Towards Continuous Software Reliability Testing in DevOps

Roberto Pietrantuono*, Antonia Bertolino†, Guglielmo De Angelis‡, Breno Miranda§ and Stefano Russo*

* DIETI - Università degli Studi di Napoli Federico II, Napoli, Italy
Email: {roberto.pietrantuono,sterusso}@unina.it

†CNR–ISTI, Pisa, Italy
Email: antonia.bertolino@isti.cnr.it

‡CNR–IASI, Roma, Italy
Email: guglielmo.deangelis@iasi.cnr.it

§Federal University of Pernambuco, Recife, Brazil
Email: bafm@cin.ufpe.br

***Abstract*—We introduce the *DevOpRET* approach for continuous reliability testing in DevOps. It leverages information monitored in operation to guide operational-profile based testing, which is conceived as part of the acceptance testing stage before each next release to production. We overview the envisaged test and monitoring pipeline, describe the approach and present a case-study evaluating how reliability assessment evolves over subsequent releases.**

***Index Terms*—Acceptance Test, DevOps, Operational Profile, Quality Gate, Software Reliability Testing**

I. Introduction

DevOps is commonly regarded (Fig. 1) as the intersection among the scopes of software Development (Dev), Operation (Ops) and Quality Assurance (QA). Despite its spread, there is no universally agreed upon definition for DevOps, as discussed in [1], [2]. Some authors describe it as a cultural shift that IT organizations should undergo to remove technical or managerial barriers between the Dev and Ops teams, and let them collaborate under shared responsibilities [3]. Others focus on the technological capabilities that are necessary to enable such culture [2]; for instance, 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*" [4].

Continuous testing and monitoring are two key DevOps practices. *Continuous testing* foresees: on the one side, short and automated testing rounds that can provide quick quality feedbacks to continuous integration (CI); on the other side, an acceptance test stage that checks whether the current software candidate is ready for release – 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...*" ([5], pag. 124). *Monitoring* consists in collecting data from the system running in production as well as users' feedback, which can be used by Dev and QA teams for measurement and optimization in next testing stage. Monitoring is essential for DevOps as, ultimately, DevOps adoption is motivated and driven by business objectives, which are quantified into measurable Key Performance Indicators (KPIs). Indeed, measurement is a central instrument in DevOps success [6]. At each release cycle, the acceptance testing must include the assessment of KPIs of interest (such as performance, security) and evaluate whether the candidate release meets the established targets. Such KPI targets constitute a *quality gate* before release.

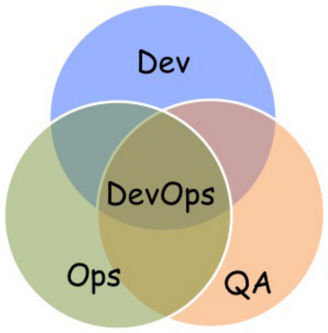


Fig. 1: DevOps scope

As noticed in [7], notwithstanding the interest of companies and consultants, DevOps has not yet received much attention by the scientific community. As for the most important KPIs for DevOps, some attention has been devoted to performance [8], [9] and security [10], [11], but further insights come form looking at the ample gray literature. In particular, since 2014, the DevOps Research and Assessment (DORA) company (recently acquired by Google Cloud) has conducted yearly surveys over more than 30K DevOps professionals. These reports provide a comprehensive and up-to-date analysis of observed trends, meant as reference by companies for benchmarking. Typical KPIs reported by DORA include: Deployment frequency, Lead time for changes, Time to restore service, and Change failure rate. Notably, in the 2018 edition [12] 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".

Indeed, providing an adequate level of reliability in operation should be an important concern in DevOps and part of the acceptance testing quality gate, as it certainly contributes to the final user satisfaction. Yet, surprisingly, the topic of DevOps reliability has not been addressed in existing work. Motivated by such argument, we propose here an approach, called *DevOpRET*, for introducing *continuous software reliability testing* in DevOps practice. The approach is inspired (as suggested by the acronym) by the software reliability engineered testing (SRET) best practice, early introduced by Musa [13]. *DevOpRET* is conceived to exploit usage data to derive representative test cases and assess reliability before each release.

In "traditional" software development, the adoption of reliability testing practice 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 optimize the operational profile, and *ii)* rely on it for the next acceptance testing cycle to refine reliability assessment or improve it.

Towards such a vision, in this paper we provide:

- an overview of the *DevOpRET* approach (Sect. II), which to the best of our knowledge (see Related Work, Sect. V) is the first attempt to apply reliability testing to DevOps;
- a walk-through example of its application to the real-world open source platform *Discourse* (Sect. III);
- the evaluation in a simulated environment to show how, by using *Ops* data, *DevOpRET* would converge towards "true" reliability assessment (Sect. IV), thus enabling DevOps teams to draw better informed decisions on whether a candidate release can pass the established quality gate.

II. *DevOpRET* OVERVIEW

As anticipated in the introduction, *DevOpRET* uses the feedback from Ops to guide the acceptance testing stage conducted by the QA team before release. The technique expects the following assumptions to be met: the input space S of user demands can be decomposed into i partitions $S_1, \ldots, S_i$; a test oracle is available and it is possible to determine if a user demand fails or succeeds, for demands issued by operational workload as well as for those issued by *DevOpRET* tests; a continuous monitoring facility is available to trace the user demands in operation. Monitoring data are the key in *DevOpRET* to create a characterization of the usage profile which, ultimately, reduces the pre-release time uncertainty about the exact knowledge of the operational profile. Formally, the operational profile is a probability distribution P over the set of partitions, with each value p_i representing the probability of issuing a user demand with inputs belonging to partition i. Its estimate is denoted as $\hat{P}$ (and $\hat{p}_i$).

Figure 2 depicts the scenario of DevOps release cycles where we envisage the adoption of *DevOpRET*, which foresees the following steps:

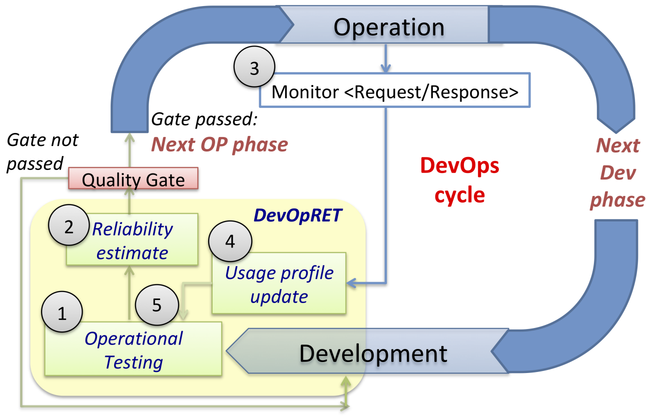


Fig. 2: Continuous reliability testing in DevOps cycles

1) In a DevOps cycle, the version ready to be released is black-box tested for reliability assessment by the QA team. Since the *actual* operational profile is unknown, testing is based on an *estimated* operational profile, $\hat{P}$. Given a budgeted number of test cases, the latter are generated by an *operational testing* approach, by which we *i)* select a partition i with probability $\hat{p}_i$, and then *ii)* select an input from the partition's input space randomly according to a uniform distribution.
2) Test results are used to estimate the probability that the software will fail on a user demand – i.e., the *probability of failure on demand (PFD)*, which is a metric of the *operational reliability* - by a frequentist approach. Reliability is computed as $R = 1 - PFD$. If reliability satisfies the quality *gate* (e.g.: a minimum reliability threshold), the software version is released, otherwise it is sent back to the development team.
3) Once released, the end users' demands are monitored, and information about input values of the demands is collected.
4) Based on the gathered information, the estimate of the operational profile is updated (i.e., the $\hat{p}_i$ values).
5) On the next release, reliability testing will be carried out based on the updated $\hat{p}_i$ values. By using the updated profile at each cycle, the estimated reliability is expected to converge to the actual operational reliability.

The next Section details the steps by a case study.

III. CASE STUDY WALK-THROUGH

We illustrate the application of *DevOpRET* through a working case study, a popular open-source community platform.

A. *Subject*

The subject we consider is *Discourse*, a discussion platform featuring services for managing mailing lists, forums and long-form chat rooms.[1] It is adopted by over 1,500 customers (e.g., companies, institutions, communities, etc.) all over the world;

[1]https://www.discourse.org/.

as an open-source project it counts on a community of almost 670 committers.

From a technical viewpoint, the front-end of Discourse is a JavaScript application that can run in any web browser; the server side is developed mainly in Ruby on Rails. An instance of Discourse exposes an API which allows to consume the contents as JSON records. Any functionality offered by the site version of Discourse can also be accessed using such API. Specifically, the API offers a set of 85 methods. The API method invocations are HTTP requests which act on a set of 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 interfaces exposed by *Discourse*. The main resources accessed by means of the API are: *categories*, *posts*, *topics*, *private messages*, *tags*, *users*, and *groups*.

B. Input space partitioning

The input space of *Discourse* is partitioned according to a specification-based criterion. The input arguments of each API method are grouped in sets of equivalence classes based on their type. These classes have been defined based on the API documentation describing the inputs the method accepts. For each input type, we also considered common corner cases (e.g., empty string, a number equal to zero), as well as values clearly invalid for that type (e.g., number with alphabetical characters), as is usually done in black-box robustness testing approaches to assess a system under test under more rare circumstances [14]. Table I lists the equivalence classes we defined for the Discourse API inputs. It is worth to note that although we derived them manually, partitioning can be automated by parsing documentation, as long as a complete API documentation is available (e.g., in the SWAGGER format). This is part of our ongoing work.

Table II reports 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. The total number of partitions is 8,802. Partition i is characterized by an estimate of probability of selection in operation ($\hat{p}_i$) – the *estimated operational profile* $\hat{P}$. These estimates (which could be assigned by the QA team) correspond to the expected probability by which those methods (and partitions) will be used in operation. The $\hat{p}_i$ values are used to derive the *testing profile* Π, namely a probability distribution over the input space that drives the test generation process.

C. Test generation and execution

We define the procedure to first select partitions, and then generate a test case from within the selected partition. The algorithm is based on conventional *operational testing*: the selection of partitions is done according to the estimated profile $\hat{P}$ values (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 i for a test is the same as selecting partition i in operation.

TABLE I: Input classes for case study partitions

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

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). A test case is a REST HTTP request to the method which the selected partitions refers to, parametrized with the selected inputs. For each method, we have verified if any precondition on the used resource (e.g., categories, posts, users, topics) must 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 is managed in the same way.

D. Output interpretation

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

1) *Correct reply*: the status code and message are consistent with the input submitted, such as:
   a) a *2xx* status code (indicating *success*) for a request with inputs belonging to the *ValidStr* 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 cor-

TABLE II: An excerpt of the case study API methods' signature, input classes and partitions

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**
..	...	...	...	...	...	...	...	...	**...**

rect replies to incorrect requests, which the API client is required to manage.

2) *Failure*: the application raises an unexpected, unmanaged, exception, sent to the client, which is reported as *5xx* status code (*server error*).

E. Reliability estimation

The quality measure we assess for the QA team is *reliability on a single demand* obtained as $R = 1-PFD$, where PFD is the *probability of failure on demand* (PFD). This corresponds to reliability of a run, a discrete reliability computation typical of testing research [15], [16]. Reliability is estimated by the conventional Nelson model [17], in which $PFD = \frac{N_F}{N}$, namely the PFD is the proportion of the number of failing demands over the N executed ones. This is an unbiased estimate as the algorithm used to select test cases mimics the way the user selects inputs, i.e., the real operational usage.

F. Monitoring and update

As software is put in operation, *DevOpRET* gathers field data to update the auxiliary information about partitions, that will be leveraged for reliability testing in the next release cycle. Specifically, we update the estimates of p_i values by looking at requests/responses. As monitoring data are gathered and knowledge about actual runtime usage cumulates, the estimates $\hat{p}_i$ tend to the true values p_i. The update rule at the end of DevOps operational cycle k is:

$$\hat{p}_i^k = \lambda(\hat{p}_i^{k-1}) + (1-\lambda)\frac{N_i^k}{N^k} \tag{1}$$

where:

N^k	is the total number of requests in current cycle k;
N_i^k	is the number of requests to partition i in cycle k;
λ	is the learning factor, $\lambda \in [0,1]$, regulating how much we account for the past history (cycle $k-1$) with respect to observations made in the current cycle k – it is $\lambda = 0.5$ in our setting.

G. Testbed

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

The *Test Generator* encapsulates the test case generation algorithm; it performs the activities of steps 1 and 4 in Fig. 2. It extracts from monitored data the list of partitions with the associated probabilities ($\hat{p}_i$), generates and executes the test cases according to the *testing profile* Π, and stores the result in a textual file. Results of testing are used by the *Estimator* to compute reliability (step 2 in Fig. 2).

The *Workload Generator* emulates the real usage of the software in operation, namely it issues requests according to an *operational profile*, which reflects the actual profile in operation resulting from the user requests (i.e., the *true* p_i values). The operational profile P is, in general, different from the estimated operational profile $\hat{P}$ used during testing. The generation of P is discussed in the next subsection.

Requests and responses are observed and logged in a textual file by the *Monitor*, corresponding to step 3 in Fig. 2. This is meant to be used for reliability testing of the next release.

H. Procedure

For our evaluation, we consider $k = 5$ consecutive releases of *Discourse*, simulating five DevOps cycles. Each cycle includes an acceptance testing session in order to decide if the software is ready to be released or not. The budget of the acceptance testing session is set to $T = 100$ at each cycle. After testing, if the quality gate (a reliability requirement, in our case) is met, then the software is "released" in the simulated environment and the operational testing phase starts. The number of requests issued by the Workload Generator in an operational cycle is set to $N = 1000$, after which we assume a new release is ready, and a new cycle starts.

1) Testing phase: DevOpRET testing is done at each DevOps cycle before release, using the information associated with partitions, namely the values $\hat{p}_i^k$ at cycle k. At cycle zero, we need to assume an initial value of $\hat{p}_i^0$ representing the initial estimated probabilities for the test partitions. Whatever the initial estimated profile, *DevOpRET* foresees the update of the profile as monitoring data are gathered after each operational cycle, getting closer to the true usage and thus yielding a reliability estimate converging to the true one. In the reality, the better the initial estimates the earlier the approach will converge.

For the purpose of the evaluation, we consider two different ways of defining the initial profile (resulting in two different scenarios). The former assumes complete ignorance by testers about the expected usage of partitions. Hence: $\hat{p}_i = \frac{1}{M}$, where M is the number of partitions, These are used as *initial testing profile* – let's denote it as Π_1^0, namely the testing profile Π at cycle 0, scenario 1. The second scenario gives a smaller usage probability to partitions with corner case and invalid input classes, to represent a more realistic situation. This is accomplished by assigning: $\hat{p}_i = \frac{|valid|}{|allClasses|}$, then normalized to sum up to 1 (i.e., $\hat{p}_i \leftarrow \hat{p}_i / \sum_j \hat{p}_j$). Partitions with more valid classes will have bigger $\hat{p}_i$. The testing profile generated using this information at cycle 0 is denoted as Π_2^0, i.e., testing profile Π at cycle 0, scenario 2.

After testing, the *Estimator* provides an assessment of the achieved PFDs.

2) Operational phase: Assuming that a quality gate is satisfied (i.e., we have a reliability above a desired threshold), software is released in a local environment and is stressed by the *Workload Generator*.

The Workload Generator issues requests according to the operational profile P. We create an operational profile (assumed to be the *true* one) that differs from the testing-time estimated profile $\hat{P}$ by a *variation factor* v. In fact, what matters to assess *DevOpRET* is the difference between the expected and the real usage, $\hat{P}$ and P. Specifically, given a variation factor $v \in [0,1]$ and an estimated profile $\hat{P}$, the procedure is as follows:

1) Split the set of M partitions in two subsets in any arbitrary way, S_1 with M_1 partitions and S_2 with M_2 partitions.
2) Generate, for S_1, M_1 random numbers between 0 and 1 such that their sum is $v/2$.
3) Generate, for S_2, M_2 random numbers between -1 and 0 such that their sum is $-v/2$.
4) Concatenate the two sets of values in one set $S \leftarrow S_1 \cup S_2$, and shuffle it.
5) Sum the elements s_i of S to $\hat{p}_i$ values, obtaining the vector $\mathbf{w}$ such that: $w_i \leftarrow \hat{p}_i + s_i$.
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 as the sum is 1. The set of p_i values is the generated true profile P.

During the operational phase, requests and responses are monitored and used to update $\hat{P}$ according to the described Eq. 1. This is repeated for each cycle, i.e., at each release. In our evaluation, we considered two values of v so as to assess *DevOpRET* under a more or less severe error in the initial profile estimate: $v = 0.3$, $v = 0.7$.

3) Evaluation metric: As evaluation metric, we consider the difference between the estimated reliability $\hat{R} = 1 - PFD$ and the true one ($\Delta = |R - \hat{R}|$). The true reliability is assessed (once for all in the whole evaluation) by running 10,000 requests generated according to the true profile (as explained in Section III-C) and using again the Nelson estimator: $R = 1 - N_F/10,000$, where N_F is the number of failed requests[2].

IV. Results

Figure 3 shows the offset between the *true* and the *estimated* reliability at each step (representing the k-th release of *Discourse*), under the two defined scenarios: *a)* the initial estimated profile is *uniform*; *b)* the initial estimated profile is *proportional* to the number of valid equivalence classes. In both cases, $T = 100$ test cases and the difference between estimated and true profile is 30%, i.e., $v = 0.3$.

[2]We assume a number of 10,000 requests are considered enough to provide an accurate estimate of true reliability, being it one to two orders of magnitude bigger than the executed test cases T (T goes from 100 to 1,000)

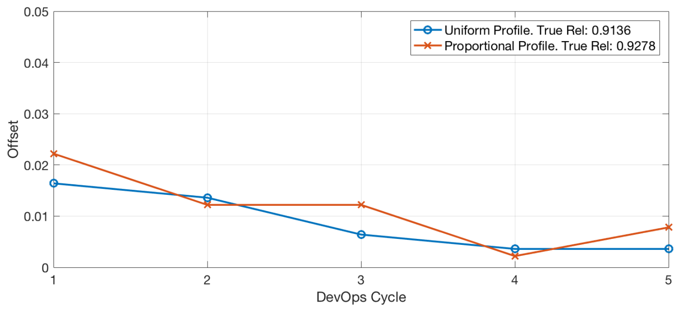


Fig. 3: Offset. $T = 100$ test cases, $v = 0.3$

In the first scenario, we observed 864 failures out of 10,000 requests, hence with a true reliability under the generated profile $R = 0.9136$. In the second scenario, we got 722 failures leading to a true reliability of $R = 0.9278$. Since the estimated profile is proportional to valid classes (i.e., partitions with invalid classes are less likely exercised) and the true profile is not much different ($v = 0.3$), the true reliability in the second scenario is more representative of the actual reliability experienced by an end user. The offset is similar in the two cases, as the it does not depend on the specific true or estimated profile, but mainly on the difference between them. In both scenarios, as information is gathered from the operational phase, the assessed reliability gets closer to the true one, and the offset closer to zero.

In Figure 4, we assess *DevOpRET* when the estimated profile differs from the true one by 70% ($v = 0.7$), meaning that the QA team, in its assessment of the usage profile, has missed by a large extent the true profile. In such a case, the initial offset is more pronounced than the case of $v = 0.3$, especially for the proportional profile. In fact, with a profile proportional to valid classes, but a high variation factor ($v = 0.7$), the derived true profile is more likely to exercise the invalid classes compared to the case with $v = 0.3$ – the true reliability turned out to be $R = 0.8974$ in this case. Using updated information is of paramount importance in such cases in order to correct the initial estimate – at step 4, the offset for both scenarios goes below 0.01.

For a given length of the learning window (i.e., number of user requests between two testing sessions), which is set to 1,000 in our evaluation, the speed at which the assessment

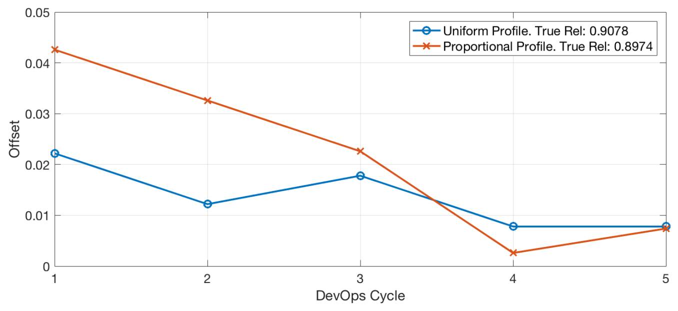
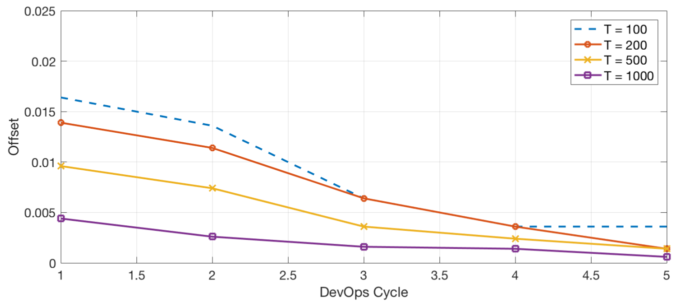
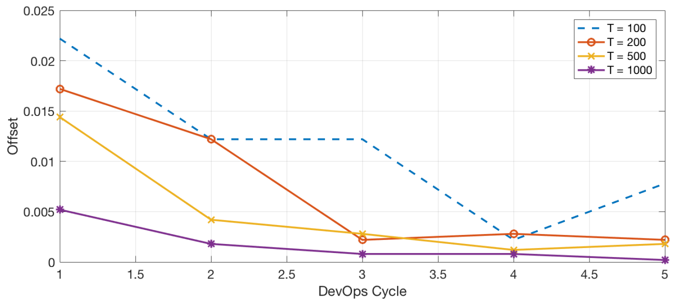


Fig. 4: Offset. $T = 100$ test cases, $v = 0.7$



(a) Initial estimated profile: uniform



(b) Initial estimated profile: proportional

Fig. 5: Offset. $T = 100, 200, 500$ and 1000 test cases, $v = 0.3$

converges to the true one depends mainly *i)* on how much the initial profile estimate deviates from the true one, as we observed from Figures 3 and 4, but also *ii)* on the number of executed test cases. Figure 5 reports the offset under 200, 500 and 1,000 test cases (we report also the previous case, T=100) for both uniform and proportional profile. Expectedly, increasing the number of tests considerably improves the assessment accuracy, with an offset under 0.005 (i.e., about 0.5%) in the best case of 1,000 test cases. In summary, considering that the test case space we derived has 8,802 partitions (which is an indirect measure of the testing complexity of the application), 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 0.995%.

V. Related work

Despite its claimed popularity within grey-literature sources, testing within the context of DevOps still appears to be underconsidered in the academic scientific literature [18]. On the one hand, to identify motivational factors driving DevOps testing several works [19] [20] [21] analyse the state-of-the-art of continuous software engineering. On the other hand, few of them actually describe studies or detailed aspects that are specifically tailored for continuous testing, and that explicitly concern settings from the DevOps environments.

With respect to the former category of related works, it is interesting to remark how our research contributes to some of the key activities in Continuous software engineering identified by [22]. Specifically, *DevOpRET* contributes to *Continuous Testing*, *Continuous Use*, and *Continuous Improvement* of the targeted system: by using the actual operational profile as a means for data-driven planning of the future reliability testing activities *DevOpRET* guides the Dev or QA teams to solve issues that are closer to the user-significant scenarios fostering their continuous engagement.

In the latter category of related works: the approach in [23] supports the study of performance issues (already revealed or potential) by analysing profiled data collected at run-time. The context referred in [23] is quite similar to the context described in our work. Nevertheless a first main difference is about the target: they are only focusing on performance-driven software refactoring and specifically on load testing. Another difference is that that paper does not clarify the rules/conditions that regulate the sampling of the data within the operational profile. Possibly in the case of load testing it is more interesting to use the whole set of data observed during operation in order to better study the nature of the failure; nevertheless this is not always the case for other testing approaches (e.g. functional, and non-functional) where in general it is either unfeasible, or not economically sustainable to exploit the whole set of monitored data also for testing purposes.

The work in [24] also focuses on software performance aspects. Specifically it aims to support robust performance estimation by efficiently evaluating the impact of fluctuation in the values of the parameters adopted within the prediction models. The authors highlight how their approach can be integrated in DevOps methodologies in order to take decisions about the next version releases. In this sense, the interesting relation with *DevOpRET* is that in our future work we could consider similar techniques in order to take into account how the operational profile observed during the current *op-cycle* could vary in the next one.

In [25] the authors proposed a method for predicting software defects within the context of projects running continuous integration, and continuous delivery practices. Even though their overall objective is similar to the goal of our work, the main difference between the two proposals is that the authors of [25] do not use the actual operational profile in order to plan the testing activities for the next release, while its 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 teams involved in the *dev* cycle can tailor their 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).

VI. Conclusions and Future Work

We have presented the *DevOpRET* approach that supports black-box testing based on the operational profile within a DevOps cycle. The idea is to leverage the information that is usually monitored in Ops for the purpose of refining an estimate of reliability during the next acceptance testing stage. We envisage that reliability-related KPIs (here for example we considered an estimate of the probability of failure on demand) should be included into the acceptance quality gate that a product should pass before each new release.

This work included the outline of a scenario of a reliability testing framework included within a DevOps cycle: this is the first proposal in this direction. We also implemented a preliminary version of *DevOpRET* and carried out an initial evaluation on a real world open source platform, aiming at ascertaining how the approach behaves in terms of reliability assessment while more usage data are collected. As expected, the results confirm that leveraging usage data contributes to quickly converge towards "exact" reliability prediction. Of course, this paper is only a first step towards a more comprehensive framework for DevOps reliability assessment and improvement, and further developments and evaluation studies are required.

In the future, our aim is to improve results and extensively validate them by: *i)* working on the learning phase with the aim of expediting the convergence of the estimated profile to the true one (we aim at investigating the adoption of machine learning to characterize the profile); *ii)* adopting more sophisticated testing algorithms based on probabilistic sampling developed in previous work [26], aimed to expedite the assessment by exploiting auxiliary information besides the profile (such as the observed probability of failure of each partition); *iii)* working on automatic partition extraction (form documentation) and update (from monitoring data) and *iv)* deploying the approach under a true workload (not synthetically generated as in the case study developed here) as well as more case studies. More specifically with respect to this last item, an interesting future work includes the study of the impacts (e.g., performance, scalability, effort, maintainability, etc.) of *DevOpRET* when it is adopted in an actual deployment pipeline.

Acknowledgment

This work has been partially supported by: the PRIN 2015 project "GAUSS" funded by MIUR (Grant n. 2015KWREMX_002); the European Project H2020 731535: ElasTest; and the Italian Research Group: INdAM-GNCS.

B. Miranda wishes to thank the postdoctoral fellowship jointly sponsored by CAPES and FACEPE (APQ-0826-1.03/16; BCT-0204-1.03/17).

References

[1] A. Dyck, R. Penners, and H. Lichter, "Towards definitions for release engineering and DevOps," in *IEEE/ACM 3rd International Workshop on Release Engineering (RELENG)*. IEEE, 2015, pp. 3–3.

[2] J. Smeds, K. Nybom, and I. Porres, "DevOps: A definition and perceived adoption impediments," in *Agile Processes in Software Engineering and Extreme Programming*, C. Lassenius, T. Dingsøyr, and M. Paasivaara, Eds. Cham: Springer International Publishing, 2015, pp. 166–177.

[3] M. Walls, *Building a DevOps culture*. " O'Reilly Media, Inc.", 2013.

[4] L. J. Bass, I. M. Weber, and L. Zhu, *DevOps - A Software Architect's Perspective*, ser. SEI series in software engineering. Addison-Wesley, 2015.

[5] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, 2011.

[6] N. Forsgren and M. Kersten, "DevOps Metrics," *Communications of the ACM*, vol. 61, no. 4, pp. 44–48, 2018.

[7] R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer, "Towards a benefits dependency network for DevOps based on a systematic literature review," *Journal of Software: Evolution and Process*, vol. 30, no. 11, p. e1957, 2018.

[8] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. R. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert, "Performance-oriented devops: A research agenda," *CoRR*, vol. abs/1508.04752, 2015. [Online]. Available: http://arxiv.org/abs/1508.04752

[9] M. Mazkatli and A. Koziolek, "Continuous integration of performance model," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: ACM, 2018, pp. 153–158. [Online]. Available: http://doi.acm.org/10.1145/3185768.3186285

[10] A. A. U. Rahman and L. Williams, "Software security in devops: Synthesizing practitioners' perceptions and practices," in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, May 2016, pp. 70–76.

[11] J. S. Lee, "The devsecops and agency theory," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2018, pp. 243–244.

[12] N. Forsgren, J. Humble, and G. Kim, "Accelerate: State of DevOps, strategies for a new economy," pp. 436–440, 2018, accessed: 2019-01-23. [Online]. Available: https://cloudplatformonline.com/2018-state-of-devops.html

[13] J. D. Musa, "Software-reliability-engineered testing," *IEEE Computer*, vol. 29, no. 11, pp. 61–68, 1996.

[14] N. Laranjeiro, M. Vieira, and H. Madeira, "A robustness testing approach for soap web services," *Journal of Internet Services and Applications*, vol. 3, no. 2, pp. 215–232, Sep 2012. [Online]. Available: https://doi.org/10.1007/s13174-012-0062-2

[15] K.-Y. Cai, "Towards a conceptual framework of software run reliability modeling," *Inf. Sci.*, vol. 126, no. 1-4, pp. 137–163, Jul. 2000. [Online]. Available: https://doi.org/10.1016/S0020-0255(00)00018-9

[16] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini, "Evaluating testing methods by delivered reliability [software]," *IEEE Transactions on Software Engineering*, vol. 24, no. 8, pp. 586–601, Aug 1998.

[17] T. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability*. North-Holland Publishing, TRW Series of Software Technology, Amsterdam, 1978.

[18] J. Angara, S. Prasad, and G. Sridevi, "The Factors Driving Testing in DevOps Setting - A Systematic Literature Survey," *Indian Journal of Science and Technology*, vol. 9, no. 48, 2017.

[19] B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, 2014, pp. 1–9.

[20] M. Soni, "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, pp. 85–89.

[21] E. Di Nitto, P. Jamshidi, M. Guerriero, I. Spais, and D. A. Tamburri, "A software architecture framework for quality-aware DevOps," in *Proceedings of the 2nd International Workshop on Quality-Aware DevOps (QUDOS)*. ACM, 2016, pp. 12–17.

[22] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017.

[23] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche, "Exploiting load testing and profiling for performance antipattern detection," *Information and Software Technology*, vol. 95, pp. 329–345, 2018.

[24] A. Aleti, C. Trubiani, A. van Hoorn, and P. Jamshidi, "An efficient method for uncertainty propagation in robust software performance estimation," *Journal of Systems and Software*, vol. 138, pp. 222–235, 2018.

[25] R. Mijumbi, K. Okumoto, A. Asthana, and J. Meekel, "Recent advances in software reliability assurance," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 77–82.

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