Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

DevOpRET: Continuous Reliability Testing in DevOps

Antonia Bertolino[1] | Guglielmo De Angelis[2] | Antonio Guerriero[3] | Breno Miranda[1,4] | Roberto Pietrantuono*[3] | Stefano Russo[3]

[1]ISTI, CNR, Pisa, Italy
[2]IASI, CNR, Rome, Italy
[3]DIETI, Università degli Studi di Napoli Federico II, Napoli, Italy
[4] Federal University of Pernambuco, Recife, Brazil

Correspondence
*Roberto Pietrantuono, Email: roberto.pietrantuono@unina.it

Present Address
DIETI, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy.

Abstract

To enter the production stage, in DevOps practices candidate software releases have to pass quality gates, where they are assessed to meet established target values for key indicators of interest. We believe software reliability should be an important such indicator, as it greatly contributes to the end-user satisfaction. We propose *DevOpRET*, an approach for reliability testing as part of the acceptance testing stage in DevOps. *DevOpRET* relies on operational-profile based testing, a common reliability assessment technique. *DevOpRET* leverages usage and failure data monitored in operations to continuously refine its estimate. We evaluate accuracy and efficiency of *DevOpRET* through controlled experiments with a real-world open source platform and with a microservice architectures benchmark. The results show that *DevOpRET* provides accurate and efficient estimates of the true reliability over subsequent DevOps cycles.

KEYWORDS:
Acceptance Test; DevOps; Operational Profile; Quality Gate; Software Reliability Testing

1 | INTRODUCTION

A relatively recent trend in software production is that of blurring the boundaries between development in laboratory and operations in production [1]. This is the philosophy behind practices named DevOps [2], which in its own name signifies the seamless connection between development and operations. Despite its spread, there is no commonly agreed definition for DevOps [3,4]. Some authors describe it as a cultural shift that IT organizations should undergo to remove technical or managerial barriers between the development and operations teams, and let them collaborate under shared responsibilities [5]. Other authors focus on necessary capabilities and on cultural and technological enablers for DevOps [4]. Bass *et al.* define DevOps as "*a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality*" [2]. As quality controls at the boundary between development and operations are central in such practices, we describe DevOps as the intersection among the scopes of software Development (Dev), Operation (Ops) and Quality Assurance (QA), as depicted in Figure 1.

QA can leverage feedback from operations to drive quality controls before releasing new product versions, in a virtuous DevOps cycling. Indeed, continuous testing and monitoring are two key DevOps practices. *Continuous testing* foresees short and automated testing rounds that can provide quick quality feedback to continuous integration (CI); an acceptance test stage can check whether the current software candidate can be released – Humble and Farley state that "*without running acceptance tests in a production-like environment, we know nothing about whether the application meets the customer's specification*" [6] (p. 124). *Monitoring* consists in collecting data from the system in production, which can be used for measurement and optimization in the next testing stage. Monitoring and measurement are crucial in DevOps success [7], as DevOps adoption is ultimately motivated and driven by business objectives, which are quantified into measurable Key Performance Indicators (KPIs).

In a release cycle, acceptance testing must include the assessment of KPIs of interest and evaluate if the candidate release meets defined target values. These KPI targets constitute a *quality gate* before release. Typically, KPIs considered in DevOps include performance [8,9] and security [10,11]. We believe that software reliability too should be considered an important KPI in DevOps, as it is related to customer satisfaction and to the organization success in service delivery; it should be part of acceptance testing of a new product release at the quality gate in a DevOps cycle.

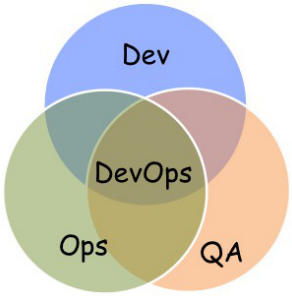


FIGURE 1 DevOps scope

Software Reliability Engineering (SRE) is a discipline pioneered thirty years ago by Musa, who defined it as the "*technology for the 1990s*" [12]. With Everett, he introduced SRE as the "*applied science of predicting, measuring and managing the reliability of software-based systems to maximize customer satisfaction*. Surprisingly, in the scientific literature on DevOps, the assessment of reliability of a new version before release seems to have received insufficient attention (as discussed in Section 2), yet in the grey literature reliability is advocated as a key user-related quality factor.

The DevOps Research and Assessment (DORA) company publishes since 2014 a periodic survey over more than 30 thousands DevOps professionals, which provides a comprehensive and up-to-date analysis of observed trends, to be used as a reference by companies. KPIs early listed by DORA included: Deployment frequency, Lead time for changes, Time to restore service, and Change failure rate. In the 2018 edition [13], for the first time the report has also included a fifth KPI, namely *Availability*, meant as "*ensuring timely and reliable access to and use of information*".

In about the same years Google introduced "Site Reliability Engineering" [14] (a term earlier coined by Google Vice President of Engineering B. Sloss, notably with the same acronym as Musa's discipline). Google SRE is conceived as an implementation of the broader DevOps principles, and practiced as a concrete set of tasks pertaining to the operations team. We find an interesting overlap between the two SREs: both state the need of establishing customer-related metrics for reliability assessment, and both identify testing as the main assessment technique. In Google's SRE "*one key responsibility of site reliability engineers is to quantify confidence in the systems they maintain*"; this task is performed "*by adapting classical software testing techniques to systems at scale. Confidence can be measured both by past reliability and future reliability. The former is captured by analyzing data provided by monitoring historic system behavior, while the latter is quantified by making predictions from data about past system behavior*" [14] (Ch. 17).

A central tool in software reliability testing is the *operational profile*, which is a quantitative characterization of how customers will use the system in production [12]. In fact, to satisfy user requirements it is important to be able to profile actual usage, so that the system can then be tested by reproducing and predicting users' experience. It is well known that deriving an operational profile before product release is hard, and the overhead costs often discourage SRE adoption in industry [15]. However, in DevOps cycles monitoring software behavior in operation is a common practice, and this provides the opportunity to *learn from history of recent executions to improve predictions about the fulfillment of quality targets by next release*.

This is what we allow DevOps testers to do by *DevOpRET*, a black-box testing approach supporting *continuous software reliability testing*. *DevOpRET* is inspired (as suggested by its name) by the traditional Software Reliability Engineered Testing (SRET) best practice [16], and a basic version of it was early presented in prior work [17]. The original contributions of this paper include:

- a refined version of *DevOpRET*: the basic version [17] uses operational testing (OT), building on a statistical sampling algorithm to generate test cases leveraging usage data from operations. The novel version, denoted as WOT, builds on a sampling algorithm based on a weighted version of the operational profile, leveraging also failure data from the Ops phase. This entails the usage of a different estimation technique to preserve unbiasedness and improve the confidence in the estimation.
- an extensive evaluation of *DevOpRET* over two case studies, namely a real-world social platform (*Discourse*), and a benchmark for microservice architectures (*TrainTicket*). The results with the first case study show the ability of the approach to provide accurate estimates of reliability as more operational data are observed. The WOT and OT variants exhibit statistically equivalent accuracy, but WOT shows a greater confidence and a greater ability in exposing failures than OT (at the cost of collecting also failure data in the Ops phase). The second case study shows the adaptivity of the reliability estimate provided by *DevOpRET* (for OT and WOT, as well) when the true reliability of the subject under test changes. While confirming that both variants get close to the true reliability over cycles as in the first case study, we could observe that OT performs better in the short term in presence of sudden changes of failure probability.

The rest of the paper is structured as follows. After a comprehensive overview of related work in the next section, Section 3 introduces the proposed *DevOpRET* approach. Section 4 presents the empirical evaluation with the two case studies, including the research questions, the evaluation metrics, and the experimental artifacts and procedure. Section 5 reports and discusses the results and the threats to validity. Finally, Section 6 draws conclusions and outlines future work.

2 | RELATED WORK

The factors driving testing in DevOps are identified by several authors, who analyzed the state-of-the-art in continuous software engineering [18,19,20,21,22]. Our research contributes to some of the key activities identified by Fitzgerald and Stol [23], namely *Continuous Testing* and *Continuous Improvement*. By using the actual operational profile as a means for data-driven planning of the future reliability testing activities, *DevOpRET* guides the QA team to solve issues that are closer to the user-significant scenarios, fostering their continuous engagement.

We believe that for a reliability testing technique to be used for quality assurance in a DevOps context, the following requirements have to be met: *i)* the technique has to consider usage-related metrics for acceptance testing; *ii)* it has to leverage data about actual failures observed in operation (possibly, automatically gathered); *iii)* it has to take into account constraints on testing time or number of test cases, for it to be used between (possibly short) release cycles. Clearly, QA teams might well leverage existing techniques, even though not explicitly conceived for DevOps processes. Although a systematic literature review within software engineering research, strictly following guidelines like those by Kitchenham *et al.* [24], is out of the scope of this paper, nevertheless we performed a comprehensive systematic search of related work, which shows that testing in the context of DevOps still appears to be under-considered in the scientific literature, thus confirming previous similar studies [21].

The search has been performed through an automated query over the main popular digital libraries (ACM Digital Library, IEEE eXplore, Scopus, Springer Link, Wiley Online Library). The search string, shown in Listing 1, relates to the three different domains covered in our work: reliability testing, continuous development practices and quality assurance. The query has been applied as a *full text* search within the body of the manuscripts in the above five libraries.

Listing 1: Literature search string over the main digital libraries

```
("reliability testing" OR "reliability estimation" OR "software reliability" OR "acceptance
    testing" OR "software testing" OR "operational testing")
AND ("DevOps" OR "dev-ops" OR "continuous integration" OR "CI")
AND ("acceptance quality" OR "acceptance quality gate" OR "product quality" OR "software
    quality" OR "quality gate" OR "operational profile")
```

The items returned by the query have been processed to remove duplicates and surveys. The remaining hundreds papers have been manually filtered to remove items out of scope, first by inspecting title and abstract, then where appropriate by examining their whole content. Papers which, based on the above three requirements, do not deal with techniques that can be used for reliability assessment at DevOps quality gates were filtered out. Among the remaining papers, we discuss here those proposing testing techniques, which we envisage are more related to our work.

Marijan [25] proposes a multi-criteria test prioritization approach for regression testing under time constraints. While not specifically thought for DevOps, the approach is conceived for continuous integration environments where teams work in short development cycles. *DevOpRET* shares several principles with it. Both are black-box approaches considering failure occurrence and failure impact, defined by Marijan as *"a user-driven measure of the severity of defects of a test case"*. However, Marijan's approach is for regression testing: it assumes test suites exist, and the problem is their *prioritization* to maximize the number of test cases detecting severe faults to be executed in a limited test time; failure data concern the frequency of failures of regression test cases in past cycles; and the failure impact *"is calculated based on historical user feedback reports collected from previous versions"*. In *DevOpRET*, test cases are *generated* based on failure occurrences actually observed in previous Ops phases; failure severity and failure exposing ability are addressed by our Weighted Operational Testing algorithm, and severity may be automatically inferred from responses to user demands; finally, time constraints are taken into account through the efficiency of the input space sampling algorithm.

An approach similar to Marijan's one [25] is proposed by Ali *et al.* [26], who present test case prioritization and selection techniques for continuous integration strategies. Like Marijan, they aim at increasing the fault detection rate by relying on the most frequently changed and failed test cases. Najaf *et al.* [27] present an experience report of using several similar test selection and prioritization approaches based on test execution history; this is performed through simulations on industrial data, where test failure results need to be manually labeled by the testers by the end of each day. In these works, test cases are engineered by developers independently of the actual demands of users and of failures in operation. *DevOpRET* takes into account usage and failures actually experienced by users in the production phase.

Révész and Pataki [28] propose a technique based on field data to drive decisions about the next release of a system. Field data sampled by means of an *A/B testing* campaign, during which users are requested to select between two versions of the considered system. Based on the users' feedback, developers decide which version to maintain. The major difference with our approach is that decisions about the testing campaign concern usability or Quality-of-Experience aspects, while in *DevOpRET* acceptance testing aims at exposing failures and it relies on failure-related operational metrics.

Mijumbi *et al.* [29] proposed a method for predicting software defects within the context of projects running continuous integration, and continuous delivery practices. Even though their objective is similar to the goal of our work, the two proposals differ mainly in that Mijumbi *et al.* do not use the actual operational profile in order to plan the testing activities for the next release; their model for early defect prediction is based on user stories together with defect data from a previous release. In our opinion, the main advantage of *DevOpRET* is that the QA team involved in the acceptance testing can tailor the decision based on actual usage of the system (i.e., by considering both correct replies and failures), and not only on the basis of a combination of revealed defects and user's intentions (i.e., stories).

Janes and Russo [30] propose the PPTAM+ tool for automating: *i)* the collection of field data about application performance degradations, *ii)* the building of the operational profile, and *iii)* regression performance tests on each build in a DevOps process. The tool is conceived for the problem of migrating a monolithic software to a microservice architecture, targeting performance testing. In this sense, it is complementary to prior related own work on microservice architectures reliability testing [31,32]; we are currently investigating the integration of the two approaches.

3 | *DevOpRET*

3.1 | Overview

We aim at an approach that guides the acceptance testing conducted by the QA team before each release cycle in a DevOps context. In the previous section we anticipated three main requirements we identified for such a testing technique, *i.e.*: it should leverage both usage-related metrics and data about actual failures in operation, and should be efficient enough for being used between the short DevOps release cycles. Our review of literature revealed that no such an approach exists, and in this section we introduce *DevOpRET*, the approach we propose to address such requirements. Ultimately, we aim at allowing the QA team to obtain an accurate and efficient estimate of reliability, and accordingly we present an assessment of *DevOpRET* in the next section.

DevOpRET relies on the following assumptions:

- the input space S of user demands can be decomposed into n partitions $S_1, \ldots, S_n$;
- a continuous monitoring facility is available to trace the user demands in operation and to record responses;
- a test oracle is available and it is possible to determine whether the response to a user demand succeeds or fails.

Partition S_i is characterized by the probability p_i of being selected by a user demand – the set of p_i being the *operational profile* P - and by the (unknown) probability f_i of that demand with inputs selected from S_i to fail – we call the set of f_i the *failure profile* F. Monitoring data are used to create a characterization of usage and failures which, ultimately, reduces the pre-release uncertainty about the exact knowledge of behavior in operation. Let us denote with $\hat{P} = (\hat{p}_1, ..., \hat{p}_n)$ the *estimated operational profile*, and with $\hat{F} = (\hat{f}_1, ..., \hat{f}_n)$ the *estimated failure profile*.

Figure 2 depicts the scenario we envisage for the adoption of *DevOpRET* within the DevOps release cycles. It foresees the following steps:

1. In a DevOps cycle, the version ready to be released is black-box tested for reliability assessment by the QA team. Since the *true* operational and failure profiles are unknown, testing is based on their estimates. Two testing algorithms are presented. *Operational testing* (OT) generates a number of test cases for partition S according to the expected usage in operation, given by $\hat{P}$. *Weighted operational testing* (WOT) generates test cases for S according to the expected usage profile $\hat{P}$ *and* to the expected failure profile $\hat{F}$. A maximum number of tests to be executed (*testing budget*) is assumed to be defined by the QA team, as a ceiling to cost or duration of step 1.
2. Test results are used to estimate the probability that the software will fail on a user demand by a frequentist approach – i.e., by the *probability of failure on demand* (PFD), which is a metric for the *operational reliability*. Reliability is computed as $R = 1$ - PFD. If the value of R satisfies the quality *gate* (e.g., a minimum threshold), the software version is released, otherwise it is sent back to the development team.
3. Once released, the end users' demands are monitored in the Ops phase: data about usage and failures (request/responses) are collected.
4. Based on the gathered information, the estimated profiles $\hat{P}$ and $\hat{F}$ are updated.

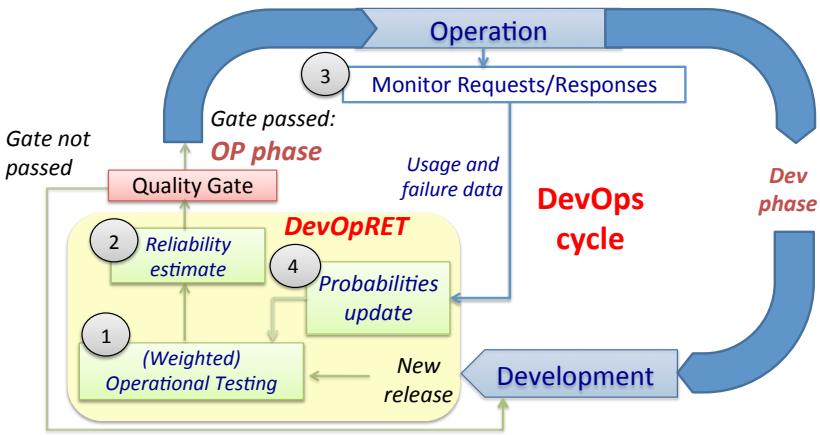


FIGURE 2 Continuous reliability testing in DevOps cycles with *DevOpRET*

On the next release, reliability testing will be carried out based on the updated profile. The estimate of R is likely to be different from the previous cycle if the estimated operational profile in the last cycle differs from the previous one, and/or if new bugs are triggered, because of different partitions being stimulated in a different way. By updating the profile at each cycle based on observations, the estimated reliability is expected to progressively converge towards the actual operational reliability.

Note how this refinement of the operational profile based on the feedback from the actual use is particularly facilitated within DevOps practice, and is the central concept of *DevOpRET*. In fact, the short circuit between users demands and acceptance testing makes it possible to automatically update the operational test generation at each new cycle, so as to continuously improve the reliability estimation. In contrast, in a traditional life cycle, a costly and more static estimation of the operational profile would be used. The next subsections provide details about the *DevOpRET* steps.

3.2 | Step 1: Test generation and execution

The first step of *DevOpRET* consists in generating and executing test cases. Test generation for reliability assessment is based on *operational testing*, which is a well-known approach using probabilistic sampling. The procedure is to first select partitions, and then generate a test case from within the selected partition. The input partitioning can be performed in many ways, e.g., based on the specification or on source code. In specification-based partitioning (the most common one), the inputs to each functionality are grouped in sets of *equivalence classes*. Ideally, each class is meant to contain inputs with the same failing behavior (i.e., either all the inputs fail or none of them); in practice, an approximation of this ideal case is obtained. We adopt specification-based partitioning based on the type of the input data.

The profiles $\hat{P}$ and $\hat{F}$ are used to derive the *testing profile* $\Pi = (\pi_1, \ldots, \pi_n)$, namely the probability distribution over the set of partitions S, driving the test generation process. For test generation we consider two sampling algorithms:

- *Operational testing* (**OT**): the selection of partitions is done according to the estimated profile $\hat{P}$ (i.e., higher $\hat{p}_i$ values have more chances to be selected), namely according to the expected usage in operation. Formally, the *testing profile* Π is such that $\pi_i = \hat{p}_i$, meaning the probability π_i of selecting partition S_i for a test is the same as the expected probability of S_i being selected in a real demand in operation.
- *Weighted operational testing* (**WOT**): the selection of partitions is done proportionally to the product of $\hat{p}_i$ and $\hat{f}_i$ values, namely preferring the partitions with a higher change of being selected *and* of failing in operation. Hence, the *testing profile* Π in this case is such that $\pi_i = \hat{p}_i \cdot \hat{f}_i$.

In both cases, the test generation within the selected partition is done by uniform random testing, i.e., by taking an input from each class of the partition according to a uniform distribution (each input having the same chance of being selected).

3.3 | Step 2: Reliability estimation

Based on the testing result, reliability is estimated. As anticipated in Section 3.1, the metric we consider for the *true reliability* R is the *reliability on a single demand*, given by:

$$R = 1 - PFD, \tag{1}$$

where PFD is the *probability of failure on demand*. This is the reliability of a run, a discrete reliability metric typical of testing research [33,34]. Let us denote with $\hat{PFD}$ the PFD *estimated* by *DevOpRET*. The OT and WOT algorithms entail two different estimators.

OT calls for the conventional Nelson model [35], in which:

$$\hat{R} = 1 - \hat{PFD} = 1 - \frac{N_F}{N} \tag{2}$$

namely the PFD is estimated as the proportion of the number of failing demands N_F over the N executed demands. This is an unbiased estimate as long as the algorithm used to select test cases mimics the way the user selects inputs, i.e., the real operational usage. Its disadvantages are that: *i)* it does not target inputs with low probability of occurrence (which often lead to failures), and *ii)* it may require many test cases to yield confident estimates (i.e., it may have low efficiency).

WOT aims at spotting those failure regions that contribute more to (un)reliability, namely those with a higher value for the product $\pi_i = \hat{p}_i \cdot \hat{f}_i$. This estimator accounts for the disproportional selection of partitions (with respect to the operational profile), computing the $\hat{PFD}$ as:

$$\hat{R} = 1 - \hat{PFD} = 1 - \frac{1}{T} \sum_{tc} \frac{\hat{p}_i \cdot z_{i,tc}}{\pi_i} \tag{3}$$

where:

- T is the number of test cases (testing budget);
- $z_{i,tc}$ is a binary value indicating if test case tc taken from partition S_i failed ($z_{i,tc}$=1) or not ($z_{i,tc}$=0),
- $\pi_i = \hat{p}_i \cdot \hat{f}_i$ (the π_i values are normalized to sum up to 1).

Dividing by π_i allows preserving unbiasedness with respect to the true PFD. This corresponds to the Hansen-Hurwitz estimator [36].

Depending on the severity of the failures observed, we compute two values for reliability. The first one considers all the failures occurred (*all-failures* reliability, denoted as R_A). Besides this, a reliability estimate considering only the high-severity failures is computed (we call it *critical-failure* reliability, denoted as R_C) – a detailed explanation will be provided in Section 4.6.5. This distinction gives more figures of interest for the QA team, which may prioritize major issues over others, thanks to a quantitative characterization of what type of problems the end user could experience and with what probability.

3.4 | Steps 3 and 4: Monitoring and update

When software is in operation, field data are gathered, and used to update the auxiliary information about partitions, which are then leveraged for reliability testing in the next release cycle. Specifically, the estimates $\hat{p}_i$ and $\hat{f}_i$ of invocation and failure probabilities for partition S_i (i=1, …, n) are updated by looking at requests/responses data gathered in the Ops phase. The update rules at the end of the *k*-th DevOps cycle is:

$$\begin{aligned} \hat{p}_i^k &= \lambda(\hat{p}_i^{k-1}) + (1-\lambda)\frac{N_i^k}{N^k} \\ \hat{f}_i^k &= \lambda(\hat{f}_i^{k-1}) + (1-\lambda)\frac{Q_i^k}{N_i^k} \end{aligned} \tag{4}$$

where:
- N^k is the total number of requests in cycle *k*;
- N_i^k is the number of requests to partition S_i in cycle *k*;
- Q_i^k is the number of failed requests to partition S_i in cycle *k*;
- λ is a *learning factor* $\lambda \in [0, 1]$, regulating how much we account for the execution history in the previous cycle with respect to observations made in cycle *k* (it is $\lambda = 0.5$ in our experimental settings).

4 | EXPERIMENTAL EVALUATION

4.1 | Research questions

The experimental evaluation is driven by the overall goal of achieving in DevOps acceptance testing an accurate and efficient estimate of reliability. We consider two case studies, whose details are provided in the next subsection. The former is an open source project. The aim of experiments with this case study is to assess performance of *DevOpRET* in both configurations (OT and WOT) in terms of estimate's accuracy and confidence, as well as in terms of ability to expose failures[1]. The second case study is a benchmark for microservice architectures. The aim of experiments with this case study is to investigate if and how the reliability estimate provided by *DevOpRET* (with both OT and WOT) changes when the true reliability of the subject under test changes, a circumstance not observed in the first case study.

In the experiments, *DevOpRET* is first used in its baseline implementation, the one adopting OT. Then, we investigate if and to what extent the adoption of *weighted operational testing* improves results over OT.

The first case study is used to investigate the following research questions:

- **RQ1**: What is the accuracy and efficiency of the reliability estimate, and what is the ability at exposing failures of *DevOpRET* with *Operational Testing*? How do they vary over successive DevOps cycles?
- **RQ2:** Does *DevOpRET* with *Weighted Operational Testing* perform better than with *Operational Testing*?

The second case study targets the following research question:

- **RQ3**: How does the accuracy of the reliability estimate change when the true reliability across releases changes?

4.2 | Experimental subjects

The first subject is *Discourse*, a discussion platform featuring services for managing mailing lists, forums and long-form chat rooms.[2] It is an open-source platform adopted worldwide by over 1,500 customers, and counting on a community of almost 670 committers. The average number of lines of code (LoC) of the five versions selected to emulate DevOps cycles is 684 KLoC. From a technical viewpoint, the front-end of *Discourse*

[1] A better reliability estimate does not necessarily imply finding more faults, although the two things are related.
[2] Available at: https://www.discourse.org/.

is a JavaScript application running in any web browser; the server side is developed mainly in Ruby on Rails. Discourse exposes an Application Programming Interface (API) allowing to consume the contents as JSON records. The API offers 85 methods. Their invocations are HTTP requests acting on resources via conventional GET, PUT, POST and DELETE operations according to the REST paradigm. In this work the acceptance tests are executed on the API interface; the main resources accessed are: *categories*, *posts*, *topics*, *private messages*, *tags*, *users*, and *groups*.

The second experimental subject is *TrainTicket*[3], a microservice system developed as a benchmark for microservice applications by Zhou *et al.*[37] It contains 41 microservices, based on several programming languages and frameworks, including Java (Spring Boot, Spring Cloud), Node.js (Express), Python (Django), and Go (Webgo), MySQL. The specification of the *TrainTicket* interfaces is in the open API documentation format Swagger[4]. The application is characterized by three main releases with 258 commits. The average number of lines of code of the 16 commits selected for experiments is 518 KLoC.

4.3 | Experiment design

To answer RQ1 and RQ2 we conduct three experiments with the *Discourse* subject, each with a number N_R=20 of repetitions for every emulated cycle. The first experiment computes the reliability estimates with OT in both the *all-failures* and the *critical-failures* scenarios. Unlike for OT, with WOT the failure severity influences the test generation algorithm (through the $\hat{f}_i$ values), hence, two experiments are needed for the two scenarios.

To answer RQ3, a fourth experiment is conducted with the *TrainTicket* subject, performing a single run for each of the 16 cycles, and considering only the *critical-failures* scenario.

4.4 | Evaluation metrics

As metrics for *DevOpRET accuracy* and *efficiency* we compute, respectively, the *mean offset* Δ and the *sample variance* V over multiple runs with respect to the *true reliability* R. The smaller the offset, the more accurate the estimate; the smaller the variance, the stronger the confidence, and the more efficient the estimator.

The *true reliability* R is computed from the true operational profile P=$\{p_i\}$ and the true failure profile F=$\{f_i\}$. It is given by:

$$R = 1 - PFD = 1 - \sum_{i=1}^{n} p_i \cdot f_i. \tag{5}$$

The computation of P and F in the experiments is described in Section 4.6.2. The true reliability is re-computed at every cycle, as the f_i values can change across the subject versions.

The *DevOpRET* estimate $\hat{R} = 1 - \hat{PFD}$ is computed by the OT and WOT algorithms with Equations 2 and 3, respectively. For statistical significance, *DevOpRET* is run N_R times per cycle, yielding N_R estimates per cycle. The mean offset Δ and sample variance V of the reliability estimate $\hat{R}$ at a given cycle over the N_R repetitions (indexed by j) are computed as:

$$\begin{aligned} \Delta_j &= |R - \hat{R}_j|, \\ \Delta &= Mean(\Delta_j), \\ Mean(\hat{R}) &= \tfrac{1}{N_R} \textstyle\sum_{j=1}^{N_R} \hat{R}_j, \\ V &= \tfrac{1}{N_R - 1} \textstyle\sum_{j=1}^{N_R} [\hat{R}_j - Mean(\hat{R})]^2. \end{aligned} \tag{6}$$

Finally, as metric for the ability at exposing failures we count the *number of failing tests*.

4.5 | Experimental artifacts

The experimental testbed includes four artifacts: a *Test Generator*, a *Workload Generator*, a *Monitor*, and an *Estimator*.

The *Test Generator* encapsulates the test cases generation algorithm and executes tests (step 1). It extracts from monitored data the list of partitions with the associated probabilities ($\hat{p}_i$ for OT, $\hat{p}_i$ and $\hat{f}_i$ for WOT), generates and executes the test cases according to the *testing profile* Π (depending on the algorithm, OT or WOT), and stores the test results. These are used by the *Estimator* to compute reliability (step 2).

The *Workload Generator* emulates the real usage of the software in the Ops phase, by issuing requests according to the *true* operational profile P (while test cases generation uses the estimated profile $\hat{P}$).

[3]Available at: https://github.com/FudanSELab/train-ticket/.
[4]https://swagger.io

The *Monitor* observes requests and responses and records them in a (textual) log file (step 3). These observations are then used by the *Test Generator* in the next cycle to update the testing profile (step 4).

For repeatability and reproducibility, we provide the code of the artifacts for running the experiments.[5]

4.6 | Experimental procedure

4.6.1 | DevOps cycles emulation

To emulate DevOps cycles, for the *Discourse* case study we consider five consecutive stable (i.e., non-beta) releases.[6] For the *TrainTicket* case study, 16 DevOps cycles are emulated by selecting 16 commits, according to the following criteria: *a)* the first selected commit corresponds to the first release (0.0.1); *b)* only "verified" commits are selected[7]; *c)* only one commit per day is selected; *d)* consecutive commits with only documentation update are removed.

Each emulated cycle includes an operational testing session (step 1) to decide whether or not the software may be released. We set the *testing budget* available per cycle to T = 1,000; this is the number of tests generated and executed to then compute the estimate of reliability $\hat{R}$ (step 2). If the reliability requirement at the quality gate is met, the software is released and the operational phase starts: the software is subject to a number of requests by the *Workload Generator*, set to N = 5,000, during which data are gathered (step 3) and used to update the probabilities (step 4). Afterwards, a new release is assumed to be ready, and next cycle starts.

We explicitly point out that, while test cases at step 1 are generated using the testing profile Π (Section 3.2), requests of the *Workload Generator* are issued according to the true profile P, computed as described in the next subsection.

4.6.2 | True profiles

For the purpose of the evaluation, an operational profile P assumed to be the *true* one is created by varying the initially estimated profile $\hat{P}$ by a *variation factor* v. In fact, what matters to assess the *DevOpRET* performance is the difference between the expected and the real usage, $\hat{P}$ and P, and how *DevOpRET* adapts and improves its reliability estimate $\hat{R}$ as $\hat{P}$ gets closer to P. Specifically, given an estimated profile $\hat{P}$ and a variation factor $v \in [0, 1]$, at the beginning of each experiment we generate the true operational profile P with the following procedure:

1. Split the set S of n partitions in two subsets in any arbitrary way, S′ with n_1 partitions and S″ with n_2 partitions (n_1+n_2 =n).
2. For S′, generate n_1 random numbers ($r'_1, \ldots, r'_{n_1}$) between 0 and 1 such that their sum is $v/2$.
3. For S″, generate n_2 random numbers ($r''_1, \ldots, r''_{n_2}$) between -1 and 0 such that their sum is $-v/2$.
4. Concatenate the two vectors of random values into a single vector r of n elements, and shuffle it.
5. Sum the elements r_i to $\hat{p}_i$ values, obtaining a new vector w such that: $w_i \leftarrow \hat{p}_i + r_i$.
6. If there is at least one $w_i < 0$, sum 1 to all values: $w_i \leftarrow w_i + 1$.
7. Normalize the obtained values: $p_i \leftarrow w_i / \sum_j w_j$, so that the sum is 1. The set of p_i values is the generated true profile P.

The true failure profile F is computed by exercising every partition to check whether it leads to failure or not. We perform multiple runs (namely, 5) to each of the thousands partitions for both subjects (with inputs picked up randomly from that partition under test); then we compute the estimate $\hat{f}_i$ as the ratio of failed requests over the number of runs. Since the partitions are relatively "small", results over runs turned out to be consistent in almost all the cases: repeated requests to a partition either always failed (hence $\hat{f}_i = 1$) or not ($\hat{f}_i = 0$). Few partitions fail occasionally, depending on the input values. For *Discourse* this happens for 67 partitions (average over the 5 cycles) for the all-failures scenario, and for 21 partitions (average) for the critical-failures scenario, corresponding, respectively, to the 0.76% and 0.24% of the total of n=8,802 partitions. For *TrainTicket*, it happens for just 2 out of the n=3,757 partitions for the critical-failures scenario (average over the 16 cycles).

4.6.3 | Profiles initialization

DevOpRET step 1 uses the testing profile Π built with the information associated with partitions. In the *k*-th cycle, the OT algorithm uses the values $\hat{p}_i^k$; the WOT algorithm used both $\hat{p}_i^k$ and $\hat{f}_i^k$. Initial values for $\hat{p}_i^0$ and $\hat{f}_i^0$ are needed. They can be assigned by the tester in several ways. If the tester has some prior knowledge about the system (or about similar services/systems), s/he can initialize the values of $\hat{p}_i^0$ considering the expected

[5] Artifacts are available at: https://github.com/dessertlab/DevOpsTesting.
[6] The *Discourse* consecutive versions considered are: *2.1.6*, *2.1.7*, *2.1.8*, *2.2.0*, *2.2.1* (available at https://github.com/discourse/discourse/releases).
[7] These commits are signed with signatures verified by GitHub, to let people know the commits come from trusted sources.

TABLE 1 Input classes for *Discourse* partitions

Input type	Input class	Class description	Input type	Input class	Class description
String	*StrValid*	Valid string, as per documentation	*Date*	*DateValid*	Value is a date (format as per documentation)
	StrEmpty	Empty string			
	StrNull	String with *null* value		*DateInvalid*	Value is not a date
	StrVeryLong	String length $> 2^{16}$		*DateEmpty*	Empty value
	StrInvalid	String with non-printable characters			
Numeric	*NumValidNormal*	Value within interval $[-2^{31}; 2^{31}]$ (Java integer limits)	*Color*	*ColValid*	Value represents a color (6-digits hexadecimal number)
	NumValidBig	Value out of interval $[-2^{31}; 2^{31}]$		*ColInvalid*	Value is not a color
	NumInvalid	Not a number		*ColEmpty*	Empty value
	NumEmpty	Empty value			
	NumAbsMinusOne	Value is equal to -1			
	NumAbsZero	Value is equal to 0			
Boolean	*BooleanValid*	Valid boolean value	*List*	*ListValid*	List with valid values
	BooleanInvalid	Value not in {True,False}		*ListEmpty*	Empty list
	BooleanEmpty	Empty value		*ListNull*	List with *null* value
Enum	*EnumValid*	Value is one of the enumeration	*Email*	*EmailValid*	Value is an email address (format 'aaa@bbb[.ccc].zzz')
	EnumInvalid	Invalid enumerative value			
	EnumEmpty	Empty value		*EmailInvalid*	Value is not an email address
				EmailEmpty	Empty value

usage in operation. In the same way the failure probability of each partition can be initialized considering the previous knowledge about the failure-proneness of partitions (for instance, the outcome of pre-release tests such as unit tests, or the tester's belief used for input space partitioning - e.g., the partitions with invalid input type are deemed to be more prone to fail). Alternatively, in case of complete ignorance, the operational profile is initialized uniformly, and the failure probability of all partitions is initialized by a same default value $\epsilon \geq 0$.

Whatever the initial profile and failure probability are, *DevOpRET* foresees their continuous update as monitoring data are gathered in the Ops phase, getting closer to the true usage and failure profile, and thus yielding a reliability estimate converging to the true one. We highlight that such capability to refine the operational profile leveraging users' demands naturally descends from the DevOps practice and makes reliability estimation easier. Clearly, the better the initial estimates the sooner the approach will converge.

As for the initialization of the values $\hat{p}_i^0$, assuming complete ignorance of the expected real usage, they are initialized by randomly assigning a probability according to a uniform distribution in (0,1) to each partition: $\hat{p}_i^0 = \mathrm{rand}(0,1)$, then normalized so as: $\sum_i \hat{p}_i^0 = 1$.

As for the initialization of the values $\hat{f}_i^0$, for the first case study we make it proportional to the number of invalid equivalence classes of the partition: $\hat{f}_i = \frac{|\mathrm{invalidClasses}|}{|\mathrm{allClasses}|}$, assuming invalid classes are more failure-prone. For the second case study, where we perform a run for each partition, we set $\hat{f}_i^0$ to 1 if the run fails, 0 otherwise.

At subsequent cycles, the probabilities are updated according to the Equations 4.

4.6.4 | Test cases generation

Test cases are generated by the OT and WOT algorithms described in Section 3.2. For both subjects, the input space is partitioned according to a specification-based criterion, on the basis of the API documentation. A fine-grain partitioning is applied; the input arguments of API methods are grouped in sets of equivalence classes based on their type. For each input type, common corner cases are considered too (e.g., empty string) as well as values clearly invalid for that type (e.g., numbers with alphabetical characters), as is usually done in black-box robustness testing [38].

Table 1 lists the equivalence classes we defined for the *Discourse* API inputs. It is worth to note that although they were derived manually, partitioning can be automated by parsing documentation, as long as a complete API documentation is available (e.g., in the Swagger format). Table 2 shows an excerpt of the API methods with their input classes and partitions. Every partition is a specific combination of input classes, one for each argument of a method. Similar classes and partitions are produced for the interfaces of *TrainTicket* microservices. The total number of partitions n is 8,802 for *Discourse* and 3,757 for *TrainTicket*.

4.6.5 | Tests execution

For both subjects a test case is a REST HTTP request to the method which the selected partition refers to. We have verified if any preconditions on the used resources (e.g., categories, posts, users, topics) hold before issuing a request: when needed, we have added the code to meet the precondition before the test (e.g., for a GET request to retrieve a resource, the precondition is that at least one instance of the resource is available; if not, our code performs a PUT before the GET). Dependencies between API methods are managed in the same way.

TABLE 2 An excerpt of the signature, input classes and partitions for *Discourse* API methods

Resource	Type	Endpoint	Arg 1 type	#Classes	Arg 2 type	#Classes	Arg 3 type	#Classes	Partitions
Categories	*POST*	*/categories.json*	*String*	*5*	*Color*	*3*	*Color*	*3*	**45**
Post	*POST*	*/posts.json*	*Numeric*	*6*	*String*	*5*	*Date*	*3*	**90**
Users	*PUT*	*/users/username /preferences/avatar/pick*	*Numeric*	*6*	*Enum*	*3*	*String*	*5*	**90**

The application responses to requests are characterized by the HTTP status code and message. We distinguish two types of output (test oracle):

1. *Correct reply*: the status code is consistent with the input submitted, e.g.: *a)* a *2xx* status code (indicating *success*) for a request with inputs belonging to the *StrValid* class, or *b)* a *4xx* status code (indicating a *client error*) for an incorrect request (e.g., a numeric input containing alphabetical characters). These responses are correct replies to incorrect requests, which the API client is required to manage.
2. *Failure*: *a)* the application raises an unmanaged exception, sent to the client, which is reported with a *5xx* status code (*server error*), or *b)* the returned status code and message are inconsistent with the input submitted. We consider the former a high-severity failure, the latter a low-severity failure.

The results of tests' execution are used to estimate the overall reliability R by the OT and WOT algorithms (through the Nelson and Hansen-Hurwitz estimators of Equations 2 and 3, respectively). Depending on the severity of failures we observe, we compute both R_A and of R_C, indicating, respectively, the case in which we consider all the failures indistinguishably (*all-failures* scenario) and the case in which we consider only the high-severity ("critical") failures (*critical-failures* scenario). In the remainder, R, F and f are used in turn to refer to both scenarios.

5 | RESULTS

5.1 | RQ1: Accuracy, efficiency, failing tests

The first research question addresses: *a)* accuracy and efficiency of the reliability estimate; *b)* number of failing tests, and *c)* how they vary over successive DevOps cycles, for *DevOpRET* with the OT algorithm. The experimental subject is *Discourse*. Table 3 reports the average true reliability values for the five cycles in the *all-failures* (R_A) and in the *critical-failures* (R_C) scenarios. *DevOpRET* tries to estimate them as accurately as possible.

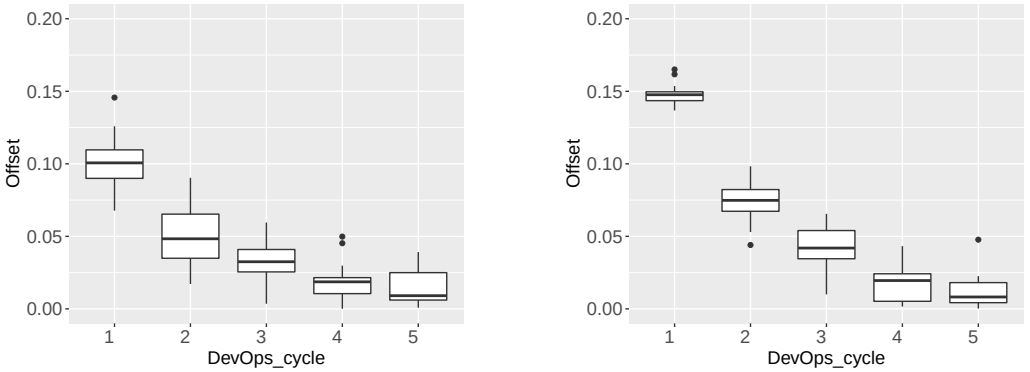
Figure 3 shows the absolute value of the offset between the *true* and the *estimated* reliabilities over cycles for the two scenarios. Expectedly, the offset decreases over releases, as the estimate of the true operational profile becomes more and more accurate. In the *all-failures* case, the offset is equal to or below 0.10 in the first cycle, and then suddenly drops to around 0.05 in the second cycle. The extent of the improvement is tied to the update rule used for monitoring (through what we called the *learning factor* λ, set to 0.5). In this experiment, a larger learning factor could further reduce the offset after the first cycle, as the true profile P does not change. In the *critical-failures* case, a larger offset is observed for the first cycle, likely due to the smaller number of detected failures, which penalizes the estimator.

Table 4 shows the sample variance and semi-interquartile range (IQR) of the R_A and R_C reliability estimates with the OT algorithm. In both cases, the values change by a small amount over releases: while the better knowledge gained about the operational profile improves the accuracy, by reducing the offset, it is scarcely influent in terms of efficiency of the estimator. Sample variance values are in the order of magnitude of 1.0E-4 in both *all-failures* and *critical-failures* cases, which denotes a quite stable performance (coefficient of variation is in the same order of magnitude, between 1.0E-4 and 2.0E-4, the average reliability being around 0.5 and 0.8 in the two scenarios). Considering only critical failures has a negligible influence on the sample variance. The same considerations stand for the interquartile range, as there is no outlier significantly affecting the result.

Figure 4 shows the box plots of the number of failing tests observed using the OT version of *DevOpRET*, for both the *all-failures* and *critical-failures* scenarios. It can be seen that the number increases over releases regardless the type of failure considered.

TABLE 3 True reliability values (subject: *Discourse*)

Subject release	Emulated DevOps cycle	*all-failures* R_A	*critical-failures* R_C
2.1.6	1	0.5297	0.7978
2.1.7	2	0.5308	0.7982
2.1.8	3	0.5305	0.7978
2.2.0	4	0.5290	0.7983
2.2.1	5	0.5279	0.7971



(a) *all-failures* **(b)** *critical-failures*

FIGURE 3 Accuracy: offset between true and estimated reliability over cycles (subject: *Discourse*, algorithm: OT)

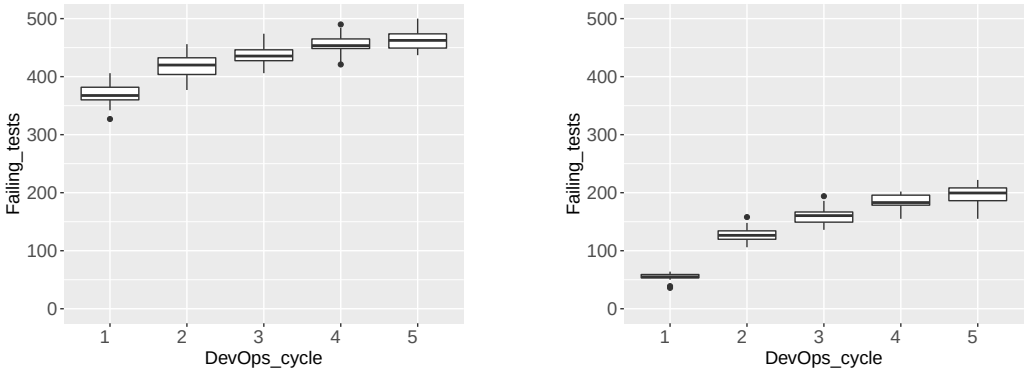
TABLE 4 Efficiency: variance and IQR of the estimated reliability (subject: *Discourse*, algorithm: OT)

(a) *all-failures*

Cycle	σ^2	IQR
1	3.33e-4	2.10e-3
2	3.44e-4	3.10e-3
3	2.43e-4	1.70e-3
4	1.59e-4	1.20e-3
5	1.31e-4	2.00e-3

(b) *critical-failures*

Cycle	σ^2	IQR
1	4.51e-5	6.00e-3
2	1.94e-4	1.70e-3
3	2.45e-4	2.10e-3
4	1.41e-4	2.00e-3
5	1.25e-4	1.40e-3



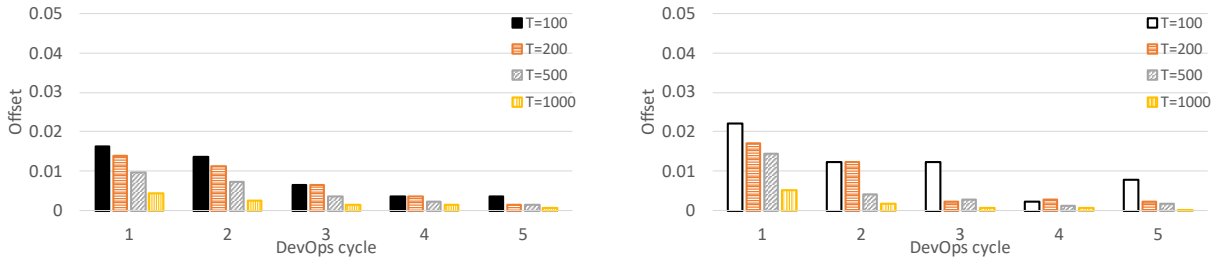
(a) *all-failures* **(b)** *critical-failures*

FIGURE 4 Number of failing tests over cycles (subject: *Discourse*, algorithm: OT)

From all these results we see that *DevOpRET*'s estimates converge to the true reliability over emulated cycles.

We now present a sensitivity analysis for the OT algorithm with respect to the number of tests performed at quality gate (the testing budget T), and the initial error on the estimate of the operational profile. In addition, we discuss two different ways to get the initial testing profile, to simulate the case of a complete ignorance and the case of the availability of an initial tester's belief based on valid/invalid classes. In the former case, the testing profile Π is generated by uniform random sampling, as explained in Section 4.6.3. In the latter case, Π is generated by giving smaller usage probability to partitions with corner cases and invalid input classes, namely by assigning: $\hat{p}_i = \frac{|\text{validClasses}|}{|\text{allClasses}|}$, normalized to sum up to 1. We call them *uniform* and *proportional* profile, respectively.

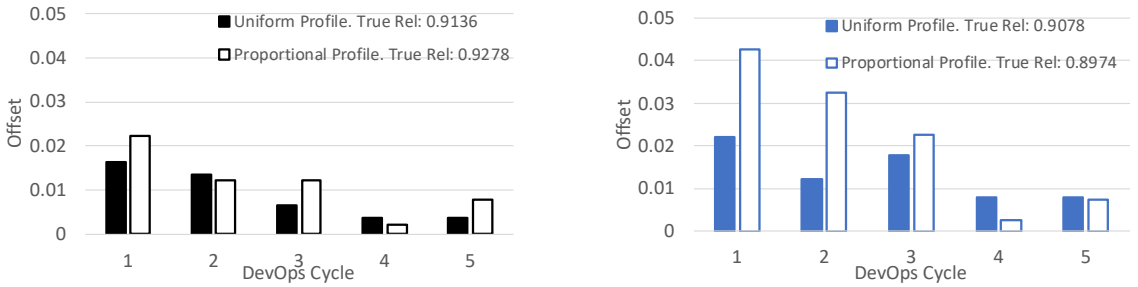
Figure 5 reports the offset varying the number of test cases for the uniform and proportional profiles, assuming the estimated profile differs from the true one by 30% ($v = 0.3$). Expectedly, increasing the number of tests improves considerably the estimate accuracy, with an offset under 0.5%



(a) Initial estimated profile: uniform

(b) Initial estimated profile: proportional

FIGURE 5 Accuracy: sensitivity to the testing budget T under a 30% error on the initial profile estimate (subject: *Discourse*, algorithm: OT)



(a) 30% error on OP (same as for T=100 in Fig. 5a and 5b)

(b) 70% error on OP

FIGURE 6 Accuracy: sensitivity to different errors in the initial operational profile estimate (subject: *Discourse*, algorithm: OT; T=100)

in the best case of 1,000 test cases. Considering that the input space we derived has 8,802 partitions (an indirect indicator of the maximum testing cost), a testing budget amounting to about 1/8 of the number of partitions has been spent to achieve an accuracy of the reliability estimate of 99.5%.

Let us consider the lowest testing budget, namely T=100 test cases. Figure 6b refers to the situation when the estimated profile differs from the true one by 70% (v=0.7), meaning that in the estimate of the usage profile the QA team has missed the true profile by a much larger extent than in Figure 6a, where v=0.3. We can see that the offset for v=0.7 is generally higher than for v=0.3, especially for the proportional profile. In fact, with a profile proportional to valid classes but under a larger initial error (v=0.7), the derived true profile is more likely to exercise the invalid classes compared to the case of a smaller error (v=0.3). We also see how *DevOpRET*'s use of data gathered in operation helps in both cases to correct the initial estimate – at cycle 4, the offset for both scenarios goes below 0.01.

To sum up, our conclusions about RQ1 are the following.

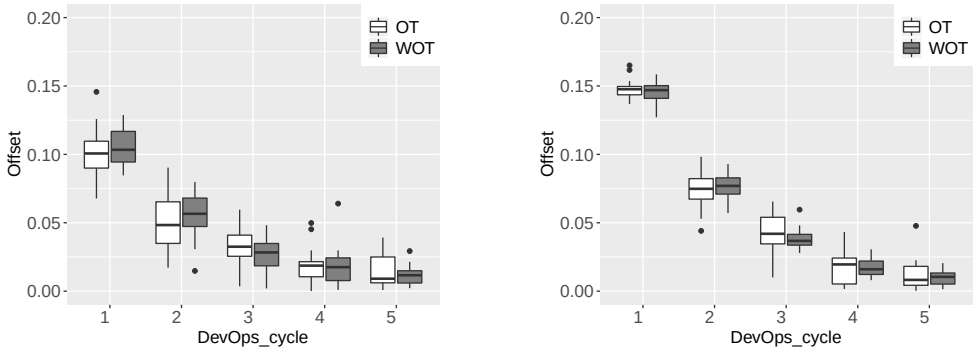
> *DevOpRET* accurately estimates the reliability of the experimental subjects.
> The estimations become more precise over subsequent DevOps cycles.
> While the knowledge of the operational profile improves the estimator accuracy, its influence in the estimator's efficiency is scarce.
> The type of failures (all or critical) has a negligible influence on *DevOpRET* efficiency.

5.2 | RQ2: WOT *vs.* OT

The second research question concerns whether or not *Weighted Operational Testing* performs better than *Operational Testing*.

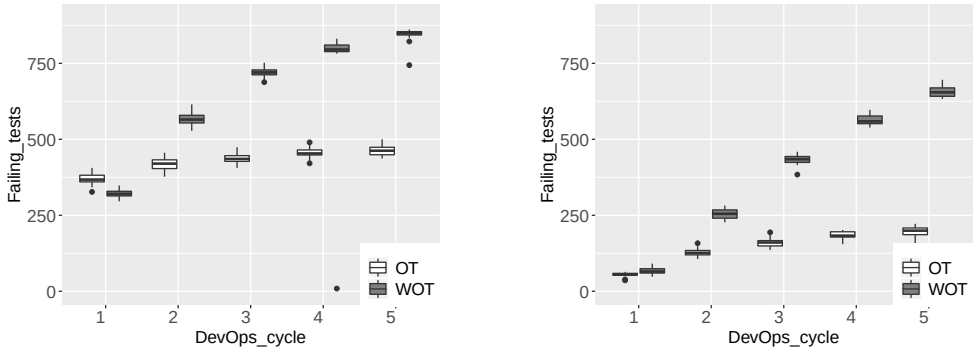
Figure 7 shows box plots of the absolute value of the offset between the *true* and the *estimated* reliability with the WOT algorithm, for both the *all-failures* and *critical-failures* scenarios. Figure 8 shows the number of failing tests with WOT. The previous results with OT from Figures 3 and 4 are shown too, for visual comparison. Table 5 reports the *p-values* of the Wilcoxon signed ranked test for paired samples applied in both scenarios to the 5 subject releases; values in bold highlight a significant difference between OT and WOT (with a significance level α=0.05). Finally, Figure 9 compares the sample variance of *DevOpRET* using WOT to the one using OT; WOT yields estimates with lower variance in most of the cases.

The results show that OT and WOT yield statistically equivalent results in terms of *offset*, namely they provide estimates with a similar level of accuracy. When they learn from histories with the same length, the results are equivalent (the comparison for all cycles does not reject the null hypothesis of no difference between the two algorithms). However, OT and WOT differ – to a significant extent - in terms of failing tests: all *p-values*



(a) *all-failures* **(b)** *critical-failures*

FIGURE 7 WOT *vs* OT - Accuracy: offset between true and estimated reliabilities over DevOps cycles (subject: *Discourse*)



(a) *all-failures* **(b)** *critical-failures*

FIGURE 8 WOT *vs* OT - Number of failing tests over DevOps cycles (subject: *Discourse*)

TABLE 5 WOT *vs* OT - Statistical comparison: *p-value*s of the Wilcoxon test (subject: Discourse)

(a) Offset

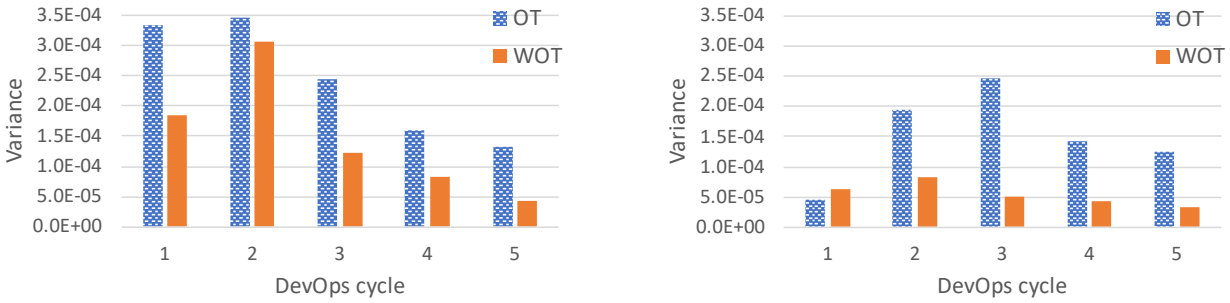
Cycle	*all-failures*	*critical-failures*
1	0.4091	0.2162
2	0.3884	0.5459
3	0.0696	0.2611
4	0.7285	0.8695
5	0.4091	0.8124

(b) Number of failing tests

Cycle	*all-failures*	*critical-failures*
1	**9.556e-05**	**5.138e-04**
2	**1.907e-06**	**9.556e-05**
3	**9.475e-05**	**9.489e-05**
4	**1.601e-03**	**9.556e-05**
5	**9.529e-05**	**9.556e-05**

in Table 5 are much smaller than 0.05. As we can see from Figure 8, the number of failing tests for the *all-failures* scenario for OT ranges from an average of 369.10 (cycle 1) to 462.15 (cycle 5), while for the WOT it ranges from 320.90 to 843.10, with a drastic improvement over versions.
An even more pronounced trend occurs for the *critical-failures* scenario: OT ranges from 54.4 (cycle 1) to 196.75 (cycle 5); WOT ranges from 66.90 to 656.55 (Figure 8b). This means that, while for the first cycle both algorithms show the same number of failing tests, WOT exploits the knowledge progressively gained about failures of partitions to direct testing toward more failing-prone partitions. The higher number of failing tests of WOT has a strong impact on the variance of the estimates, which are always smaller compared to OT, especially in the *critical-failures* scenario. This may be attributed to the adopted testing strategy. Testing is done at the API level in a black box fashion based on input types: because of the several invalid equivalence classes used to generate the tests, we exposed many failures due to the incorrect management of invalid requests. The WOT algorithm favors cases whose probability of failure is expected to be high, in order to expedite the reliability assessment (see the initialization of the failure profile in Section 4.6.3). It should be noted also that more than one failure of an API method may be related to a same fault.

In principle, the better ability of WOT at exposing failures could also impact the accuracy (offset) of its estimates. Indeed, prior own work on reliability testing has shown that techniques like WOT (based on a sampling scheme that exploits auxiliary information) are able to provide better



(a) *all-failures* **(b)** *critical-failures*

FIGURE 9 WOT *vs* OT - Sample variance over DevOps cycles (subject: *Discourse*)

offset, besides detecting much more faults and with higher efficiency (i.e., smaller variance), especially under a scarce testing budget [39,40]. However, in this case study we do not observe this phenomenon. This is due mainly to two reasons: *i)* since the number of partitions is large (8,802), the failure probability of each partition (i.e., the f_i value) is small (and is also multiplied by p_i): therefore, a more accurate knowledge of such f_i values, which are those exploited by WOT with respect to OT, has a negligible impact on the final estimate; *ii)* the number of tests is enough for both techniques to provide a close estimate (they, in a sense, are both saturated). We also explicitly point out that WOT entails an additional cost with respect to OT, namely the need for monitoring failures, besides the usage profile. Overall, we advise to use WOT whenever the cost of gathering failure data is acceptable and the testing budget is significantly limited.

To sum up, our conclusions about RQ2 are the following.

> The OT and WOT versions of DevOpRET yield statistically equivalent results in terms of accuracy of estimates.
> WOT is generally able to provide estimates with higher confidence (lower variance).
> WOT results in more failing tests compared to OT (consistently with its ability to direct testing towards more failing-prone partitions).
> WOT requires failure data; it should be preferred to OT when the testing budget is low and the failure data gathering cost is acceptable.

5.3 | RQ3: Adaptivity

The third research question investigates the adaptivity of the *DevOpRET* reliability estimate to changes of the true reliability over DevOps cycles. The goal is to assess if and how promptly *DevOpRET* reacts to changes of the actual failure probability across product releases. This experiment is conducted with the *TrainTicket* case study. The true operational profile, generated with a variation factor $v = 0.8$, is kept constant in the experiment.

Figure 10 plots the true reliability over emulated cycles. Figure 11 plots the absolute value of the offset between the *estimated* and the *true* reliability. The results show that the estimate tends to follow the true value using both OT and WOT. The two techniques show however some differences. We see that WOT exhibits performance worse than OT in cycles number 2 and 8. These two commits correspond to two significant changes in the reliability of the case study which can be seen in Figure 10. OT is less impacted by the changes of failure probability. However, WOT performs better than OT 12 times out of 16; in particular, WOT provides better estimates of the true reliability (the offset is lower) starting from a couple of cycles after the sudden changes (from 4 to 7 and from 10 on).

We draw the following conclusion about RQ3.

> OT is less affected than WOT by perturbations of the true reliability due to sudden changes of the failure probability on user demands.

5.4 | Threats to validity

The results of experiments must be considered in light of potential threats to validity. Following the guidelines by Runeson and Höst [41], we consider four aspects: construct validity, internal validity, external validity, and reliability.

Construct validity (Is the experiment we designed appropriate to answer our research questions?) The metrics we used for answering the research questions (probability of failure on demand, true and estimated reliability, sample variance, number of failing tests) are standard in the software

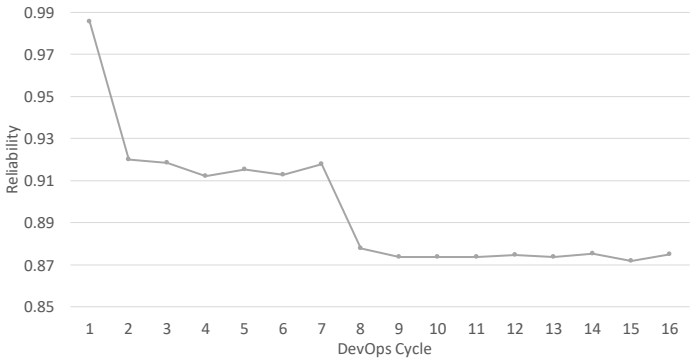


FIGURE 10 True reliability under constant operational profile over cycles (subject: *TrainTicket*)

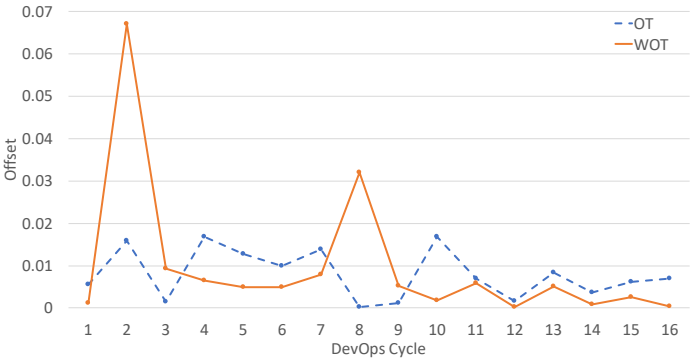


FIGURE 11 Offset between *estimated* and *true* and reliability over cycles (subject: *TrainTicket*)

reliability literature. The choice of true reliability might bias the result. While we could assume a true operational profile (p_i values) for our purposes, to control this threat we have computed the true failure probability (f_i values) by exercising each partition 5 times, for each of the 5 releases of *Discourse* and for each of the 16 commits of *Train Ticket*. This allowed us to spot failing partitions with a satisfactory confidence: on average over the releases, just 0.76% and 0.24% of *Discourse* partitions and 0.05% of *Train Ticket* ones gave inconsistent results, whereas the vast majority were either always failing or not. Two further factors in the construction of experiments which might have influenced the results are the number of test cases and the initial profile. To control this threat we have performed a sensitivity analysis, which shows that *DevOpRET* estimates are anyway close to the true reliability, although they influence the speed of the convergence. Another factor in the construction of experiments which might influence the results is the partitioning of the input space into equivalence classes, which could be performed differently by different experts. This, however, is not really a threat to validity: it only implies that the proposed approach has to be evaluated by further researchers applying different partitionings.

Internal validity (Are there factors different from the treatment that could affect the observed behavior?) A common internal threat is the accuracy of measures, which can be affected by random factors. To mitigate it, all the code developed was carefully inspected, and all experiments related to RQ1 and RQ2 (concerning the performance of *DevOpRET* and the comparison between its OT and WOT strategies) were repeated multiple times. Another common threat is the selection of experimental subjects. To favor *repeatability* of experiments under different possibly influencing factors, we chose an open-source and a publicly available benchmark as subjects, and all code and artifacts are made available for further inspection. The results might change depending on the value N of user requests we emulate with the *Workload Generator*. To control this threat, we did a conservative choice for N with respect to the number of tests T, since in a real deployment the ratio N/T would be much greater. Greater values of N with respect to T allow for larger data as feedback from Ops, and a more accurate learning of the usage and failure probabilities.

External validity (Is it possible to generalize the results beyond the experimental subject?) Although we stated the *DevOpRET* assumptions, our results might be influenced by specific characteristics of the chosen subjects. From current observations we cannot draw fully general conclusions, claiming results hold for any application; to this aim, more case studies are needed.

Reliability (Whether and to what extent can the observations be reproduced by other researchers?) To support *reproducibility* by other researchers, and possible experimentation with different subjects, we make available the code of the components of the *DevOpRET* testbed.

6 | CONCLUSIONS AND FUTURE WORK

In "traditional" software development, reliability testing practices can be hindered by the cost and difficulty of specifying the operational profile. In DevOps, thanks to the short-circuit between development and production, we claim that reliability testing is facilitated because the development and QA team can *i)* leverage usage data coming from *Ops* through monitoring as a feedback to adjust estimates of the operational profile, and *ii)* rely on it for the next acceptance testing cycle to refine reliability assessment.

In this view, we propose the *DevOpRET* approach that supports continuous testing based on the operational profile within DevOps cycles. The idea behind the approach – earlier introduced in preliminary own work [17] - is to leverage usage information monitored in operation for refining the estimate of the operational profile, which is then used during the next acceptance testing cycle. The approach has been here enhanced to also leverage actual failure information, by which a weighted version of the operational profile is obtained. We have empirically evaluated both versions of *DevOpRET* in two case studies, a real-world open source platform and a microservice architectures benchmark. The controlled experiments rely on synthetically generated workload. The results show how from one release to the next the operational profile estimate improves steadily for both versions, but the weighted version is more effective as for the number of failing tests.

Our survey of literature showed that *DevOpRET* is the first framework supporting reliability-assessment in DevOps using actual data monitored in operation. As such, it contributes to include reliability as a key performance indicator at quality gates in DevOps practices [13,42].

We envisage several future research directions for improving the approach:

- We intend to work on automatic partition extraction (from documentation) and update (from monitoring data);
- To further expedite the convergence of the estimated profile to the true one, we aim at investigating the adoption of machine learning to characterize the profile;
- We intend to experiment more sophisticated testing algorithms based on probabilistic sampling, as developed in previous work [39];
- We plan to consider similar approaches that leverage usage data for improving the testing of other non-functional properties, such as performance or usability;
- Finally, we aim at performing extensive empirical evaluations, by deploying the approach under a true workload.

In pursuing the above future research directions, we foresee both technical challenges related to the need of minimizing the impact of the approach to the users and to the tight and agile DevOps schedules, and practical challenges, descending from the paucity of real-world benchmarks for experimentation. We hope that our promising results can motivate other researchers and practitioners that reliability estimation can and should be naturally included in DevOps acceptance testing.

ACKNOWLEDGMENTS

This work has been partially supported by the PRIN 2015 project "GAUSS" funded by MIUR (Grant n. 2015KWREMX_002). The work by G. De Angelis has also been supported by the Italian Research Group INdAM-GNCS.

References

1. Baresi L, Ghezzi C. The Disappearing Boundary Between Development-time and Run-time. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER)*, ACM; 2010: 17–22
2. Bass LJ, Weber IM, Zhu L. *DevOps - A Software Architect's Perspective.* SEI series in software engineering. Addison-Wesley . 2015.
3. Dyck A, Penners R, Lichter H. Towards definitions for release engineering and DevOps. In: *IEEE/ACM 3rd International Workshop on Release Engineering (RELENG)*, IEEE; 2015: 3–3.
4. Smeds J, Nybom K, Porres I. DevOps: A Definition and Perceived Adoption Impediments. In: Lassenius C, Dingsøyr T, Paasivaara M., eds. *Agile Processes in Software Engineering and Extreme Programming*, Springer International Publishing; 2015; Cham: 166–177.
5. Walls M. *Building a DevOps culture.* O'Reilly Media, Inc. . 2013.
6. Humble J, Farley D. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Addison-Wesley . 2011.
7. Forsgren N, Kersten M. DevOps Metrics. *Communications of the ACM* 2018; 61(4): 44–48.

8. Brunnert A, Hoorn vA, Willnecker F, et al. Performance-oriented DevOps: A Research Agenda. *CoRR* 2015; abs/1508.04752.

9. Mazkatli M, Koziolek A. Continuous Integration of Performance Model. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18. ACM; 2018: 153–158

10. Rahman AAU, Williams L. Software Security in DevOps: Synthesizing Practitioners' Perceptions and Practices. In: *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, IEEE; 2016: 70–76

11. Lee JS. The DevSecOps and Agency Theory. In: *IEEE 29th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE; 2018: 243–244

12. Musa JD, Everett WW. Software-Reliability Engineering: Technology for the 1990s. *IEEE Software* 1990; 7(6): 36–43.

13. Forsgren N, Humble J, Kim G. Accelerate: State of DevOps, Strategies for a New Economy. https://cloudplatformonline.com/2018-state-of-devops.html; 2018. Accessed: 2020-03-02.

14. Beyer B, Jones C, Petoff J, Murphy NR. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly . 2016.

15. Lyu MR. Software reliability engineering: A roadmap. In: *Future of Software Engineering (FOSE)*, IEEE; 2007: 153–170.

16. Musa JD. Software-Reliability-Engineered Testing. *IEEE Computer* 1996; 29(11): 61–68.

17. Pietrantuono R, Bertolino A, De Angelis G, Miranda B, Russo S. Towards Continuous Software Reliability Testing in DevOps. In: *Proceedings of the 14th International Workshop on Automation of Software Test*, IEEE; 2019: 21–27

18. Fitzgerald B, Stol KJ. Continuous software engineering and beyond: trends and challenges. In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, ACM; 2014: 1–9.

19. Soni M. End to end automation on cloud with build pipeline: the case for DevOps in insurance industry, continuous integration, continuous testing, and continuous delivery. In: *IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, IEEE; 2015: 85–89.

20. Di Nitto E, Jamshidi P, Guerriero M, Spais I, Tamburri DA. A software architecture framework for quality-aware DevOps. In: *Proceedings of the 2nd International Workshop on Quality-Aware DevOps (QUDOS)*, IEEE; 2016: 12–17.

21. Angara J, Prasad S, Sridevi G. The Factors Driving Testing in DevOps Setting - A Systematic Literature Survey. *Indian Journal of Science and Technology* 2017; 9(48): 1–8.

22. Soares Cruzes D, Melsnes K, Marczak S. Testing in a DevOps Era: Perceptions of Testers in Norwegian Organisations. In: Misra S. et al. ., ed. *International Conference on Computational Science and Its Applications (ICCSA)*, . 11622 of *LNCS*. Springer, Cham; 2019: 442–455.

23. Fitzgerald B, Stol KJ. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software* 2017; 123: 176–189.

24. Kitchenham B, Charters S. Guidelines for performing Systematic Literature Reviews in Software Engineering. Tech. Rep. EBSE 2007-001, *Keele University and Durham University Joint Report*; 2007.

25. Marijan D. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In: *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE; 2015: 157–162.

26. Ali S, Hafeez Y, Hussain S, Yang S. Enhanced regression testing technique for agile software development and continuous integration strategies. *Software Quality Journal* 2019.

27. Najafi A, Shang W, Rigby PC. Improving test effectiveness using test executions history: an industrial experience report. In: *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE- SEIP)*, IEEE; 2019: 213–222.

28. Révész Á, Pataki N. Containerized A/B Testing. In: *Proceedings of the 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA)*, CEUR Workshop Proceedings; 2017.

29. Mijumbi R, Okumoto K, Asthana A, Meekel J. Recent Advances in Software Reliability Assurance. In: *IEEE 29th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE; 2018: 77–82

30. Janes A, Russo B. Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE; 2019: 163–168.

31. Pietrantuono R, Russo S, Guerriero A. Run-time Reliability Estimation of Microservice Architectures. In: *IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE; 2018: 25–35.

32. Pietrantuono R, Russo S, Guerriero A. Testing microservice architectures for operational reliability. *Software Testing Verification and Reliability* 2020; 30(2). doi: 10.1002/stvr.1725

33. Frankl PG, Hamlet RG, Littlewood B, Strigini L. Evaluating testing methods by delivered reliability [software]. *IEEE Transactions on Software Engineering* 1998; 24(8): 586–601.

34. Cai KY. Towards a Conceptual Framework of Software Run Reliability Modeling. *Information Sciences* 2000; 126(1-4): 137–163.

35. Thayer TA, Lipow M, Nelson EC. *Software Reliability*. North-Holland Publishing, TRW Series of Software Technology, Amsterdam . 1978.

36. Chaudhuri A. *Survey Sampling Theory and Methods.* Chapman & Hall/CRC, Second Edition, Taylor & Francis Group . 2005.

37. Zhou X, Peng X, Xie T, et al. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* 2018.

38. Laranjeiro N, Vieira M, Madeira H. A robustness testing approach for SOAP Web services. *Journal of Internet Services and Applications* 2012; 3(2): 215–232.

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

40. Pietrantuono R, Russo S. Probabilistic Sampling-Based Testing for Accelerated Reliability Assessment. In: *18th IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE; 2018: 35–46

41. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 2009; 14: 131–164.

42. Sloss BT, Nukala S, Rau V. Metrics That Matter. *Queue* 2018; 16(6): 86–105.

How to cite this article: A. Bertolino, G. De Angelis, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo (2020), DevOpRET: Continuous Reliability Testing in DevOps, *Journal of Software: Evolution and Process*.