Project funded by the EU Horizon 2020 programme under the Marie

Sk ldowska-Curie grant agreement No 871342

uDevoOps

Software Quality Assurance for Microservice Development Operations Engineering

Deliverable D3.1. Sampling-based Testing Techniques Design and Algorithms for QoS Testing

February 2023

Abstract

This document reports the results of Deliverable D3.1 of the μDevOps project, entitled "Sampling-based Testing Techniques Design and Algorithms for QoS Testing". The type of the deliverable is marked as *Report*, and its dissemination level is *Public*. The document will be made available through the project's website, `https://udevops.eu/`.

The document describes the techniques designed for testing Quality-of-Service (QoS) attributes for microservices, namely reliability, performance, security. The techniques also support testing for checking functional correctness and robustness.

The structure is as follows: first, an overview on the key challenges for testing, viewed in the context of Microservice and DevOps practices, is given. Then, a background section explains the adopted terminology and assumptions, the role of *probabilistic sampling* and of the auxiliary process-related information it requires about the system under test. Then, the testing techniques will be described with reference to the quality assessment and quality improvement perspective.

CONTENTS

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

The ultimate aim of testing is to expose potential *failures* that may impact the user experience in operation. Engineers select and prioritize test inputs according to the objective, e.g., checking functional correctness and/or satisfying quality requirements, such as robustness, reliability, performance, security. *Failures*, therefore, refer to deviations from such functional and non-functional requirements (i.e., from the expected behaviour) – namely, the system is said to fail when it does not satisfy these requirements.

Once failures are exposed, the natural following objective is then to *improve* the software, by removing the identified failure's cause. Therefore, the next activity is *debugging*, which identifies and removes the underlying cause (also known as *fault* or *defect*) of the exposed failures. However, it is also extremely important for engineers (designers, developers and testers) to know the extent to which the *operating software* will possess the quality attribute of interest once will be deployed. Therefore, *assessment*, besides

the *improvement*, is the further important objective, as it gives information useful for decision-making (e.g., if the product is ready to release or needs more testing, if quality is improving or not from release to release, ...). This distinction resembles the *debug* testing *vs operational* testing view present in the literature of 90's Beizer (1997). The two perspectives differ in how tests are derived – i.e., test generation criteria may differ because the objective is different, as the former aims at fault detection while the latter accounts for the expected operational profile to assess the runtime failure probability. Also, in a testing-for-assessment the software is typically kept unchanged for the whole duration of the testing session (debugging is done afterwards to not bias the assessment), although this is not always the case (in fact, the assessment can also be done during a testing-for-improvement session via models – hence with no dedicated testing-for-assessment session - as discussed in the next Section).

Apart from this latter case, testing for improvement and testing for assessment usually occur in different phases of the lifecycle, depending on the adopted development process, and require different pieces of information. In a DevOps-driven development process for Microservices, like the one we consider in this project, testing for improvement occurs: *i)* during implementation (unit tests by developers), *ii)* during the *continuous integration*, when evolving regression test suites are continuously run[1],

[1]Continuous integration is a well-known practice in Microservice-Devops context, which runs automatic integration tests to always have a working build of the system

iii) during system testing, where the software is checked for functional correctness and possibly robustness with respect to the specified system requirements. Testing for assessment is typically run by the QA team in the *acceptance testing* stage, i.e., to assess if predefined *quality gates* (which are exactly those mentioned quality requirements, e.g., minimum required reliability, performance, security) are satisfied. Acceptance testing, unlike system testing, accounts for the user requirements (besides the specified system requirements); a good acceptance testing session should, in fact, consider the way in which the user is expected to exercise the system, namely the operational profile. It aims at the quality-in-use, and for a system to be accepted it does not just need to be compliant with the specified system requirements, it must meet the user needs. The main project's aim is to support quality assessment – techniques presented in Chapter 3. Though, we have also developed techniques for improvement to be used in the system testing stage for checking functional correctness and robustness, which will be presented in Chapter 4.

1.2 THE TESTER'S CHALLENGE

When testing for assessment, there are several challenges for testers. The most important issue is the following: assumed that not all faults will be detected in reasonable testing time, the problem for testing practitioners is to identify which are those failure-causing inputs that are more likely to

impact the user experience in operation. Failure-causing inputs are, clearly, unknown upfront, thus the challenge for testers is to select those that have the largest impact on the expected failure probability in operation, so as to get the best result with few tests.[2] This goal is challenged by the following issues:

- *Efficiency and representativeness*. "Getting the best result with few tests" is of paramount importance for feasibility and scalability of testing: with the huge input space of today's large software systems, it becomes harder and harder to find the "best" failure-causing inputs in reasonable testing time. In critical systems, it may be very hard to expose the few failures, and a lot of tests might be required.

  It is worth to note that the goal for a tester should not merely be to make the system fail (which can be achieved, for instance, by robustness testing), but should be to spot failure-causing inputs most frequently occurring in operation Cotroneo et al. (2013), which is much trickier – that is what *representativeness* refers to.

  This challenge involves not only the efficient test generation/selection phase, but also the planning phase when testing resources need to be allocated to different parts (e.g., components, modules, or input partitions) of a system.

[2] Note that *large impact* is not (or at least not only) meant in terms of *severity* of the caused failure, but is in terms of frequency of occurrence of the failure caused. Severity can be a further variable to consider if relevant.

- *Uncertainty*. The way in which the software will be used is generally not known (at least not exactly) at testing time – technically speaking, the *operational* profile is unknown. Generating representative tests is therefore difficult. Luckily, this problem – historically a big hurdle for techniques based on the operational profile estimates - can be smoothed today by the agile development practices: in fact, with the need of continuous feedback, data from the filed are readily available and much more is known about how the software is being used and about its failures – a rich source of information that testers should exploit to drive to improve efficiency and accuracy of testing.

  This is the situation we encounter in Microservice-DevOps development, where continuous monitoring is a key principle and turns out to be very useful to drive testing.

1.3 THE MAIN STRATEGIES

Acting without knowing the effect of taken actions is a further source of uncertainty. Testers need to know with reasonable accuracy what quality level their software achieved. Does it need more testing? Are there parts that need more testing than others? Which part is contributing more to quality? Is the product ready for release? Can I demonstrate, with a certain level of confidence, that the product is ready? Quality assessment is paramount for these questions. Depending on the quality attribute, there

are several techniques for assessment, roughly divided in model-based, measurements-based, hybrid. Some of these are mentioned in Deliverable D2.1 of the project. If we look at *reliability*, one for which the literature is abundant, the assessment can be achieved *i) during* testing or *ii) by* testing:

Software reliability growth models (SRGM).

SRGM observe the fault detection and correction process occurring *during* testing (i.e., they do not influence testing, but just use observed data) to fit a model projecting the expected improve reliability as result of testing and debugging. With SRGM, failure data observed during testing and debugging are used to build (parametric and non-parametric) models predicting the next time to failure, thus failure intensity at the end of testing. In this case, detected faults are removed (i.e., code is changed), and reliability grows during testing: the goal is to figure out when debug testing can be stopped. A plenty of SRGMs exist in the literature, all trying to capture the possible fault detection patterns of a testing process (e.g., Goel (1985); Goel and Okumoto (1979); Gokhale and Trivedi (1998); Ohishi et al. (2009)). The criticisms of this approach lie in their numerous assumptions due to the difficulties in modeling the complex factors involved in a real testing and debugging process Almering et al. (2007). When used, SRGM consider data during "development" testing Musa (1996), hence during testing-for-improvement, preferably during system testing since they are considered closer to the

expected operational usage.

Sampling-based testing

In alternative to, or besides, assessing during testing, researchers proposed testing techniques to probabilistically assess reliability *by* running a dedicated testing session, without changing the code with debugging (hence with code frozen) in which the software is exercised with inputs more likely to occur at runtime (i.e., according to the operational profile). This is an acceptance testing scenario, and in the case of Microservice-DevOps process, it matches with the quality gate identification by the QA team before release.

In this case, probabilistic sampling techniques can be used, which allows for treating the failure probability estimation problem as statistical estimation problem of a parameter of interest. Although test cases can be selected by a uniform distribution (i.e., what is known as random testing), the idea to get an unbiased estimate of the failure probability in operation is to sample test cases according to the operational profile (i.e.,, by a distribution depending on the expected usage of functionalities). Many papers referred to the latter as operational testing, and it is a pillar of reliability testing (although can be used for other quality attributes too, as we claim in this project). It was adopted for certification testing in the Cleanroom methodology Mills et al. (1987), Currit et al. (1986), Cobb and

Mills (1990), Linger and Mills (1988), Poore (1990), and in the Software Reliability Engineering Test process Musa (1996). More recent work improved operational testing either in terms of adaptiveness to allocate test cases or of test selection scheme. *Adaptive testing* was proposed by Cai et *al.*, based still on operational profile but foreseeing adaptation in the assignment of test cases to input domains Cai et al. (2004), Cai et al. (2008), Cai (2002). The authors formulate testing as an adaptive control problem using controlled Markov chains, with the goal of minimizing the variance of reliability estimator. In Lv et al. (2014b), it is used along with a gradient descent method to the same aim, while in Lv et al. (2014a), it exploits confidence intervals as driving criterion to select tests adaptively. In terms of test selection, few approaches went beyond the basic simple random sampling with replacement (SRSWR) scheme. In Podgurski et al. (1999), authors propose to estimate reliability by stratified sampling. Cluster analysis is applied to execution profiles to stratify captured operational executions, and then sampling within strata is without replacement, which is known to be more efficient than the with-replacement counterpart. There is no adaptiveness to online test outcomes, though. In a PhD proposal Omri (2014), (non-adaptive) stratified sampling is still proposed, combined with symbolic execution to stratify profiles. Further approaches to assess reliability (and/or its bounds) are available that use failure data and possibly other evidence, based, for instance, on Bayesian approaches or uncertainty quantification Gashi et al. (2009); Popov (2002); Singh et al. (2001); Strigini

and Povyakalo (2013); Strigini and Wright (2014), but are outside the scope of this work, as they do not target testing strategies.

Note that sampling-based testing also suffers from the same problems of efficiency and uncertainty mentioned at the beginning of this Section. We aim to overcome them via advanced sampling techniques and proper modelling of the operational profile, as presented in the next Chapter.

2 BACKGROUND ON SURVEY SAMPLING TECHNIQUES

We advocate the use of sampling to address the *efficiency* issue, and the continuous monitoring available in a Microservice-DevOps context for the *representativenss* issue. These support the testing-for-assessment strategy described in the next Chapter.

The objective is to provide an estimate of the quality attribute of interest that is unbiased (hence, its expectation is the true value) and efficient (namely, with a minimal variance, that implies *high confidence*, or, conversely, with a small number of test cases given a minimum confidence in the estimate we want to have).

Statistical sampling methods are a natural way to cope with this problem, as their goal is to design sampling plans tailored for a population to study, and provide estimators with the mentioned properties. Specifically, while unbiasedness (and other basic properties, like consistency and sufficiency Pham (2006)) are easier to obtain, the driving principle to select

an estimator is its *efficiency* in relation to the number of observations required.

However, the literature on sampling-based software testing proposed very few attempts to go beyond the conventional random or operational testing. The latter ones have been extensively proposed in the past to assess *reliability*, meant as probability of not failing in operation Musa (1996), Currit et al. (1986); Poore (1990); Selby et al. (1987); but all are instances of simple sampling schemes that, even though provide unbiased estimates, require a large number of test cases for a desired confidence, especially when few residual faults are in the software (e.g., in critical systems).

2.1 TERMINOLOGY

This Section introduces the terminology adopted in the following. Testing a program is the process of *i)* exercising it with different test cases, selected from the set of all possible inputs according to a selection criterion, and *ii)* observing the output, comparing it with the expected one such that, if they are discordant, a failure is said to have occurred. For what said previously, a deviation from the expectation regards also non-functional quality attributes, such as reliability (e.g., value failure, crash) performance (e.g., response time is greater than specified) or security (e.g., vulnerability exploited). Inputs provoking failures are called failure-causing inputs or *failure points*. When a failure occurs, a change is made to the program to

remove what is believed to be the cause of the failure, or "fault".[1] Since there may be several possible changes able to avoid the failure, the fault related to an observed failure is not uniquely defined. We thus rely on the notion of failure, rather than that of fault, and borrow the concept of *failure region* of the input space (as in, e.g., Frankl et al. (1998), Zachariah and Rattihalli (2007)). A failure region is the set of failure points that is eliminated by a program change aimed at removing the fault. An input point t is characterized by a predicate: $z_t = 1$ if the execution leads to a failure, namely, it is a failure point; $z_t = 0$ otherwise.

An operational profile is a quantitative characterization of how a system will be used. It is built by assigning probability values to all input cases representing the probability that each will occur in operation. Thus, it can be thought as a probability distribution over the set of the input points D. We denote this distribution with P, that assigns a probability p_t to each input $t \in D$. In operational testing, assuming a perfect estimate of the operational profile, p_t is also the probability that the input t will be selected during testing. But in the real world, the profile estimate is affected by an error, and another probability distribution is actually used to select test cases. We denote this distribution with $\hat{P}$, and its probability values with $\hat{p}_t$.

[1] A vulnerability is viewed as a fault for what said.

2.2 ASSUMPTIONS

We do the following assumptions:

1. Each test case leads the software under test to failure or success; we assume we are able to determine when a test is successful or not (i.e., perfect oracle).

2. Test case runs are independent; namely, all the non-executed test cases are admissible each time. The execution of a test case is not constrained by the execution of some other test case before. This affects the way in which a "test case" is defined, since, if the assumption is not met, a set of tasks can be grouped together in a single test case, so that at the end of the test case the system goes back to the initial state Lv et al. (2014b).

3. The output of a test case is independent of the history of testing; in other words, a failing test case is always such, independently from the previously executed test cases (i.e., the failing behaviour is not masked by the execution of some previous test cases).

4. If a test case exposes a failure, the debugging action is performed without introducing new faults (*perfect debugging*) and all the failure points of the corresponding failure region are corrected, so that re-executing an input of that region does no longer cause a failure. In any case (successful or not), the test case will be no

longer repeated in the future (*sampling without replacement*).

5. The input domain D is decomposed into a set of m subdomains: $\{\mathrm{D}_1, \mathrm{D}_2, \ldots, \mathrm{D}_m\}$. The number of subdomains and the partitioning criterion are decided by the tester. In general, there are several ways in which a tester can partition the test suite, provided that test cases in a partition have some properties in common (e.g., based on functional, structural, or profile criteria). These are usually dependent on the information available to test designers and on tester's objective. The choice does not affect the proposed strategy, which just assumes the presence of subdomains, but of course different results can be obtained according to it. The effect of different partitioning on results is out of the scope of this paper and is left to future research.

For each subdomain, we define the probability of selecting a failure point from D_i as: $\varphi_i = \theta_i \sum_{t \in D_i} p_t$, where $\sum_{t \in D_i} p_t$ is the probability of selecting an input from D_i, and θ_i is the probability that an input selected from D_i is a failure point. Thus, the true value of the quality attribute of interest for a given randomly selected input is computed as:

$$Q = 1 - \Phi = 1 - \sum_{i=1}^{m} \varphi_i \tag{2.1}$$

where Φ is the operational failure probability. This is the typical derivation done with reference to reliability ($Q = R$) in the literature Cotroneo et al.

(2016); we generalize to the other quality attribute, given the notion of failure discussed above. In an entire execution with N independent demands, the estimation becomes:

$$Q_N = Q^N \tag{2.2}$$

2.3 THE ROLE OF AUXILIARY INFORMATION

The key to improve the efficiency of sampling, hence to get the estimate with a high accuracy and few tests, is to use *auxiliary variables*. In general, we do not know the value of the variable to estimate; but if know some information that we guess is correlated with the variable of interest, than we can exploit that variable in the sampling process. This is known as probability-proportional-to-size (PPS) sampling.

Also, if we know that the variable of interest can be partitioned in classes in which it is likely to have homogeneous values (e.g., equivalence classes in testing), then we can exploit this knowledge to stratify the population (i.e., the input domain) and sample from the classes (called strata). This can improve efficiency too. We have assumed in the previous Section that such partitioning is possible.

Our variable of interest is the probability of failure ϕ, that is the sum of φ_i over all partitions. So, we can exploit all what we know about this variable. Since we are considering the acceptance testing stage in Microservice-DevOps and the continuous feedback coming from the field,

we have access to a large amount of information that can be used to orient the sampling strategy. For instance:

- If the tested units corresponds to equivalence classes in partition-based testing, the partitioning criterion is itself an example of belief of tester, who judges some ranges of values more prone to failure while others are deemed correct. It is constitutive of partitioning to assume that inputs within a partition have a homogeneous failing behaviour, and the partitioning criterion establishes this assignment. For instance, boundary values are usually expected to fail more often than in-range values. A similar concept applies for defining the "choices" within categories in category-partition testing Ostrand and Balcer (1988). The idea is to exploit such a belief not only for fault detection during development-time testing, but also for quality assessment during acceptance testing.

- If the tested units corresponding to D_i are components in a component-based system, then the observed *failure data* during the previous phases of testing, or from the field (hence from previous releases), are a source of knowledge to exploit. In particular, inter-failure times can be used to build *software reliability growth models* (SRGMs) for the components under test, or other kinds of models (e.g., machine learning models) to predict the failure of each component/module or input partition.

- When the tested units are software modules, then results of module-level testing (e.g., detected/corrected defects, level of coverage, amount of testing or, generally, V&V effort) are informative about their quality.

- Other examples of information contributing to form the tester's belief are discussed in several papers proposing Bayesian inference to formalise and quantify the belief Neil et al. (2000); Singh et al. (2001); Smidts et al. (2002), such as code characteristics (e.g. complexity metrics are often used as predictor for defect proneness by machine learning Catal and Diri (2009)), domain expert opinion, characteristics of the testing and of development process.

Sampling-based testing uses this auxiliary information combined with the operational profile expectation, whose estimate is readily available from field data in a DevOps context, in an unequal probability sampling design to select tests most impacting the failure probability. The *sampling design* establishes which (combination of) sampling techniques, within the family of probabilistic sampling, is better to use for the particular input domain of interest. Thus, the specific algorithm will depend on: *i)* the input space (inputs are modelled as 0/1 values, denoting correct/failing inputs, respectively), and on *ii)* the information available about failure proneness and profile. For instance, as for the input space: if the input domain can be easily split in homogeneous subdomains (i.e., with low intra-group

variance) and so that the variance between subdomains (i.e., inter-group variance) is high, then *stratification with unequal sampling probability of strata* and *with replacement* (to allow multiple tests for each subdomain) is a good sampling strategy. Instead, if stratification is not advisable, *unequal probability sampling* of single inputs is preferred Lohr (2009). In such a case, *without-replacement selection* is better, even though its mathematical treatment is more complex, because it is known to be more efficient than with-replacement schemes. Generally, *unequal probability sampling* is the required underlying framework in all the cases, as it allows having selection probabilities deviating from the operational profile (hence, integrating any testing profile in the sampling strategy) while preserving unbiasedness and improving efficiency.

In this project, we have defined several sources of information about quality attributes that can be used. Figure 4.1 reports the information we defined in WP2. This regards usage (i.e., profile) and failing behaviour as mentioned, but also architectural and behavioural models that can help identifying the dependencies between the modules and tell which module requires more testing.

As explained in Deliverable D2.2, the learning engine takes data gathered from monitoring and a specification of the decision (i.e., the SQA objective) to pursue. Based on this, the proper learning algorithm is used, with associated pre-processing steps when needed, and gives, as output, the prediction supporting that decision.

2.3. THE ROLE OF AUXILIARY INFORMATION

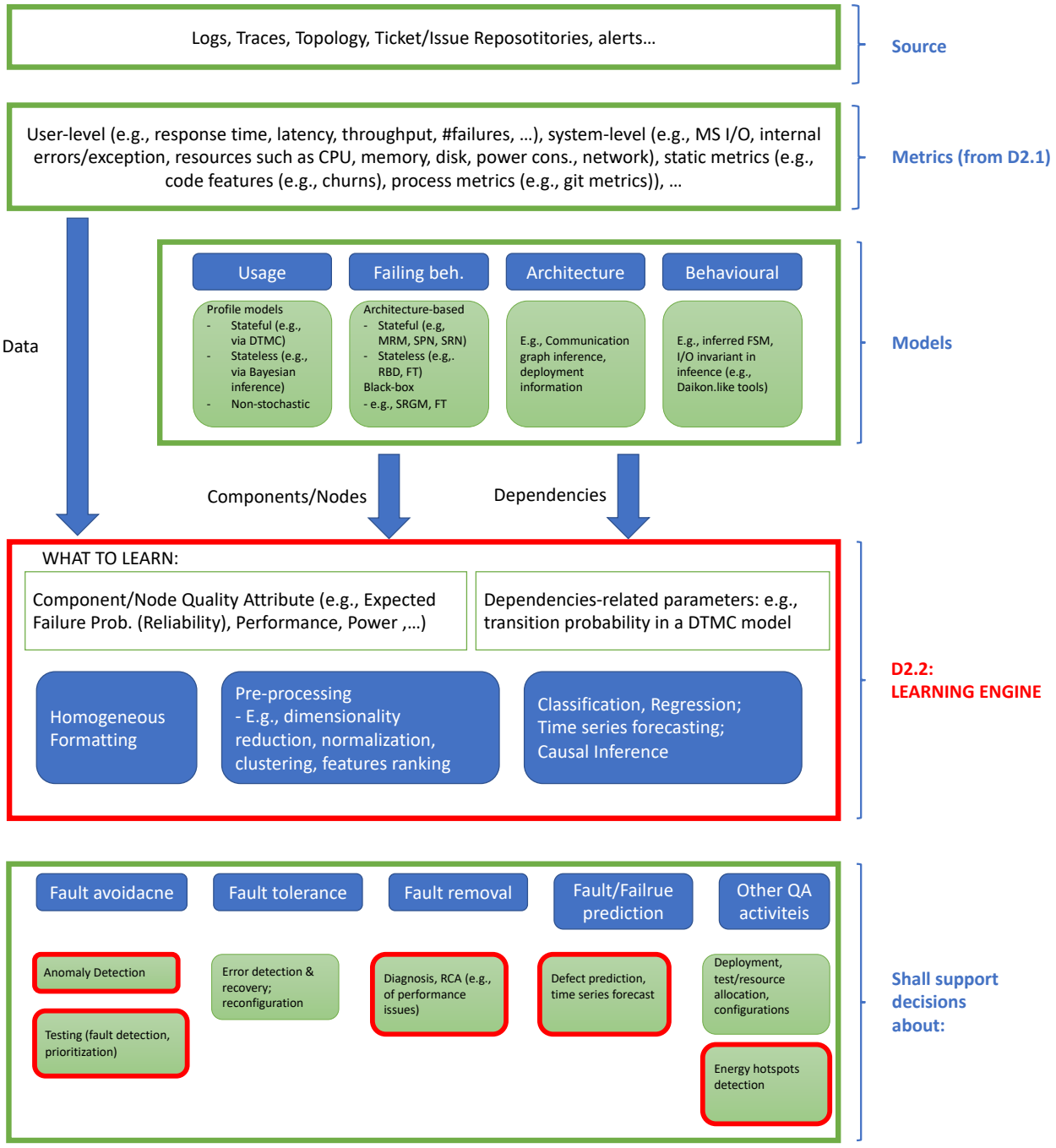

Figure 2.1. Context of use for the learning engine.

The last box shows that such knowledge can support several decisions (not only testing), although in this WP3 we care about testing. Some of these decisions, such as defect prediction, performance/energy bottlenecks detection, root cause analysis, will indirectly support testing: e.g., predicting a module as more defective or diagnosing it as more frequent failure root cause, suggests more testing for that module.

In this WP we focused the attention on: *i)* operational profile information, *ii)* observed failure data about incorrect output, long response times, security issues. These are learnt through ML models and used by the sampling schemes we define in the next Section. The same algorithms can be used with any quality-assessment task and with any variable of interest (e.g., static metrics for defect prediction).

A final note about the objective of testing: so far we have described everything as a test generation problem, in which we have to select inputs from the whole input domain and use them as test cases. In the project, we also face another problem, namely how to select or prioritize tests from a set of existing test cases (i.e., a test suite) in order to augment the fault detection (which is a different goal than quality-assessment testing, it is for quality-improvement). This is a situation typical of DevOps cycles, where regression testing is a pillar. Test selection & prioritization for quality improvement are better dealt with machine learning models, as they are not amenable to be formulated as sampling (there is no quantity to estimate). We used learning-to-rank strategies for this, as explained in D2.2, for correctness issue. This is

particularly important for security testing. In fact, in that case, tests are sets of attacks that try to exploit vulnerabilities, and often come with pre-defined attempts to exploit the vulnerability. In such a case, the problem resembles more tests selection & prioritization than tests generation – namely, which vulnerability is worth to investigate first, which type of attack is worth to be launched first.

3 ACCEPTANCE TESTING AS SAMPLING

3.1 OVERVIEW

This Section reports the sampling-based testing algorithms we use for the acceptance testing stage (e.g., to check for the *quality gates*) in a Microservice-DevOps context. These algorithms use the auxiliary information defined in Section 2.1 to assess a quality attribute of interest. We first present the algorithms we developed. Then, examples of application for reliability and performance assessment of Microservice applications are reported.

The objective of sampling-based testing is to provide an unbiased estimate of quality attribute of interest Q , denoted as $\hat{Q}$. A "good" estimator is sought, namely an estimator that is *unbiased* and *efficient* (i.e., with *variance as low as possible* given T tests to run).

Assume a system can be represented, without loss of generality, as a set of modules indicated with D_i interacting to each other. In the context

of this project, the modules are microservices in a microservice architecture (MSA). In general, they can be architectural components, or partitions of the input domain.

The two main stages required for testing are:

- **Test cases allocation**, where the number of tests for each service are decided. This could be done by several methods, such as: giving more tests to bigger services; giving more tests to services judged as more critical (by domain experts); using historical data or design information about the expected defectiveness to spot critical services (e.g., via defect prediction) or to allocate tests by optimization models (e.g., Huang et al. (2002)Pietrantuono et al. (2010)). Whatever the initial allocation is, it can be then adjusted with time, especially in a DevOps context where continuouos feedback can be helpful. Several strategies use the feedback over iterations to change the allocation of tests to each module for the next iteration Cotroneo et al. (2016); Pietrantuono et al. (2020a); Pietrantuono and Russo (2016). The output of the allocation stage is the assignment of a number of test cases to run to each service D_i, denoted as T_i.

- The second stage is about **input selection**, where the algorithm(s) derive the test inputs by selecting an input from domain D_i of the microservice i under test. These will form the T_i test cases

from domain D_i. In the following, several *selection techniques* are presented, whose applicability are a trade-off between the knowledge that a tester could exploit to improve the input selection, the technique performance, and its implementation complexity. Note that the term "selection" refers to the input space; from the testing point of view, this selection is a "generation" of test cases.[1]

All the techniques select test cases based on a more or less in-depth knowledge of the operational profile. A profile P is defined as a probability distribution where each input t has an expected occurrence probability p_t. With respect to knowledge of P, the techniques will generally consider each input either singularly or grouped by classes with similar characteristics (e.g., all inputs of a functionality, inputs of an equivalence class, etc.). To take the more general case, we consider an occurrence probability p_t assigned to each input $t \in D$. Thus, if no information is available at all about expected occurrence of inputs, we have $p_t = 1/|D|$ (i.e., same probability to all inputs). If a tester has information at (micro)service level, a p_i value is assigned to the entire domain D_i assuming the within-domain distribution being uniform with $p_t = p_i/|D_i|$. If a tester distinguishes between classes of inputs within D_i, then different p_j values are given to each class (and uniform distribution within the class). Knowledge of the profile is initially assumed to be exact, like in most related literature Cai et al. (2004), Cai et al. (2008), Lv et al. (2014b), Cai (2002), but it is progressively updated with

[1]Test selection is a different problem, where tests from an existing test suite need to be selected

observations coming from the monitoring, hence the assumption quickly becomes non-impacting thanks to the availability of field observations in the Microservice-DevOps context. The profile estimation updated is done by a strategy we devloepd based discussed in the next Section. It could also be done by ML models.

To recall the above notation: for each domain D_i, we have: $\varphi_i = \theta_i \sum_{t \in D_i} p_t$, where $\sum_{t \in D_i} p_t$ is the probability of selecting an input from D_i, and θ_i is the probability that an input selected from D_i is a failure point. The quality attribute (e.g., reliability) for a randomly selected demand is: $Q = 1 - \Phi = 1 - \sum_{i=1}^{m} \varphi_i$ where Φ is the operational failure probability. After N demands, it becomes: $Q_N = Q^N$. The estimate of Q is computed from domain-level estimates:

$$\hat{Q} = 1 - \sum_{i=1}^{m} \hat{\varphi}_i = 1 - \sum_{i=1}^{m} p_i \cdot \hat{\theta}_i \tag{3.1}$$

where $p_i = \sum_{t \in D_i} p_t$, while $\hat{\theta}_i$ is the sought estimate of the probability that an input selected from D_i is a failure point. In the following, we can therefore refer to the estimation of $\hat{\theta}_i$ values. The variance of the estimator, which is of interest to evaluate its efficiency, being the $\hat{\theta}_i$ values independent of each other, is:

$$V(\hat{Q}) = V(\hat{\Phi}) = \sum_{i=1}^{m} p_i^2 V(\hat{\theta}_i)$$

(3.2)

3.2 ADAPTIVE ALLOCATION OF TEST CASES

Adaptiveness aims at periodically re-allocating tests to improve the estimate efficiency in terms of variance. It iteratively assigns a subset of total test cases (T) available at every iteration (e.g., at every release) to domains, giving more tests to domains (e.g., to microservices or microservices' equivalence classes) with a bigger expected variance. At iteration $k = 0$, a subset $T^{(0)}$ of tests is distributed to the domains. As mentioned, we have used several alternatives for this (e.g., Pietrantuono et al. (2020a), Cotroneo et al. (2016)); we assume the simplest one, assuming that no information is available, and perform a size-proportional allocation[2]: $T_i^{(0)} = T^{(0)} \cdot \frac{|D_i|}{|D|}$. At next iterations, test cases are distributed by weighting the number of tests ($T^{(k+1)}$) foreseen for iteration *(k+1)*: $T_i^{(k+1)} = T^{(k+1)}\omega_i^{(k)}$. In the following, we describe the method implemented to determine $\omega_i^{(k)}$ and $T^{(k+1)}$. A simple solution is to keep on allocating tests proportionally to domains size, hence $\omega_i^{(k)} = \frac{|D_i|}{|D|}$. However, as the goal is to minimize the estimate's variance, allocation needs to be proportional not only to size, but also to variance. Assuming the costs to select a test case across domains approximately equal, it can be shown that the optimal allocation scheme is the *Neyman* allocation Lohr (2009),

[2]In adaptive allocation, the number of samples at the first iteration ($T^{(0)}$) is only required to be *much* smaller than T Sridharan and Namin (2010), in order to start the algorithm

where weights are proportional to size and standard deviation:

$$T_i^{(k+1)} = T^{(k+1)} \cdot \omega_i^{(k)} = T^{(k)} \frac{|D_i|\sqrt{V(\theta_i)^{(k)}} \cdot p_i}{\sum_{j=1}^{m} |D_j|\sqrt{V(\theta_j)^{(k)}} \cdot p_j} \tag{3.3}$$

However, the true within-domain variances of θ_i are unknown. Thus, at each iteration, the estimates of $V(\theta_i)$ have to be provided by the test selection scheme adopted at domain-level (discussed in the next Section). To implement a robust adaptation with respect to fluctuations of such variance estimates, we do not directly use Equation 3.3, but an *adaptive importance sampling* (AIS) algorithm.

Importance sampling aims at approximating the *true* distribution of a variable of interest Fox (2003). Our true unknown distribution is the best number of test cases for each domain that minimizes the variance of reliability estimator. The algorithm represents the beliefs (i.e., hypotheses) about this distribution by means of sets of "samples". Each sample is associated with a probability that the belief is true: at each iteration, these probabilities are updated by examining some new samples of that hypothesis, and a larger number of samples (i.e., test cases) are drawn from hypotheses with a larger probability. The goal is to converge, in few iterations, to the "true" best distribution of test cases.

To establish how the probability of each hypothesis is updated based on new collected samples, an update rule is defined. Let us denote with $\pi^{(k)}$ the probability vector representing, for each domain, the likelihoods that *testing from that domain contributes to minimizing the variance of the*

estimator. This information is well represented by weights $\omega_i^{(k)}$. Using the variance estimates of θ_i in lieu of true (unknown) variances in Equation 3.3, the update rule of the probability vector $\pi^{(k)}$ is defined as follows:

$$\pi_i^{(k)} = \gamma\pi_i^{(k-1)} + (1-\gamma)\cdot(1-\pi_i^{(k-1)})\cdot\hat{\omega}_i^{(k)} \tag{3.4}$$

The rule tends to assign progressively more tests to domains with higher variance of the estimator, so as to diminish its impact on the overall variance. Given the same weights $\omega_i^{(k)}$, the increase is larger for domains that had fewer resources at the previous iteration. The smoothness of adaptiveness is further is determined by $\gamma \in [0,1]$, regulating how the algorithm considers past iterations' results with respect to current ones. The $\pi_i^{(k)}$ values are normalized, since they are probabilities ($\pi_i^{(k)} = (\pi_i^{(k)})/(\sum_{i\in D}\pi_i^{(k)})$). Starting from $\pi_i^{(k)}$, the bucket-filling procedure reported in **?** is used to distribute the tests to domains, so as $T_i^{(k+1)} \approx T_i^{(k)}\pi_i^{(k)} \propto T_i^{(k)}\hat{\omega}_i^{(k)}$.

To determine the proper $T_i^{(k)}$ at each iteration, we consider the *adaptive* implementation of importance sampling Fox (2003). Based on a desired error and confidence, this variant tends to progressively reduce the number of required samples as more information becomes available, so as to approximate the sought distribution earlier. Accordingly:

$$T^{(k+1)} = \frac{1}{2\xi}\chi^2_{\rho-1,1-\delta} \approx \frac{\rho-1}{2\xi}\{1-\frac{2}{9(\rho-1)}+\sqrt{\frac{2}{9(\rho-1)}}z_{1-\delta}\}^3 \tag{3.5}$$

where: ξ is the error that we want to tolerate between the sampling-based

estimate and the true distribution; $1 - \delta$ is the confidence we want in this approximation; ρ is the number of domains from which at least one test case has been drawn in the k-th iteration; $z_{1-\delta}$ is the normal distribution evaluated with significance level δ.

The resulting number of $T_i^{(k+1)}$ test cases are run within each domain, e.g., for each microservice, according to one of the techniques described in the next Section: test results are in turn used to estimate the variances $V(\theta_i)$, hence allowing to update $\hat{\omega}_i$ (and π_i) based on the new information.

3.3 SELECTION OF TEST CASES

We describe test selection techniques within domain D_i of the *i*-th Microservice by providing formulas to compute the failure rate estimator $\hat{\theta}_i$ (needed in Equation 3.1), its variance $V(\hat{\theta}_i)$, and a correct estimator of such variance $\hat{V}(\hat{\theta}_i)$ (needed in Equation 3.1 as well as in Equation 3.3 in lieu of the unknown $V(\hat{\theta}_i)$). The algorithms are based on our previous work **?**. The following description starts with the simpler case where simple random sampling is exploited to select tests, and then proceeds by refining the sampling scheme to better exploit available information for efficiency improvement. Hence, the below techniques require increasing pieces of information about the program under test, and this is a possible additional criterion to choose between them, besides efficiency and bias. All the steps described in the following refer to a given iteration k; we omit the

superscript k in all the Equations for readability of formulas (e.g., T_i^k is T_i). Also, we denote: $|D_i| = N_i$.

3.3.1 SRSWR-based testing

This first technique makes no assumption about *(i)* which input or class of inputs (e.g., equivalence class) is more prone to fail within a domain D_i; *ii)* what is the expected operational usage of (class of) inputs/functionalities. Tester just has information at entity level, namely, p_i value is assigned to the entire domain D_i assuming the within-domain distribution being uniform, i.e., for each input t: $p_t = p_i/N_i$. The simplest form, which is the common one in the existing literature (e.g., Cai et al. (2008), Lv et al. (2014a), Cai et al. (2004), Cai (2002), Lv et al. (2014b)), allows the same input t to be selected more times, i.e., a *simple random sampling with replacement* (SRSWR) scheme. Test outcomes are a series of independent Bernoulli random variables $z_{i,t}$ such that $z_{i,t} = 1$ if the execution leads to a failure, $z_{i,t} = 0$ otherwise. Probability that $z_{i,t} = 1$ corresponds to proportion: $\theta_i = \frac{\sum_{t=1}^{N_i} z_{i,t}}{N_i}$. An unbiased estimator of θ_i is the observed proportion of failure points over the number of trials T_i:

$$\hat{\theta}_{i_{SRSWR}} = \frac{\sum_{t=1}^{T_i} z_{i,t}}{T_i}. \tag{3.6}$$

Accordingly, having assumed independent variables, the variance of

the θ estimator is:

$$V(\hat{\theta}_{i_{SRSWR}}) = \frac{\theta_i(1-\theta_i)}{T_i} \tag{3.7}$$

being the numerator of Eq. 3.6 a binomial random variable. An unbiased estimator of $V(\hat{\theta}_{i_{SRSWR}})$ (i.e., such that $E[\hat{V}] = V$) is:

$$\hat{V}(\hat{\theta}_{i_{SRSWR}}) = \frac{\hat{\theta}_i(1-\hat{\theta}_i)}{T_i - 1} \tag{3.8}$$

using the Bessel-corrected version as unbiased estimator of a sample variance V: $\hat{V} = \frac{n}{(n-1)}V$ (n being the sample size). Although SRSWR keeps the mathematical treatment relatively simple, it is unable to exploit additional information a tester might have. The following techniques improve the efficiency in terms of variance.

3.3.2 SRSWOR-based testing

This technique still makes no assumption about knowing failure proneness of (classes of) inputs/functionalities or their operational profile. Differently from the previous one, this technique uses a sampling *without replacement* (SRSWOR), namely, the same test case is not selected twice. This technique is expected to be more efficient in terms of estimator's variance, as it avoids sampling an input twice. The proportion estimator is still obtained as ratio of observed failure points over tests executed:

$$\hat{\theta}_{i_{SRSWOR}} = \frac{\sum_{t=1}^{T_i} z_{i,t}}{T_i} = p_i \cdot \hat{\theta}_i \tag{3.9}$$

Variance of the estimator, $\hat{\theta}$, is different. Being a without-replacement scheme, the population units from which to sample are less and less. Thus,

observations are not really independent. At the first draw, a test case t of T_i tests to run is drawn out of N_i units; at the second draw, another test case from the remaining $T_i - 1$ is drawn from a population of $N_i - 1$ units, and so on. Defining a random variable $\pi_t = 1$ if unit i is in the sample, $\pi_t = 0$ otherwise, $\hat{\theta}_i$ can be expressed as $\sum_{t=1}^{T_i} \pi_t \frac{z_{i,t}}{T_i}$. Since π_t are 0/1 variables, $E[\pi_t] = E[\pi_t^2] = T_i/N_i$, and $V(\pi_t) = E[\pi_t^2] - E[\pi_t]^2 = \frac{T_i}{N_i}(1 - \frac{T_i}{N_i})$. Moreover: $E[\pi_t \pi_{t'}] = P(\pi_{t'} = 1|\pi_t = 1)P(\pi_t = 1) = (\frac{T_i - 1}{N_i - 1})(\frac{T_i}{N_i})$ - namely, if we know that test t is in the sample, we do have a small amount of information about whether test t' is in the sample, reflected in the conditional probability $P(\pi_{t'} = 1|\pi_t = 1)$. Thus covariance is not null and: $Cov(\pi_t, \pi_{t'}) = E[\pi_t \pi_{t'}] - E[\pi_t]E[\pi_{t'}] = -\frac{1}{N_i - 1}(1 - \frac{T_i}{N_i})(\frac{T_i}{N_i})$. Given these preliminaries, and using properties of covariance:

$$V(\hat{\theta}_{i_{SRSWOR}}) = \frac{1}{T_i^2} V(\sum_{t=1}^{N_i} \pi_t z_{i,t}) =$$

$$\frac{1}{T_i^2} \sum_{t=1}^{N_i} \sum_{t'=1}^{N_i} z_{i,t}, z_{i,t'} Cov(\pi_t \pi_{t'})) = \tag{3.10}$$

$$\frac{1}{T_i^2} [\sum_{t=1}^{N_i} z_{i,t}^2 V(\pi_t) + \sum_{t=1}^{N_i} \sum_{t' \neq t}^{N_i} z_{i,t}, z_{i,t'} Cov(\pi_t \pi_{t'})]$$

Using variance and covariance of π_t, $\pi_{t'}$ and taking out of the summation:

$$V(\hat{\theta}_{i_{SRSWOR}}) =$$

$$\frac{1}{T_i^2} \frac{T_i}{N_i} (1 - \frac{T_i}{N_i}) [\sum_{t=1}^{N_i} z_{i,t}^2 - \frac{1}{N_i - 1} \sum_{t=1}^{N_i} \sum_{t' \neq t}^{N_i} z_{i,t}, z_{i,t'}] = \tag{3.11}$$

$$\frac{1}{T_i} \frac{T_i}{N_i} (1 - \frac{T_i}{N_i}) (\frac{1}{N_i (N_i - 1)}) [N_i \sum_{t=1}^{N_i} z_{i,t}^2 - (\sum_{t=1}^{N_i} z_{i,t})^2] =$$

$$\frac{N_i - T_i}{N_i} \frac{N_i}{N_i - 1} \frac{\theta_i (1 - \theta_i)}{T_i} = \frac{N_i - T_i}{N_i - 1} \frac{\theta_i (1 - \theta_i)}{T_i}$$

Hence, with respect to the SRSWR case, variance is modified by adding what is called the *finite population correction* factor $\frac{(N_i - T_i)}{N_i}$, accounting for the fact that the population is finite, and using the $\frac{N_i}{N_i - 1}$ factor to make it unbiased.

An unbiased estimator of $V(\hat{\theta}_{i_{SRSWOR}})$ is:

$$\hat{V}(\hat{\theta}_{i_{SRSWOR}}) = \frac{N_i - T_i}{N_i} \frac{\hat{\theta}_i(1 - \hat{\theta}_i)}{T_i - 1} \tag{3.12}$$

since:

$$E[\tfrac{N_i - T_i}{N_i} \tfrac{\hat{\theta}_i(1-\hat{\theta}_i)}{T_i - 1}] = \tfrac{N_i - T_i}{N_i} E[\tfrac{\hat{\theta}_i(1-\hat{\theta}_i)}{T_i - 1} \tfrac{T_i}{T_i}] = \tfrac{N_i - T_i}{N_i} \tfrac{\theta_i(1-\theta_i)N_i}{N_i - 1} \tfrac{1}{T_i} = \tfrac{N_i - T_i}{N_i - 1} \tfrac{\theta_i(1-\theta_i)}{T_i} \tag{3.13}$$

using the fact that $\frac{\hat{\theta}_i(1-\hat{\theta}_i)T_i}{T_i - 1}$ unbiasedly estimates $\frac{\theta_i(1-\theta_i)N_i}{N_i - 1}$.

Assuming $T_i \geq 1$, SRSWOR is expected to be more efficient than SRSWR, since its variance is expected to be smaller:

$$\frac{V(\hat{\theta}_{i_{SRSWR}})}{V(\hat{\theta}_{i_{SRSWOR}})} = \frac{N_i - 1}{N_i - T_i} \geq 1 \tag{3.14}$$

Since both SRSWR- and SRSWOR-based testing make the same assumptions about the knowledge available to tester, the latter is preferred: we use SRSWOR in the following for efficiency comparison, neglecting the SRSWR case.

3.3.3 Stratified SRS testing

The above two strategies can be improved if a tester has knowledge about which classes of inputs within D_i are expected to have a common behaviour,

i.e., by partitioning D_i (e.g., equivalence classes for the domain D_i of microservice i). Regardless partitioning criteria, we denote as $C_{i,h}$ the h-th class within domain i, and $N_{i,h}$ the number of elements within $C_{i,h}$.

If such information is available, stratified sampling (S-SRS) can be used to instead of SRSWOR and SRSWR. In S-SRS testing, the proportion of failure points is estimated by combining the proportions obtained in each class:

$$\hat{\theta}_{i_{S-SRS}} = \frac{1}{N_i} \sum_{h=1}^{M_i} N_{i,h} \hat{\theta}_{i,h} \tag{3.15}$$

where M_i is the number of classes and $\hat{\theta}_{i,h}$ the estimate obtained by Equation 3.9 for each class. Since the selection from classes is independent, variance of the estimator is the linear combination of within-class variances:

$$V(\hat{\theta}_{i_{S-SRS}}) = \frac{1}{N_i^2} \sum_{h=1}^{M_i} N_{i,h}^2 V(\hat{\theta}_{i,h_{SRSWOR}}) \tag{3.16}$$

Similarly, its unbiased estimator is:

$$\hat{V}(\hat{\theta}_{i_{S-SRS}}) = \frac{1}{N_i^2} \sum_{h=1}^{M_i} N_{i,h}^2 \hat{V}(\hat{\theta}_{i,h_{SRSWOR}}) \tag{3.17}$$

using Equation 3.11 and Equation 3.12 in the two cases.

A task required by S-SRS is the assignment of test cases to classes. This is the same problem we faced at domain-level, and assume, without loss of generality, the same solution here: a "proportional allocation" in the first stage (i.e., $T_{i,h} = \frac{N_{i,h}}{N_i} T_i$), and "optimal Neyman allocation" (Equation 3.3) in the next stages when an estimate of variances becomes available.

3.3.4 PPS-based testing

Besides information that allows partitioning of D_i, let us assume to have an estimate of the operational profile at class-level, along with some auxiliary indication about the failure proneness of a class with respect to the others. As discussed, the latter should be a driving principle of partitioning, wherein classes of inputs are separated with respect to their supposed failing behaviour. There are several methods to support the tester's intuition with quantitative figures about which functionality or class of inputs is more likely to fail, especially considering that assessment is done at the end of the development process, and much information is available. For instance, the amount of testing, inspection or, generally, quality assurance activities that a microservice received or the achieved code coverage suggest where a high effort was devoted to assure few residual faults; historical failure data, domain expert opinion, and other evidences can be used for such an assessment as previously discussed. These all can contribute to have a relative assessment of classes with higher expected failure rate[3]. However is assessed, we call it failure likelihood, denoted as $\vartheta \in [0, 1]$. The two techniques explained in this Section just assume a rough proportionality of the auxiliary information ϑ with the true (unknown) failure rate. Note that

[3]Failure rate of a class is meant as probability of failing given that an input is selected from that class; the actual failure probability depends, of course, not only on the faults within the class, but also on the probability of selecting an input from that class in operation, namely on the operational profile. Thus, this information is later combined with the class-level operational profile

this knowledge is just supposed to be better than knowing nothing about the relative difference among failure rates.

In such a scenario, we change the problem formulation. Let us consider the quantity to estimate being not the proportion of failure points, but the total: $\varphi_i = \sum_h p_{i,h}\theta_{i,h} = \sum_{h=1}^{M_i} \frac{p_{i,h}}{N_{i,h}} \sum_{t \in h} z_{i,t} = \sum_{t \in D_i} p_t z_{i,t}$, where $p_{i,h}$ is the probability of selecting an input from class $C_{i,h}$, and: $p_t = p_{i,h}/N_{i,h}$, because of equal selection probability within classes[4]. We define the auxiliary variable x associated with each input t such that: $x_{i,t} = p_t \vartheta_{i,h}$ where $\vartheta_{i,h}$ is the failure likelihood of the class. The corresponding probability of selection of each input point t as test case is: $\pi_t = \frac{x_{i,t}}{\sum_t x_{i,t}}$. This is called *proportional to size* (PPS) selection Lohr (2009), where the "size" is the variable x. If no knowledge about failure likelihood is available, the method still works, but the higher the correlation between x and φ_i the higher the efficiency.

Given this general scheme, selection of test cases can be done, again, with or without replacement. Since $N_{i,h}$ and $p_{i,h}$ values are known, we need to estimate the total number of failure point $Z_i = \sum_t z_{i,t}$ to get $\hat{\theta}_i$ and $\hat{\varphi}_i$. In case of with-replacement selection, the estimator is the sample mean of observed values rescaled by the inverse of their selection probability π_t, namely: $\hat{Z}_i = \frac{1}{T_i} \sum_{t=1}^{T_i} \frac{z_{i,t}}{\pi_t}$, known as the Hansen-Hurwitz estimator. Variance

[4]Note that unequal probability of selection could be seamlessly used in the method formulation, but the information on the operational profile is rarely available at such fine level of granularity.

is:

$$V(\hat{Z}_i) = E[(\hat{Z}_i - Z_i)^2] = \frac{1}{T_i}[\sum_{t=1}^{N_i} \pi_t(\frac{z_{i,t}}{\pi_t} - Z_i)^2] = \frac{1}{T_i}\left(\sum_{t=1}^{N_i} \frac{z_{i,t}^2}{\pi_t} - Z_i^2\right) \quad (3.18)$$

With respect to the simple random sampling counterpart (SRSWR), this is a generalization, since in SRSWR π_t are equal to $1/N_i$[5]. If we consider the corresponding without-replacement case (namely, PPS sampling without replacement), we expect to obtain better variance than Equation 3.18. Hence, we now consider the RHC scheme to estimate Z_i.

3.3.5 PPS-RHC technique

This uses the Rao, Hartley and Cochran (RHC) sampling for selecting tests according to PPS Rao et al. (1962). It acts as follows:

1. Given the T_i test cases to execute in D_i, divide randomly the N_i units of the population into $g = T_i$ groups, by selecting G_1 inputs with a SRSWOR for the first group, then G_2 inputs out of the remaining $(N_i - G_1)$ for the second, and so on. This will lead to g groups of size $G_1, G_2, \ldots, G_g$ with $\sum_{r=1}^{g} G_r = N_i$. The group size is arbitrary, but we select $G_1 = G_2 = \cdots = G_g = N_i/T_i$, as this minimizes the variance Rao et al. (1962).

[5]Note that, the case of proportions θ of Equation 3.7 for SRSWR is similar, since $\theta(1-\theta) = \theta - \theta^2 = \sum_t z_{i,t}/N_i - \sum_t z_{i,t}^2/N_i^2 = \sum_t z_{i,t}^2/N_i - \sum_t z_{i,t}^2/N_i^2$, since $z_{i,t} = z_{i,t}^2$ being $z_{i,t}$ a dihcotomic (0/1) variable. Since proportions are "means" of the variable $z_{i,t}$, while here we have a total, Equation 3.7 multiplied by N_i^2 yields the variance of the total's estimator $\hat{Z}_i$ that is the same as Equation 3.18 with $\pi_t = 1/N_i$

2. One test case is then drawn by taking an input t in each of these g groups independently and with a probability proportional to size – in our case, according to π_t values.

3. Denote with $\pi_{t,r}$ the probability associated with the t-th unit in the r-th group, and with $q_r = \sum_{t \in G_r} \pi_{t,r}$ the sum in the r-th group. An unbiased estimator of Z_i is:

$$\hat{Z}_i = \sum_{r=1}^{g} \frac{\pi_t z_{i,t}}{\pi_r / q_r} \tag{3.19}$$

with $z_{i,t} = 1$ if t is a failure point, 0 otherwise. The suffixes $1, 2, \ldots, r$ denote the g test cases selected from the g groups separately. This leads to: $\hat{\theta}_{i_{RHC}} = \frac{\hat{Z}_i}{N_i}$, which is the sought proportion of failure points.

The estimator is unbiased since $\mathsf{E}[\hat{Z}_i] = \mathsf{E}_1\mathsf{E}_2[\hat{Z}_i] = \mathsf{E}_1[Z_i] = Z_i$, where E_2 is the expectation for a given split and E_1 the expectation over all possible splits into T_i groups of the chosen sizes. Variance of $\hat{Z}_i$ is derived by observing that, under unbiasedness, $V(\hat{Z}_i) = E_1 V_2(\hat{Z}_i)$, where V_2 is the variance within a split:

$$V(\hat{Z}_{i_{RHC}}) = \frac{\sum_r G_r^2 - N_i}{N_i(N_i - 1)} \left(\sum_{t=1}^{N_i} \frac{z_{i,t}^2}{\pi_t} - Z_i^2 \right) \tag{3.20}$$

with $\sum_r$ denoting the sum over the $g = T_i$ groups. Its unbiased estimator is derived in Rao et al. (1962) is:

$$\hat{V}(\hat{Z}_{i_{RHC}}) = \frac{\sum_r G_r^2 - N_i}{N_i^2 - \sum_r G_r^2} \left(\sum_{r=1}^{g} q_r \left(\frac{z_{i,r}}{\pi_r} - \hat{Z}_i \right)^2 \right). \tag{3.21}$$

Choosing $G_1 = G_2 = \cdots = G_g = N_i/T_i$, we have:

$$\frac{\sum_r G_r^2 - N_i}{N_i(N_i - 1)} = \frac{T_i (N_i/T_i)^2 - N_i}{N_i(N_i - 1)} = \frac{1}{T_i} \frac{(N_i - T_i)}{(N_i - 1)} \tag{3.22}$$

Hence:

$$V(\hat{Z}_{i_{RHC}}) = \frac{1}{T_i}\frac{(N_i - T_i)}{(N_i - 1)}\left(\sum_{t=1}^{N_i}\frac{z_{i,t}^2}{\pi_t} - Z_i^2\right) \tag{3.23}$$

which clearly less than the with-replacement case in Equation 3.18. Thus the without-replacement case is better, in terms of efficiency, than the with-replacement case by a factor $\frac{(N_i - T_i)}{(N_i - 1)}$. The sought variance of $\hat{\theta}_{i_{RHC}}$ and its estimator are:

$$V(\hat{\theta}_{i_{RHC}}) = \frac{V(\hat{Z}_i)}{N_i^2} \qquad \hat{V}(\hat{\theta}_{i_{RHC}}) = \frac{\hat{V}(\hat{Z}_i)}{N_i^2} \tag{3.24}$$

Let us compare RHC against the SRSWOR case (denoted, for brevity, SRS). From Equation 3.11, writing $\theta_i = \frac{\sum_{t=1}^{N_i} z_{i,t}}{N_i}$ and recalling that $z_{i,t} = z_{i,t}^2$, being $z_{i,t}$ a 0/1 variable), we have that:

$$V(\hat{Z}_{i_{SRS}}) = N^2 V(\hat{\theta}_{i_{SRS}}) = \frac{1}{T_i}\frac{(N_i - T_i)}{(N - 1)}\left(\sum_t N_i z_{i,t}^2 - Z_i^2\right) \tag{3.25}$$

Therefore, RHC (Equation 3.23) is more efficient if this condition is verified:

$$\sum_{t=1}^{N_i}\frac{z_{i,t}^2}{\pi_t} < \sum_{t=1}^{N_i} N_i z_{i,t}^2 \tag{3.26}$$

Considering that $\pi_t = \frac{x_{i,t}}{\sum_t x_{i,t}} = \frac{x_{i,t}}{X_i} = \frac{x_{i,t}}{\bar{X}_i N_i}$, and $Z_i = \bar{Z}_i N_i$ ($\bar{X}$ and $\bar{Z}$ are the population means), the RHC variance becomes:

$$V(\hat{Z}_{i_{RHC}}) = \frac{(N_i - T_i)}{(N_i - 1)}\bar{X}_i\frac{N_i}{T_i}\sum_{t=1}^{N_i}\frac{1}{x_{i,t}}(z_{i,t} - \frac{\bar{Z}_i}{\bar{X}_i}x_{i,t})^2 \tag{3.27}$$

Expanding the expression and recalling that $Cov(X, \frac{Z^2}{X}) = E[X, \frac{Z^2}{X}] - E[\frac{Z^2}{X}]E[X]$, condition in Equation 3.26 is verified if and only if $Cov(X, \frac{Z^2}{X}) > 0$. But in PPS sampling, X is supposed to be roughly proportional to Z, thus their covariance should be at least positive. RHC turns out to be worse than

SRSWR only in the case that auxiliary information is negatively correlated with the variable to estimate, which is a worse situation than a complete absence of knowledge about more or less failure-prone classes (i.e., knowledge is even misleading). In practice, an even partial knowledge (e.g., inputs from boundary-value regions more likely to fail than others) can suffice to distinguish more failure-prone classes; without such knowledge, partition testing is not convenient from the assessment point of view.

3.3.6 Adaptive sampling technique

Network structure

With this strategy, the test case space is represented as a network where each *node* is a module D_i and *links* between nodes represent a dependency between the tester's beliefs about the failure probability of the linked services. The idea is that the failure probability assigned to a service may affect the belief about the failure probability of another service (e.g., if there is a strong similarity between the two, according to some similarity metric).

Given the failure probability $P(i)$=$\hat{f}_i$ and $P(j)$=$\hat{f}_j$ of two services, the *link* is intended to capture the joint belief that a test case from both modules 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 service i to fail conditioned on the fact that

a failure is observed in the j-th service. $P(i|j)$ depends on the distance in an inversely proportional way: the smaller the distance, the more similar the two services, 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 $d > 0$, 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 can be also updated at run time by monitoring data.

Test generation algorithm

The 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 Lohr (2009) – 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. we generate 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 Horvitz and Thompson (1952). 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, we combine 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:

$$q_{k,i} = r\frac{w_{a_{k,i}}}{w_{a_{k+}}} + (1-r)\frac{1}{N - n_{s_k}} \tag{3.28}$$

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. 3.46. 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[6] – second term of Eq. 3.46. The selected test frame is added to

[6]The scheme can be either with- or without-replacement, with few changes in the estimator Horvitz and Thompson (1952); Eq. 3.46 is the without-replacement version, the with-replacement variant replaces $\frac{1}{N-n_{s_k}}$ with $\frac{1}{N}$.

the active set. All is repeated until the testing budget is over.

3.3.7 Estimation

After testing, the estimation is carried out. Let us consider the quality attribute to estimate, for instance reliability R: $R = 1 - \sum_i x_i = 1 - \sum_i p_i f_i$, 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:

$$\Phi = \frac{1}{n}(N\hat{f}_i' + \sum_{i=2}^{n} z_i); \tag{3.29}$$

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.

3.4 ESTIAMTING THE OPERATIONAL PROFILE

In the above techniques, we have assumed to be able to estiamte the operational prifle faithfully, thanks to the availability of data in Microservice-DevOps context. In the following, we summarize the work done to define a framework for operational profile estimation from data, published in Pietrantuono et al. (2020a). The method is with reference to reliability, although can be generalized as per the above discussion on other quality attributes whose estimation depends on the operational profile.

3.4.1 Software operational profile

Consider a software service, whose inputs are requests made to the service through its API. Denote with $prob(d)$ the probability that input $d \in D$ is

submitted to software for processing [7]

Assume to have n non-overlapping partitions Cai et al. (2008), S=$\{S_1, \ldots, S_n\}$, with the corresponding domains: D=$\{D_1, \ldots, D_n\}$ and $D_i \cap D_j = \emptyset \mid i \neq j$. In this case, the operational profile is often defined in two stages:

- A probability distribution is defined on the set of partitions S, which defines the probability $prob(d_r \in D_i)$ – denoted with $\mathcal{P}_i$ - of selecting at random an input d_r from partition's domain D_i:

$$\mathcal{P}_i prob(d_r \in S_i) = \sum_{d \in S_i} prob(d) \quad (i = 1, \ldots, n). \tag{3.30}$$

- The conditional probabilities $p(d \mid d \in D_i)$ of selecting input d from within partition's domain D_i can be expressed as:

$$prob(d \mid d \in D_i) = \frac{prob(d)}{\mathcal{P}_i} \quad (i = 1, \ldots, n). \tag{3.31}$$

Note that the probabilities (3.30) and (3.31) are defined over different domains: the former over the set of partitions (we refer to it as *operational profile on partitions*, OPP); the latter over the inputs of a partition (*operational profile within partitions*). We explicitly point out that the n probabilities $\mathcal{P}_i$, summing up to 1 by their nature, have $(n\text{-}1)$ degrees of freedom, i.e., $(n\text{-}1)$ partition probabilities can be defined so that $\sum_{i=1}^{n-1} \mathcal{P}_i \leq 1$, and the last one is given by $\mathcal{P}_n = 1 - \sum_{i=1}^{n-1} \mathcal{P}_i$.

[7] Selecting an input from the input space D and submitting it to the service corresponds to issue a request to the service through its API; thus, "Input" and "request" are used as synonymous in the following.

3.4.2 Dealing with profile uncertainty

The probabilities (3.30) and (3.31) capture the *aleatory uncertainty* about the likelihood of an input being selected at random from the input space. In principle they are estimable with an arbitrary accuracy: it would suffice to observe software in operation for unlimited period of time. If this were possible, then these probabilities will be known with certainty.

Unlimited observations of software in operation cannot be afforded since D is, in general, very large. While with limited observations one might be able to estimate quite accurately the partition probabilities $\mathcal{P}_i$, precisely estimating the conditional probabilities $prob(d \mid d \in D_i)$ for every single input is infeasible. The very idea of partitioning D and having a much smaller number of partitions than that of inputs is motivated by the desire for a coarser model for the OP.

Infeasibility of estimating $prob(d \mid d \in D_i)$ can be dealt with by making additional assumptions. Finding *plausible* assumptions is difficult and instead *convenient* assumptions are often made in practice, which may be incorrect. One such assumption is that all inputs of a partition are *equally likely*. Another one is that conditional probabilities $prob(d \mid d \in D_i)$ are not affected by a change of the likelihoods of partitions. In this paper we adopt the latter assumption, i.e. that the operational profile changes only affect the probabilities of partitions.

Given a limited knowledge about the true OP (e.g., due to limited

observations of software in operation), the estimates of probabilities (1) and (2) are subject to epistemic uncertainty. We focus on epistemic uncertainty of the probabilities of partitions (OPP), which are treated as random variables with their corresponding distributions. We apply Bayesian inference to update the epistemic uncertainty about OPP and discuss practical implications.

3.4.3 Reliability modeling framework

We assume, in line with the literature Cai et al. (2008); Frankl et al. (1998); Lv et al. (2014b), that reliability is expressed as the probability of not failing on a randomly chosen input $d_r \in D$. Let $\mathcal{F}$ be a random variable (r.v.) that represents this probability. The service reliability then can be expressed via the r.v. $\mathcal{R} = 1 - \mathcal{F}$.

Let $\mathcal{F}_i$ be the r.v. representing the probability of service failure on an input d_r selected from partition's domain D_i. Assuming that the profile on partitions does not affect the likelihood of the inputs within partitions, each conditional probability $\mathcal{F}_i$ is suitably represented as a r.v. with *pdf* $f_{\mathcal{F}_i}(x)$. We assume that Beta distribution with shape parameters a_i, b_i – $Beta(a_i; b_i)$ is appropriate for $\mathcal{F}_i$, since it offers flexibility and simplifies Bayesian inference, as is detailed below. The expected value of each $\mathcal{F}_i$ with Beta distribution is Albert and Denis (2012):

$$E[\mathcal{F}_i] = a_i/(a_i + b_i). \tag{3.32}$$

Consider the case that OPP is known *with certainty*, i.e., the values $\mathcal{P}_1$=$\overline{P_1}$, ...,$\mathcal{P}_n$=$\overline{P_n}$ are known constants. In this case, the probability of failure on an input d_r selected from D according to that profile is a weighted sum of the n conditional $\mathcal{F}_i$, the weights being the known probabilities $\overline{P_1}, \ldots, \overline{P_n}$:

$$\mathcal{F} = \sum_{i=1}^{n} \overline{P_i} \cdot \mathcal{F}_i, \qquad E[\mathcal{F}] = \sum_{i=1}^{n} \overline{P}_i \cdot E[\mathcal{F}_i] \tag{3.33}$$

Let us further assume that the n conditional $\mathcal{F}_i$ are *independently distributed random variables*. This is a *plausible assumption* in those cases when an assessor is not going to change the belief (i.e., epistemic uncertainty) associated with $\mathcal{F}_i$ if (s)he sees evidence of poor/good conditional probability of failure in some of the other partitions. We observe that the product $\overline{P_i} \cdot \mathcal{F}_i$ in Equation (3.33) is itself a random variable. Denoting with $f_{\mathcal{F}_i}^{P_i}(x)$ its marginal distribution,[8] the *pdf* of $\mathcal{F}_i$ can be expressed as a convolution:

$$f_{\mathcal{F}}(x \mid \mathcal{P}_1 = \overline{P_1}, ..., \mathcal{P}_n = \overline{P_n}) = f_{\mathcal{F}_1}^{P_1}(x) \circledast \cdots \circledast f_{\mathcal{F}_n}^{P_n}(x). \tag{3.34}$$

We can now remove the assumption that the profile is known with certainty (captured by $\mathcal{P}_1 = \overline{P_1}$,..., $\mathcal{P}_n = \overline{P_n}$). Since partition probabilities are dependent, following Adams (1996) we model the epistemic uncertainty about OPP using a *multivariate distribution*, namely the Dirichlet distribution, $D(\alpha_1, ..., \alpha_n)$, with parameters $(\alpha_1, ..., \alpha_n)$ for n variates with $(n\text{-}1)$ degrees of freedom, defined by Albert and Denis (2012):

[8] This distribution can be trivially derived from $f_{\mathcal{F}_i}(x)$.

$$f_{\mathcal{P}_1,...,\mathcal{P}_n}(p_1,...,p_n) = \frac{\Gamma(A)}{\prod_{i=1}^{n}\Gamma(\alpha_i)}\left(\prod_{i=1}^{n-1} p_i^{\alpha_i-1}\right)\left(1-\sum_{i=1}^{n-1} p_i\right)^{\alpha_n-1} \tag{3.35}$$

where $A = \sum_{i=1}^{n}\alpha_i$, and $\Gamma()$ is the Gamma function.

The marginal distribution of each $\mathcal{P}_i$ variate is a Beta distribution with shape parameters (α_i, A-α_i), $Beta(\alpha_i, A\text{-}\alpha_i)$ [9]. The moments of the $\mathcal{P}_i$ variates are given by Albert and Denis (2012):

$$E[\mathcal{P}_i] = \frac{\alpha_i}{A}, \tag{3.36}$$

$$Var(\mathcal{P}_i) = \frac{\alpha_i\cdot(A-\alpha_i)}{A^2\cdot(1+A)}, \quad Covar(\mathcal{P}_i,\mathcal{P}_j) = \frac{-\alpha_i\cdot\alpha_j}{A^2\cdot(1+A)}, \ j\neq i. \tag{3.37}$$

Using the formula of the total probability, Equation (3.34) becomes:

$$\begin{aligned} f_{\mathcal{F}}(x) &= \int f(x \mid \mathcal{P}_1,...,\mathcal{P}_n) f_{\mathcal{P}_1,...,\mathcal{P}_n}(p_1,...,p_n)dp_1\ldots dp_n \\ &= \int \left[f_{\mathcal{F}_1}^{\mathcal{P}_1}(x)\cdot\ldots\cdot f_{\mathcal{F}_n}^{\mathcal{P}_n}(x)\right] f_{\mathcal{P}_1,...,\mathcal{P}_n}(p_1,...,p_n)dp_1\ldots dp_n. \end{aligned} \tag{3.38}$$

Equation (3.38) provides the *marginal distribution of the service probability of failure*, which accounts for the epistemic uncertainty related to the profile and the n conditional probabilities of failure, $\mathcal{F}_i$.

The marginal distribution of $\mathcal{F}$ given by Equation (3.38) can be used to compute various metrics of interest for the service under study. One can compute the expected value (and other moments) of the service probability

of failure, hence the expected value of the service reliability $\mathcal{R}$, which are given by:

$$E[\mathcal{F}] = \sum_{i=1}^{n} E[\mathcal{P}_i] \cdot E[\mathcal{F}_i], \qquad E[\mathcal{R}] = 1 - E[\mathcal{F}]. \tag{3.39}$$

Moreover, one can compute the risk that the true probability of failure can turn out to be *unacceptably high* (i.e. exceed a given threshold). This risk is represented by the *tail of the distribution* of the service probability of failure:

$$prob(\mathcal{F} \geq T) = \int_{T}^{1} f_{\mathcal{F}}(x)dx. \tag{3.40}$$

Another question of interest is knowing the *likelihood of surviving the next M input requests without a failure*. This can be obtained as:

$$prob(no\ failure\ in\ next\ M\ inputs) = \int_0^1 (1-x)^M \cdot f_{\mathcal{F}}(x)\, dx. \tag{3.41}$$

The above expressions may be computed from $\mathcal{F}$, which in turn depends on the data observed in operation: *i)* the number of inputs processed correctly and incorrectly in partitions – these will be used to update the uncertainty about conditional probabilities of failure in partitions, $f_{\mathcal{F}_i}(x)$; *ii)* the number of inputs selected from each partition, to update the uncertainty about the partition probabilities, captured by $D(\alpha_1, ..., \alpha_n)$.

Application scenarios

We envisage two important circumstances of interest:

- The operational profile is *fixed*. In this case the epistemic uncertainty about the OP will diminish as more and more observations come from monitoring the service in use. The distributions of the conditional probabilities of failure in partitions ($\mathcal{F}_i$), too, will become narrower and narrower as more observations are collected, and asymptotically their whole mass will be concentrated in a single point. This asymptotic case may require observations much longer than one can afford prior to deployment. Thus, we foresee the method to be useful in the initial period after deployment.

- The operational profile and/or the service itself is *subject to change* (e.g., due to new functionalities, which may affect the way the service is used). In this case, one can monitor the service behavior for possible changes of the OP on partitions and of the $\mathcal{F}_i$, and re-compute the service reliability and other metrics of interest, like those of Equations (3.40) and (3.41). The asymptotic case for the stable profile outlined above may be simply *unattainable* due to frequent changes in the case of a variable profile. Hence, one may wish to discard "old" observations, if the current profile differs significantly from the past. For such circumstances we define a procedure to capture the relevance of the observations in judging the operational profile, where recent observations are given a higher weight than those reflecting the profile in the more distant past.

Since the Microservice-DevOps context is more likely interested by this second case, we herefater report only the variable-profile case. For details about the stable profile, see our paper Pietrantuono et al. (2020a).

3.4.4 Estimation of variable profile

In case the OP changes over time, estimates of the service reliability, which are accurate yet promptly reactive to changes, can be made using a range of schemes, depending on how the history of observations in operation is taken into account when updating the Dirichlet distribution. In the case of "small" changes, accounting in the prior for the entire history might be acceptable Adams (1996). In the case of significant and rapid profile changes, discarding the possibly irrelevant history and re-starting with "ignorance" might be preferable. Cases may also be envisaged, whereby accounting in the prior only for the very recent history might be best. Within this range of schemes – from keeping all past observations in the prior to discarding them all - we propose an iterative method to chose the prior best suited for the pace of change.

We propose to run several Bayesian models in parallel, and to select for reliability predictions the model which provides *the most accurate prediction of the operational profile at the time a prediction is made*. The candidates are all Dirichlet models, using various history lengths. To this aim, we divide the history of observations into *iterations*. A candidate

model, M_h, will account for the history up to h previous iterations, with $h = 1,...,K$, K being the maximum number of past iterations to consider. With this approach, each candidate prior remains a Dirichlet distribution, allowing for analytic inference (based on the conjugate property of Dirichlet and the multinomial likelihood of the observations). Specifically:

- At iteration i, $N_{1,i},...,N_{n,i}$ requests – with $N_{1,i}+...+N_{n,i}=N_i$ - are observed for partitions $S_1, ..., S_n$, respectively, and the profile is updated.
- K Dirichlet distributions are computed, using the requests observed in the last iteration, and considering the history up to the previous $h = K$ iterations. At iteration i $(i \geq K)$, we have K models:

$$M_h = f_{P_1,...,P_n}(p_1, ..., p_n) = D(\alpha_{1,i-h} + N_{1,i}, \ldots, \alpha_{n,i-h} + N_{n,i}) \quad (3.42)$$

  where h=1 to K. The parameters $\alpha_{1,i-h},...,\alpha_{n,i-h}$ account for the cumulative number of observations per partition between iterations (i-h) and i. These models consider histories of various lengths, representing the observations more or less well depending on when and to what extent the profile changed.
- These candidate models are then pair-wise compared by means of *posterior odds* on the posterior Dirichlet distributions only. If we are indifferent between the candidate prior beliefs in the OP, using the *Bayes Factor* B is the same as using the *posterior odds*, which is equal to the likelihood ratio. For instance, with two candidate models, M_1

and M_2, for the prior of the operational profiles, the posterior odds can be expressed as:

$$posterior\ odds = \frac{P(M_1|data)}{P(M_2|data)} = \frac{P(data|M_1)}{P(data|M_2)} \cdot \frac{P(M_1)}{P(M_2)} = B \cdot [prior\ odds] \quad (3.43)$$

where *data* represents the requests $N_{1,i}$,...,$N_{n,i}$ observed in the last iteration. Given the same prior, $prior(M_1) = prior(M_2)$, hence *prior odds*$= 1$, model M_1 is preferred if the Bayes factor is greater than 1, meaning that it describes better the observed data (i.e., how the observed requests are split among partitions).

Comparing models allows addressing a well-known problem in Bayesian inference, often left to intuition, namely how long the history needs to be (i.e., how to choose K) for learning properly. Indeed, this can bring to scalability problems, but the estimate's accuracy *vs* computational cost trade-off is decided by the user depending on the needs/resources. The most expensive choice is to compare, at every iteration, all the models and take the best one. While the least expensive choice is to compare only two models and only at "relevant" iterations. A practical compromise strategy is to compare the model considering only the observations in the last iterations (i.e., without history) against a model with either *a)* all the observations from the beginning ($K = i$), or *b)* the observations up to a given number of past iterations ($K < i$) deemed to be relevant for the problem under study, or *c)* up to known/hypothesised change points of the process (for instance, if the tester has reasons to think that the operational

profile has changed – e.g., a new functionality is released - or a change is detected through other techniques).

This method has been shown to work well to follow the operational profile change. Examples and case studies of this approach can be found in our paper Pietrantuono et al. (2020a).

3.5 RELIABILITY TESTING

We have developed and applied some of the above sampling schemes for testing RESTful web services and Microservices. We hereafter report the application of an adaptive sampling algorithms for reliability-assessment testing, and of a with-replacement sampling according to the state-based operational profile for performance and reliability testing of microservices.

We first 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 favor seamless management of microservices' failures via fault tolerance means, what finally matters is the reliability of the overall MSA actually observed during operation (*operational reliability*). A microservice scarcely resilient in its operational context may have small impact on the user perception if it is rarely stimulated. Conversely, a robust yet highly invoked microservice 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 operation - need to be aware of how much reliable is the MSA in the operating environment. This would drive strategic decisions, e.g. about effort allocation to maintenance or re-engineering activities.

Traditional software reliability assessment techniques have limited applicability to MSAs. Indeed, static attempts to gauge reliability are almost useless, as the application and the usage profile change over time due to frequent releases, services' upgrades, dynamic service interactions, and to how customers use the application – a scenario we have dealt with in the previous Chapter.

The technique we present uses an adaptive sampling scheme for reliability assessment of MSA in its operational context. It acts as a run-time testing strategy, triggered upon request by a stakeholder who needs an estimate of the MSA operational reliability. 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. We use a testing algorithm based on adaptive statistical sampling, which exploits data gathered in operation to drive the test generation and accelerate the exposure of failures.

3. **Estimation**: The 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.

3.5.1 Usage scenarios

Two use cases are foreseen:

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 which can be performed in operation a value as high as the number of test frames – a test frame is a jth equivalence class within domain D_i, $C_{i,j}$. In this situation, a without-replacement sampling scheme is adopted, as in Pietrantuono et al. (2018).

In use case UC2, higher accuracy and/or stronger confidence in the

reliability estimate are required. The tester wants to achieve them even at the cost of a possibly high number of test cases. In this scenario, without-replacement sampling is not applicable, and we use a with-replacement sampling scheme.

3.5.2 The method

The steps of the testing method includes pre-release activities, to be performed once before release, and *in vivo* activities, to perform the reliability assessment in operation.

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 each edge microservice method are grouped in *equivalence classes*, $C_{i,j}$. 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 the `username` input can be grouped into five classes according to the string length (in-range, out-of-range) and content (alphanumeric, ASCII, 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).

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; the estimates $\hat{p}_i$ and $\hat{f}_i$ of the true values are used instead. 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. It then refines the estimates dynamically as more information becomes available from monitoring, using the simple probabilities update formulas described later, or the previously-explained Bayesian approach.

Graph construction. As required by the adatpvie sampling scheme presented in the previous Chapter, an graph model is needed. In this case, a graph 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 greater the distance, the bigger the chance for two demands to execute different control flow paths within the method's code.

The *weight* $w_{i,j}$ associated with the arc (i, j) captures the belief about the joint failure probability of test frames i and j. Indeed, as demands 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: $w_{i,j} = \hat{f}_i \cdot \frac{1}{d} \cdot \hat{f}_j$.

Run-time monitoring and update

Monitoring. The *in vivo* activities 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, e.g. Amazon CloudWatch

(Amazon) and Nagios (Nagios Enterprises). 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. The goal here 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 modeled 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. Instead of the Bayesian approach described earlier, we adopt here a simpler criterion, i.e., 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:

$$\hat{p}_i^u = \hat{p}_i^{u-1} \cdot [H + (1-H) \cdot (1 - \frac{R}{W})] + \hat{op}_i^u \cdot (1-H) \cdot (\frac{R}{W}) \quad (3.44)$$

$$\hat{f}_i^u = \hat{f}_i^{u-1} \cdot [H + (1-H) \cdot (1 - \frac{R}{W})] + \hat{of}_i^u \cdot (1-H) \cdot (\frac{R}{W}) \quad (3.45)$$

where:

- $\hat{p}_i^{u-1}$ is the previous occurrence probability of the i-th test frame;
- $\hat{f}_i^{u-1}$ is the previous failure probability of the i-th test frame;

- R: is the number of executed demands (less than W if the estimate is requested within a window);
- $\hat{op}_i^u$: is the occurrence probability for test frame i at the current step, estimated as the ratio between the number of failed demands to the i-th 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 i-th test frame and number of total demands to that test frame;
- H: is a value between 0 and 1, which weighs the history considered in the update (set to 50% 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.

Test generation algorithm

The test cases generation and execution phase consists of an iterative algorithm using the adaptive sampling scheme, hecne the above-defined graph. 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 to a uniform distribution. Then, one of two sampling schemes is used to select the next test frame: *weight-based sampling* (WBS) and *simple random sampling* (SRS)[9]. 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. 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 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:

$$q_{k,i} = r \cdot \frac{\sum_{j \in s_k} w_{i,j}}{\sum_{h \notin s_k, j \in s_k} w_{h,j}} + (1 - r) \cdot \frac{1}{m - n_{s_k}}, \tag{3.46}$$

with:

[9] 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.

- $q_{k,i}$: probability to select test frame i at step k;
- m: total number of test frames;
- s_k: current sample, namely the set of all test frames selected up to step k;
- n_{s_k}: size of the current sample s_k;
- $w_{i,j}$: weight of arc from node (test frame) j in the current sample s_k to node (test frame) i;
- $w_{h,j}$: weight of arc from node j in the current sample s_k to node h not in s_k;
- r: 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 Eq. 3.46 becomes:

$$q_{k,i} = r \cdot \frac{\sum_{j \in s_k} w_{i,j}}{\sum_{h=1,\dots m, j \in s_k} w_{h,j}} + (1 - r) \cdot \frac{1}{m}. \tag{3.47}$$

The first addendum in Eq. 3.46 and Eq. 3.47 accounts for the contribution proportional to the weights of the graph (WBS in Fig. 4.2), which capture the failure dependency between test frames. The second addendum in Eq. 3.46 and Eq. 3.47 accounts, 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.

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 Eq. 3.46 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 *i*-th 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 Hansen and Hurwitz (1943):

$$z_k = \frac{1}{n} \sum_{k=1}^{n} \frac{\hat{p}_i \cdot \hat{f}'_{k,i}}{q_{k,i}}, \tag{3.48}$$

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:

$$z_k = \sum_{h \in s_k} \hat{p}_i \cdot \hat{f}'_{h,i} + \frac{\hat{p}_i \cdot \hat{f}'_{k,i}}{q_{k,i}}. \tag{3.49}$$

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

$$\hat{\Phi} = \frac{1}{n}(N \cdot \hat{p}_i \cdot \hat{f}'_{1,i} + \sum_{k=2}^{n} z_k) \tag{3.50}$$

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

The overall MSA reliability is then computed as:

$$R = 1 - \hat{\Phi}. \tag{3.51}$$

3.6 PERFORMANCE AND RELIABILITY ASSESSMENT TESTING VIA OP-BASED SAMPLING

This work has been presented in a software testing conference Camilli et al. (2022a). We hereafter report the peculiar aspects.

The focus is on performance and reliability, and on their inter-relationship, considered in a DevOps context where *continuous testing* and *monitoring* represent two key practices. To assess if a release meets a desired quality, tests are performed in production, or in a staging environment with realistic users' behaviour and workload intensity . The setpes are depicted in Figure 3.1.

The main steps of the strategy are as follows:

1. definition of the operating conditions (based on the usage data collected from Ops), composed of workload intensity and behaviour of the actors using the system;

2. execution of ex-vivo testing sessions, loading the system under test (SUT) with the specified workloads;

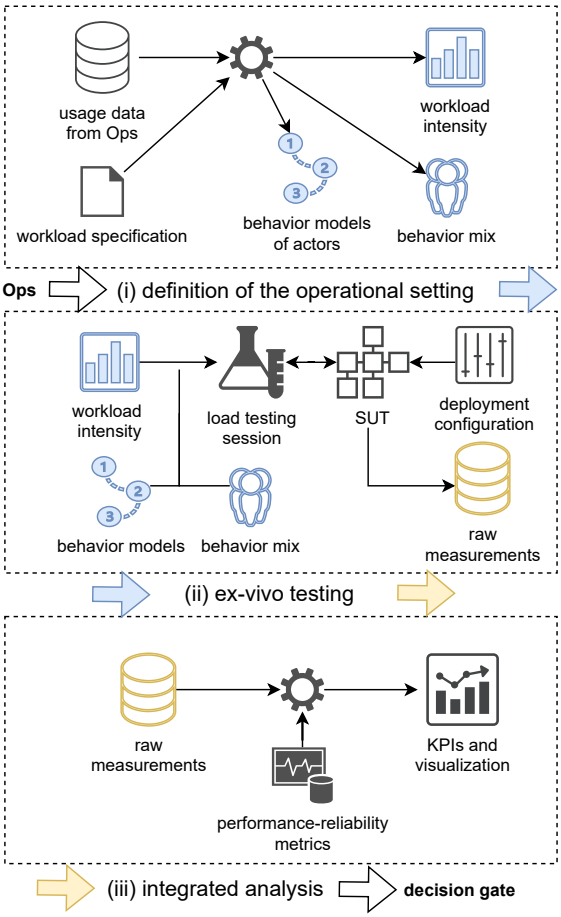

Figure 3.1. The steps of the proposed technique. From Camilli et al. (2022a)

3. integrated analysis, fed by raw measurements, to compute and visualize performance and reliability estimates.

3.6.1 Definition of the operating conditions

The first stage consists in defining the operating conditions to be reproduced for testing the system. It includes the following elements:

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

A user interacts with the system according to a behavioural model. The model is generated by combining the information extracted from the

documentation (i.e., the workload specification) and the frequency of requests issued by different actors extracted from usage data.

The technique developed foresees a behavioural model that provides a probabilistic representation of user sessions in terms of a Discrete Time Markov Chain (DTMC) Camilli et al. (2022b); Norris (1997), where the nodes represent the requests that can be issued to the system by providing either a *valid* or *invalid* input values, according to the API specification. Thus, the input space for each request is partitioned into valid and invalid classes, henceforth referred to as *request classes*. The transitions in the DTMC specify the probability of moving from a given request class to the next one.

The DTMCs are used to drive the generation of instances of synthetic users (i.e., actors) for the testing sessions. The behaviour mix defines the percentage of concurrent users to be sampled for each actor. For instance, assuming to have three different actors in a ticket reservation system, for a workload intensity of N concurrent users, the following behaviour mix:

$$(\textit{guest:}\ 0.5;\ \ \textit{buyer:}\ 0.3;\ \ \textit{refund\_claimer:}\ 0.2) \tag{3.52}$$

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

The operating conditions (behavioural models, behaviour mix, and workload intensity) are extracted automatically from usage data collected during the Ops stages of a DevOps cycle and raw sessions are automatically recorded in session logs and then analyzed to extract the workload intensity

and DTMCs using *clustering* algorithms Bertolino et al. (2020a); Vögele et al. (2018). In this case, a cluster represents an actor and is a set of sessions represented by similar DTMCs. Thus, to automatically generate the operating conditions, we first need the following data in a session log: "session identifier", "request start time", "request end time", "request relative path"and combinations of "valid" and "invalid values" for the arguments of each request. Once the DTMCs are generated, the frequency associated with DTMC is computed as frequencies of sessions in clusters over all sessions. Thus, the frequencies defines the empirical categorical distribution for workload intensity.

3.6.2 Ex-vivo testing

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

- the DTMC behavioural models of the users;
- the behaviour mix categorical distribution;
- a set Λ of workload intensity values;

- a set of deployment configurations $\mathcal{C}$ (e.g., memory, CPU, and replicas per each microservice).

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

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

3.6.3 Performance-reliability analysis

Metrics

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

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

$$\hat{P}(p) = \begin{cases} \frac{L(p)-\mu(p)}{L(p)} & \mu(p) < L(p) \\ 0 & otherwise \end{cases} \tag{3.53}$$

The lower the value, the worse is performance. It is worth noting that the parametric threshold $L(p)$ in Eq. 3.53 can be set for any class p. There are essentially two ways known in literature to set this threshold: according to a user-based experience Nielsen (1994) or a scalability requirement Avritzer et al. (2018). The former approach follows usability engineering practices for web-based applications. In this case, $L(p)$ can be set to $1\ sec$ if we want to represent the limit for the user's flow of thought to stay uninterrupted, or $10\ sec$ for keeping the user's attention focused. According to the latter approach and existing literature Avritzer et al. (2020); Camilli and Russo (2022), $L(p)$ can be empirically derived as a *scalability threshold*: $L(p) = \mu_0(p) + 3 \cdot \sigma_0(p)$, with $\mu_0(p)$ and $\sigma_0(p)$ average and standard

deviation of the response time for the request class p, measured during a testing session carried out under ideal operating conditions, like a small number of users and full availability of system resources. We further define *Performance Degradation* (PD) as $1 - \hat{P}(p)$, so that the higher its value, the worse is the performance.

We then define the *Reliability estimator*, $\hat{R}(p)$, as the ratio of non-failing requests in T, according to the *NelsonÄìAalen* non-parametric estimator Nelson (2000); Pietrantuono et al. (2020b):

$$\hat{R}(p) = 1 - \frac{F(p)}{N(p)} \tag{3.54}$$

with $N(p)$ total number of issued requests in p, and $F(p)$ number of failed requests in p, so that the lower the value, the worse is reliability. Then we define the ratio of *Failed Requests* (FR) as $1 - \hat{R}(p)$, so that higher values correspond to worse reliability. In our work, detect a failure or success of a request on the HTTP status code. Specifically, every status code other than `2xx` (success) is considered as a failed request.

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

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

out of all the failed requests in p over of all the requests of the test session.

- *Server Errors ratio* (SE): requests that return a server error out of all the failed requests in p over of all the requests of the test session.

Visualization facilities are also provided for a more comprehensive reporting.

The solution is fully automated and requires the following inputs: the RESTful API specification, the target operating conditions, and the performance threshold for each class of requests. Raw measurements are collected during each test session to compute the performance and reliability estimators as well as the additional indices per each individual class. At the end of the sessions, the tester visualizes metrics as well as plots in a interactive notebook implemented using Apache Zeppelin.

Further details about the proposal and expeirmentation are in the paper Camilli et al. (2022a).

4 SYSTEM TESTING FOR FUNCTIONAL AND ROBUSTNESS

This chapter reports our proposal for supporting the *system testing* phase, with the aim of checking the functional correctness and robustness of the MSA under test. Since we do not use information from the expected usage (i.e., the operational profile), the results of such a testing are oriented toward the development team (rather than for assessing quality-in-use attributes such as operational reliability and performance). The technique is therefore meant for he system testing phase rather than for acceptance testing.

It also integrates an approach for tests prioritization, which will be used to support security testing.

The solution being developed and its preliminary results on a case study are presented in two papers we have published in the context of the project Giamattei et al. (2022a)Giamattei et al. (2022b).

4.1 THE SPECIFICATION-BASED TESTING METHOD

The proposed solution is a black-box testing technique based on API scpeficiation of an MSA. This is preferable as MSA code is polyglot and distributed across various repositories. Automatic techniques for specification-based black-box testing of RESTful web-services can be also applied for MSA testing, as they can generate test cases from documentation of their microservices interface Arcuri (2019); Atlidakis et al. (2019); Corradini et al. (2021).[1] This practice is adopted in black-box testing of service-oriented architectures for fault detection Karlsson et al. (2020); Martin-Lopez et al. (2020), as well as to test against requirements while achieving some degree of coverage Corradini et al. (2021); Martin-Lopez et al. (2019).

However, the characteristics of real-scale MSA can make black-box testing fall short. When many microservices are involved, with complex inter-dependencies, a black box view gives no information about the internal behaviour (both in terms of achieved internal-microservices coverage and of their failing behaviour). Black-box testing exercises functionalities from an external perspective, with requests directed to *edge microservices*. The output of an edge microservice is usually dependent on the interaction with other *internal* microservices, which can be edge for other functionalities, or inaccessible from the outside. The absence of an internal perspective

[1]The most notable open format for specifying web services and MSA Application Programming Interfaces (API) is OpenAPI/Swagger Ma et al. (2018) (https://www.openapis.org).

does not allow a tester to distinguish if a failure observed on a request to a microservice is due to the microservice being faulty or to another, interacting, microservice that propagated its failure to the one under test. Also, internal microservices can be invoked by different edge microservices; if one of them is faulty, several different failures can be observed at edge level, in possibly different microservices. Testing without an internal perspective considers these as independent failures.

The solution proposed is a grey-box specification-based strategy for automatic tests generation and interactions monitoring. The strategy is supported by a tool, called MacroHive, deployed as a collection of microservices according to a *service mesh* pattern. This provides observability of internal interactions, which is crucial for microservice testing Ghani et al. (2019). The tool is applied to the *TrainTicket* benchmark Zhou et al. (2018), and turns out to perform comparably to black-box state-of-the-art techniques in edge-level coverage; it however: *i)* exposes a number of internal failures undetected by black-box testing (distinguishing propagated from masked failures), thus easing the identification of faulty microservices and of failure propagation chains; *ii)* gives details about internal dependencies, errors, and exceptions – of great importance to practitioners Waseem et al. (2021); *iii)* and requires a lower number of tests. Moreover, being itself a (set of) microservices deployed with the MSA, it does not need to run separate testing sessions for each microservice to test.

4.2 OVERVIEW

The grey-box strategy for testing an MSA, aims to expose and characterize failures[2] and to provide internal coverage information. It focuses on observability, which is important when debugging a distributed system such as an MSA Indrasiri and Siriwardena (2018). MSA are usually characterized by:

- *edge* microservices, exposing APIs to external users to access the functionality offered by the systems;
- *internal* microservices, exposing APIs to other microservices to implement complex business functions.

A microservice can be edge for some functions and internal for others. Black-box testing may not be able to allow testers to evaluate the test suite's ability to cover internal interactions. Moreover, they cannot spot when a microservice fails due its own fault or due to the failure of an internal microservice.

MacroHive generates tests starting from the microservices' API, and for every executed test observes the chain of requests among internal microservices. It supports the proposed grey-box testing strategy via

[2]In the MSA literature, a failure is considered as a request yielding a *5xx* HTTP response code, indicating an error condition, an unhandled exception, or in general the inability to serve the request Arcuri (2019); Laranjeiro et al. (2021); Martin-Lopez et al. (2020).

automated test suite generation, then execution and monitoring thanks to an infrastructure - designed according to the *service mesh pattern* Li et al. (2019) - deployed with the MSA under test.

At the end of a session, the following results concerning edge and internal microservices are provided to the tester:

- the set of executed tests with the corresponding outcome;
- the path of requests of each test through the internal microservices;
- a set of metrics at both edge and internal microservices levels (e.g., number of failures, average response time);
- a set of metrics for each level of dependency, namely the depth of a microservice in the requests chain.

With this information, the tester can discriminate different kinds of failures involving internal microservices, such as *masked failures* (corresponding to correct responses from edge microservices, despite failures of internal microservices), and *propagated failures* (incorrect responses of the edge microservices due to failures of internal microservices).

4.3 MACROHIVE

MacroHive is conceived to automatically expose both edge and internal failures, so that a tester does not need to manually inspect request paths. This functionality allows catching internal failures, undetectable by black-box strategies. It also allows identifying the true cause of edge-level failures, namely if due to the edge itself or to internal microservices. Since the testing process targets microservices of the same MSA, it is possible to detect common cause failures (e.g., a single faulty microservice that causes failures of other microservices).

Figure 4.1 shows the MacroHive infrastructure. It has three main components: *uTest*, *uSauron* and *uProxy* (uP). The first is responsible for test cases generation and execution. The other components form a support inter-service communication infrastructure Li et al. (2019) to be deployed with the SUT. An MSA is composed of many microservices with independent deployments, often controlled by multi-container management tools such as Docker Compose Gouigoux and Tamzalit (2017); Jaramillo et al. (2016). MacroHive automatically manipulates a docker-compose YAML file to add a sidecar proxy to each microservice to test/monitor.

uTest

This service generates and executes a test suite. It adopts a pairwise generation strategy that could help testers to detect multi-factor faults, which are a high percentage in software systems Hu et al. (2020). Compared

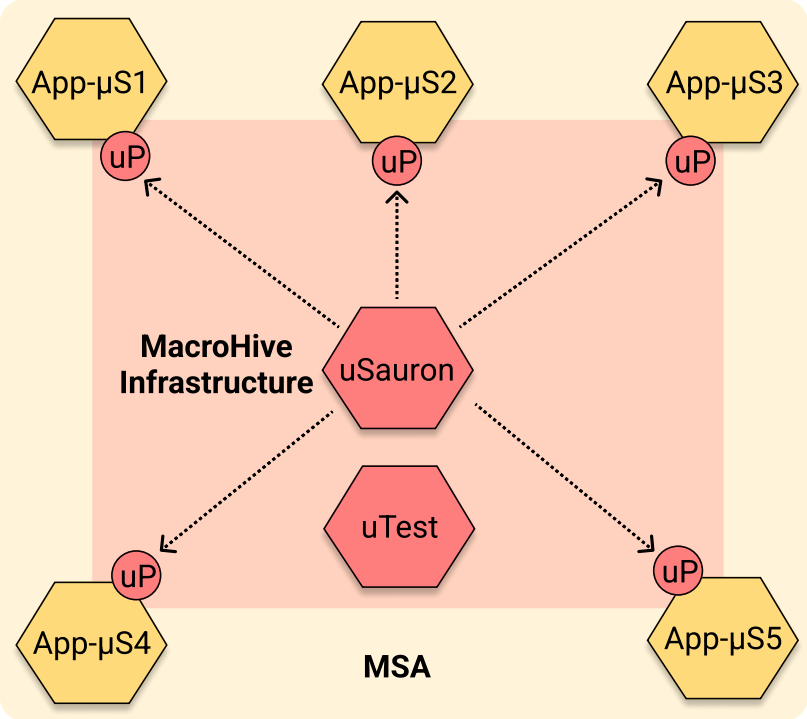

Figure 4.1. The MacroHive infrastructure

to other state of the art techniques, we expect a combinatorial design to substantially reduce testing cost, while providing good coverage and fault detection ability Cohen et al. (1996). *uTest* automatically retrieves the specification (in the OpenAPI/Swagger format) of the edge microservices of the MSA under test. The API are parsed to extract an Input Space Model consisting of HTTP methods, URIs and body templates, HTTP status codes and parameters' details (type, bounds, default value, etc.); equivalence classes Bertolino et al. (2020b) are defined for each parameter and then categorized into valid and invalid.[3]

[3]A class is valid if it contains only input parameter values which do comply to the microservice specification, and invalid if it contains only values that do not.

Table 4.1 shows an example of input space partitioning for a request with three parameters. By selecting two equivalence classes per parameter, *test case specifications* are produced with a pairwise combinatorial strategy: a *2-way* test suite is generated, covering all pairs of parameter classes. Table 4.2 shows a sample test case specification: a test case generated from this specification shall have for p_1 a value chosen from class $c_{1,2}$ (the *example* value); for p_2 a value from class $c_{2,2}$ (negative value in range), and for p_3 the value *true* or *false*.

We call *valid* test cases those containing parameter values all belonging to valid input classes; *invalid* test cases those containing at least a parameter value belonging to an invalid class. To generate a nominal test suite (composed of only *valid* test cases), only valid classes per parameter are selected (when available, examples valid and default values are preferred), otherwise *valid* and *invalid* classes per parameter are chosen to generate a mixed test suite (e.g., for robustness testing).

The generated tests are executed by sending HTTP requests. MacroHive allows generating requests also in case of authentication, by specifying credentials or tokens in the configuration file. The test outcome is automatically determined by evaluating the received HTTP status code.

uSauron and uProxy

These two components constitute a service mesh infrastructure to trace service dependencies and log request-response couples during a testing

session. Although many monitoring tools are available in the literature (e.g., Prometheus[4], Jaeger[5], etc.), we preferred to build our infrastructure in favor of automation and flexibility with minimum instrumentation.

uProxy (*uP*) is deployed alongside each microservice to test/monitor, complying with the sidecar pattern Burns and Oppenheimer (2016); Jamshidi et al. (2018). Each proxy performs two tasks:

- acting as a reverse proxy for the coupled microservice;
- sending to *uSauron* an information packet whenever it collects a request-response couple.

Different threads run these tasks to minimize communication delay. The information packet is composed of: request/response URL, request/response body, HTTP response code, response time, sender/receiver address.

uSauron is a microservice responsible for the collection of information provided by proxies. In particular, it aims to log proxies packets and compute fine-grained metrics (e.g., coverage, dependencies) for each test. For this purpose, *uSauron* runs a distributed algorithm during a testing session to link collected information to executed tests.

Test execution algorithm

The tests execution algorithm run by MacroHive (Figure 4.2) is realized by

[4] https://prometheus.io/

[5] https://www.jaegertracing.io/

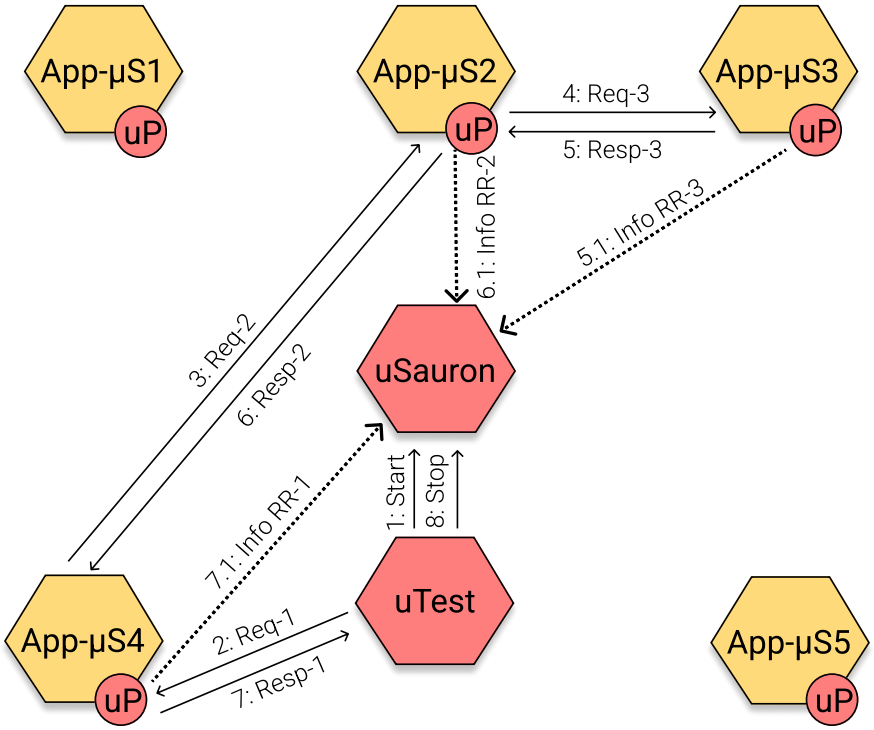

Figure 4.2. Example test execution sequence

uTest (the test executor), *uSauron* (the collector), and *uProxies* (the probes). The example in Figure 4.2 shows a test involving microservices *uS4* (edge) and *uS2*, *uS3* (internal); it entails the following messages: a *start recording* message (number 1) is sent by *uTest* to *uSauron*; it notifies the intent to run test t and that every subsequent message received by uSauron needs to be linked to t. Then, *uTest* actually starts the test t, sending an HTTP request to the *uP* proxy coupled with the edge microservice (message number 2). The involved proxies intercept the request-response couples with the edge microservice (2,7) and the internal interactions (3,6 and 4,5). For every intercepted request/response, the proxies send information packets to *uSauron* (messages 7.1, 6.1, and 5.1), which links them to test t. When *uTest*

receives the response for t (message number 7), it sends a *stop record* message to *uSauron* (message number 8). On receipt, *uSauron* stops the packets recording and saves the collected records.

This algorithm is executed for every test in a testing session. The way it is designed, the monitoring infrastructure can capture any concurrent calls of internal microservices made within the same test execution. At the end of a session, *uSauron* outputs a set of statistics.

Details about the implementation and experimentation are in the papers we published Giamattei et al. (2022a)Giamattei et al. (2022b).

Table 4.1. Example of input space partitioning

Parameter	Type	Input Classes	Category
p_1 4*(*required*, *in path*)	4*string	$c_{1,1}$: in range	*valid*
		$c_{1,2}$: specified example value(s)	*valid*
		$c_{1,3}$: empty string	*invalid*
		$c_{1,4}$: no string	*invalid*
p_2 4*(*required*, *in body*)	4*integer	$c_{2,1}$: positive value in range	*valid*
		$c_{2,2}$: negative value in range	*valid*
		$c_{2,3}$: alphanumeric string	*invalid*
		$c_{2,4}$: no value	*invalid*
p_3 4*(*optional*, *in body*)	4*boolean	$c_{3,1}$: {true,false}	*valid*
		$c_{3,2}$: no value	*valid*
		$c_{3,3}$: empty string	*invalid*
		$c_{3,4}$: alphanumeric string	*invalid*

Table 4.2. A sample test case specification

URI template	http://exampleHost:8080/examplePath/$\{c_{1,2}\}$
HTTP method	POST
body template	$\{$Äúp_2Äù:$\{c_{2,2}\}$,Äúp_3Äù:$\{c_{3,1}\}\}$
HTTP status code	201, 400

5 CONCLUSION

This document presented the work done for the definition of testing strategies aimed at quality assessment and improvement. Specifically, we have first described the main challenges for testing particularly relevant in the context of Microservice-DevOps systems. Then, we reported about the algorithms we are using to support both the acceptance testing stage, wherein an assessment of quality is required (e.g., for checking quality gates), and for the specification-based system testing stage to check for functional correctness and robustness.

The implementation of the above-mentioned techniques is expected in Deliverable D3.2. Within the project, we are also working on the integration with Artificial Intelligence (AI): how AI can support testing and quality assurance in general, and how we should test systems (possible microservices architectures) containing AI/ML components, viewed in the context of a DevOps development and deployment. Also, further quality attributes, such as energy consumption, are under investigation.

REFERENCES

Adams, T. (1996). "Total variance approach to software reliability estimation." *IEEE Trans. on Software Engineering*, 22(9), 687–688.

Albert, I. and Denis, J.-B. (2012). "Dirichlet and multinomial distributions: properties and uses in Jags." *Rapport technique 2012-5*, INRA.

Almering, V., Genuchten, M. V., Cloudt, G., and Sonnemans, P. (2007). "Using software reliability growth models in practice." *IEEE Software*, 24(6), 82–88.

Amazon. "Cloudwatch, <http://aws.amazon.com/it/cloudwatch (Last checked 27/2/2019)>.

Arcuri, A. (2019). "RESTful API Automated Test Case Generation with EvoMaster." *ACM Transactions on Software Engineering and Methodology*, 28(1).

Atlidakis, V., Godefroid, P., and Polishchuk, M. (2019). "RESTler: Stateful REST API Fuzzing." *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 748–758.

Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., and van Hoorn, A. (2018). "A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing." *Proceedings of the 12th European Conference on Software Architecture (ECSA)*, Vol. 10469 of *Lecture Notes in Computer Science*, Springer, 159–174.

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

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

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

Bertolino, A., De Angelis, G., Guerriero, A., Miranda, B., Pietrantuono, R., and Russo, S. (2020b). "DevOpRET: Continuous reliability testing in DevOps." *Journal of Software: Evolution and Process* e2298 smr.2298.

Burns, B. and Oppenheimer, D. (2016). "Design patterns for container-based distributed systems." *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, USENIX Association, <https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns>.

Cai, K.-Y. (2002). "Optimal software testing and adaptive software testing in the context of software cybernetics." *Information and Software Technology*, 44(14), 841–855.

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

REFERENCES

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

Camilli, M., Guerriero, A., Janes, A., Russo, B., and Russo, S. (2022a). "Microservices integrated performance and reliability testing." *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, AST '22, New York, NY, USA, Association for Computing Machinery, 29–39, <https://doi.org/10.1145/3524481.3527233>.

Camilli, M., Janes, A., and Russo, B. (2022b). "Automated test-based learning and verification of performance models for microservices systems." *Journal of Systems and Software*, 187, 111225.

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

Catal, C. and Diri, B. (2009). "A systematic review of software fault prediction studies." *Expert Systems with Applications*, 36(4), 7346–7354.

Cobb, R. and Mills, H. (1990). "Engineering software under statistical quality control." *IEEE Software*, 7(6), 45–54.

Cohen, D., Dalal, S., Parelius, J., and Patton, G. (1996). "The combinatorial design approach to automatic test generation." *IEEE Software*, 13(5).

Corradini, D., Zampieri, A., Pasqua, M., and Ceccato, M. (2021). "Empirical comparison of black-box test case generation tools for restful apis." *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 226–236.

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

Cotroneo, D., Pietrantuono, R., and Russo, S. (2016). "Relai testing: A technique to assess and improve software reliability." *IEEE Transactions on Software Engineering*, 42(5), 452–475.

Currit, P., Dyer, M., and Mills, H. (1986). "Certifying the reliability of software." *IEEE Trans. on Software Engineering*, SE-12(1), 3–11.

Fox, D. (2003). "Adapting the Sample Size in Particle Filters Through KLD-Sampling." *Int. Journal of Robotics Research*, 22, 2003.

Frankl, P., Hamlet, D., Littlewood, B., and Strigini, L. (1998). "Evaluating testing methods by delivered reliability." *IEEE Trans. on Software Engineering*, 24(8), 586–601.

Gashi, I., Popov, P., and Stankovic, V. (2009). "Uncertainty explicit assessment of off-the-shelf software: A bayesian approach." *Information and Software Technology*, 51(2), 497–511.

Ghani, I., Wan-Kadir, W., Mustafa, A., and Imran Babir, M. (2019). "Microservice testing approaches: A systematic literature review." *International Journal of Integrated Engineering*, 11(8), 65–80.

Giamattei, L., Guerriero, A., Pietrantuono, R., and Russo, S. (2022a). "Assessing black-box test case generation techniques for microservices." *Quality of Information and Communications Technology*, A. Vallecillo, J. Visser, and R. Pérez-Castillo, eds., Cham, Springer International Publishing, 46–60.

REFERENCES

Giamattei, L., Guerriero, A., Pietrantuono, R., and Russo, S. (2022b). "Automated grey-box testing of microservice architectures." *The 22nd IEEE Conference on Software Quality, Reliability, and Security*, IEEE.

Goel, A. L. (1985). "Software reliability models: Assumptions, limitations and applicability.." *IEEE Trans. on Software Engineering*, SE-11(12), 1411–1423.

Goel, A. L. and Okumoto, K. (1979). "Time-dependent error-detection rate model for software reliability and other performance measures." *IEEE Trans. on Reliability*, R-28(3), 206–211.

Gokhale, S. and Trivedi, K. (1998). "Log-logistic software reliability growth model." *Proc. 3rd Int. High-Assurance Systems Engineering Symposium (HASE)*, 34–41.

Gouigoux, J. and Tamzalit, D. (2017). "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture." *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, 62–65.

Hansen, M. H. and Hurwitz, W. N. (1943). "On the theory of sampling from finite populations." *The Annals of Mathematical Statistics*, 14(4), 333–362.

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

Hu, L., Wong, W., Kuhn, D., and Kacker, R. (2020). "How does combinatorial testing perform in the real world: an empirical study." *Empirical Software Engineering*, 25.

Huang, C.-Y., Lo, J.-H., Kuo, S.-Y., and Lyu, M. (2002). "Optimal allocation of testing resources for modular software systems." *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th Int. Symposium on*, 129–138.

Indrasiri, K. and Siriwardena, P. (2018). *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress, USA, 1st edition.

Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., and Tilkov, S. (2018). "Microservices: The journey so far and challenges ahead." *IEEE Software*, 35(3), 24–35.

Jaramillo, D., Nguyen, D. V., and Smart, R. (2016). "Leveraging microservices architecture by using Docker technology." *SoutheastCon 2016*, IEEE, 1–5.

Karlsson, S., Čaušević, A., and Sundmark, D. (2020). "QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs." *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, 131–141.

Laranjeiro, N., Agnelo, J., and Bernardino, J. (2021). "A Black Box Tool for Robustness Testing of REST Services." *IEEE Access*, 9.

Li, W., Lemieux, Y., Gao, J., Zhao, Z., and Han, Y. (2019). "Service mesh: Challenges, state of the art, and future research opportunities." *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, IEEE, 122–127.

Linger, R. and Mills, H. (1988). "A case study in cleanroom software engineering: the ibm cobol structuring facility." *12th Int. Computer Software and Applications Conference, COMPSAC 88*, 10–17 (Oct).

REFERENCES

Lohr, S. L. (2009). *Sampling Design and Analysis*. Duxbury Press; 2 edition.

Lv, J., Yin, B.-B., and Cai, K.-Y. (2014a). "Estimating confidence interval of software reliability with adaptive testing strategy." *Journal of Systems and Software*, 97, 192–206.

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

Ma, S., Fan, C., Chuang, Y., Lee, W., Lee, S., and Hsueh, N. (2018). "Using service dependency graph to analyze and test microservices." *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 02, IEEE, 81–86.

Martin-Lopez, A., Segura, S., and Ruiz-Cortés, A. (2019). "Test Coverage Criteria for RESTful Web APIs." *Proc. of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST)*, ACM, 15–21, <https://doi.org/10.1145/3340433.3342822>.

Martin-Lopez, A., Segura, S., and Ruiz-Cortés, A. (2020). "RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs." *Service-Oriented Computing*, E. Kafeza et al., ed., Springer, 459–475.

Mills, H., Dyer, M., and Linger, R. (1987). "Cleanroom software engineering." *IEEE Software*, 4(55), 19–24.

Musa, J. (1996). "Software reliability-engineered testing." *Computer*, 29(11), 61–68.

Nagios Enterprises. "Nagios Monitoring Solutions, <www.nagios.org (Last checked 27/2/2019)>.

Neil, M., Fenton, N., and Nielson, L. (2000). "Building Large-scale Bayesian Networks." *Knowl. Eng. Rev.*, 15(3), 257–284.

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

Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

Ohishi, K., Okamura, H., and Dohi, T. (2009). "Gompertz software reliability model: Estimation algorithm and empirical validation." *Journal of Systems and Software*, 82(3), 535–543.

Omri, F. (2014). "Weighted statistical white-box testing with proportional-optimal stratification." *19th International Doctoral Symposium on Components and Architecture*, WCOP'14, ACM, 19–24, <http://doi.acm.org/10.1145/2601328.2601333>.

Ostrand, T. J. and Balcer, M. J. (1988). "The category-partition method for specifying and generating fuctional tests." *Commun. ACM*, 31(6), 676–686.

Pham, H. (2006). *Software System Reliability*. New York, NY, USA: Springer-Verlag.

Pietrantuono, R., Popov, P., and Russo, S. (2020a). "Reliability assessment of service-based software under operational profile uncertainty." *Reliability Engineering System Safety*, 204, 107193.

REFERENCES

Pietrantuono, R. and Russo, S. (2016). "On adaptive sampling-based testing for software reliability assessment." *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 1–11.

Pietrantuono, R., Russo, S., and Guerriero, A. (2018). "Run-Time Reliability Estimation of Microservice Architectures." *29th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 25–35.

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

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

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

Poore, J. (1990). "A case study using cleanroom with box structures adl." *Report no.*, Software Engineering Technology CDRL 1880.

Popov, P. (2002). "Proc. 21st int. conference on computer safety, reliability and security." SAFECOMP, Springer, 139–150, <http://dx.doi.org/10.1007/3-540-45732-1$_1$5¿.

Rao, J., Hartley, H., and Cochran, W. (1962). "On a simple procedure of unequal probability sampling without replacement." *Journal of the Royal Statistical Society. Series B (Methodological)*, 24(2), 482–491.

Selby, R., Basili, V., and Baker, F. (1987). "Cleanroom software development: An empirical evaluation." *IEEE Trans. on Software Engineering*, SE-13(9), 1027–1037.

Singh, H., Cortellessa, V., Cukic, B., Gunel, E., and Bharadwaj, V. (2001). "A bayesian approach to reliability prediction and assessment of component based systems." *ISSRE 2001. Proceedings. 12th International Symposium on Software Reliability Engineering*, 12–21 (Nov).

Smidts, C., Cukic, B., Gunel, E., Li, M., and Singh, H. (2002). "Software reliability corroboration." *Proceedings 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, IEEE, 82–87 (Dec).

Sridharan, M. and Namin, A. (2010). "Prioritizing mutation operators based on importance sampling." *21st Int. Symposium on Software Reliability Engineering (ISSRE)*, 378–387 (Nov).

Strigini, L. and Povyakalo, A. (2013). *Computer Safety, Reliability, and Security: 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Software Fault-Freeness and Reliability Predictions, 106–117.

Strigini, L. and Wright, D. (2014). "Bounds on survival probability given mean probability of failure per demand; and the paradoxical advantages of uncertainty." *Reliability Engineering & System Safety*, 128, 66–83.

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

REFERENCES

Waseem, M., Liang, P., Shahin, M., Di Salle, A., and Márquez, G. (2021). "Design, monitoring, and testing of microservices systems: The practitioners' perspective." *Journal of Systems and Software*, 182, 111061.

Zachariah, B. and Rattihalli, R. N. (2007). "Failure size proportional models and an analysis of failure detection abilities of software testing strategies." *IEEE Trans. on Reliability*, 56(2), 246–253.

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., and Zhao, W. (2018). "Benchmarking microservice systems for software engineering research." *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE-Companion)*, ICSE '18, New York, NY, USA, ACM, 323–324.