Run-time Reliability Estimation of Microservice Architectures

Roberto Pietrantuono, Stefano Russo, Antonio Guerriero
DIETI, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Napoli, Italy
{roberto.pietrantuono, stefano.russo}@unina.it, antonio.guerriero8@studenti.unina.it

***Abstract*—Microservices are gaining popularity as an architectural paradigm for service-oriented applications, especially suited for highly dynamic contexts requiring loosely-coupled independent services, frequent software releases, decentralized governance and data management. Because of the high flexibility and evolvability characterizing microservice architectures (MSAs), it is difficult to estimate their reliability at design time, as it changes continuously due to the services' upgrades and/or to the way applications are used by customers.**

This paper presents a testing method for on-demand reliability estimation of microservice applications in their operational phase. The method allows to faithfully assess, upon request, the reliability of a MSA-based application under a scarce testing budget, at any time when it is in operation, and exploit field data about microservice usage and failing/successful demands. A new in-vivo testing algorithm is developed based on an adaptive web sampling strategy, named *Microservice Adaptive Reliability Testing* (*MART*). The method is evaluated by simulation, as well as by experimentation on an example application based on the Netflix Open Source Software MSA stack, with encouraging results in terms of estimation accuracy and, especially, efficiency.

***Index Terms*—Reliability assessment, microservice architecture, software testing, runtime testing, operational testing, reliability testing, adaptive testing, in-vivo testing, Netflix.**

I. Introduction

Microservice architectures (MSAs) are a new way of architecting applications as suites of independently deployable services, running in their own processes and interacting via lightweight mechanisms [1]. They are gaining momentum especially for those contexts where a loose coupling between software services, a prompt reaction to changes and a fast software release cycle provide benefits in terms of delivered value. Many organizations, like Netflix, Amazon, eBay, Twitter have already evolved their business applications to MSA [2].

While on the design and development side research on MSA is evolving at a rapid pace, little effort is being devoted to reliability of applications developed by this paradigm. The actual value of a microservice application, especially when deployed on a large scale as is often the case, is greatly dependent on the user-perceived reliability. Failure in satisfying reliability requirements entails a large part of the business risk associated with these applications, and the continuous awareness of run-time reliability in the context where the application operates is paramount for decision-makers.

Testing is a common means for reliability assessment. A fundamental strategy is *operational testing* and its evolutions, which exploit the operational profile to derive test cases resembling the expected usage in operation [3]. However, in a MSA scenario, assessing reliability at development time is difficult and, in a sense, even pointless, since the software and the operational profile change continuously due to the frequent release, services' upgrades, dynamic service interactions, and/or to the way the applications are used by customers.

This paper proposes a run-time testing method to estimate reliability of microservice applications during their operational phase upon a reliability assessment request (e.g., periodically, at every new release of a microservice). The core of the method is a new testing algorithm called *Microservice Adaptive Reliability Testing (MART)*. It is formulated as an *adaptive sampling* scheme that navigates the input space of the application to draw test cases with higher chance of improving the reliability estimate. The algorithm is fed with updated information about the microservice usage profile and about failing/successful demands of the version of microservices deployed at assessment time.

The advantages of the proposed testing method are both in terms of *accuracy* and *efficiency*. Accuracy is enforced by exploiting the field data gathered via monitoring, which contribute to provide an estimate close to the actual reliability at assessment time, and assured by the unbiasedness of the *MART* estimator. Efficiency is favoured by the adaptivity feature of the *MART* algorithm, which is conceived to identify the most relevant test cases in few steps, providing a small-variance estimate under a scarce testing budget – a feature particularly important for a run-time testing strategy.

We use both simulation and experimentation to show the benefits of the method. Simulation is on a set of 26 configurations resembling possible different characteristics of the input space. Experimentation is on an application based on the Netflix Open Source Software platform. Comparison is against operational testing. Results under several simulation and experimentation scenarios highlight a remarkable improvement of both the estimation accuracy and efficiency. The rest of the paper is organized as follows: Section II surveys the related literature; Section III presents the method and the *MART* algorithm; Sections IV and V report results of simulation and experimentation, respectively; Section VI closes the paper.

II. Related Work

A considerable number of papers about MSA are being published since the last years. Besides architectural and design issues, researchers started targeting quality concerns and how this new architectural style impacts them. Among the

quality attributes of interest, *performance* and *maintainability* are the most investigated ones according to a recent study [2]. Reliability is considered in few studies and always in its broader acceptation related to dependability (i.e., fault tolerance, robustness, resiliency, anomaly detection) – no study deals with reliability meant as probability of failure-free operation. Moreover, reliability-related considerations rarely appear as the main proposal of a research, but more often as a side concern in a design-related proposal.

Toffetti *et al.* propose an architecture to enable a resilient self-management of microservices on the cloud, by monitoring application and infrastructural properties to provide timely reactions to failures [4]. A framework for software service emergence in pervasive environments is proposed by Cardozo [5], where the problem of evaluating reliability under changing environment-dependent services is recognized but left to future work. Kang *et al.* present the design, implementation, and deployment of a microservice-based containerized OpenStack instance [6]. The authors implement a recovery mechanism for microservices to increase availability. Butzin *et al.* investigate the adoption of MSA for Internet of Things applications, proposing a fault management mechanism based on the circuit-breaker pattern, which prevents a failed service from receiving further requests until its recovery is complete, so as to avoid cascading failures [7]. Similarly, the service discovery mechanism proposed by Stubbs *et al.*, based on the *Serf* project [8], is equipped with monitoring and self-healing capabilities [9]. Failure detection ability provided by *Serf* is also exploited in the design of a decentralized message bus for communication between services [10]. Along the same line, Bak *et al.* [11] include an anomaly detection microservice in their MSA for location and context-based applications. Also testing is still an underestimated issue in MSA and mostly focuses on robustness/resiliency assessment. Heorhiadi *et al.* propose a framework for testing the failure-handling capabilities of microservices by emulating common failures observable at network level. Fault injection is also used by Nagarajan *et al.* at Groupon [12], to assess resiliency of MSA-based systems, and by Meinke *et al.* [13] where a learning-based testing approach evaluates the robustness of MSAs to injected faults.

While, to the best of our knowledge, there is no attempt for a reliability assessment solution for MSAs, existing techniques could be borrowed. *Operational testing*, where a testing profile is derived in accordance with the expected *operational profile*, is a reference technique for software reliability engineers [3]. The two main problems it has always suffered from are: *i)* the difficulty in determining the operational profile at development time [14]–[16], and *ii)* the scarce ability to deal with low-occurrence failures (as it targets mainly high-occurrence ones). These issues cause estimates with large variance (due to the few failures exposed) and bias (due to inaccurate profiles).

As in many other scenarios, for MSAs the assumption of an operational profile known at development time is easily violated. In addition, in MSA the changing of the software itself needs to be considered too, as continuous service upgrades occur. The method proposed here feeds the run-time testing algorithm with the updates of the profile and of the services failure probability as well, so as to assess the actual reliability depending on current usage and deployed software.

Indeed, generating and executing tests at run-time further stresses the second issue criticized to operational testing, namely the high cost required to expose many failures besides the high-occurrence ones. To this aim, evolutions of operational testing could be considered, which improve the fault detection ability by a partition-based approach and through adaptation. For instance, Cai et *al.* published several papers on *Adaptive Testing*, in which the assignment of the next test to a partition is based on the outcomes of previous tests to reduce the estimation variance [17]–[19]. The profile (assumed known) is defined on partitions, and selection within partitions is done by simple random sampling with replacement (SRSWR). Adaptiveness is also exploited in our recent work, where we use *importance sampling* to allocate tests toward more failure-prone partitions [20]–[22]. *Adaptive random testing* (ART) [23] exploits adaptiveness to evenly distribute the next tests across the input domain, but it aims at increasing the number of exposed failures rather than at assessing reliability. The sampling procedure is another key to improve the efficiency while preserving unbiasedness. In [24] and [25], the authors adopt stratified sampling and/or sampling without replacement (SRSWOR). In our recent work, we introduce a family of testing algorithms that enable the usage of more efficient sampling schemes than SRSWR/SRSWOR [26]. The proposed method goes beyond these: it includes a new sampling-based testing algorithm, *MART*, conceived to quickly detect clusters of faults with very scarce testing budget – hence, suitable for run-time testing – and it considers the updated profile and services version at assessment time. Estimation efficiency (i.e., small-variance) and accuracy (w.r.t. the real reliability at assessment time) are both pursued thanks to these features.

III. The Reliability Assessment Method

A. Assumptions

The following assumptions, typical of reliability assessment testing studies (e.g., [17]–[20]), are made:

1) Consider a *demand* as an invocation of a microservice's method: a demand leads to failure or success; we are able to determine when it is successful or not (perfect oracle).
2) The code is not modified during the testing activity; code can be modified (and detected faults can be removed) after the assessment.
3) A test case is a demand executed during testing: test cases are drawn independently from a demand space.
4) The demand space D of a microservice can be partitioned into a set of m subdomains: $\{D_1, D_2, \ldots, D_m\}$. The number of subdomains and the partitioning criterion are decided by the tester. In general, there are several ways in which the test suite can be partitioned (e.g., based on functional, structural, or profile criteria), provided that test cases in a partition have some properties

in common; these are usually dependent on the information available to test designers and on the objective. The choice does not affect the proposed strategy, but of course different results can be obtained according to it.

5) The operational profile P can be described as a probability distribution over the demand spaces of all microservices. Differently from most of literature on reliability testing, no assumption is made on the upfront knowledge of the profile, but we assume the ability to monitor the invocations to each microservice at run-time: the dynamic nature of the assessment process makes use of the updated information coming from the field in order to provide an estimate in line with the observed usage profile and failure probability of services.

Assumptions 3 and 5 are typically met in a MSA, since Microservces are, by their nature, implemented as loosely-coupled units and the monitoring/feedback mechanism is a common facility for what said in Section II.

For each subdomain D_i of a microservice, we define the probability of selecting a failing demand from D_i as: $x_i = f_i p_i$, where p_i is the probability of selecting an input from D_i, and f_i is the probability that an input selected from D_i is a failing demand. The assessment method aims at unbiasedly and efficiently estimating the reliability R upon a request during the operational phase, with R defined as:

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

B. Overview of the method

Figure 1 outlines the phases of the assessment process. The assessment includes "development-time" one-off activities aimed at mapping the demand space to a mathematical structure used to derive test cases, and then "run-time" activities to actually carry out the reliability estimation upon request, by test cases generation and execution. The assessment exploits the feedback coming from the field (thanks to monitoring facilities) in order to estimate reliability in line with the current *usage probability*, that is p_i, and the *probability of failure on demand (PFD)*, that is f_i, of microservices under test. In the following, each step is detailed.

C. Partitioning and Initialization

As first step, the demand space D of each microservice is partitioned in a set of subdomains D_i. Partitioning is applied to the arguments of each microservices' method, whose values are grouped in sets of *input classes*. Any partitioning criterion applies, inasmuch it reflects the tester's belief about the failure probability of such sets of inputs. Consider, for instance, the method `Login(String username, String password)`: the input `username` can be associated with 5 input classes according to the string length (in-range, out of range) and content (only alphanumeric or ASCII, plus the empty string); `password` can be associated with 7 input classes according to the length and content (as for `username`), but also to the satisfaction of application-specific

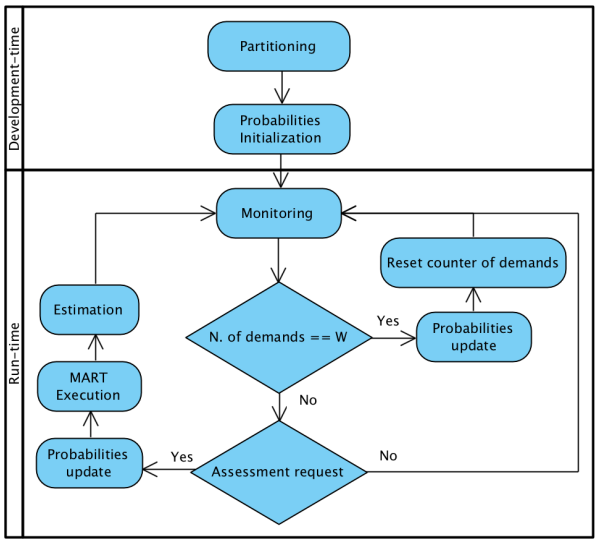

Fig. 1: The assessment steps

requirements (e.g., one upper case letter, one special character, etc., broadly split in the classes: requirements satisfied, not satisfied). As a result, a set of 35 *test frames* are derived, meant as the cartesian product of all input classes. In our formulation, test frames correspond to the subdomains D_i. For each pair of test frames $< i, j >$ referring to a same method of a microservice, a *distance* d is defined 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 represents the potential difference in the demands; the greater it is, the bigger the chance of executing different control flow paths within the method's code.

Each test frame i of the method h of microservice m is associated with the current estimate of the two values introduced in Section III-A: *a)* the probability that a demand is done selecting an input from that test frame ($\hat{p}_{i,h,m} \in [0,1]$, such that $\sum_{i,h,m} \hat{p}_{i,h,m} = 1$), which is the estimate of the usage probability of subdomain D_i; *b)* the probability that a demand from that test frame fails ($\hat{f}_{i,h,m} \in [0,1)$, which is the estimate of the PFD for subdomain D_i. We refer to $\hat{p}_{i,h,m}$ and $\hat{f}_{i,h,m}$ simply as $\hat{p}_i$ and $\hat{f}_i$ when there is no ambiguity. At the beginning, ignorance of the profile and of the expected failure probabilities can be dealt with by assigning the same probability to all $\hat{p}_i$ (summing up to 1) and the same probability $\hat{f}_i$ to all test frames in order to start up the algorithm (*Probabilities initialization* in Fig. 1).[1] Then, the estimate will be refined at run time as more information becomes available from the feedback cycle (*run-time* phase in Figure 1). Of course, any preliminary estimate of p_i and f_i of test frames (or, at coarser grain, of methods or microservices) would expedite the assessment. In real cases, rarely a tester is in a complete ignorance about expected usage or failure proneness of microservices at release time; some information

[1] Alternatively, $\hat{p}_i$ and $\hat{f}_i$ can be assigned random values each drawn from a uniform distribution in [0,1] and, in the case of $\hat{p}_i$, normalized to sum to 1.

is usually available either as quantitative evidence or at least as tester's belief. The partitioning criterion is itself an example of belief of tester, who judges some classes as more prone to failure while others are deemed correct. It is shown that an even partial knowledge distinguishing the more failure-prone partitions improves the assessment w.r.t. the uniform case [26].

D. Monitoring and probabilities update

The run-time assessment (*Run-time* activities in Figure 1) requires the ability of monitoring and updating the usage profile and the *PFD* for each test frame. Common monitoring tools can be used to gather data, such as Wireshark, Amazon Cloudwatch, Nagios. We customize a tool developed by our research group, called MetroFunnel[2], tailored for microservice applications. In general, monitoring should be able to gather (at least) the number of request/response to each microservice method and the outcome (correct/failure). Note that a rough reliability estimate could be derived by analyzing the gathered data, but, due to the passive nature of monitoring, the demand space might be not explored adequately (e.g., never exercising failing demands, or never using a test frame), yielding large-variance estimates. Our aim is to assure a small-variance (i.e., high-confidence) estimate by actively spotting those demands more informative about the current reliability.

The update of monitored data is done periodically (each W demands) and upon a reliability assessment request. Each $\hat{p}_i$ value is updated by using a sliding window of size W representing the maximum number of demands (i.e., the length of the history taken into account). The update rule is:

$$\hat{p}_i^u = \hat{p}_i^{u-1} \cdot [H + (1-H) \cdot (1 - \tfrac{R}{W})] + \hat{o}_i^u \cdot (1-H) \cdot (\tfrac{R}{W})$$

where:

- $\hat{p}_i^{u-1}$ is the occurrence probability of the i-th test frame in the previous update request;
- $\hat{o}_i^u$: is the occurrence probability at the current step estimated as the ratio between demands targeted at the i-th test frame and the total number of demands;
- H: is a value between 0 and 1, representing the minimum percentage of history kept in the update operations (in our case it is set to 50%);
- R: is the total number of executed demands (at most W).

The update of the *PFD* follows exactly the same rule, with $\hat{f}_i$ instead of $\hat{p}_i$ and $\hat{o}_i$ referring to the ratio between failed over executed demands with inputs belonging to the i-th test frame. The rule is to guarantee that possible changes of the operational profile (e.g., due to new users) and of the *PFD* (e.g., due to new releases of microservices) are detected in few steps, unlike the case of all the history being considered. The window W needs to be tuned depending on the context. Other update approaches can be used, like the black-box ones, where adjustments to the frequentist or Bayesian estimators are done at a profile changes [27], [28], or the white-box approach where the control flow transfer among components is captured [29]. However, investigating the best update strategy is outside the scope of this work and matter of future work.

[2]MetroFunnel is available at: *https://github.com/iraffaele/MetroFunnel.*

E. MART algorithm

1) Network structure: In *MART*, the test case space is represented as a network where each *node* is a test frame and *links* between nodes represent a dependency between the tester's beliefs about the failure probability of test frames of a same method. Since demands drawn from two test frames of a same 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 test frame of the same method proportionally to the distance between the two. Given the failure probability $P(i)$=$\hat{f}_i$ and $P(j)$=$\hat{f}_j$ of two test frames, the *link* is intended to capture the joint belief that a test case from both frames will fail. To this aim, each link between a pair of nodes $< i, j >$ is associated with a *weight* $w_{i,j}$ defined as the *joint probability* of failure of i and j: $P(i \cap j) = P(i|j) \cdot P(j)$. The conditional probability of failure $P(i|j)$ is the probability for a test frame i to fail conditioned on the fact that a failure is observed in the j-th test frame. $P(i|j)$ depends on the distance in an inversely proportional way: the smaller the distance, the more similar the two frames, and the bigger the conditional probability of failure. We represent this relation by: $F(d) = \frac{1}{d}$, hence: $P(i|j) = P(i)\frac{1}{d}$ with $d > 0$ (as at least one input class differs between two test frames), but other distance functions can be conceived. Consequently, weights are defined as: $w_{i,j} = \hat{f}_j \hat{f}_i \frac{1}{d}$, and, since they are based on failure probabilities, they are also updated at run time by monitoring data.

2) Test generation algorithm: The *MART* algorithm for test cases generation is encoded as an adaptive sampling design on the defined network structure, in which the generation of the next test case depends on the outcome of the previous ones. Given a testing budget in terms of number of test cases to run, the goal is to derive tests contributing more to an efficient (i.e., low variance) unbiased estimate. Sampling adaptivity is a feature that allows spotting rare and clustered units in a population so as to improve the efficiency of the estimation [30] – this makes such a type of sampling suitable for testing problems, especially in late development and/or operational phase, since failing demands are relatively rare with respect to all the demands space and are clustered. *MART* generates one test case at each step. In a given step, the algorithm aims at selecting the test frame with higher chance of having failing demands. The exploited design is the adaptive web sampling defined by Thompson for survey sampling problems [31]. Within the selected test frame, a test cases is generated by drawing a demand according to a uniform distribution – namely, each demand with equal probability of being selected.

Specifically, at the k-th step, *MART* combines two techniques (i.e., two *samplers*): a *weight-based sampler* and a *simple random sampler* to select the next test frame. The *weight-based sampler* (WBS) follows the links between frames, in order to identify possible clusters of failing demands. This depth exploration, useful when a potential "cluster" of failing demands is found, is balanced with the *simple random sampler* (SRS) for a breadth exploration of the test frame space, useful

to escape from unproductive local searches. At each step k, the next test frame is selected by a mixture distribution according to the following equation:

$$q_{k,i} = r\frac{w_{a_{k,i}}}{w_{a_{k+}}} + (1-r)\frac{1}{N - n_{s_k}} \quad (2)$$

where:

- $q_{k,i}$ is the probability to select test frame i;
- N: is the total number of test frames;
- s_k is the current sample, namely the set of all selected test frames up to step k;
- n_{s_k} is the size of the current sample s_k;
- a_k is the active set, which is a subset of s_k along with the information on the outgoing links;
- $a_{k,i}$ is the set of the outgoing links from test frame i to test frames not in the current sample s_k;
- $w_{a_{k,j}} = \sum_{i \in a_k} w_{i,j}$ is the total of weights of links outgoings from the active set;
- $w_{a_{k+}} = \sum_{i \in a_k, j \in \bar{s_k}} w_{i,j}$;
- r between 0 and 1 determines the probability to use the weight-based sampler or the random sampler.

The selection of the first test frame is done by SRS, and the active set is updated. Then, at each iteration, if there are no outgoing links from the active set (i.e., no link with a weight greater than 0), the SRS is preferred, so as to explore other regions of the test frame space. Otherwise, the selection of the sampler is dependent on r. When WBS is used, the selection is done proportionally to the weights – first term of Eq. 2. Such a disproportional selection is then counterbalanced in the estimator preserving unbiasedness. When SRS is used, the not-yet-selected test frames have equal selection probability[3] – second term of Eq. 2. The selected test frame is added to the active set. All is repeated until the testing budget is over.

3) Dynamic update of the sampler selection: Besides the basic version of *MART*, a further variant is implemented where the choice of r is made adaptive itself. We denote this variant *MART*$_D$ opposed to the previous one denoted as *MART*$_S$ (D and S standing for *dynamic* and *static*, respectively). In *MART*$_D$ an initial value of $r = r_0$ is specified, which, in a sense, encodes the initial trust that the tester has in the WBS compared to SRS. $r_0 \geq 0.5$ to assure r will always be in $[0, 1]$. The approach in Fig. 2 is used for the dynamic update of r. The update resembles the mechanism of a serial input-parallel output shift register. A binary cell is 1 if the corresponding scheme is selected, 0 otherwise. Each binary cell is associated with a decreasing percentage (starting from cell 1) of the $(1 - r_0)$ quantity, that is equally spread across the cells and depends on the size of the register (namely, if i is the index of the cell and s the number of cells, the percentages $v[i]$ are: $v[i] = i \cdot \frac{100}{\sum_{i=1}^{s} i}$). For instance, for a shift register of size 4, the assignment is: 40% for the first cell, then 30%, 20% and 10% for the second, third and fourth ones. The initial

[3]The scheme can be either with- or without-replacement, with few changes in the estimator [31]; Eq. 2 is the without-replacement version, the with-replacement variant replaces $\frac{1}{N-n_{s_k}}$ with $\frac{1}{N}$.

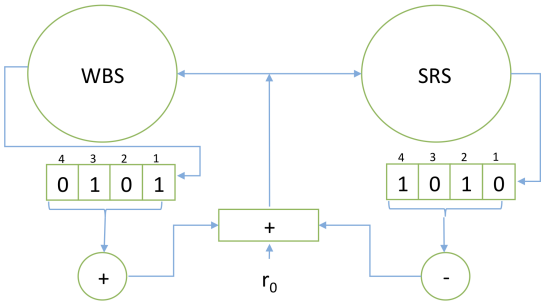

Fig. 2: *MART*$_D$: update of r

r_0 value is added to the quantity resulting from the sum of products between the first register and $v[i]$ and subtracted the sum of products between the second register and $v[i]$ (e.g., in the Figure, $r = r_0 + (1-r_0)(0.4+0.2) - (1-r_0)(0.3+0.1)$). At each step k of the algorithm, the register corresponding to the chosen scheme (WBS or SRS) is shifted by one position, by writing a '1' or '0' if the sampler revealed a failing test frame, and r is updated as described.

F. Estimation

After testing, the estimation is carried out. Let us consider the quantity to estimate: $R = 1 - \sum_i x_i = 1 - \sum_i p_i f_i$, where x_i is the probability that a test case from i is selected and fails. During testing, results in terms of failed/correct test cases are collected. Let us denote with $y_{i,t}$ the observed outcome of a test case t taken from test frame i, $y_{i,t}$=0/1. In the general case, in which some failure data for test frame i is available from the field, the estimate of 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). When no data is observed for a test frame during operation, the estimate becomes: $\hat{f}_i' = \frac{\sum_{t=0}^{m_i} y_{i,t}}{m_i}$. Additionally, in a without-replacement scenario, which can be preferred under a scarce budget, $m_i = 1$ and $\hat{f}_i' = 0/1$. The Thompson estimator is tailored for our assessment problem, whose idea is to take the average of the (SRS or WBS) estimators obtained at each step. The total failure probability Φ is unbiasedly estimated as:

$$\Phi = \frac{1}{n}(N\hat{f}_i' + \sum_{i=2}^{n} z_i); \quad (3)$$

where:

$N\hat{f}_i'$	is the total estimator at the first step $k = 0$ (the first observation taken by the SRS);
z_i	is the total estimator obtained at step $k = i$, and
$z_i =$	$\sum_{j \in s_k} \hat{x}_j + \frac{\hat{x}_i}{q_{k,i}} = \sum_{j \in s_k} \hat{p}_j \hat{f}_j' + \frac{\hat{p}_j \hat{f}_j'}{q_{k,i}}$;
n	is the number of executed test cases;
N	is the total number of test frames.

IV. SIMULATION

Simulation and experimentation are used to assess performance of *MART* against operational testing (*OT*). Partition-based operational testing is used, where partitions (in our case, test frames) are selected according to an operational profile, and test cases are randomly taken from the equally-probable inputs of the selected partition [32]. For simulation, a uniform distribution is assumed as operational profile, since the interest is to evaluate the approaches with respect to characteristics of the demand space and problem size regardless of a specific profile; for experimentation, specific profiles are defined.

A. Simulation scenarios

MART$_S$ and *MART*$_D$ and *OT* are tested in 26 simulated scenarios obtained by varying the following characteristics (spread and amount of failures) and problem size:

- *Type of partitioning*. Partitioning is about separating correct from failing test cases into test frames, so as all test cases of a frame have the same outcome. In a perfect case, a test frame believed to be *failing* (*correct*) contains only failing (correct) test cases – hence its failure probability (as proportion of failing test cases) is 1 (0). We refer to this configuration as *perfect partitioning*. As we depart from this, the failure probability of a test frame believed to be failing is smaller, meaning that partitioning is not perfect. We consider two configurations representing a non-perfect partitioning, by assigning a failure probability of 0.9 (0.1) and of 0.75 (0.25) to failing (correct) test frames. Bigger errors would go toward a uniform belief about failing/correct test frame (i.e., assigning 0.5 and 0.5 to failing and correct test frames), i.e., there would not be a discriminative criterion to perform partitioning and random testing is expected to be better – hence, we treat this as a separate configuration.
- *Failing test frame proportion*. This is the proportion of the *failing test frames* over the total, for which we consider two values, 0.1 and 0.2.
- *Clustered population*. For each of the above 3x2=6 configurations, we further consider the case of test frames grouped in clusters. These are obtained by considering the number of failing test frames F in that configuration and determining the cluster size as $S = C \cdot F$, with the clustering factor C =10% or 20% depending on the failing test frame proportion. For the resulting F/S clusters; a number of F/S test frames are randomly chosen as centroids, and, for each of them, S test frames with the minimum distance d are selected as cluster's member.
- *Total number of test frames*, N. Two order of magnitudes are considered: $N = 100$, $N = 1,000$.

These combinations generate 24 scenarios. Besides, the mentioned uniform case (0.5/0.5) is added (under both $N = 100$ and $N = 1000$), getting to 26 scenarios. The assessment is made at 9 checkpoints: $n_1 = 0.1N, n_2 = 0.2N, \ldots 0.9N$.

Prior to the comparison, we performed a sensitivity analysis on *MART* parameters.[4] Specifically, *MART*$_S$ has been run by varying the value of r as: $r = 0.2$, $r = 0.4$, $r = 0.6$, $r = 0.8$. The *MART*$_D$ variant has been run by varying the value of r_0 as: $r = 0.5$, $r = 0.6$, $r = 0.7$, $r = 0.8$, $r = 0.9$ and 3 size values of the register: $size = 3$, $size = 4$, $size = 5$ under all configurations. The best values, in terms of mean squared error (MSE) and variance, turned out to be: $r = 0.8$, $r_0 = 0.8$ and $size = 4$. The results that follow refer to *MART*$_S$ and *MART*$_D$ parametrized with these values.

B. Evaluation criteria

Accuracy and *efficiency* are considered as evaluation criteria. A simulation scenario j is repeated 100 times; r denotes one of such repetitions. At the end of each repetition, the estimate $\hat{R}_{r,j}$ is computed by the technique under assessment as well as the *true* reliability R_j – for simulation, we know in advance which input t is failing. For each scenario j, we compute the sample mean (M), sample variance (S) and MSE:

$$
\begin{aligned}
M(\hat{R}_j) &= \tfrac{1}{100} \textstyle\sum_{r=1}^{100} \hat{R}_{r.j} \\
MSE(\hat{R}_j) &= \tfrac{1}{100} \textstyle\sum_{r=1}^{100} (\hat{R}_{r.j} - R_j))^2 \\
S(\hat{R}_j) &= \tfrac{1}{100-1} \textstyle\sum_{r=1}^{100} (\hat{R}_{r.j} - M(\hat{R}_j))^2
\end{aligned}
\tag{4}
$$

Comparison of estimation accuracy is done by looking at the MSE. Comparison of efficiency is done by the sample variance S. The number of runs is: 3 techniques x 26 simulation scenarios x 9 checkpoints x 100 repetitions = 70,200 runs.

C. Results

Figures 3-6 show results in several representative configurations[5]. Figure 3 reports the configuration: <*type of partitioning*, *failing test frames proportion*, *number of test frames*> = $< 1/0, 0.1, 1000 >$. In this ideal case of a perfect partitioning, results are clearly in favour of *MART* for both evaluation criteria, with *MART*$_D$ performing better than the static case. OT gets close to *MART* with the increase of the number of tests, as it approaches to N. Figure 4 shows that in what can be considered a good partitioning ($0.9/0.1$), *MART*$_S$ is the best one and outperforms even *MART*$_D$. The case of $0.75/0.25$ (Figure 5) shows that as partitioning becomes worse and worse the performances of both *MART* algorithms decrease in terms of MSE but remains still remarkably superior in terms of variance. In clustered population, performance of *MART* is slightly further better with respect to the corresponding non-clustered cases. Finally, in the extreme case ($0.5/0.5$), the MSE of OT is better up to the first 30% of test cases, but it is still worse in terms of variance (Figure 6). Summarizing:

- In the case of uniform distribution of failing inputs across the input space (and with no clusters), the random testing approach, in which sampling resembles the characteristics of the population, could be preferred in terms of MSE,

[4]Detailed results of sensitivity analysis are not reported for lack of space; they are made available at *https://github.com/AntonioGuerriero/MART.*

[5]The results for all configurations are available at *https://github.com/AntonioGuerriero/MART.*

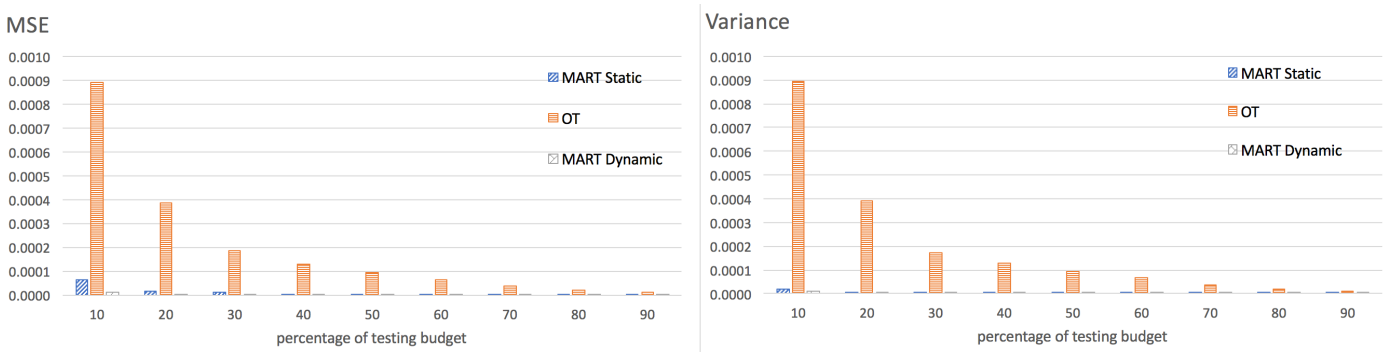

Fig. 3: Simulation - perfect partitioning, failing test cases proportion 10%, 1,000 test frames: MSE and variance

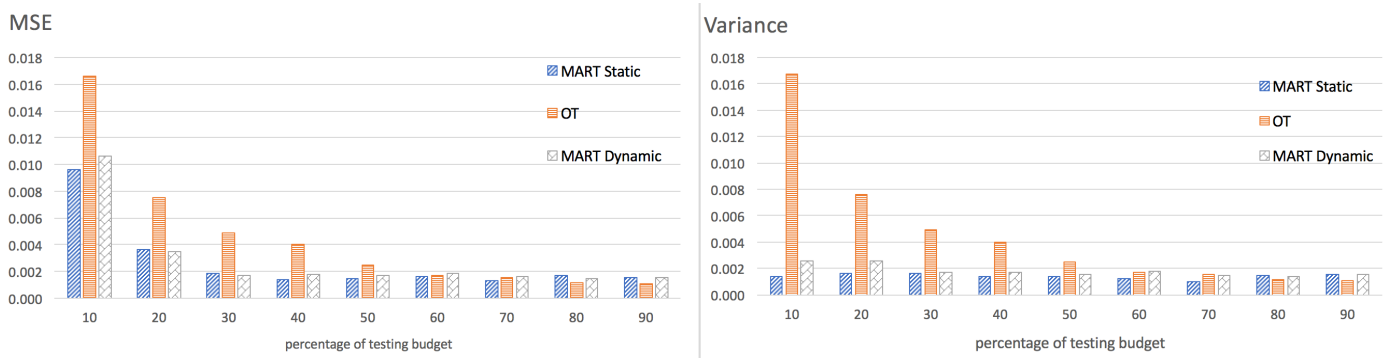

Fig. 4: Simulation - type of partitioning 0.9/0.1, failing test cases proportion 10%, 100 test frames: MSE and variance

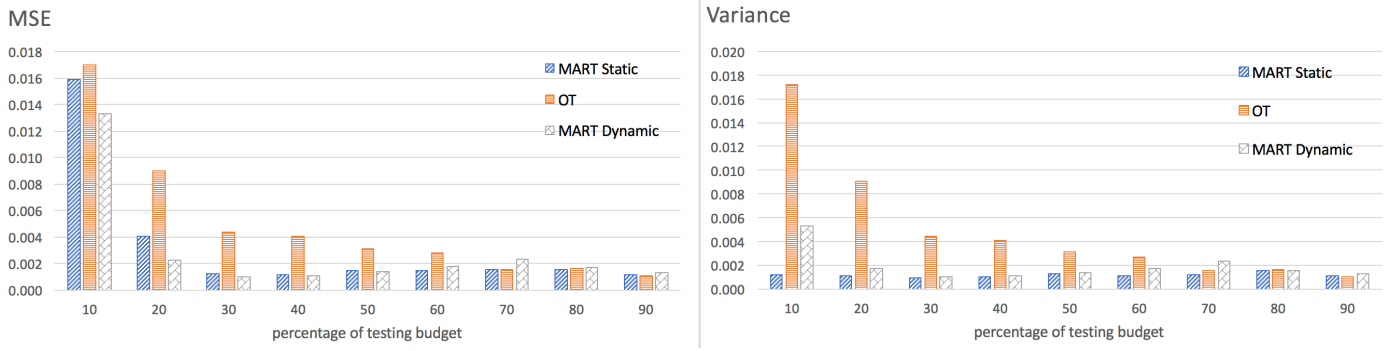

Fig. 5: Simulation - type of partitioning 0.75/0.25, failing test cases proportion 20%, 100 test frames: MSE and variance

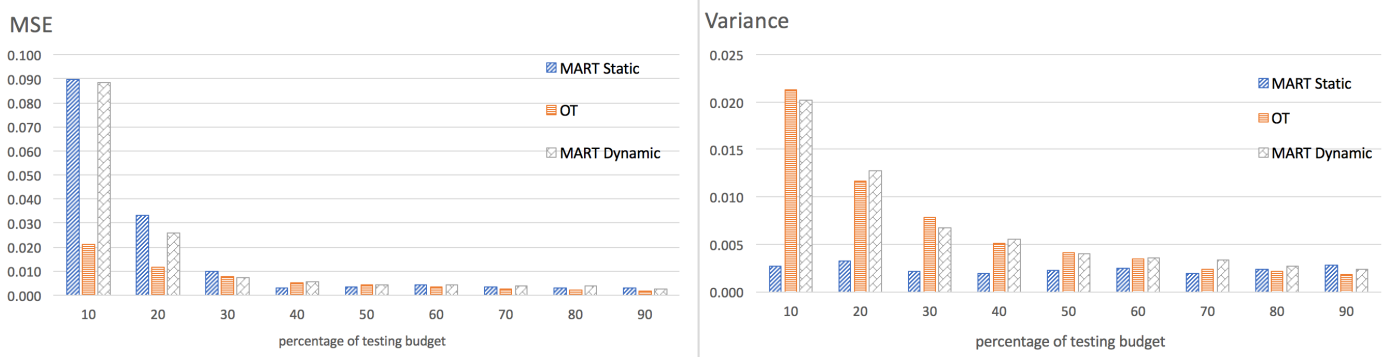

Fig. 6: Simulation - *random population*: MSE and variance

but not in terms of *variance*; in all the other cases, wherein partitioning has some chance of putting together common test cases from the failing behaviour point of view, *MART* show considerable advantages.
- *MART* performs much better with low number of test cases; this is due to the adaptivity feature, which allows picking up and exploring entire clusters of failing inputs soon. This is confirmed by the number of failure points found, which was always bigger in all the 26 tested configurations than the ones found by *OT*.
- $MART_D$ is generally better than the static counterpart, but the advantage of $MART_S$ in some cases is worth to be noted considering its simpler formulation.
- Results are unvaried under $N = 100$ and $N = 1,000$. The trends are approximately the same.

V. Experimentation

MART is experimented on a Netflix-based application example, named *Pet Clinic Microservices System*, for the management of a veterinary clinic. It is built by using the *Spring Cloud Netflix* technology stack, which provides Spring Boot apps with interoperability with Netflix Open Source Software (OSS) components, such as the *Eureka* service and the *Zuul* service. The application consists of five services: the *API gateway application* (built on *Zuul*), the *discovery server* (*Eureka*), the *config server application* to manage external properties for applications across environments, the *customers service application*, the *vets service application* and *visits service application*, which are services for customers, vets and visits management, respectively. The application has 13 methods and we manually derived 262 test frames.

A. Experiments

The experiments aim at evaluating *MART* in assessing reliability in operation. Four experiments are designed. The goal of experiments 1, 2 and 3 is to evaluate performance with three different initial operational profile estimates (i.e., a tester's estimate before operation). At the beginning of the experiment (before any operational data is still observed), we run both *MART* and OT and compare their estimate. Then, during the operational phase, *MART*, unlike OT, *i)* collects field data by monitoring, *ii)* updates the profiles, *iii)* runs the testing algorithm and *iv)* computes a new estimate. We evaluate the updated estimates at 3 subsequent steps during the operational phase, representing, ideally, a *request for a new reliability assessment*. Experiment 4 evaluates the method in presence of a change of the *true* operational profile, in order to assess performance under a *variable profile* scenario (opposed to the *stable profile* of experiments 1, 2 and 3). In this case, there are 3 update steps before the profile change, and 3 further steps after the profile change, i.e., under a new profile.

B. Initial probabilities assignment

Test frames could be assigned an initial *failure probability* and *occurrence probability* by assuming ignorance as mentioned in Section III-C, like we did in the simulation case. However, to evaluate the approach also in another setting, probabilities are hereafter assigned by assuming some knowledge of the tester about the test frames. To this aim, we preliminarily analyzed test frames to provide a realistic testing-time characterization. We ran 30 (uniform) random test cases for each test frame. The proportion of failures was used as an "equivalent" prior knowledge about failure probability, as suggested by the seminal work by Voas et al. [28], and to support distinguishing more or less failure-prone test frames. Based on them, we split test frames in three categories, and assigned an initial failure probability to each. Categories are:

- First category: 25 test frames that exhibited no failure. The initial failure probability to these test frames is set to: $\hat{f}_i = \epsilon = 0.01$. The assignment of $\epsilon > 0$ represents the uncertainty due to the limited number of observations.
- Second category: 46 test frames, which failed at any of the 30 executions. The initial failure probability for these test frames is set to: $\hat{f}_i = 1 - \epsilon = 0.99$.
- Third category: 191 test frames, the rest of test frames, which failed sporadically. Based on observed proportion of failures, approximately 1 failure every 10 requests, the initial probability is set to: $\hat{f}_i = 0.1$.

As for operational profiles, for the experimental purpose we need to consider both a *true* profile and an *estimated* profile. The *true profile* is used during the operational phase to submit demands to microservices, simulates the run-time usage of microservices. Based on the inspection of microservices' methods (e.g., looking at the size of test frames in terms of number of inputs, at their failure probability assessed as just mentioned, and at the functionality they support within the system), the following probabilities are set for test frames of the three categories: $p_i = 0.8/|F_1|$, $p_i = 0.05/|F_2|$, $p_i = 0.15/|F_3|$ (where $|F_1| = 26$, $|F_2| = 46$, $|F_3| = 191$ test frames), for each test frame of the first, second and third category, respectively. Clearly, while any other *true profile* can be assumed, we are interested in assessing how the difference between an *estimated profile* and a *true* one impacts the assessment. Hence, three *estimated profiles* are considered, deviating, respectively, by 10%, 50% and 90% from the true profile. The "deviation" is measured as the sum of absolute differences of the true *vs* estimated occurrence probability of each test frame: $e = \sum_{i=0}^{|F|} \left| p_i^E - p_i^T \right|$ where: $|F|$ is the number test frames; p_i^E is the estimated occurrence probability of a test frame; p_i^T is the true occurrence probability. Experiment 1, 2 and 3 use these three profiles. Experiment 4 needs to use two true profiles, since it aims at evaluating the method in presence of a true profile variation. A second profile is defined as: $p_i = 0.55/|F_1|$, $p_i = 0.05/|F_2|$, $p_i = 0.35/|F_3|$ for test frames of the first, second and third category, respectively. As initial estimated profile for this experiment, the average one is taken, hence the one deviating by 50% from the true one.

C. Experimental scenarios

Each experiment is repeated under a different number n of executed test cases. This is set as percentage of the 262 test frames, assuming to be in a situation with scarce testing

budget (i.e., less test cases than test frames). Specifically, we consider two values for a very scarce budget, which is particularly interesting in a MSA scenario, with n being 20% and 40% of test frames, and one value between 60% and 80% to assess performance when some more tests are available, namely: n = 70% of test frames. Considering 3 values for n and 4 experiments, the experimental scenarios are 12. In the *stable profile* case (experiment 1, 2 and 3), *MART* is run during the operational phase 3 times (3 update steps), while it is run 6 times in experiment 4 – hence 45 times in total; operational testing is run 12 times, at the beginning of each scenario, since no run-time update is foreseen by *OT*.

D. Evaluation criteria

MSE and *variance* are used again, as for simulation. However, an experimental scenario j is repeated 30 times for each technique instead of 100, being experiments more time-consuming. The *true* reliability R_j is computed by preliminary running $T = 10,000$ test cases under the true profile and using $R_j = 1 - \frac{F}{T}$, with F being the number of observed failures.

E. Testbed

The code to run the experiments is in a *testing service* interacting with the application's microservices. It includes:

- *Workload generator*: this module executes demands according to a given true profile, emulating a client of the application, with the possibility to change the profile and the failure probability (i.e., emulating an upgrade of a service) after a number of requests (5,000 in our case).
- *True reliability estimator*: this module runs T test cases according to a given true profile, and computes the frequentist reliability estimate (T=10,000 in our case).
- *Monitor & parser*: these modules are implemented by customizing a tool for microservice monitoring developed in our group, called *Metro Funnel*, and adding a log parser to extract the number of (correct/failing) demands to each microservice used for the probabilities update.
- *Tester*: this module implements the *MART* algorithm; it runs test cases according to what defined above, and provides the estimate of the MSE and *variance*.
- *Operational tester*: the module implements the *OT* algorithm, runs tests and estimates the MSE and *variance*.

F. Results

Fig. 7-9 report the results of experiments 1, 2 and 3. The values for *MART* are reported for step 1 to 3 during the operational phase representing the three subsequent assessment requests occurring every 5,000 demands. OT is a static approach, hence it is run only at step 1, before the operational phase starts, and then the same value is reported at step 2 and 3 for easing the visual comparison. The results show that:

- In all the cases, *MART*'s estimates have a smaller MSE and a much smaller *variance* than OT;
- *MART* is better since the beginning, regardless of the dynamic update of failure and usage probabilities, as the algorithm based on the adaptive web sampling scheme is able to spot clusters of failures more rapidly than *OT*;
- The update step based on monitoring makes *MART* considerably improve its performance, as profile and *PFD* estimates converge to the true ones and exploited by the algorithm. This progressive improvement is more evident for the MSE; variances are small since step 1.
- Looking at the difference between the 3 profiles, it turns out that with profile 1 (10% of deviation), Figure 7, the MSE decreases suddenly at step 2; with profile 2 (50% of deviation), there is a smoother decrease and the MSE is slightly bigger than the profile 1 case; with profile 3 (90% of deviation), MSE is expectedly bigger at step 1, but suddenly decreases at step 2. This means that, even under a profile much different from the true one, the adaptivity allows correcting the estimate, of course with different speed of convergence toward the true reliability.
- The difference between the percentage of executed tests (20%, 40% and 70%), reported in each graph, is of interest too: what can be observed is that the MSE and *variance* obtained with only $n = 20\%$ is comparable, or even better in several cases, with those obtained with 40% and 70%. At step 2, and especially at step 3, the 20% case has no difference with 40% and 70% in terms of MSE, and is even better in terms of variance.

Figure 10 reports the results of experiment 4, i.e., under a variable profile scenario. After step 3, the profile changes – namely, the demands are issued in accordance to the second true profile and the assessment is computed again by both approaches at step 4. The decrease of MSE of *MART* at step 5 highlights how the proposed method suddenly detects the change of the profile and "correct" the estimate according to the new profile; at step 6, the results are like in the steps 1 to 3. Results on *variance* are not affected by the changed profile: the *MART*'s estimate is stable and much more consistent over the 30 repetitions compared to *OT*. The superiority of *MART* is statistically confirmed by the Wilcoxon test for both MSE and *variance*, in both cases yielding a *p-value*$< .0001$.

VI. Conclusion

This paper presented a run-time testing method for on-demand assessment of reliability in MSA. Results suggest that both the *run-time adaptivity* to the real observed profile and failing behaviour and the *testing-time adaptivity* implemented by the new *MART* algorithm (allowing to spot failures with few tests while preserving the estimate unbiasedness) are good starting points to further elaborate in the future. Improvements can be achieved by investigating what other information can be useful to expedite the assessment (e.g., about service interactions), by exploring other approaches for the information update (e.g., Bayesian updates), by exploring different partitioning criteria and/or by integrating optimal partition allocation strategies [33]–[35]. Finally, further extensive experiments are planned to improve generalization of results.

Acknowledgment

This work has been supported by the PRIN 2015 research project "GAUSS" (2015KWREMX_002) funded by MIUR.

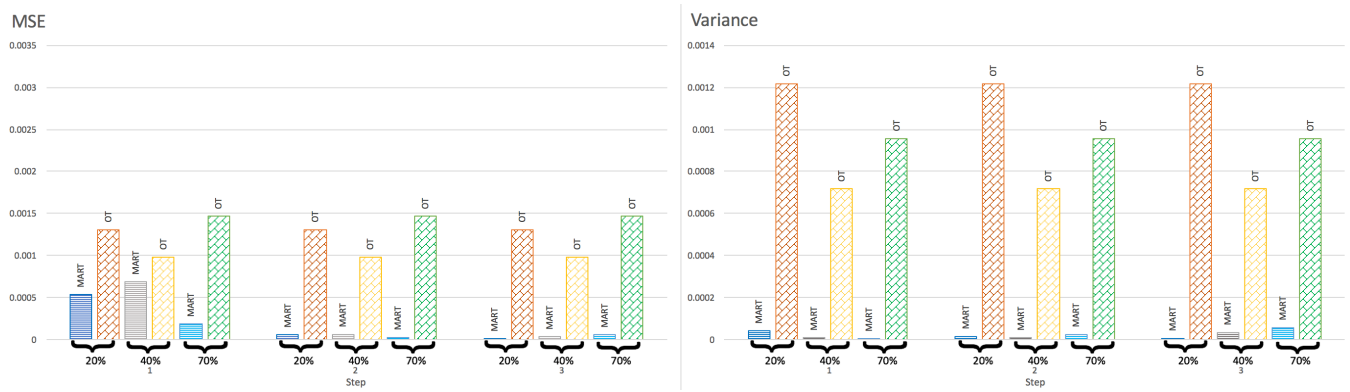

Fig. 7: Stable profile, experiment 1 - 10% deviation of estimated vs true profile: MSE and variance

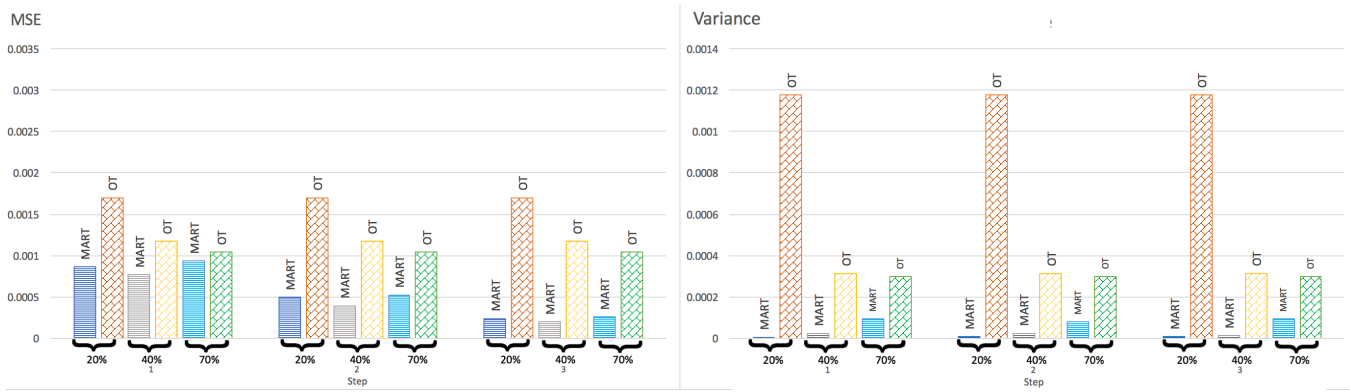

Fig. 8: Stable profile, experiment 2 - 50% deviation of estimated vs true profile: MSE and variance

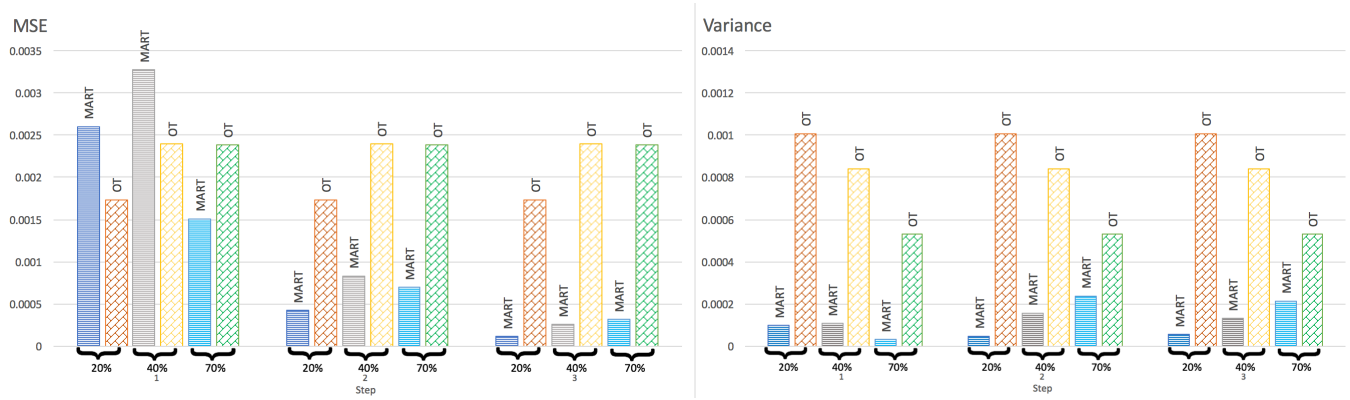

Fig. 9: Stable profile, experiment 3 - 90% deviation of estimated vs true profile: MSE and variance

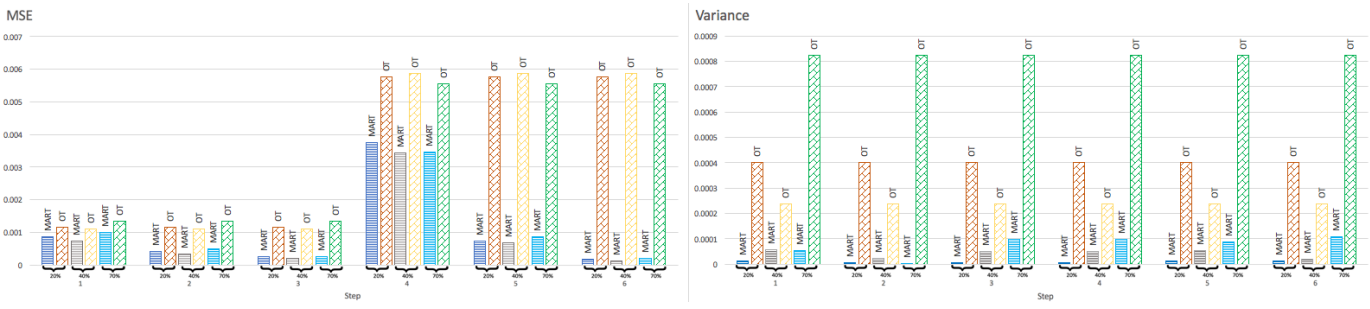

Fig. 10: Experiment 4, variable profile. MSE and variance

REFERENCES

[1] J. Lewis and M. Fowler. Microservices - a definition of this new architectural term. Available at: http://martinfowler.com/articles/microservices.html, 2014.

[2] P. Di Francesco, I. Malavolta, and P. Lago. Research on architecting microservices: trends, focus, and potential for industrial adoption. In *IEEE International Conference on Software Architecture*, pages 21–30, 2017.

[3] M. R. Lyu, editor. *Handbook of software reliability engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[4] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds. An architecture for self-managing microservices. In *International Workshop on Automated Incident Management in Cloud*, AIMC '15, pages 19–24, 2015.

[5] N. Cardozo. Emergent software services. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 15–28, 2016.

[6] H. Kang, M. Le, and S. Tao. Container and microservice driven design for cloud infrastructure devops. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 202–211, 2016.

[7] B. Butzin, F. Golatowski, and D. Timmermann. Microservices approach for the internet of things. In *21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6, 2016.

[8] HashiCorp. The Serf tool - Decentralized Cluster Membership, Failure Detection, and Orchestration. http://www.serfdom.io (last access: 2018-05-12).

[9] J. Stubbs, W. Moreira, and R. Dooley. Distributed systems of microservices using Docker and Serfnode. In *7th International Workshop on Science Gateways (IWSG)*, pages 34–39, 2015.

[10] P. Kookarinrat and Y. Temtanapat. Design and implementation of a decentralized message bus for microservices. In *13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–6, 2016.

[11] P. Bak, R. Melamed, D. Moshkovich, Y. Nardi, H. Ship, and A. Yaeli. Location and context-based microservices for mobile and internet of things workloads. In *IEEE International Conference on Mobile Services (MS)*, pages 1–8, 2015.

[12] A. Nagarajan and A. Vaddadi. Automated fault-tolerance testing. In *IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 275–276, 2016.

[13] K. Meinke and P. Nycander. Learning-based testing of distributed microservice architectures: Correctness and fault injection. In D. Bianculli, R. Calinescu, and B. Rumpe, editors, *Software Engineering and Formal Methods*, pages 3–10, 2015.

[14] B. Beizer. Cleanroom process model: a critical examination. *IEEE Software*, 14(2):14–16, 1997.

[15] B. Littlewood and L. Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.

[16] D. Cotroneo, R. Pietrantuono, and S. Russo. Combining Operational and Debug Testing for Improving Reliability. *IEEE Transactions on Reliability*, 62(2):408–423, 2013.

[17] K-Y. Cai, Y-C. Li, and K. Liu. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology*, 46(15):989–1000, 2004.

[18] J. Lv, B.-B. Yin, and K.-Y. Cai. On the asymptotic behavior of adaptive testing strategy for software reliability assessment. *IEEE Transactions on Software Engineering*, 40(4):396–412, 2014.

[19] J. Lv, B-B. Yin, and K-Y. Cai. Estimating confidence interval of software reliability with adaptive testing strategy. *Journal of Systems and Software*, 97:192–206, 2014.

[20] D. Cotroneo, R. Pietrantuono, and S. Russo. RELAI Testing: A Technique to Assess and Improve Software Reliability. *IEEE Transactions on Software Engineering*, 42(5):452–475, 2016.

[21] A. Bertolino, B. Miranda, R. Pietrantuono, and S. Russo. Adaptive coverage and operational profile-based testing for reliability improvement. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 541–551, May 2017.

[22] D. Cotroneo, R. Pietrantuono, and S. Russo. A learning-based method for combining testing techniques. In *Proc. 35th Int. Conference on Software Engineering (ICSE)*, pages 142–151. IEEE, 2013.

[23] T.Y. Chen, H. Leung, and I.K. Mak. Adaptive random testing. In MichaelJ. Maher, editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321 of *Lecture Notes in Computer Science*, pages 320–329, 2005.

[24] A. Podgurski, W. Masri, Y. McCleese, F.G. Wolff, and C. Yang. Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodology*, 8:263–283, 1999.

[25] F.b.N. Omri. Weighted statistical white-box testing with proportional-optimal stratification. In *19th International Doctoral Symposium on Components and Architecture*, WCOP'14, pages 19–24. ACM, 2014.

[26] R. Pietrantuono and S. Russo. On Adaptive Sampling-Based Testing for Software Reliability Assessment. In *Proceedings 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–11, 2016.

[27] J. R. Brown and M. Lipow. Testing for software reliability. *SIGPLAN Not.*, 10(6):518–527, 1975.

[28] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, 1992.

[29] K. Goseva-Popstojanova and S. Kamavaram. Software reliability estimation under certainty: generalization of the method of moments. In *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings.*, pages 209–218, 2004.

[30] S. L. Lohr. *Sampling Design and Analysis*. Duxbury Press; 2 edition, 2009.

[31] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47(260):pp. 663–685, 1952.

[32] K-Y. Cai, C-H. Jiang, H. Hu, and C-G. Bai. An experimental study of adaptive testing for software reliability assessment. *Journal of Systems and Software*, 81(8):1406–1429, 2008.

[33] R. Pietrantuono, S. Russo, and K.S. Trivedi. Software Reliability and Testing Time Allocation: An Architecture-Based Approach. *IEEE Trans. on Software Engineering*, 36(3):323–337, 2010.

[34] R. Pietrantuono, P. Potena, A. Pecchia, D. Rodriguez, S. Russo, and L. Fernandez. Multi-objective testing resource allocation under uncertainty. *IEEE Transactions on Evolutionary Computation*, 22(3):347–362, 2017.

[35] G. Carrozza, R. Pietrantuono, and S. Russo. Dynamic test planning: a study in an industrial context. *International Journal on Software Tools for Technology Transfer*, 16(5):593–607, 2014.