0 otherwise – so weights are omitted in the graph.

DOI: 10.1002/stvr

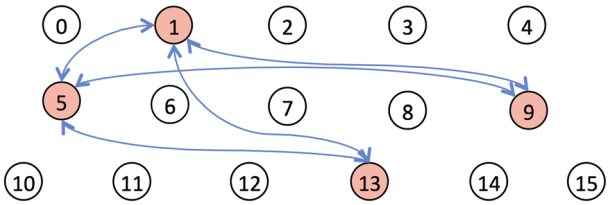

Figure 2. Example of graph. The numbers are the ID of the test frame listed in Table II.

3.3.2. Run-time monitoring and update.

Monitoring. The *in vivo* activities in Figure 1 require run-time data about the usage and failure probability of test frames, to compute an estimate aligned with the current reliability in operation. To this aim, a monitoring facility traces the requests to each microservice's method (name of the method and input values, so as to map the demand to a test frame), and their outcome (success/fail, so as to count the failed requests per test frame). Many monitoring tools are available to gather such data, for example, Amazon CloudWatch [31] and Nagios [32]. Note that a rough reliability estimate could be computed directly by the gathered data, but the demand space is not guaranteed to be explored adequately by normal workload. EMART's goal is to provide faithful estimates by actively spotting (through the generated tests) those demands more informative about the current reliability.

Probabilities update. The unknown usage and failure probabilities p_i and f_i are modelled as random variables, whose estimate is updated as more evidences (monitoring data) become available. The length of the history of observations to consider should to be defined so as to promptly react to changes of the usage profile and failure probabilities. EMART adopts a *sliding window* of length W on the history of the demands issued to edge microservices. The update rule for $\hat{p}_i$ and $\hat{f}_i$ are

$$\hat{p}_i^u = \hat{p}_i^{u-1} \cdot \left[H + (1-H) \cdot \left(1 - \frac{R}{W}\right)\right] + \hat{op}_i^u \cdot (1-H) \cdot \left(\frac{R}{W}\right), \tag{2}$$

$$\hat{f}_i^u = \hat{f}_i^{u-1} \cdot \left[H + (1-H) \cdot \left(1 - \frac{R}{W}\right)\right] + \hat{of}_i^u \cdot (1-H) \cdot \left(\frac{R}{W}\right), \tag{3}$$

where

- $\hat{p}_i^{u-1}$ is the previous occurrence probability of the ith test frame;
- $\hat{f}_i^{u-1}$ is the previous failure probability of the ith test frame;
- R is the number of executed demands ($R \leq W$);
- $\hat{op}_i^u$ is the occurrence probability for test frame i at the current step, estimated as the ratio between the number of failed demands to the ith test frame and R;
- $\hat{of}_i^u$ is the failure probability for test frame i at the current step, estimated as the ratio between the number of failed demands to the ith test frame and number of total demands to it;
- H is a value between 0 and 1, which weighs the history considered in the update (set to 0.5 in the experiments).

These rules allow changes of the operational profile and of the failure probability to be detected more promptly than it would be by considering the whole history.

3.3.3. Test generation algorithm. Figure 3 details the test cases generation and execution phase of EMART. It consists of an iterative algorithm navigating the graph built as in Section 3.3.1. Assuming n test cases to spend, the algorithm generates and executes one test case per step. The first test frame is selected by simple random sampling, namely all test frames have equal probability of being selected initially. In an iteration, a test case is generated and executed for the selected test frame by drawing a demand for it (i.e., taking values from the corresponding inputs classes), according

DOI: 10.1002/stvr

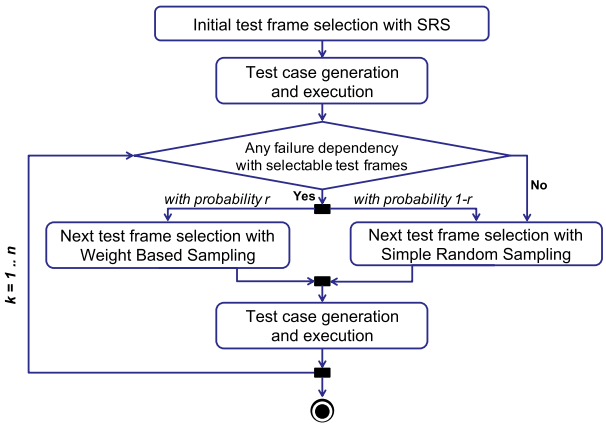

Figure 3. Enhanced Microservice Adaptive Reliability Testing tests generation and execution phase. SRS, simple random sampling.

to a uniform distribution. Then, one of two sampling schemes is used to select the next test frame: *weight-based sampling* (WBS) and SRS.[**] The former is chosen with probability r and follows the arcs between graph nodes (i.e., failure dependency between test frames), so as to explore possible clusters of failing demands; this feature is useful when failure points are clustered, as it often happens in software testing. This depth exploration is balanced by SRS, chosen with probability $1 - r$, for a breadth exploration of the test frame space, useful to escape from unproductive cluster searches. The steps are repeated until the testing budget n is over.

The test generation algorithm varies depending on the usage scenario (Section 3.2). In use case UC1, without replacement WBS and SRS schemes are used, in which a test frame can be selected only once. Clearly, this implies that the number of tests must not exceed the number of test frames. This EMART variant is useful when just 'few' tests can be executed in operation; it is a mere *best effort* approach within an upper bounded sample size (i.e., number of tests). In this scenario, a test frame is selected at step k by a distribution according to equation:

$$q_{k,i} = r \cdot \frac{\sum_{j \in s_k} w_{i,j}}{\sum_{h \notin s_k, j \in s_k} w_{h,j}} + (1 - r) \cdot \frac{1}{m - n_{s_k}}, \tag{4}$$

with:

- $q_{k,i}$ is the probability to select test frame i at step k;
- m is the total number of test frames;
- s_k is the current sample, namely the set of all test frames selected up to step k;
- n_{s_k} is the size of the current sample s_k;
- $w_{i,j}$ is the weight of arc from node (test frame) j in the current sample s_k to node (test frame) i;
- $w_{h,j}$ is the weight of arc from node j in the current sample s_k to node h not in s_k;
- r is the probability of using WBS (hence, probability of using SRS: $1 - r$).

In the Scenario UC2 (with an unconstrained number of tests), with replacement sampling is adopted, where a test frame can be selected more times. In this case, Equation (4) becomes

$$q_{k,i} = r \cdot \frac{\sum_{j \in s_k} w_{i,j}}{\sum_{h=1,\ldots m, j \in s_k} w_{h,j}} + (1 - r) \cdot \frac{1}{m}. \tag{5}$$

The first addendum in Equations (4) and (5) account for the contribution proportional to the weights of the graph (WBS in Figure 3), which capture the failure dependence between test frames.

[**]If there is no arc outgoing from the current set of selected test frames (thus, no failure dependency between the current sample and any other test frame), the SRS scheme is used.

DOI: 10.1002/stvr

The second addendum in Equations (4) and (5) account, respectively, for the selection probability of not-yet-selected test frames in SRS without replacement and the selection probability in SRS with replacement. The algorithm is adaptive as the q values change depending on which test frame is in the current sample.

3.3.4. Estimation. The testing algorithm is fed with information from monitoring, namely $\hat{p}_i$ and $\hat{f}_i$ of each test frame. Testing is expected to improve $\hat{f}_i$ by spotting more failing test frames, yet it cannot tell anything about the usage probability $\hat{p}_i$. Therefore, the $\hat{p}_i$ values remain unchanged during testing and are used only at the end to compute the estimate. The $\hat{f}_i$ values are updated at each step considering the 0/1 (success/failure) outcome of tests. We denote by $y_{i,t}$ the observed outcome of a test case t taken from test frame i, $y_{i,t} = 0/1$.

In Scenario UC2, the estimate of $\hat{f}_i$ is the updated ratio of the number of failing over executed demands with inputs taken from test frame i: $\hat{f}_i' = \frac{\hat{f}_i \cdot n_i + \sum_{t=0}^{m_i} y_{i,t}}{n_i + m_i}$, where n_i is the number of demands with an input from test frame i observed during operation and m_i is the number of demands taken from test frame i during testing (i.e., test cases).

In Scenario UC1, where $m_i \leq 1$, $\hat{f}_i$ is unchanged if $m_i = 0$; if $m_i = 1$, it is given by: $\hat{f}_i' = \frac{\hat{f}_i \cdot n_i + y_{i,t}}{n_i + 1}$.

The monitoring data and the results of testing are used to compute the estimate of the failure probability $\Phi = \sum_i p_i \cdot f_i$. The estimate is updated at step k accounting for the change of the selection probability for each test frame ($q_{k,i}$) and of the failure probability $\hat{f}_i'$. The estimator properly accounts for the disproportional selection (with respect to the operational profile) made through Equation (4) so as to preserve unbiasedness, by using weights equal to $1/q_{k,i}$ (values selected with high probability will contribute less to the estimation, and vice versa), as detailed hereafter.

In Scenario UC1, the estimator at step $k = 1$ (the first observation taken by the SRS) is $z_1 = N \cdot \hat{p}_i \cdot \hat{f}_{1,i}'$, where N is the total number of test frames, $\hat{p}_i$ is the probability of selecting the ith test frame (that does not depend on the step) and $\hat{f}_{1,i}'$ is the failure probability of the selected test frame i at Step 1. At step $k > 1$, the estimator is the one by Hansen–Hurwitz [33]:

$$z_k = \frac{1}{n} \sum_{k=1}^{n} \frac{\hat{p}_i \cdot \hat{f}_{k,i}'}{q_{k,i}}, \tag{6}$$

where n is the number of executed tests.

In Scenario UC2, the initial estimator z_1 is the same as before, while at step $k > 1$ it becomes

$$z_k = \sum_{h \in s_k} \hat{p}_i \cdot \hat{f}_{h,i}' + \frac{\hat{p}_i \cdot \hat{f}_{k,i}'}{q_{k,i}}. \tag{7}$$

In both use cases, the final estimator is the average of the values obtained at each step:

$$\hat{\Phi} = \frac{1}{n} \left(N \cdot \hat{p}_i \cdot \hat{f}_{1,i}' + \sum_{k=2}^{n} z_k \right) \tag{8}$$

representing the expected probability to experience a failure on a random demand to the MSA.

The overall MSA reliability is then computed as

$$R = 1 - \hat{\Phi}. \tag{9}$$

4. EXPERIMENTATION

4.1. Research questions

The performance of EMART in the envisaged Scenarios UC1 and UC2 are the target of research questions RQ1 and RQ2, respectively. An additional research question concerns the speed of convergence of the estimate to a stable value as the number of test cases increases, so as to figure out a trade-off between a desired quality of the estimate and the effort required (number of tests generated and run). The three research questions are stated as follows.

DOI: 10.1002/stvr

RQ1 - How does EMART perform with a constrained testing budget (UC1)?

- *RQ 1.1* - What is the *accuracy* of the reliability estimate?
- *RQ 1.2* - What is its *confidence*?

RQ2 - How does EMART perform with an unconstrained testing budget (UC2)?

- *RQ 2.1* - What is the *accuracy* of the reliability estimate?
- *RQ 2.2* - What is its *confidence*?

RQ3 - What is the *efficiency* of EMART?

- *RQ 3.1* - How much does *accuracy* improve as the number of test cases increases?
- *RQ 3.2* - How much does *confidence* improve as the number of test cases increases?

4.2. Subjects

An empirical study would demand for a large set of experimental subjects. However, as pointed out by Arcuri, 'finding the right MSA projects that do not require complex installations ... is not a trivial task' [34]. We chose to run controlled yet repeatable experiments with three subjects, publicly available from *Github*; they are listed in Table III. These, although simple, are good examples of MSA applications, as each implemented MS provides a fine-grain (high-cohesive) functionality, loosely coupled with the others, using the (lightweight) REST paradigm for communication, and stressing the independent and rapid deployment (using Docker for containerization in the first two cases (AWS and NLP) and the Spring Boot development framework in the third case (feature service) particularly suited for rapid deployment.

The first subject, here called AWS, is a demo program for deploying microservices on the Amazon Web Service Cloud with various configuration options. It manages information about users stored in an XML database, and it consists of three edge microservices, with 62,546 lines of code in several languages (mainly JavaScript, JSON, and YAML). Two microservices take one input, the *user id* (uid); the third one takes no input. Table IV lists the seven test frames for each of the former two microservices; because there is only one input, test frames correspond to the seven equivalence classes devised for *user id*. The total number of frames for AWS is (7 classes) × (2 methods) + 1 = 15, where the method taking no input is counted as one test frame, as noted in Section 3.3.1.

The second subject, called NLP building blocks (here simply: BB), is a natural language processing engine; it offers eight services, for 1,980 lines of code (mostly in Java) for language detection, sentence extraction, etc.; we defined for it 44 test frames.

The last subject, features service (here: FS), is a REST microservice for managing products feature models (compact representations of the features of products in a software product line); it has seven methods, for 1,712 lines of code mostly in the Java and SQL languages; we devised for it 163 test frames.

4.3. Test infrastructure

The test infrastructure for the experiments, shown in Figure 4, consists of the following components:

Table III. Experimental subjects (MSA applications available on Github).

Subject	$\mu services$	LOC	Frames
AWS demo URL: https://github.com/aws-samples/aws-microservices-deploy-options	3	62,546	15
NLP building blocks URL: https://github.com/mtnfog/nlp-building-blocks	8	1,980	44
SPL features service URL: https://github.com/JavierMF/features-service	7	1,712	163

Abbreviations: AWS, Amazon Web Service; LOC, lines of code; MSA, microservice architectures; NLP, natural language processing; SPL, software product line.

DOI: 10.1002/stvr

Table IV. Test frames for each of the two microservices of the Amazon Web Service subject with one input (*user id*).

Input	Test frame	Description
uid	GET http://127.0.0.1:8080/{uid0}	integer in range [0; 6]
uid	GET http://127.0.0.1:8080/{uid1}	integer in $]min(int32); 0[$
uid	GET http://127.0.0.1:8080/{uid2}	integer in $]6; max(int32)[$
uid	GET http://127.0.0.1:8080/{uid3}	$max(int32)$
uid	GET http://127.0.0.1:8080/{uid4}	$min(int32)$
uid	GET http://127.0.0.1:8080/{uid5}	special char
uid	GET http://127.0.0.1:8080/{uid6}	empty
-	GET http://127.0.0.1:8081/resources/greeting	service with no input
uid	GET http://127.0.0.1:8082/resources/names/{uid0}	integer in range [0; 6]
uid	GET http://127.0.0.1:8082/resources/names/{uid1}	integer in $]min(int32); 0[$
uid	GET http://127.0.0.1:8082/resources/names/{uid2}	integer in $]6; max(int32)[$
uid	GET http://127.0.0.1:8082/resources/names/{uid3}	$max(int32)$
uid	GET http://127.0.0.1:8082/resources/names/{uid4}	$min(int32)$
uid	GET http://127.0.0.1:8082/resources/names/{uid5}	special char
uid	GET http://127.0.0.1:8082/resources/names/{uid6}	empty

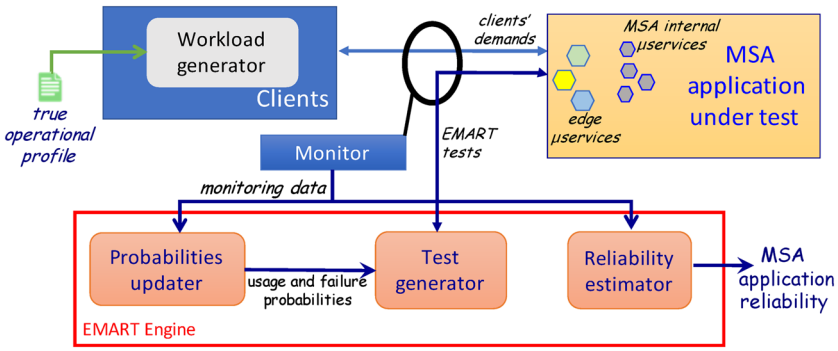

Figure 4. Experimental test infrastructure. EMART, Enhanced Microservice Adaptive Reliability Testing; MSA, microservice architectures.

- Workload generator: This component emulates MSA clients, issuing demands according to the true profile. Requests are generated according to the operational profile, namely selecting a given test frame with probability $\hat{p}_i$, and randomly choosing an input from the selected test frame. To emulate a variable profile, for example, due to an upgrade of a service, the profile and the failure probabilities are changed after a number of requests (set to 5,000 in the experiments);
- Monitor: It performs MSA monitoring, feeding the EMART engine. To this aim, we use the MetroFunnel monitoring tool tailored for microservices, developed in our research group.[††] The tool is in charge of monitoring the request/response pairs (possibly from multiple clients) so as to figure out what partition is being invoked and if it failed or not; such results are used to update the probabilities estimates $\hat{p}_i$ and $\hat{f}_i$ used for test generation.
- EMART engine: Its three subcomponents perform the EMART *in vivo* tasks of Figure 1:
  - Probabilities updater: It parses monitoring data to extract the number of correct/failing demands (coming from both the workload and the generated tests); then, usage and failure probabilities are updated, according to what has been described in Section 3.3.2;
  - Test generator: It implements the algorithm described in Section 3.3.3, generating and running tests based on the current usage profile and failure probabilities;
  - Reliability estimator: It computes the final reliability estimate in use cases UC1 and UC2, as described in Section 3.3.4.

[††]MetroFunnel is available at: https://github.com/dessertlab/MetroFunnel.

DOI: 10.1002/stvr

4.4. Parameters initialization

To simulate some knowledge of the tester about the failure proneness and occurrence probability of test frames (see Section 3.3.1), we performed an initial characterization of test frames, computing the initial probabilities $\hat{f}_i$ and $\hat{p}_i$ as follows. We ran 30 random test cases for each frame. The proportion of failures is used as an 'equivalent' prior knowledge about failure probability, as suggested by the seminal work by Miller *et al.* [35]. We recall that a failure in this context is either a server error (*5xx* HyperText Transfer Protocol code) or a reply inconsistent with the input request (cf. with Section 3.1). Faults causing these failures are real (not injected). We envisaged three categories of test frames based on failure proneness and assigned an initial failure probability to each. The categories are the following:

- No-failure frames: It encompasses frames for which all 30 tests exhibited no failure. The initial failure probability to these test frames is set to $\hat{f}_i = \epsilon = 0.01$. The small yet not null assignment of $\epsilon > 0$ represents the uncertainty due to the limited number of observations;
- All-failures frames: It includes test frames whose test cases failed at any of the 30 executions. The initial failure probability for these test frames is set to $\hat{f}_i = 1 - \epsilon = 0.99$;
- Sporadic-failures frames: It comprises frames that failed in some of the 30 executions. The initial failure probability for them is the ratio between failed requests and executed requests.

For all three subjects, we observed only *no-failure* and *all-failure* frames: the 30 test cases for every frame failed either never or always.[‡‡]

As for the $\hat{p}_i$ values (i.e., the *estimated profile)*, instead of taking a uniform profile (i.e., assuming ignorance) or any arbitrary distribution, we opted for deriving them by deviating a *true profile* by a given percentage, so as to simulate the incorrect knowledge of the tester about the true profile and assess how much it impacts on results. To this aim, we first derived the *true profile* as follows. We fix a *target true reliability* R_E (0.90, 0.95, 0.99) and then assign *occurrence probabilities* p_i to failing or nonfailing partitions so as to attain the desired value. Specifically, considering the abovementioned categories of frames, we assign the same occurrence probability to all frames in the first category, computing it as R_E divided by the number of frames in that category. Each frame in the second category is assigned a probability computed as $(1 - R_E)$ divided by the number of frames in that category. For instance, for the AWS subject, 11 out of the 15 test frames belong to the first category (no-failure) and four to the second category (all-failure): for a desired reliability R_E=0.9, the no-failure test frames are assigned an occurrence probability equal to $\frac{0.9}{11} = 0.0\overline{81}$; the all-failure test frames are assigned a probability equal to $\frac{(1-0.9)}{4} = 0.025$. For instance, assuming that the first 11 rows of Table IV are the *no-failure* test frames, each of them will be selected with probability $0.0\overline{81}$, and a test case (namely, a GET request as those listed in the first 11 rows of the Table) will be issued to the service. In the same way, assuming that the last four rows of the Table correspond to the *all-failure* partition, each of them will be selected with probability 0.025 for generating the test case. This process gives a true reliability approximately equal to 0.9 — the approximation is because of the usage of $\hat{f}_i$ values in lieu of the unknown f_i ones, obtained on a limited number of 30 runs.

We finally deviated the true profile by a desired percentage, measured as the sum of absolute differences of the estimated versus the true occurrence probability of each test frame:

$$e = \sum_{i=0}^{|TF|} |\hat{p}_i - p_i| \,, \tag{10}$$

where $|TF|$ is the number of test frames; $\hat{p}_i$ and p_i are the estimated and the true occurrence probability of test frame i, respectively.

[‡‡]This happened to be different, for instance, from what experimented in the case study of previous *MART* method [11].

DOI: 10.1002/stvr

4.5. Experimental factors

For every subject, three experiments are set up, one per RQ. Each experiment foresees a number of scenarios, depending on a number of independent variables potentially affecting the reliability estimate, called *experimental factors* [36].

The first factor is the true operational profile. For construction validity, all the experiments are repeated for three true profiles, emulating workloads of clients' requests yielding different expected reliability R_E values, namely 0.90 (True Profile 1), 0.95 (True Profile 2) and 0.99 (True Profile 3).

The second experiment design factor is the error on the initial estimate of the profile. Two *estimated profiles* are considered, deviating by 10% and 90% from the true one, respectively.

The third design factor is the testing budget, that is, the number of test cases generated and run; this serves to study the improvement in *accuracy* and *confidence* of the reliability estimate as the number of tests increases (RQ3). To this aim, we run EMART in five configurations, namely generating a number of tests corresponding to 0.25×, 0.50×, 0.75×, 5× and 10× times the number of test frames. Rather than performing a sensitivity analysis to find the best value of r for every specific application under test, we prefer to set the value of r to 0.5 for all testing sessions, so as to have a balanced sampling between WBS and SRS.

The fourth design factor is the number of times that EMART is run to provide a reliability estimate in a single scenario execution. This allows assessing to what extent its performance improves thanks to learning from monitoring data. First, we run EMART without the emulated clients' workload, yielding a first reliability estimate when no field data is observed yet; we call this Step 1. Then, we run EMART two more times, as if it were in the MSA operational phase — after 5,000 demands (Step 2) and after 10,000 demands (Step 3) — in order to evaluate the improvements brought by exploiting the knowledge gained from monitoring the clients' demands and their success/failure.

The stability of the profile is the fifth factor. In real settings, it is unlikely that the usage profile of an MSA remains unchanged over time. To study how EMART reacts to a *variable* operational profile, we consider an additional experimental scenario with six assessment steps, where the true profile changes after three steps.

4.6. Evaluation metrics

As proxies for EMART *accuracy* and *confidence*, we compute, respectively, the mean squared error MSE and sample variance S of multiple EMART runs with respect to the *true reliability*. The smaller the MSE, the more accurate the estimate; the smaller the variance S, the more efficient the estimator, hence the stronger the confidence. To have a true reliability, we fix a profile (whatever it is), deemed as the 'true' profile in operation — in our experimental setting, it is derived as described in Section 4.4, but any other profile is fine for comparison purpose — and then we issue requests according to that profile. The true reliability according to the conventional Nelson model is

$$R_T = \lim_{T \to \infty} (1 - F/T), \tag{11}$$

where F is the number of observed failures over the T executions. The Nelson estimator is simply

$$R = 1 - F/T \tag{12}$$

provided that the T requests are issued according to the true profile. This, of course, is closer to the true value as T tends to infinite. In our experiment, we set $T = 10,000$ to have a good approximation of the true reliability, being 10,000 two orders of magnitude bigger than the test budget used by EMART. Thus, R is our best-effort approximation of the true reliability R_T.

Since the true profile is, in practice, unknown, EMART (as any operational testing technique) needs to use an *estimate* of the operational profile, refined over time, to provide its estimate of reliability. The accuracy and confidence metrics are computed with reference to the estimate of reliability obtained by EMART under the estimated profile, denoted as $\hat{R}$, and the 'true' reliability estimate obtained under the true profile, denoted as R in Equation (12). In particular, the mean

DOI: 10.1002/stvr

value M, the variance S and the MSE of the reliability estimate $\hat{R}$ over $N_R = 30$ repetitions are computed as

$$\begin{aligned} lM(\hat{R}) &= \frac{1}{N_R}\sum_{i=1}^{N_R}\hat{R}_i, \\ MSE(\hat{R}) &= \frac{1}{N_R}\sum_{i=1}^{N_R}(\hat{R}_i - R))^2, \\ S(\hat{R}) &= \frac{1}{N_R - 1}\sum_{i=1}^{N_R}(\hat{R}_i - M(\hat{R}))^2. \end{aligned} \tag{13}$$

As baseline for comparison, we consider OT; clearly, OT is used just to compare the performance of the EMART test generation algorithm with a competing one without the clients' workload (before it starts to learn from field data).

4.7. Experiments

RQ1 (Scenario UC1). In the experiment to answer RQ1 the number of tests generated is set to 25% of the number of test frames (i.e., 0.25×), so as to simulate a scarce testing budget. We consider the two extreme cases for the estimate error, that is, an error of 10% and 90%, for each of the three profiles. This leads for every subject to six *scenarios* (3 true profiles × 2 error values) in which the true operational profile is *stable*, that is, it is not subject to significant variations. In a seventh scenario accounting for the variable profile, the initial deviation is set to $e = 0.5$. As anticipated in Section 4.6, the performance figures of EMART are computed after $N_r = 30$ repetitions. For RQ1 we performed 3 subjects × (3 stable profiles × 2 errors × 3 steps + 1 variable profile × 1 error × 6 steps) × 1 testing budget = 72 testing sessions, each with 30 repetitions.

RQ2 (Scenario UC2). The experiment targeting RQ2 foresees seven scenarios per subject, similarly as for UC1; the number of tests is set to ten times the number of test frames (i.e., 10x), so as to simulate a large testing budget and evaluate the gain in accuracy and confidence. For RQ2 we performed 72 testing sessions, as for RQ1.

RQ3 (performance vs test budget). In the experiment to target RQ3, we vary the factor testing budget. For each session, we generate a number of tests corresponding to 0.25×, 0.50×, 0.75×, 5× and 10× times the number of test frames. For what stated in Section 3.2, WOR sampling is used in the former three cases, and with replacement sampling in the 5× and 10× cases. For answering RQ3, we performed further 72 testing sessions for each of the test budgets 0.50×, 0.75× and 5×.

Overall, we performed 360 testing sessions — each with 30 repetitions — of which 270 sessions for the stable profiles and 90 for the variable profile. Supplemental material, including the EMART engine source code and all the obtained results, is available at: https://zenodo.org/badge/latestdoi/205180385.

5. RESULTS

5.1. RQ1 (constrained testing budget)

Figures 5 and 6 show, respectively, the MSE (for RQ1.1) and the variance (for RQ1.2) for the three experimental subjects with a *stable profile*. They show results for the two values of the estimated profile error, namely 10% and 90%, under the *True Profile* 3, for which the true reliability is 0.99. Figure 7 shows MSE and variance with a *variable profile* over six assessment steps, for the AWS subject.[§§] EMART is compared with OT as baseline technique under the same conditions (Step 1 in the Figures, when EMART does not exploit field data yet). We recall that these results are obtained running a number of tests as low as 25% of test frames. They show that

[§§]Results for the other profiles and subjects are essentially the same; they are not reported for brevity and are available at https://github.com/dessertlab/EMART.

DOI: 10.1002/stvr

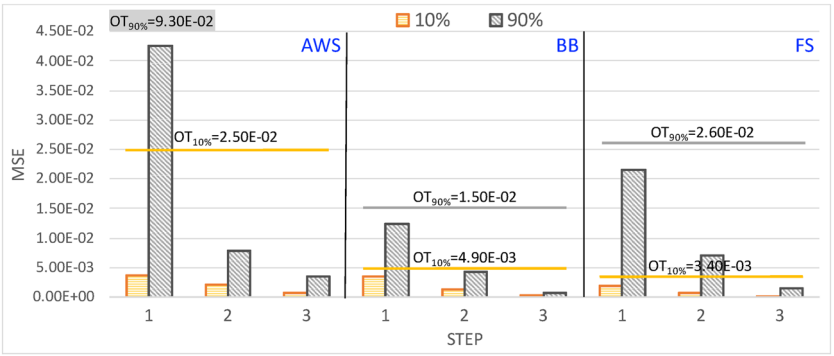

Figure 5. RQ1, stable profile: accuracy (MSE) of the estimate for subjects AWS, BB and FS. Testing budget equal to 25% of test frames number. Starting profile with 10% error (yellow), and with 90% error (grey). AWS, Amazon Web Service; BB, building blocks; FS, features service, MSE, mean squared error; OT, operational testing.

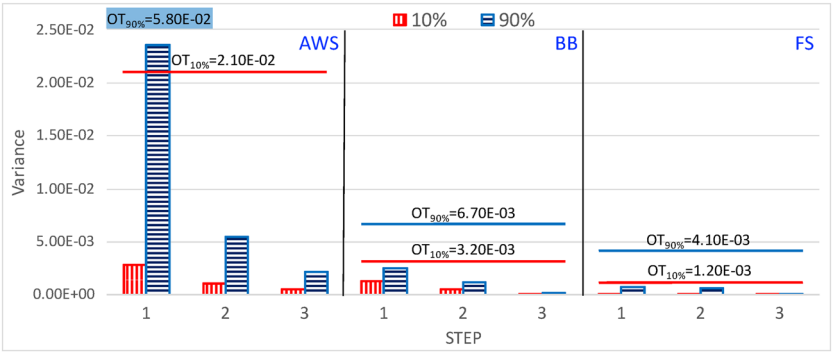

Figure 6. RQ1, stable profile: confidence (variance) of the estimate for subjects AWS, BB and FS. Testing budget equal to 25% of test frames number. Starting profile with 10% error (red), with 90% error (blue). AWS, Amazon Web Service; BB, building blocks; FS, features service, OT, operational testing.

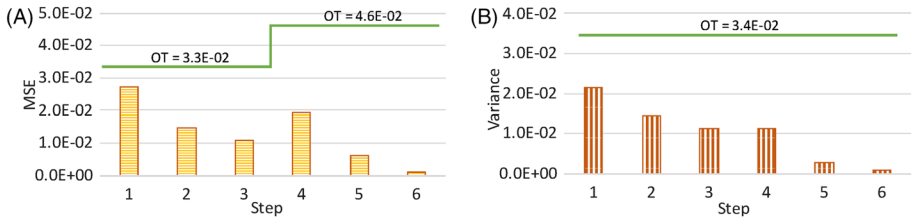

Figure 7. RQ1, variable profile: (a) MSE and (b) variance for subject AWS. Testing budget equal to 25% of test frames number. AWS, Amazon Web Service; MSE, mean squared error; OT, operational testing.

- For all subjects, the order of magnitude of both MSE and variance is as low as 1.0E-3 with an initial profile error of 10%, and as low as 1.0E-2 with an initial profile error of 90%;
- At Step 1, EMART exhibits lower MSE and variance than OT. This means that the sampling algorithm improves the estimate accuracy and efficiency with respect to OT even when field data are not exploited yet. Considering the 18 test sessions with stable profiles (three subjects, three true profiles, two estimated profiles) and the three experiments with variable profile, this happens in 18 out of 21 cases for MSE and in 20 out of 21 cases for variance;
- Both MSE and variance decrease, expectedly, as field data about the (*stable*) usage profile and failure probability of test frames become available;
- In the case of *variable* profile (Figure 7), it can be noted how, upon the change of the true profile, the MSE suddenly increases (while variance is not greatly affected), and then at Steps

DOI: 10.1002/stvr

5 and 6 it decreases as the new profile is 'learnt' from the field. This highlights the ability of EMART to detect the profile change and to correct the estimate.

5.2. *RQ2 (unconstrained testing budget)*

Figures 8 and 9 show, respectively, the MSE (RQ2.1) and the variance (RQ2.2) for the three subjects, with a *stable profile*. Figure 10 is for the *variable* profile. We observe that

- As for accuracy, the comparison with OT (Figure 8, Step 1) highlights a lower MSE difference with respect to Scenario UC1; with an unconstrained testing budget, EMART performs better in half of the cases (11 out of 21 test sessions), and absolute differences are very small. This happens because of the high number of test cases, which compensate the inaccuracy of OT due to low-occurrence faults detection ability;
- As for confidence (Figure 9, Step 1), EMART outperforms OT in 17 out of 21 test sessions, and further improves at subsequent steps; the estimate tends to become stable. As expected, when EMART exploits monitoring data (Steps 2 and 3), it outperforms OT in both the stable and variable profile experiments, in accuracy (Figure 8) as well as in confidence (Figure 9);
- Like for RQ1, both MSE and variance expectedly decrease as field data about the (*stable*) usage profile and failure probability of test frames become available.
- In the case of *variable* profile (Figure 10), EMART promptly reacts to the profile change, as in Scenario UC1.

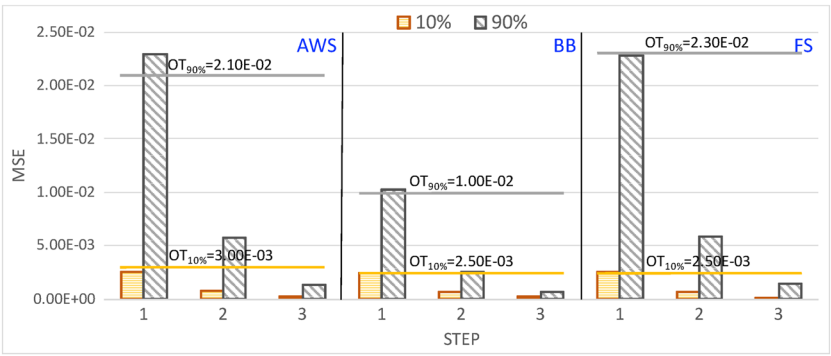

Figure 8. RQ2, stable profile: accuracy (MSE) of the estimate for subjects AWS, BB and FS. Testing budget equal to 10x the test frames number. Starting profile with 10% error (yellow), with 90% error (grey). AWS, Amazon Web Service; BB, building blocks; FS, features service, MSE, mean squared error; OT, operational testing.

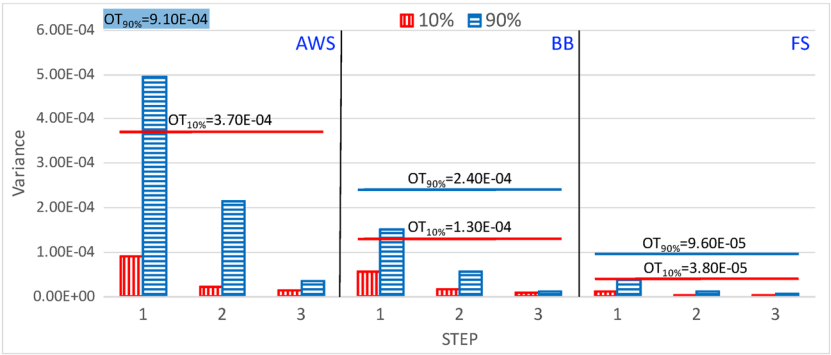

Figure 9. RQ2, stable profile: confidence (variance) of the estimate for subjects AWS, BB and FS. Testing budget equal to 10× the test frames number. Starting profile with 10% error (red), with 90% error (blue). AWS, Amazon Web Service; BB, building blocks; FS, features service, OT, operational testing.

DOI: 10.1002/stvr

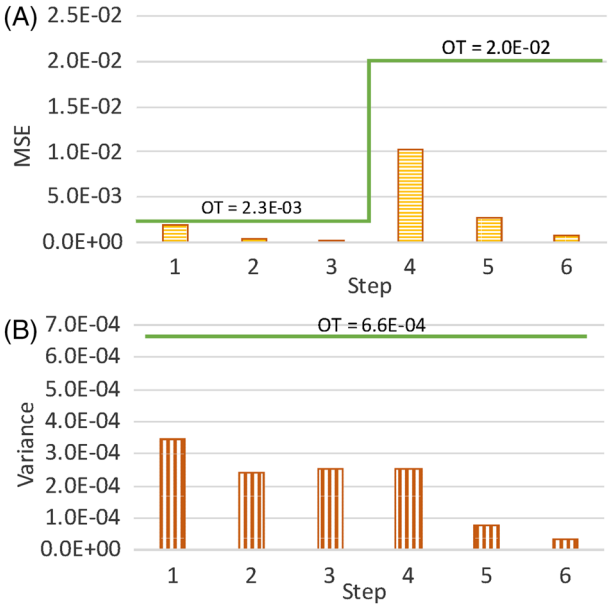

Figure 10. RQ2, variable profile: (a) MSE and (b) variance for subject AWS. Testing budget equal to 10x the test frames number. Variable profile. AWS, Amazon Web Service; MSE, mean squared error; OT, operational testing.

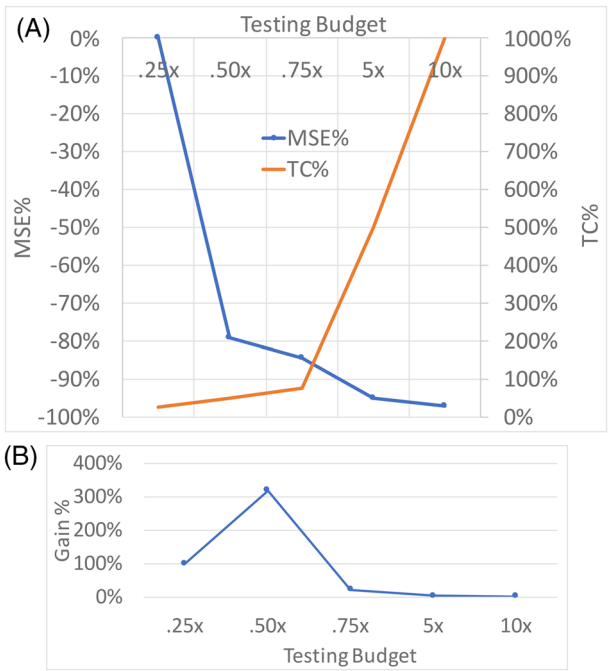

Figure 11. RQ3: (a) Estimate's accuracy (MSE) and (b) gain versus number of tests for subject AWS. AWS, Amazon Web Service; MSE, mean squared error.

5.3. RQ3 (performance vs testing budget)

MSE and variance of EMART are assessed with respect to a varying number of test cases enabling a cost-benefit analysis. The five configurations defined in Section 4.7 (test budget equal to 0.25×, 0.50×, 0.75×, 5× and 10× times the number of test frames) are labelled from 1 to 5, respectively.

DOI: 10.1002/stvr

We evaluate cost in terms of percent increase of required test cases (TC), and the benefit in terms of percentage decrease of MSE and variance.

In Figure 11(a) the blue line (left y-axis) plots the percentage MSE with respect to the first configuration (assumed as baseline), namely $\Delta_{MSE} = \frac{MSE_i - MSE_1}{MSE_1}\%$, where MSE_i and MSE_1 are the MSE values obtained under the ith configuration and under the first configuration. Figure 12(a) plots similar percentages in blue for the variance. In both figures, the red line (right y-axis) plots the percent increase in the number of executed test cases: it just transposes the x-axis with the five configurations (from 0.25× to 10×) in order to have a clearer view of the cost-benefit trade-off. We denote the percentage increase as Δ_{TC}.

Figures 11(a) and 12(a) refer to one out of the 72 testing sessions, for the *FS* subject under Profile 1 and error value 10%, at assessment Step 3. We see how MSE decreases with the increase of test cases. To figure out the best trade-off, we consider the *incremental ratio* computed as $\frac{|\Delta_{MSE}|}{|\Delta_{TC}|}\%$ (similarly for variance). This represents the actual *percentage gain*: a ratio higher than 100% means that the benefit in terms of percentage reduction of MSE (or variance) overcomes the cost in terms of percent increase of number of test cases. For instance, Figures 11(b) and 12(b) plot the incremental ratios for the mentioned *FS* case: the without replacement variant working with a number of tests half of the test frames (0.50×) is the best choice, as it yields a high gain with respect to the 0.25× configuration, with relatively few more test cases.

To compare configurations, we counted how many times each configuration outperforms the others in terms of incremental ratio over all three subjects. The results are shown in Figure 13(a). The best configuration turned out to be the without replacement variant working with a number of test cases 0.5 times the number of frames. This is the best configuration 36 out of 72 times, followed by the 0.25× configuration (27/72), then by the 0.75× (6/72). The results for the variance are shown in Figure 13(b) (70/72 wins for the 0.50× configuration, and 2/72 for the 0.75× configuration).

As for the percentage gain, we conclude that EMART is particularly suited when a scarce testing budget can be spent (Scenario UC1), as its performance allows obtaining an estimate of the MSA reliability with considerable accuracy (low MSE) and confidence (low variance) with a number of test cases just as high as 25% of the number of test frames; the best results, in terms of percentage gain, are achieved with a number of tests as high as 50% of test frames. On the other hand, if a tester

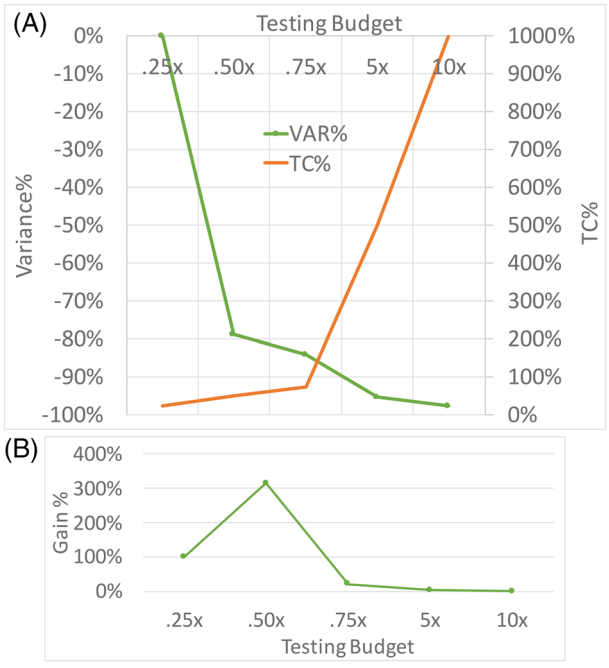

Figure 12. RQ3: (a) Estimate's confidence (variance) and (b) gain versus number of tests for subject AWS. AWS, Amazon Web Service.

DOI: 10.1002/stvr

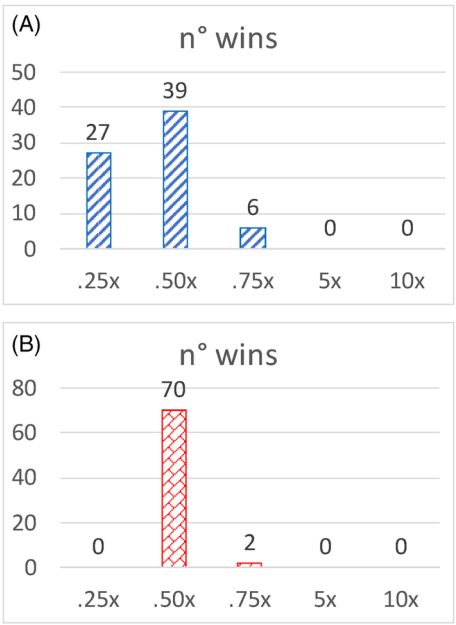

Figure 13. Number of wins per test budget over all subjects: (a) accuracy (mean squared error); (b) confidence (variance).

needs accuracy or confidence higher than those obtained with 50% or 25% of the tests, regardless of the number of tests needed (namely, the accuracy and/or confidence of the assessment is deemed relatively more important than the cost of the assessment), then the with replacement configuration (Scenario UC2) is needed: the gain will be lower, but the configuration achieves the goal of a better assessment, which the without replacement configuration cannot accommodate.

6. THREATS TO VALIDITY

We discuss threats that may affect the validity of the results beyond our best efforts in the design and execution of experiments.

6.1. Internal validity

In the experiments, we derived test frames by a functional partitioning criterion, in which equivalence classes are defined based on the input arguments in a method's signature. While any partitioning criterion can be applied, different criteria may lead to different results, depending on the overlap of equivalence classes. Exploring the sensitivity of results to alternative partitionings (e.g., structural) was out of the scope of this paper and is left to future research.

The *true* profile of an application under test is generally unknown before release; the procedure we used to derive a true profile for experiments foresees to fix a target reliability value, and to assign occurrence probabilities to failing/nonfailing partitions so as to attain the desired value. This could not be representative of a true profile; however, any profile generation procedure would be subject to the same threat. It is important to notice that the experiments were mainly focused on the *difference* between the estimated profile (representing the belief of a tester about the true profile) and the true one, and on whether EMART is able to converge both in case of a small and a large difference.

Other experimental settings may affect the results; the update cycle between two reliability assessment requests is set to W = 5,000 demands, and the history parameter of the monitoring framework is set to H = 0.5. Different values are likely to change the speed of convergence, hence the reader should be aware that these parameters need to be tuned before applying the method.

Additional threats to internal validity include the correctness of scripts for data collection process and of implementation of the experimental test bed performed by the authors.

DOI: 10.1002/stvr

6.2. Construct validity

The estimate of reliability is based on the model $R = 1 - \sum_i p_i \cdot f_i$. This model is widely used in the literature of software testing, but it assumes that partitions (in our case, test frames) are independent. Violation of this assumption can affect the results. However, compared with other studies based on this model (e.g., [8, 25–27, 37]), the assumption is more likely satisfied in MSA applications, because of the loose coupling that this kind of architecture enforces.

6.3. External validity

The experiment is performed on three microservice applications selected from *Github*. Thus, care must be taken in extending conclusions to other programs. A high number of testing sessions considering several factors has been performed in order to mitigate this threat (360 testing session, each repeated 30 times, under various true and estimated profiles, including stable and variable profile, under several testing budgets and number of estimate requests). Beside MSA, we left to future research extending the experiment to different types of applications for which the *in vivo* assessment is deemed important (e.g., self-adaptive systems).

7. CONCLUSIONS

The microservice architecture style has currently great momentum in the engineering of modern scalable web-based or cloud-based service-oriented applications. Traditional techniques for software reliability assessment, based on testing with respect to an expected operational profile, can hardly be applied to MSAs, because of their highly dynamic nature — with frequent updates — and to the uncertainty about their actual usage profile with respect to what can be statically foreseen at design time.

This paper has presented the EMART *in vivo* testing method for the estimate of the reliability of microservice architectures. EMART allows to estimate the reliability of a MSA application during the operational phase, featuring accuracy and confidence. These are obtained through (i) the adaptivity to the real observed usage profile and failing behaviour (thanks to monitoring data) and to (ii) the adaptivity of the statistical sampling-based test cases generation algorithm, which allows to spot failures with relatively few tests while preserving the estimate unbiasedness.

EMART has been evaluated experimentally with various controlled experiments with three publicly available subjects. The results show how EMART adapts to the real usage profile of the microservice application, as well as to its changes over time, yielding estimates with good accuracy (mean squared error), high confidence (small variance) and good efficiency (low number of tests). These advantages come to the cost of gathering run-time data about number of requests and failures for each microservice, which dynamically feed the EMART statistical algorithm. Such data are typically captured by monitoring tools usually in place in service-oriented execution environments.

EMART deals with providing a high-confidence estimate of operational reliability. However, beside the estimate, the results of testing are also a very good starting point to improve reliability in the next releases, because failed tests reveal the presence of defects. To this aim, the information provided by EMART can greatly help (i) pinpoint the partitions of the input of a service that leads to a failure and (ii) figure out if the partition exhibiting the failure is a much used one or not. This information can be useful for the prioritization of fixing actions, by preferring to fix the more used functionalities, which contribute more to improve reliability and/or for balancing the load in a different way to prevent some services to be stressed much more than others.

ACKNOWLEDGEMENTS

This work has been supported by the PRIN 2015 project ‘GAUSS’ funded by MIUR.

DOI: 10.1002/stvr

REFERENCES

1. Di Francesco P, Malavolta I, Lago P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. *Int. Conf. on Software Architecture (ICSA)*: IEEE: Gothenburg, Sweden, 2017; 21–30.
2. Kang H, Le M, Tao S. Container and microservice driven design for cloud infrastructure DevOps. *Int. Conf. on Cloud Engineering (IC2E)*: IEEE: Berlin, Germany, 2016; 202–211.
3. Bak P, Melamed R, Moshkovich D, Nardi Y, Ship H, Yaeli A. Location and context-based microservices for mobile and internet of things workloads. *Int. Conf. on Mobile Services (MS)*: IEEE: New York, NY, USA, 2015.
4. Butzin B, Golatowski F, Timmermann D. Microservices approach for the internet of things. *21st Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*: IEEE: Berlin, Germany, 2016.
5. Heorhiadi V, Rajagopalan S, Jamjoom H, Reiter MK, Sekar V. Gremlin: Systematic resilience testing of microservices. *36th Int. Conf. on Distributed Computing Systems (ICDCS)*: IEEE: Nara, Japan, 2016; 57–66.
6. Meinke K, Nycander P. Learning-based testing of distributed microservice architectures: correctness and fault injection. In *Software Engineering and Formal Methods*, vol. 9509, LNCS. Springer: Heidelberg, 2015; 3–10.
7. Nagarajan A, Vaddadi A. Automated fault-tolerance testing. *9th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*: IEEE: Chicago, IL, USA, 2016; 275–276.
8. Cai K-Y, Li Y-C, Liu K. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology* 2004; **46**(15):989–1000.
9. Chen TY, Leung H, Mak IK. Adaptive random testing. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, vol. 3321, Maher MJ (ed.), LNCS. Springer: USA, 2005; 320–329.
10. Musa JD. Software reliability-engineered testing. *Computer* 1996; **29**(11):61–68.
11. Pietrantuono R, Russo S, Guerriero A. Run-time reliability estimation of microservice architectures. *29th Int. Symp. on Software Reliability Engineering (ISSRE)*, IEEE: Memphis, TN, USA, 2018; 25–35.
12. Pietrantuono R, Russo S. On adaptive sampling-based testing for software reliability assessment. *27th Int. Symp. on Software Reliability Engineering (ISSRE)*: IEEE: Ottawa, ON, Canada, 2016.
13. Lyu MR (ed..) *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc.: Hightstown, NJ, USA, 1996.
14. Toffetti G, Brunner S, Blöchlinger M, Dudouet F, Edmonds A. An architecture for self-managing microservices. *1st Int. Workshop on Automated Incident Management in Cloud*: ACM: Bordeaux, France, 2015; 19–24.
15. HashiCorp. The Serf tool - decentralized cluster membership, failure detection, and orchestration. [Online]. Available: http://www.serfdom.io (Last checked: 2019-7-12).
16. Stubbs J, Moreira W, Dooley R. Distributed systems of microservices using docker and serfnode. *7th Int. Workshop on Science Gateways (IWSG)*: IEEE: Budapest, Hungary, 2015; 34–39.
17. Kookarinrat P, Temtanapat Y. Design and implementation of a decentralized message bus for microservices. *Int. Joint Conf. on Computer Science and Software Engineering (JCSSE)*: IEEE: Khon Kaen, Thailand, 2016.
18. Cardozo N. Emergent software services. *Onward! 2016 - ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software*: ACM, 2016; 15–28.
19. Schermann G, Schöni D, Leitner P, Gall HC. Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies. *17th Int. Middleware Conf*: ACM: Trento, Italy, 2016; 12:1–12:14.
20. Musa JD. Operational profiles in software-reliability engineering. *IEEE Software* 1993; **10**(2):14–32.
21. Chen M-H, Mathur AP, Rego V. A case study to investigate sensitivity of reliability estimates to errors in operational profile. *5th IEEE Int. Symp. on Software Reliability Engineering (ISSRE)*: Monterey, CA, USA, 1994; 276–281.
22. Littlewood B, Strigini L. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM* 1993; **36**(11):69–80.
23. Cotroneo D, Pietrantuono R, Russo S. A learning-based method for combining testing techniques. *35th Int. Conf. on Software Engineering (ICSE)*: IEEE: San Francisco, CA, USA, 2013; 142–151.
24. Bertolino A, Miranda B, Pietrantuono R, Russo S. Adaptive coverage and operational profile-based testing for reliability improvement. *39th Int. Conf. on Software Engineering (ICSE)*: IEEE: Buenos Aires, Argentina, 2017; 541–551.
25. Cotroneo D, Pietrantuono R, Russo S. RELAI testing: A technique to assess and improve software reliability. *IEEE Transactions on Software Engineering* 2016; **42**(5):452–475.
26. Lv J, Yin B-B, Cai K-Y. Estimating confidence interval of software reliability with adaptive testing strategy. *Journal of Systems and Software* 2014; **97**:192–206.
27. Lv J, Yin B-B, Cai K-Y. On the asymptotic behavior of adaptive testing strategy for software reliability assessment. *IEEE Transactions on Software Engineering* 2014; **40**(4):396–412.
28. IEEE. *IEEE Recommended Practice on Software Reliability*, 2017. IEEE Std 1633-2016.
29. Lohr SL. *Sampling Design and Analysis*, 2nd edn. Duxbury Press: Boston, USA, 2009.
30. Ostrand TJ, Balcer MJ. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM* 1988; **31**(6):676–686.
31. Amazon. *Cloudwatch*. http://aws.amazon.com/cloudwatch(Lastchecked2019/07/12),.
32. Nagios Enterprises. *Nagios Monitoring Solutions*. www.nagios.org (Last checked 2019/07/12).
33. Hansen MH, Hurwitz WN. On the theory of sampling from finite populations. *The Annals of Mathematical Statistics* 1943; **14**(4):333–362.
34. Arcuri A. RESTful API automated test case generation. *Int. Conf. on Software Quality, Reliability and Security (QRS)*: IEEE: Prague, Czech Republic, 2017; 9–20.

DOI: 10.1002/stvr

35. Miller KW, Morell LJ, Noonan RE, Park SK, Nicol DM, Murrill BW, Voas M. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering* 1992; **18**(1):33–43.
36. Montgomery DC. *Design and Analysis of Experiments*. John Wiley & Sons: USA, 2006.
37. Pietrantuono R, Russo S. Probabilistic sampling-based testing for accelerated reliability assessment. *IEEE Int. Conf. on Software Quality, Reliability and Security (QRS)*: IEEE: Lisbon, Portugal, 2018; 35–46.

DOI: 10.1002/stvr