Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet?

Zishuo Ding, Jinfu Chen, and Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{zi_ding,fu_chen,shang}@encs.concordia.ca

ABSTRACT

Performance is one of the important aspects of software quality. Performance issues exist widely in software systems, and the process of fixing the performance issues is an essential step in the release cycle of software systems. Although performance testing is widely adopted in practice, it is still expensive and time-consuming. In particular, the performance testing is usually conducted after the system is built in a dedicated testing environment. The challenges of performance testing make it difficult to fit into the common DevOps process in software development. On the other hand, there exist a large number of tests readily available, that are executed regularly within the release pipeline during software development. In this paper, we perform an exploratory study to determine whether such readily available tests are capable of serving as performance tests. In particular, we would like to see whether the performance of these tests can demonstrate performance improvements obtained from fixing real-life performance issues. We collect 127 performance issues from *Hadoop* and *Cassandra*, and evaluate the performance of the readily available tests from the commits before and after the performance issue fixes. We find that most of the improvements from the fixes to performance issues can be demonstrated using the readily available tests in the release pipeline. However, only a very small portion of the tests can be used for demonstrating the improvements. By manually examining the tests, we identify eight reasons that a test cannot demonstrate performance improvements even though it covers the changed source code of the issue fix. Finally, we build random forest classifiers determining the important metrics influencing the readily available tests (not) being able to demonstrate performance improvements from issue fixes. We find that the test code itself and the source code covered by the test are important factors, while the factors related to the code changes in the performance issues fixes have a low importance. Practitioners may focus on designing and improving the tests, instead of fine-tuning tests for different performance issues fixes. Our findings can be used as a guideline for practitioners to reduce the amount of effort spent on leveraging and designing tests that run in the release pipeline for performance assurance activities.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380351

KEYWORDS

Performance testing, Performance issues, Software performance

ACM Reference Format:
Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet?. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380351

1 INTRODUCTION

Performance is one of the most important aspects of software quality. Performance can directly affect the user experience of large-scale systems, such as Amazon, Ebay, and Google [33]. A Prior study finds that field issues reported in such systems are more associated with the performance of the system, instead of functional issues [50].

Performance issues exist widely in software systems [26], and are difficult to avoid during the software development processes [38]. The performance issues have various effects on the system. Some lead to high resource (like CPU or memory) utilization, and some can cause a long response time to user requests. An example performance issue excerpt from *Hadoop* issue tracking system [1] describes that when *NetworkTopology* calls *add()* or *remove()*, it calls *toString()* for *LOG.debug()* which requires extra resources. As indicated in the issue report, the *toString()* method is used for logging messages, which can lead to the unnecessary slowdown of the operation and extra resource utilization.

Performance testing is challenging. It is often an expensive and time-consuming process [3, 25]. Performance tests often need to run with carefully designed sophisticated test plans, on top of the support of special software (like JMeter [1]) and are executed for a long period of time (days) [25]. On the other hand, such performance tests typically exercise the entire system as a whole instead of an optimized "Targeted Therapy". In particular, such long-running and un-targeted performance testing is difficult to fit into the widely adopted DevOps process, when releases are frequent and contain smaller changes between two releases.

On the other hand, there exist a large number of tests that are typically executed regularly during every build in the release pipeline of software development [49]. For instance, in a recent release of *Cassandra*, more than 500 tests are executed by default in a regular build process during the release pipeline; while more than 4,000 tests are executed in a recent release of *Hadoop*[2]. Prior studies find that such tests are often complex, covering various scenarios of the

[1] https://issues.apache.org/jira/browse/HADOOP-14369
[2] https://github.com/apache/hadoop/releases/tag/rel/release-3.1.2

usage of the software [4, 5, 40]. More importantly, these tests are readily available and are executed by default on a regular basis.

Due to the expensive performance testing as well as the wide availability and maturity of tests that run in the release pipeline, recent research has been advocating the use of such tests in performance assurance activities[11, 22, 23, 45]. However, there exists little knowledge about to what extent can the tests in the release pipeline behave as performance tests. Therefore, in this paper, we study the use of the readily available tests in the release pipeline of two open-source projects, i.e., *Hadoop* and *Cassandra*, as performance tests. We identify 127 performance issues that are fixed in the two subject systems and the snapshots of the source code before and after the fix of each performance issue. By evaluating the performance of the tests with the snapshots of the source code, we aim to answer the following research questions[3]:

RQ 1: *Can the readily available tests from the release pipeline demonstrate performance improvements from performance issues fixes?*
Most of the performance improvements after an issue fix can be demonstrated by at least one test. However, for each performance issue, only a very small (9.2% and 20.6%) portion of the tests can demonstrate the performance improvements.

RQ 2: *What are the reasons that some tests in the release pipeline cannot be used as performance tests?*
We identify eight reasons that a test from the release pipeline cannot demonstrate performance improvements from a performance issue fixes. The reasons can be used as a guideline for practitioners to design micro-performance tests.

RQ 3: *What are the important factors for a test to be useful as a performance test?*
We build classifiers to model whether a test can demonstrate the performance improvements of a particular performance issue. By exploring the important factors in our classifiers, we find that the factors related to the test itself and the covered source code of the test are important in the classifiers. On the other hand, the factors related to the code changes in the performance issue fixes have a low importance. Our results imply that practitioners may focus on designing and selecting tests, instead of optimizing tests especially for different performance issues.

Our findings demonstrate the capability and the challenges of using the readily available tests from the release pipeline in performance assurance activities. Our paper calls for future research that assists in designing and selecting tests that can be used in various (e.g., functional and non-functional) scenarios for the development of software systems.

Paper organization. The rest of this paper is organized as follows: Section 2 presents the prior research that is related to this paper. Section 3 presents our approach for collecting the performance data from the readily available tests and manual labelling with test the performance metrics. Section 4 presents our three research questions and our results to answer the three research questions. Section 5 presents the threats to the validity of our study. Finally, Section 6 concludes this paper.

[3]The data from our study is shared at https://github.com/senseconcordia/ICSE2020-Performance

2 RELATED WORK

In this section, we discuss the prior research that is related to this paper.

Empirical studies on performance issues

Empirical studies are conducted in order to gain a deep understanding of the nature of performance issues. Jin et al. [26] conducted an empirical study on 109 real-world performance issues that are collected from five representative software projects. Zaman et al. [54] study a random sample of 400 performance and non-performance issues from *Mozilla Firefox* and *Google Chrome*. Huang et al. [24] study 100 randomly selected real-world performance regression issues from three open source systems. Based on the study results, prior research found that that it is difficult to reproduce performance issues and more time is spent on discussing performance issues than other kinds of issues [54]. Therefore, automated approaches are designed in order to assist in detecting performance issues [26] and prioritizing performance tests [24] based on the study results. Prior research illustrates the importance of addressing performance issues in practice. Our work can be adopted by practices in tandem with the prior research on the topic of performance issues.

Performance issues detection

Prior research builds predictive models in order to predict performance issues [30, 52]. Lim et al. [30] formulate the performance issue identification as a Hidden Markov Random Field based clustering problem. Xiong et al. [52] leverage statistical models to model the system performance in the cloud. Luo et al. [31] propose a recommendation system, called PerfImpact to identify code changes that may potentially cause performance regressions. Such approaches are applied with a new version of the software in order to detect performance issue. However, such prior research on performance issue modeling depends on a large amount of performance data with complex modeling techniques. Such approaches, although proven to be effective, are difficult to adopt in practice [6], due to their extra overhead and the required resources. Moreover, such approaches are often conducted at the last stage of the release. Leveraging these approaches to detect every performance issue is difficult and impractical. Therefore, our findings in this paper may complement existing approaches in order to detect performance issue fixes more frequently during the rapid development processes.

Micro-scale performance tests

Extensive prior research has proposed automated techniques to design, execute and analyze large-scale performance testing [25]. Due to the complexity and the resources needed for such large-scale performance testing, in recent years, research has been conducted in order to study and design performance testing in a small scale (micro-scale performance test).

Leitner et al. [29] conduct a study on 111 open-source java projects to understand the state of art of performance testing. Similarly, Stefan et al. [45] conduct a study on the practices of using performance unit testing frameworks, including *Caliper*, *ContiPerf*, *Japex*, *JMH*, *JunitPerf*. Both studies show that most of the performance tests are smoke tests and the projects often use *JUnit* to test the performance combined with functional test; while only few open source projects use any performance unit testing framework.

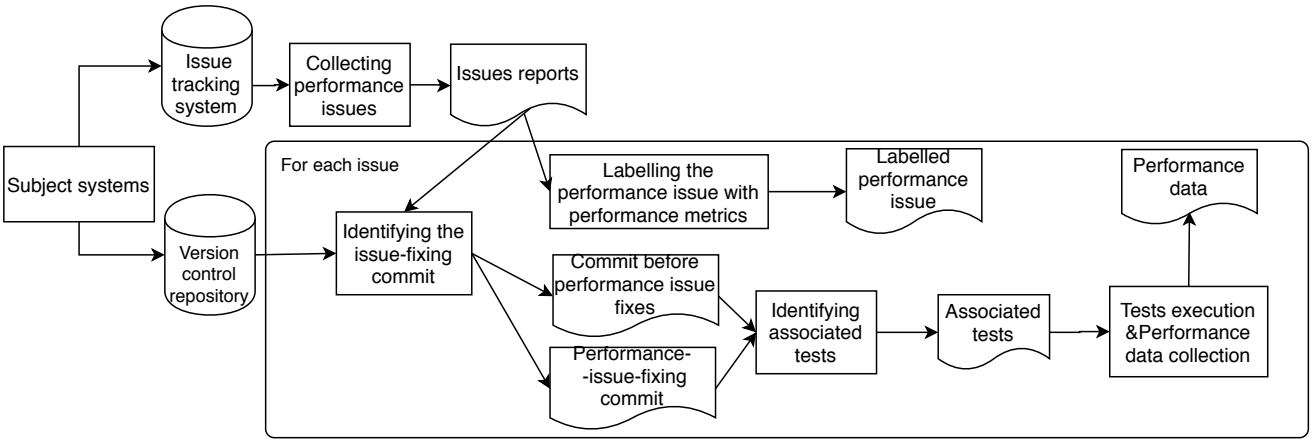

Figure 1: An overview of our case study setup and performance data collection.

These prior papers motivate our work in order to support a more flexible and low-friction performance testing practice.

Approaches are designed to improve the existing micro-performance testing. Bulej et al. [11] present a statistic approach to express performance requirements on unit testing. In addition, Horký et al. [23] propose an approach to use performance unit tests to increase performance awareness.

The prior research on micro-performance testing motivates the need of knowing the effectiveness of the readily available tests in performance assurance scenarios. Our findings can complement prior research in order to advance the practice of testing system performance in a targeted manner.

3 CASE STUDY SETUP

In this section, we first present the subject systems of our study and the collection of performance issues from the subject systems. Then we present our approach and experiment to collect performance data and we also present the experimental environment. Figure 1 shows an overview of these steps.

3.1 Subject systems

We base our study on two open-source projects, *Hadoop* and *Cassandra*. *Hadoop* is a distributed data processing system. *Cassandra* is a free and open-source distributed NoSQL database management system. We choose *Hadoop* and *Cassandra* since they are highly concerned with their performance and have been studied in prior research in mining performance data [15, 46].

3.2 Collecting performance issues

We first collect the performance issues in the two subject systems. We follow an approach similar to the one used in prior studies [54] for performance issues collection. In order to ensure that there exists a performance improvement after the issue fixes, we only focus on the issue reports that have the type *Bug* and are labeled as *Resolved* or *Fixed*.

We use keywords as the heuristics to identify performance issue reports. We start by using the keywords that are used in prior research [26, 54]. In order to avoid missing performance issues, we expand our list of keywords by using word embedding. We adopt a word2vec model trained over 15GB of textual data from *Stack Overflow* posts [19] to identify the words that are semantically related to the existing list of keywords. Examples of the uncommon words that related to performance issues include "sluggish", and "laggy", which may not be used in previous research, but can help collect performance issue reports.

By expanding the list of keywords, we gathered a total of 953 and 966 issue reports in *Hadoop* and *Cassandra*, respectively [4]. Intuitively, not all issue reports are indeed related to performance issues. Therefore, the first and last authors manually examine every issue report independently to confirm that the issue report is related to a performance issue. The two authors achieve an agreement of 73.9%. Afterwards, the two authors discuss each disagreement to reach consensus. When the consensus cannot be reached, a third author examines the issue report and makes a final decision. Finally, we collect 88 and 121 performance-related issue reports in *Hadoop* and *Cassandra*, respectively. The amount of issue reports is comparable to prior study on performance issues [24, 26, 54].

3.3 Labelling performance issues with performance metrics

Each performance issue has its corresponding performance metrics that can be measured and used to demonstrate the symptom of the performance issue and the improvement after fixes. For example, issue *HADOOP-6502*, has a description of ". . . DistributedFileSystem#listStatus is very slow when listing a directory with a size of 1300 . . . ". Based on the description, we know that the performance issue can be observed by measuring elapsed time of the execution and the elapsed time should decrease after the issue is fixed. The first two authors manually label all of the collected performance issues with their corresponding performance metrics. In total, we identify five performance metrics in our labelling of the performance issues in our subject systems, i.e., elapsed time, CPU usage,

[4]The time period of the data collection is from the start date of each project to the day we collected the issues (17, September 2018).

memory usage, I/O read and I/O write. For Hadoop, 70, 19, 17, 6, and 4 issues are labeled with elapsed time, CPU usage, memory usage, I/O read and I/O write, respectively. 77, 32, 29, 33, and 29 issues from *Cassandra* are labeled with elapsed time, CPU usage, memory usage, I/O read and I/O write, respectively. Note that an issue report can have performance issues with multiple performance metrics. The two authors have an agreement of 89.0% on the labelling and a similar approach as the last step is followed when labelling disagreement occurs.

3.4 Evaluating the fixes of performance issue

In this subsection, we present how do we study the use of the readily available tests from the release pipeline to evaluate performance. We first identify the performance issue fixing commits, in order to identify the two snapshots of the source code, i.e., before and after fixing each performance issue. We then present the selection and execution of the associated tests that cover the issue fixing source code. Finally, we present the performance evaluation for each test in order to study whether each test can demonstrate a performance improvement for the performance issue fixes.

3.4.1 Identifying performance issue fixing commits. We clone the *git* version control repositories of our subject systems, and use *git log* to extract all the code commits together with the corresponding commit messages. The commit messages typically contain an issue ID, indicating the issue that each commit addresses. With this information, we collect all the associated commits for each collected performance issue.

We note that there may exist multiple commits for fixing one issue. One reason is that an issue may be too complex to fix in one commit. Therefore, developers may divide the fix of an issue into several commits. In addition, developers might have thought that the issue is fixed, while actually is found not fixed, reopened [51] and fixed in a later commit. In these cases, we consider the chronological last commits as the issue fixing commits. We also exclude the commits that do not have any code changes. Finally, if an issue ID is not contained in any commit message, we remove the issue from our study.

After this step, 46 issues are filtered out. And then, we can collect two snapshots of source code for each performance issue, i.e., one before issue fixing, and one after issue fixing. We checkout both snapshots of the source code for each performance issue.

3.4.2 Executing associated tests. Both of our subject systems have a large number of tests that are available in the release pipeline. We first search for all tests based on their build files. *Hadoop* has four different sub-modules. We select the tests by each sub-module to minimize the large amount of irrelevant tests to save computational resources. For *Cassandra*, we include all the retrieved tests.

Intuitively, not all tests execute the source code that is changed by the performance issue fixes. Hence, for each performance issue, we identify the tests that execute the source code that is changed by the fixes (impacted tests) and the tests that do not (un-impacted tests). We leverage code coverage tools to identify the executed lines in the source code for each test. Different code coverage tools are used in the subject systems. In particular, *Cobertura* and *JaCoCo* are used for *Cassandra*. *Hadoop* depends on *Atlassian Clover* to calculate code coverage. Since *Atlassian Clover* needs licenses to execute, and all support was discontinued at April 11, 2018, we turn to *OpenClover*, which is an open-sourced version of *Atlassian Clover*, to measure the code coverage in *Hadoop*. If a test executes the added or modified lines in the source code between two versions (before and after the performance issues fixes), we consider the test impacted. In addition, for deleted lines of code, we consider a test covering the code if the test executes the lines before and after the deleted lines. By doing this, we identify 127 issues that have the impacted tests.

Afterwards, we run every test (both impacted and un-impacted) individually to evaluate performance that is associated with each test. In particular, the tests for each performance issue are executed on one virtual machine with 8GB memory and 16 cores CPU hosted by Google Compute Engine (GCE) [5]. Each test is independently executed with 30 repetitions to minimize noise. Prior research studies the use of cloud environment on performance evaluation and shows the successful use of such a number of repetitions [28]. Note that we also exclude the commits and the issues where the project fails to build and run. In total, we spent more than 11,642 machine hours for executing all the tests for the 127 performance issues in our subject systems.

3.4.3 Evaluating the performance of each test. To evaluate the performance that is associated with each test, we collect the five performance metrics, including the elapsed time, CPU usage, memory usage, I/O read and I/O write, as the labelling of performance issues. We use *psutil* (python system and process utilities) [41] for monitoring the CPU usage, memory usage, I/O read, and I/O write of the process that executes the tests. *Psutil* has been used widely in prior research on software performance [13, 53]. We use test summary reports generated via *Ant/Maven* and *Junit* to measure the elapsed time of each test. After this step, we have collected performance data for all the tests (both impacted and un-impacted) that are associated with two versions of source code (before and after each performance issue fix) of each performance issue. We then use this data to answer our research questions.

4 CASE STUDY RESULTS

In this section, we aim to answer the following research questions:

RQ1: Can the readily available tests from the release pipeline demonstrate performance improvements from performance issues fixes?

Motivation. Performance issue reports are often used as a great source of knowledge in system performance assurance activities in prior research [24, 26]. The certainty of having performance improvements, the description of the reports and the available patches make performance issues a great subject for prior research on software performance. This research question concerns whether the performance of the readily available tests from the release pipeline can demonstrate performance improvements from performance issue fixes. If not, the readily available tests would not be capable of serving as performance tests for other performance assurance activities with even higher difficulty.

[5] https://cloud.google.com/compute/

Approach. **Analyzing performance evaluation results.** For each test, we leverage statistical tests on the performance evaluation results to determine whether the performance of the test has changed after fixing the performance issue. In particular, for each performance issue, we first select only the tests that are impacted by the performance issue fixes. Afterwards, we check the label of the performance metrics (e.g., elapsed time) (see Section 3.3) that are associated with the symptoms of the performance issues. We would like to determine whether the corresponding performance metrics have different statistical significance values before and after the performance issues fixes.

Due to the non-normality of the performance data, we use *Mann-Whitney U test*, as does prior work [14, 55]. Our *null hypothesis* and *alternative hypothesis* are given below,

H_0 **:** The two performance result (i.e., test and control group – the same test before and after performance issue fixes) are equal.

H_1 **:** The two compared tests do not have the same performance.

and we run the test at the 5% level of significance (i.e., $\alpha = 0.05$). That is, if the P-value of the test is not greater than 0.05 (i.e., $P - value \leq 0.05$), we would reject the *null hypothesis* in favour of the alternative hypothesis. In other words, there exists a statistically significant performance change between the performance metrics, and the change is unlikely by chance.

However, a statistical significance test does not contain the information about *the size of the effect* [17], and when the performance data points under study are formed by a great number of items, the statistically significant differences are more frequently observed [12, 28]. Therefore, we further adopt the effect size as a complement of the statistical significance test. Considering the non-normality of our data points, we utilize *Cliff's Delta* [16], which does not require any assumptions about the shape or spread of the two distributions [28]. The effect size is assessed using the thresholds provided in prior research [42],

Filtering false-positive results. To avoid the False Positives, and eliminate the influence of the negligible or small changes of the performance, we only consider the performance changes that have a *large* effect size. In short, if the performance metric of an impacted test is changed, in particular improved (e.g., lower CPU usage), after the performance issue fixes, with statistically significant difference and large effect size, and the performance metric is also labelled for the performance issue, we consider the test to be capable of verifying the performance issues fixes.

In order to further avoid false positive results, we would like to understand the patterns of false-positive results and use such patterns to filter out our data. In order to identify the most obvious false-positives, we check the largest ten performance changes (in effect sizes, c.f., Section 4) in the un-impacted tests (no modification committed on the source code covered by the tests) in each subject system. We manual study on the possible causes of the false positive changes that reside in the source code. We find two reasons: 1) some functional tests contain random operations, which can lead to the unstable performance and 2) frequent I/O operations. Therefore, we do not consider the results of a test if the test is corresponding to either of these two reasons.

Finally, we manually examine all the cases of each performance issue (c.f., Section 4) to ensure that the tests indeed demonstrate a performance improvement after a performance issue fix.

Results. **Most performance fixes' improvements can be demonstrated by at least one readily available test.** We find that for 56 out of 60 of the performance issues in *Hadoop* and 46 out of 67 performance issues in Cassandra, at least one test from the release pipeline can be used to demonstrate performance improvements with all their associated performance metrics. In addition, for seven additional performance issues in *Cassandra*, performance improvements with part of the performance metrics can be demonstrated. For example, the commit *#9afc209* fixes the issue *CASSANDRA-7401*, which describes an endless loop in the source code. Based on the report, there should be improvements on both elapsed time and CPU usage from the issue fix. Among all the impacted tests, elapsed time and CPU usage are indeed improved significantly with large effect size in three tests. Such results show the potential capability of the readily available tests from the release pipeline to serve as performance tests.

Only a small portion of the tests from the release pipeline can be used to demonstrate performance improvements. Figure 2 shows the percentage of tests that can or cannot be used to demonstrate the improvements from performance issues fixes. The results show that it would be challenging for practitioner to directly use the readily available test in the release pipeline as performance tests. In particular, on average, only 9.2% and 20.6% of the tests in *Cassandra* and *Hadoop*, respectively, can demonstrate performance improvements for all associated performance metrics. 13.9% and 5.1% of the tests in *Cassandra* and *Hadoop*, respectively, can demonstrate performance improvements with part of the associated performance metrics. On the other hand, 76.9% and 74.3% of the tests in *Cassandra* and *Hadoop*, respectively, cannot demonstrate any performance improvement, even though these tests all execute the changed source code for the issue fixes. For example, to fix issue *CASSANDRA-3344*, 25 tests are impacted by the code change; while only two tests can demonstrate the performance improvement from the issue fix. Due to the large number of total available tests in the release pipeline, practitioners may be overwhelmed by the influx of performance results from the tests in the release pipeline and the difficulty of selecting the useful ones.

On one hand, most of performance improvements from performance issue fixes can be demonstrated using the readily available tests in the release pipeline. On the other hand, it is challenging to use these tests in practice since only a very small portion of the tests can demonstrate the improvements.

RQ2: What are the reasons that some tests in the release pipeline cannot be used as performance tests?

Motivation. In the last research question, we find that many of the readily available tests in the release pipeline cannot demonstrate a performance improvement from the performance issue fixes, even though the changed source code for the issue fixes is executed by these tests. Therefore, in this research question, we would like to understand the reason that these tests cannot serve as performance

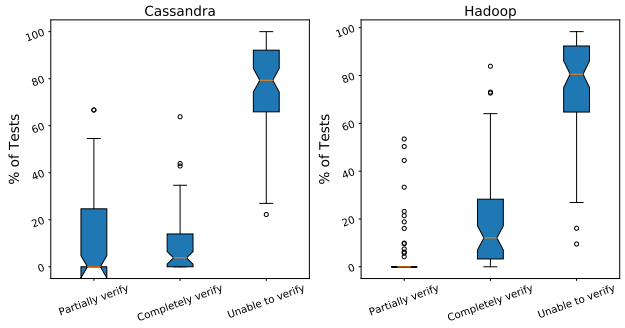

Figure 2: The percentage of tests that can or cannot be used to demonstrate performance improvements from issue fixes for each issue.

tests. The findings of this research question can assist practitioners in avoiding the use of certain tests in performance assurance activities and in improving tests to serve as performance tests.

Approach. We follow a four-step open coding approach to analyze the reasons that can cause a test to not be able to demonstrate performance improvements, even though the test is impacted by the issue fix.

Based on the results from RQ1, we collect all the impacted tests for the performance issues, i.e., the tests that cover the changed source code of the corresponding issue fix, but do not demonstrate performance improvements on the performance metrics of the issue. Two authors independently examine each test to uncover reasons of not being able to demonstrate performance improvements. In particular, the authors examine the following information that is associated with each test: 1) the performance issue report, which contains the high-level information for the issues' description, 2) the test code, which contains the low-level information of the tests and the changed parts of the committed files and 3) the source code covered by the test, which tells us which lines have been executed by the tests.

Step 1. The first two authors independently generate categories of reasons that a test cannot demonstrate performance improvements. In particular, each author iteratively investigates all the tests to identify the reasons, until no more new reasons can be found. The outcome of the first step is the different category of reasons by each of the two authors.

Step 2. Intuitively, the two authors would not generate identical categories. Hence, the two authors meet and discuss their categories. The goal is to generate final categories of reasons that both of the two authors agree on. The two authors discuss each of their generated categories of reasons and reach consensus on the final categories.

Step 3. The two authors use the agreed categories from the second step. The two authors independently put each test into one category.

Step 4. Finally, the two authors examine the results where the two authors do not agree. The two authors discuss their rationale to try to reach consensus. If consensus cannot be made, the third author will examine the corresponding test to make the final decision. The two authors have an agreement of 71.1%.

Results. **We identify eight possible reasons that a test cannot be used to demonstrate performance improvements.** We discuss each reason in detail with examples in the rest of this RQ.

Too light workload (185 tests). We find that some performance issues can only be triggered with a rather large data size. However, functional tests may not be written with such a large data size as input, making it impossible to demonstrate the issue fixes. For example, the issue reported in *CASSANDRA-581*, can be triggered with a very large number of *sstables*. It is fixed in the commit *#2b62df2*. However, the impacted tests do not have a large enough amount of *sstables* as input to reproduce the performance issue.

Not enough repetition (9 tests). Some performance issues have a rather small effect, while becoming impactful with a large number of repetitions. For such performance issues, the tests often can detect the performance improvements but only with a *small* or *medium* effect size, which are not considered in our experiments to minimize noise. However, with more repetitions, the effect can increase. For example, in the report of performance issue *CASSANDRA-581*, developers mention that the method *convertFromDiskFormat* using *split* is slow only when being tested with more than 1,000 keys. Although a test *RandomPartitionerTest* covers the code changed by the issue fix, the method *convertFromDiskFormat* is called only once in the test and the elapsed time is slightly improved with a small effect size. Based on the description of the issue report, if there were more repetitions around this method, the performance improvement would be demonstrated by the test.

Race conditions (2 tests). The race condition related performance issues can only happen when given a certain set of circumstances. For example, the commit *#6158c64* fixed the deadlock issue in the streaming code. With the description provided in the report, *CASSANDRA-5699*, we find that we need a specific execution condition to trigger the deadlock.

Limited line coverage of the performance related codes (24 tests). We notice that developers may change a large amount of source code to fix performance issues, but the test only covers a small portion of the committed changes. In this situation, the performance of the test can be misleading since it does not tell the full picture of the issue fixes. For example, the commit *#67ccdab* fixed a performance issue in the streaming code. By using the *git diff* command, we know that there are 10 files changed with 437 line additions and 243 line deletions. However, among these changes, only one line is covered by the test *SessionInfoTest*. Moreover, the covered line is a refactoring operation (Rename Variable), and the performance sensitive operations are never performed by the tests to demonstrate the performance improvement.

Partial branch coverage (34 tests). If the performance issue is caused by the code inside the *if* statement, and without the 100% coverage of the conditions, the code snippets cannot be tested, and thus, the tests cannot demonstrate the fix to the performance issue. A representative example can be found in the fixing process of issue *CASSANDRA-3234*. The performance issue is caused by the

echoedRow function, while this function cannot be invoked as it lies inside the *if* statement without a 100% branch coverage.

Indirect performance influence (1 test). In this situation, the behavior of performance issue related code is based on the return value of another function. Therefore, covering the fix locations of the issue may not be useful to demonstrate the fix to the performance issue. For example, in the fixing process of the issue *CASSANDRA-8550*, while benchmarking *CQL3* secondary indexes, developers noticed substantial performance degradation as the volume of indexed data increases. The issue is caused by the page size selection, which is returned by another function. We notice that the tests can cover the use of the return value while missing its caller. Therefore, the tests cannot demonstrate the performance changes as expected.

Frequent access of external resources (31 tests). Frequent access operations of external resources may introduce noise into the performance evaluation of the tests. We find tests that may have 1) frequent I/O operations, including tables' creation, deletion, update and data insertion and selection, or 2) frequent memory operations, like the *flush* operations. For example, test *DefsTest* covers the fix in commit *#3ad3e73* for the issue *CASSANDRA-3234*. However, the test cannot demonstrate the improvement due to the noise from its large number of flush operations.

Idle during execution (6 tests). Some tests may proactively wait for a period of time, introducing an idle time that is much longer than the actual execution time, which reduces the observed performance improvement after issue fixes. For example, in the commit *#3ad3e73* that fixes issue *CASSANDRA-3234*, test *CleanupTest* contains a 10-second *Thread.sleep* operation with a total 11.685s elapsed test time. In this case, the elapsed time is dominated by the sleep time, hiding the performance improvement after the issue fixes.

We identify eight possible reasons that a test in a release pipeline cannot serve as a performance test. The reasons can be used as a guideline for practitioners to avoid and improve the use of certain tests from the release pipeline.

RQ3: What are the important factors for a test to be useful as a performance test?

Motivation. Prior research has studied the use of micro-scale performance tests in performance evaluation [11, 22, 23, 45]. However, the findings in our prior research questions illustrate the challenges and show the reasons why we cannot directly adopt those tests in performance evaluation. On the other hand, there exist tests from the release pipeline that successfully demonstrate performance improvements. By understanding the characteristics of tests that are able to demonstrate performance improvements, we may gain a better understanding of these tests and thus can provide more general guidance to a developer for writing new tests that run in the release pipeline for performance assurance activities.

Approach. To answer this research question, we adopt random forest, an ensemble learning method [8], as it is one of the most used machine learning algorithms for its performance and has been adopted in various software engineering research [48]. We build a binary classifier to identify whether a test can be used to demonstrate performance improvements.

Step 1: Raw data collection. In RQ1, we have identified the impacted tests of each performance issue, and whether the test can demonstrate performance improvements. However, the ability of a test to serve as a performance test may vary among different performance metrics. For example, a test that can successfully demonstrate memory usage improvement may not be able to show the improvement with elapsed time. Therefore, in this step, we separate the data based on each performance metric, i.e., we build one classifier for each performance metric. For example, to collect the raw data of elapsed time for project *Cassandra*, we first only take all the performance issues that are manually labelled with elapsed time. Then we collect the impacted tests of each performance issue. For each impacted test, we use the results shown in RQ1 to determine whether the test can demonstrate a performance improvement. The results in RQ1 are considered the ground truth data for our classifier.

Step 2: Metrics extraction. To build classifiers, we extract metrics for the raw data collected from the previous step. The effectiveness of a test can be associated with many metrics. In this work, we extract metrics from three aspects of the tests:

- test code, which contains the information about the test itself.
- source code covered by the test, where we can find the test coverage rate and the characteristics of covered source code.
- source code impacted by the issue fix, which measures the characteristics of committed changes of the source code while fixing the performance issue.

The intuition behind the selection of the three aspects is straightforward, as we are running the test to evaluate the performance of the covered source code and the performance improvements from issues fixes should be caused by the committed changes.

Inspired by the work on defect prediction [27, 34, 36, 37], and the prior findings on performance issues and performance regressions [2, 13, 18, 24, 26, 44], we extract metrics from each of the three aspects. Some metrics exists in multiple aspects. The details of the metrics are shown in Table 1.

Step 3: Training and testing random forest classifiers. In this step, we build random forest classifiers to model whether a test can demonstrate the performance improvements or not. In particular, we build five classifiers, each predicting for one performance metric (i.e., elapsed time, CPU usage, memory usage, I/O read and I/O write). For each classifier, we use a 10×10-fold cross-validation implementation in *scikit-learn* [6] with random shuffle [39]. We fit a classifier on the training data, and use the validation data to test the classifier. For our binary classification problem, we use the area under the receiver operating characteristic (ROC) curve (AUC) as a performance measurement [7]. AUC ranges in value from 0 to 1, showing the capacity of the classifier on distinguishing between classes. A higher AUC means a better classifier at predicting. Finally, we have 10×10 models and corresponding AUC values. In this study, we use the random forest implementation [7]

[6] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

[7] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Table 1: An overview of our extracted metrics to build random forest classifiers.

T	S	F	Category	Metrics	Level	Description
•	•	•	Complexity and size	FanOut	Method	Number of unique methods that are called by the code snippet.
•	•	•		FanIn	Method	Number of unique methods that call the methods of the code snippet.
•	•	•		CyclomaticComplexity	Method	McCabe Cyclomatic complexity of the code snippet.
•				SLOC	File	Number of source code lines in the code snippet.
•				CodeElementsSize	Method	Code elements divided by Size.
		•	Diffusion	Entropy	Commit	Distribution of modified code across files in one commit.
•	•	•	History	DeveloperCount	Commit	Number of developers that changed the code snippet.
•	•	•		TimeInterval	File	Average time interval between the last and the current change of the code snippet.
•		•	Human factor	DeveloperCommitCount	File	Average number of commits of the developers who modified the code snippet.
•		•		RecentDeveloperCommitCount	File	Average number of commits made in last 12 months of the developers who modified the code snippet.
•	•	•	Code elements	Condition	Method	Number of condition statements of the code snippet.
•	•	•		Loop	Method	Number of loop statements of the code snippet.
•	•	•		ExceptionHandling	Method	Number of try-catch statements of the code snippet.
•	•	•		Synchronization	Method	Number of synchronization statements of the code snippet.
•	•	•		FinalStatic	Method	Number of final or static statements of the code snippet.
•	•	•		ExpensiveVariableParameter	Method	Number of expensive parameters/variables of the code snippet.
•	•	•		ExternalCall	Method	Number of external function call of the code snippet.
•	•	•		Control	Method	Number of control statements of the code snippet.
•	•		Code change	CodeChurn	File	Total sum of lines added into and deleted from the code snippet across all the commit history.
•	•			LineAdded	File	Total sum of lines added into the the code snippet across all the commit history.
•	•			LineDeleted	File	Total sum of lines deleted from the code snippet across all the commit history.
	•		Coverage criteria	LineCoverage	File	Line coverage ratio of the test.
	•			BrahchCoverage	File	Branch coverage ratio of the test.

Note: T, S and F in the heading are abbreviations for the three aspect of metrics: test code, source code covered by the test and source code impacted by the issue fix. • means that the metric is calculated for the corresponding aspect.

and *roc_auc_score* [8] function in *scikit-learn* [39] to train and evaluate our classifiers. Note that for I/O read of project *Hadoop*, we only have 13 and 345 functional tests that can and cannot demonstrate performance improvements. The dataset is small for training a classifier, resulting in the misleading conclusions. Therefore, we do not train our classifier for I/O read with *Hadoop*.

Step 4: Determining importance of each group of metrics. In this step, we examine the importance of each group of metrics. In particular, we extract three groups of metrics, i.e., fix impacted source code, test code, and test covered source code. We remove each group of metrics from our data and rebuild the classifiers. Afterwards, we measure the AUC values of each classifier and compare with the AUC values of the original classifiers with all metrics. The more the AUC values decrease, the more important the group of metrics are.

Step 5: Determining the importance of each metric. To evaluate the importance of each metric on our random forest classifiers, we adopt the *Mean Decrease Impurity* (MDI) (also called Gini importance) [9, 10]. In a tree algorithm, it calculates each metric's importance as the sum over the number of splits that include the metric, proportionally to the number of samples it splits. For our random forest, the importance is averaged over all trees of the ensemble. We use the function *feature_importances_* of the *scikit-learn* [9] [39] in Python to compute the metrics importance values.

After we repeat the 10-fold cross-validation for 10 times, each metric has 100 importance scores. We then perform Scott-Knott Effect Size Difference (ESD) test [43] on the metrics importance. The Scott-Knott ESD test uses hierarchical clustering analysis to partition different metrics into distinct groups. With this analysis, each metric has a rank. In this study, we use the *sk_esd* function of the ScottKnottESD package [10] in R [47].

Finally, to examine the direction of the relationship between each metric and the likelihood of a test being successful on demonstrating performance improvements, we measure the correlation between each metric and the targets/classes using a Spearman rank correlation (*rho*). A positive Spearman rank correlation indicates that the metric shares a positive relationship with the likelihood of a test being successful on demonstrating performance improvements, whereas a negative correlation indicates an inverse relationship.

Results. **Our random forest classifiers achieve high AUC values, considerably outperforming a random classifier.** For project *Cassandra*, Table 2 shows that, our random forest classifiers achieve an average AUC of 0.86, 0.59, 0.69, 0.72, and 0.73 for elapsed time, CPU usage, memory usage, I/O read and I/O write, respectively. Similarly, for project *Hadoop*, our classifiers achieve an average AUC of 0.90, 0.68, 0.66, 0.79 for elapsed time, CPU usage, memory usage, and I/O write, respectively. These results indicate that our random forest classifiers outperform random classifiers when determine whether a test can be used for demonstrating performance improvements. By analyzing the results, we find that the higher AUC value of elapsed time than the CPU usage, Memory usage, I/O read and I/O write classifiers may be due to the larger number of available tests that can be used to demonstrate improvements in elapsed time over other performance metrics. In addition, we find that the AUC values of all the classifiers are stable, especially the models from the elapsed time. The stable AUC values of our classifiers suggest that our classifiers achieve stable performance in

[8] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html

[9] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.feature_importances_

[10] https://github.com/klainfo/ScottKnottESD

Table 2: An average of AUC, and AUC changes after removing some metrics. −, +, and 0 means there is a decrease, increase and no change of AUC.

		All Metrics	Metrics without source code impacted by the issue fix		Metrics without test code		Metrics without source code covered by the test	
		AUC	AUC	Change	AUC	Change	AUC	Change
Cassandra	Elapsed time	0.86	0.79	**-0.07**	0.85	-0.01	0.8	-0.06
	CPU usage	0.59	0.58	-0.01	0.57	-0.02	0.56	**-0.03**
	Memory usage	0.69	0.67	-0.02	0.67	-0.02	0.64	**-0.05**
	I/O read	0.72	0.68	**-0.04**	0.68	**-0.04**	0.68	**-0.04**
	I/O write	0.73	0.73	0	0.67	**-0.06**	0.7	-0.03
Hadoop	Elapsed time	0.90	0.90	0	0.87	-0.03	0.86	**-0.04**
	CPU usage	0.68	0.68	0	0.59	**-0.09**	0.68	0
	Memory usage	0.66	0.66	0	0.61	**-0.05**	0.67	0.01
	I/O write	0.79	0.79	0	0.74	**-0.05**	0.8	0.01

determining the effectiveness of using these readily available tests in the release pipeline in performance assurance activities.

The metrics extracted from the source code covered by the test play an important role in the usefulness of a test. Table 2 shows that for *Cassandra*, the metrics from the source code covered by the tests always have a strong influence on the AUC values among the classifiers for all performance metrics. Table 3 presents the top three most important metrics to the classifiers. To have a better understanding of these metrics, we also present their metrics importance measured using MDI, the direction (i.e., the sign of ρ) of the relationship between these metrics and the likelihood of a test being successful on demonstrating performance improvements. By examining Table 3, we find that for *Cassandra*, the metrics from the source code covered by the test always have the largest MDI for all classifiers. The *LineCoverage* and *BranchCoverage* metrics lie in the top two ranks across all the classifiers. The results also show that these two metrics have a positive impact on the unit usage, I/O read, and I/O write performance metrics. It indicates that a test tends to successfully demonstrate a performance improvement from a performance issue fix, if the test has a relatively higher line or branch coverage. These findings confirm the results in our preliminary manual study in RQ2, i.e., the tests with a lower line or branch coverage have difficulty triggering the performance issues, thus cannot demonstrate the improvements from the performance issues fixes. This finding suggests the importance of coverage criteria in developing performance tests.

The metrics of the test itself play an important role in the usefulness of a test. Shown in Table 2, for *Hadoop*, the metrics related to the test code have a large influence on all the classifiers. By examining the top three most important metrics to the classifiers (see Table 3), the *Size* and *TimeInterval* metrics from test code and are also important on whether a test can demonstrating performance improvements. For project *Cassandra*, Table 3 shows that *SLOC* metric of the test code ranks first in the I/O write classifier. This *SLOC* metric is also one of the top three important metrics in the elapsed time, CPU usage, memory usage, and I/O read classifiers. The *SLOC* metric has a positive impact in all the five performance metrics. It indicates that a test tends to successfully demonstrate performance improvements, if it has a relatively higher source lines of code. Meanwhile, for project *Hadoop*, the metric *TimeInterval* also lie in the top three most important metrics. The negative sign indicates that if a test code is updated long time ago, it may result in a low likelihood demonstrating the performance improvements. Finally, for *Hadoop*, the importance of the metric *RelativeExpensiveVariableParameter*, from test code, indicates that a readily available test tends to successfully demonstrate the performance improvements from performance issues, especially memory issues , if it has a relatively higher call of expensive variables in the test.

The metrics of the changed source code by a performance issue fix do not often play an important role in the usefulness of a test. We find that for *Hadoop* the average AUC our random forest classifiers do not change when the metrics extracted from the source code impacted by the issue fix category (see Table 2). In addition, none of the metrics that are related to the source code impacted by the issue fix lies in the top three important metrics of the classifiers. These findings suggest that developers may pay more attention to the test code and the source code covered by the test. Some practitioners may like to fine tune the tests for every performance issue fix. However, our results suggest that such fine-tuning may not be cost-effective since the characteristics from the changed source code of a performance issue do not typically play an important role in whether the test can demonstrate the performance improvements from performance issue fixes.

Metrics related to the test itself and the source code covered by the test are important in the classifiers. On the other hand, the metrics related to the code changes in the performance issues fixes have a low importance. Practitioners should focus on designing and improving the tests, instead of optimizing tests for different performance issue fixes.

5 THREATS TO VALIDITY

This section discusses the threats to the validity of our study.

External validity. Due to the large amount of time and computing resources for execution to identify performance improvements and the code coverage of tests, our evaluation is conducted on two open-source software systems, i.e., *Hadoop* and *Cassandra*. Although our study only focuses on 127 performance issues, the scale of our study is comparable to prior research on performance issues [26, 54]. Our findings might not be generalizable to other systems. Future studies

Table 3: Average rank of the top three influential metrics and the Spearman rank correlation (ρ). Note: A + (or −) sign of ρ indicates a positive (or an inverse) relationship of the metric with the likelihood that a functional being able to demonstrate the performance improvements. The larger MDI that a metric has, the more influential the metric is.

Cassandra			
Rank	**Aspect::Metrics**	**MDI±SD**	ρ
Elapsed time			
1	S::LineCoverage	0.068±0.001	+
2	S::BrahchCoverage	0.068±0.001	+
3	T::RelativeExceptionHandling	0.044±0.001	+
CPU usage			
1	S::LineCoverage	0.059±0.002	+
2	S::BrahchCoverage	0.059±0.002	+
3	T::TimeInterval	0.046±0.002	+
Memory			
1	S::BrahchCoverage	0.051±0.001	−
2	S::LineCoverage	0.051±0.001	−
3	T::TimeInterval	0.045±0.001	−
I/O read			
1	S::BranchCoverage	0.060±0.001	+
2	S::LineCoverage	0.059±0.001	+
3	T::TimeInterval	0.048±0.001	+
I/O write			
1	S::BrahchCoverage	0.049±0.002	+
	S::LineCoverage	0.049±0.002	+
	T::SLOC	0.049±0.002	+
2	T::RelativeExpensiveVariableParameter	0.040±0.001	−
3	T::TimeInterval	0.038±0.001	+

Hadoop			
Rank	**Category::Metrics**	**MDI±SD**	ρ
Elapsed time			
1	S::LineAdded	0.038±0.001	+
2	S::TimeInterval	0.037±0.001	−
3	S::LineDeleted	0.033±0.001	+
CPU usage			
1	T::TimeInterval	0.036±0.001	−
2	T::RelativeExceptionHandling	0.033±0.001	+
3	T:RelativeExpensiveVariableParameter	0.032±0.001	+
Memory			
1	T::RelativeExpensiveVariableParameter	0.035±0.001	+
2	S::TimeInterval	0.034±0.001	−
3	T::TimeInterval	0.033±0.001	−
I/O write			
1	S::LineAdded	0.040±0.002	+
2	T::TimeInterval	0.030±0.001	−
3	T::RelativeExpensiveVariableParameter	0.030±0.001	+

Note: T and S in the aspects are abbreviations for the two aspect of metrics: test code and source code covered by the test.

can apply our approach on other systems, such as commercial closed source systems.

Internal validity. Our issue report selection in the *JIRA* tracking system may be biased by the keyword definition. Although we use a manual identification process to verify whether the filtered issue reports are related to performance, we may still miss performance issue that do not contain any of our listed keywords. Our approach requires performance metrics to measure performance of available tests. In particular, we only study five performance metrics while there may be others if other people study and label other performance issues. Future studies can include more performance issues and metrics to complement the findings of our study. The manual labelling and manual study results may be subjective to the two authors. More user studies and surveys on practitioners may address this threat.

We use software metrics based on the findings from prior research and also extract new metrics highly related to test. We choose our prediction model (Random Forest), based on its widespread use in prior software engineering research [20], and since it typically provides a high accuracy in the modeling. There may exist other metrics and machine learning models that can be leveraged in our study, where future research can explore to complement our findings.

Construct validity. There exist other performance assurance activities, such a performance regression detection [21, 32]. Our study chooses to use performance issues because of the knowledge and quality of issue reports and the certainty in performance improvements. Future research can complement our study by using readily available tests in other performance assurance activities as performance tests.

There always exists noise when monitoring performance [35]. In order to minimize such noise, for each readily available test, we repeat the execution 30 times independently. Then we use a statistically rigorous approach to measuring performance improvements. Further studies may opt to increase the number of repeated executions to further minimize the threat based on their time and resource budget. Our approach is based on the system performance that is recorded by *Psutil* [14]. Further studies may evaluate our approach by varying such performance monitoring tools, i.e., *pidStat*.

In our context, we evaluate the performance of tests in a Google Cloud Platform performance evaluation environment. Although we minimize the noise in the environment to avoid bias, such an environment is not exactly the same as in-field environment of the users. To minimize the threat, we only consider the performance improvements that have large effect size. In addition, with the advancing of DevOps, more operational data will become available for future mining software repository research. Research based on field data from the real users can address this threat.

6 CONCLUSION

In this paper, we evaluate the performance of readily available tests in the release pipeline, and then examine whether these tests can be used as performance tests, in particular, to demonstrate the performance improvements from performance issues fixes. By performing an exploratory study on a total of 127 performance issues in two open-source projects, i.e., *Hadoop* and *Cassandra*, we find that most of improvements from performance issues can be demonstrated using the readily available tests in the release pipeline. Moreover, through a manual study, we identify eight reasons that may lead a test not being able to demonstrate the performance improvements. Finally, we build random forest classifiers to identify the most important metrics that influence the tests' capability on demonstrating performance improvements.

To summarize, this paper makes the following contributions:

- To the best of our knowledge, our work is the first to study the use of readily available tests in performance assurance activities.

- We uncover eight reasons why a readily available test cannot be used as a performance test.
- We find that a test itself and the source code covered by the test are the important factors for tests to be able to serve as performance tests.

Our findings shed light on the opportunities and challenges in leveraging the readily available tests in performance assurance activities. Practitioners can use our uncovered reasons and factors as guidelines to design and improve tests that run in the release pipeline for performance assurance activities.

REFERENCES

[1] [n. d.]. Apache JMeter - Apache JMeter. https://jmeter.apache.org/. (Accessed on 03/29/2019).

[2] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 298–313. https://doi.org/10.1145/3064176.3064186

[3] Hammam M. Alghmadi, Mark D. Syer, Weiyi Shang, and Ahmed E. Hassan. 2016. An Automated Approach for Recommending When to Stop Performance Tests. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 279–289. https://doi.org/10.1109/ICSME.2016.46

[4] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Trans. Software Eng.* 40, 11 (2014), 1100–1125. https://doi.org/10.1109/TSE.2014.2342227

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2015. How (Much) Do Developers Test?. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 559–562. https://doi.org/10.1109/ICSE.2015.193

[6] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Monica Villavicencio, Jürgen Walter, and Felix Willnecker. 2019. How is Performance Addressed in DevOps?. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*. ACM, New York, NY, USA, 45–50. https://doi.org/10.1145/3297663.3309672

[7] Andrew P. Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition* 30, 7 (1997), 1145–1159. https://doi.org/10.1016/S0031-3203(96)00142-2

[8] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[9] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[10] Leo Breiman. 2002. Manual on setting up, using, and understanding random forests v3. 1. *Statistics Department University of California Berkeley, CA, USA* (2002).

[11] Lubomír Bulej, Tomás Bures, Vojtech Horký, Jaroslav Kotrc, Lukás Marek, Tomás Trojánek, and Petr Tuma. 2017. Unit testing performance with Stochastic Performance Logic. *Autom. Softw. Eng.* 24, 1 (2017), 139–187. https://doi.org/10.1007/s10515-015-0188-0

[12] Ruth Cano-Corres, Javier Sánchez-Álvarez, and Xavier Fuentes-Arderiu. 2012. The effect size: beyond statistical significance. *EJIFCC* 23, 1 (2012), 19.

[13] Jinfu Chen and Weiyi Shang. 2017. An Exploratory Study of Performance Regression Introducing Code Changes. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. 341–352. https://doi.org/10.1109/ICSME.2017.13

[14] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2016. CacheOptimizer: helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 666–677. https://doi.org/10.1145/2950290.2950303

[15] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 1001–1012. https://doi.org/10.1145/2568225.2568259

[16] Norman Cliff. 1996. Ordinal methods for behavioral data analysis. (1996).

[17] Robert Coe. 2002. It's the effect size, stupid: What effect size is and why it is important. (2002).

[18] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, Walter Binder, Vittorio Cortellessa, Anne Koziolek, Evgenia Smirni, and Meikel Poess (Eds.). ACM, 389–400. https://doi.org/10.1145/3030207.3030221

[19] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. 2018. Word embeddings for the software engineering domain. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 38–41. https://doi.org/10.1145/3196398.3196448

[20] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. 2015. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 789–800. http://dl.acm.org/citation.cfm?id=2818754.2818850

[21] Christoph Heger, Jens Happe, and Roozbeh Farahbod. 2013. Automated Root Cause Isolation of Performance Regressions During Software Development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2479871.2479879

[22] Vojtěch Horký, František Haas, Jaroslav Kotrč, Martin Lacina, and Petr Tůma. 2013. Performance Regression Unit Testing: A Case Study. In *Computer Performance Engineering - 10th European Workshop, EPEW 2013, Venice, Italy, September 16-17, 2013. Proceedings (Lecture Notes in Computer Science)*, Maria Simonetta Balsamo, William J. Knottenbelt, and Andrea Marin (Eds.), Vol. 8168. Springer, 149–163. https://doi.org/10.1007/978-3-642-40725-3_12

[23] Vojtech Horký, Peter Libic, Lukás Marek, Antonín Steinhauser, and Petr Tuma. 2015. Utilizing Performance Unit Tests To Increase Performance Awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*. 289–300. https://doi.org/10.1145/2668930.2688051

[24] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance regression testing target prioritization via performance risk analysis. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 60–71. https://doi.org/10.1145/2568225.2568232

[25] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Trans. Software Eng.* 41, 11 (2015), 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

[26] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 77–88. https://doi.org/10.1145/2254064.2254075

[27] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Software Eng.* 39, 6 (2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[28] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software Microbenchmarking in the Cloud. How Bad is it Really? *Empirical Software Engineering* (2019), 1–46.

[29] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*. 373–384. https://doi.org/10.1145/3030207.3030213

[30] Meng-Hui Lim, Jian-Guang Lou, Hongyu Zhang, Qiang Fu, Andrew Beng Jin Teoh, Qingwei Lin, Rui Ding, and Dongmei Zhang. 2014. Identifying Recurrent and Unknown Performance Issues. In *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*. 320–329. https://doi.org/10.1109/ICDM.2014.96

[31] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2016. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. 25–36. https://doi.org/10.1145/2901739.2901765

[32] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2016. Mining Performance Regression Inducing Code Changes in Evolving Software. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2901739.2901765

[33] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. 2013. Automatic detection of performance deviations in the load testing of large scale systems. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 1012–1021. https://doi.org/10.1109/ICSE.2013.6606651

[34] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180. https://doi.org/10.1002/bltj.2229

[35] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March*

7-11, 2009. 265–276. https://doi.org/10.1145/1508244.1508275

[36] Nachiappan Nagappan and Thomas Ball. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, Washington, DC, USA, 364–373. https://doi.org/10.1109/ESEM.2007.87

[37] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining Metrics to Predict Component Failures. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*. ACM, New York, NY, USA, 452–461. https://doi.org/10.1145/1134285.1134349

[38] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: detecting performance problems via similar memory-access patterns. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 562–571. https://doi.org/10.1109/ICSE.2013.6606602

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[40] Stuart Reid. 2005. The Art of Software Testing, Second edition. Glenford J. Myers. Revised and updated by Tom Badgett and Todd M. Thomas, with Corey Sandler. John Wiley and Sons, New Jersey, USA, 2004, ISBN 0-471-46912-2. *Softw. Test., Verif. Reliab.* 15, 2 (2005), 136–137. https://doi.org/10.1002/stvr.322

[41] Giampaolo Rodola. 2016. Psutil package: a cross-platform library for retrieving information on running processes and system utilization. https://github.com/giampaolo/psutil

[42] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *annual meeting of the Florida Association of Institutional Research*. 1–33.

[43] AJ Scott and M Knott. 1974. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* (1974), 507–512.

[44] Linhai Song and Shan Lu. 2017. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 370–380. https://doi.org/10.1109/ICSE.2017.41

[45] Petr Stefan, Vojtech Horký, Lubomír Bulej, and Petr Tuma. 2017. Unit Testing Performance in Java Projects: Are We There Yet?. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*. 401–412. https://doi.org/10.1145/3030207.3030226

[46] Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, and Ahmed E. Hassan. 2017. Continuous validation of performance test workloads. *Autom. Softw. Eng.* 24, 1 (2017), 189–231. https://doi.org/10.1007/s10515-016-0196-8

[47] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Trans. Software Eng.* 43, 1 (2017), 1–18. https://doi.org/10.1109/TSE.2016.2584050

[48] Patanamon Thongtanunam, Weiyi Shang, and Ahmed E. Hassan. 2019. Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones. *Empirical Software Engineering* 24, 2 (2019), 937–972. https://doi.org/10.1007/s10664-018-9645-2

[49] Nikolai Tillmann and Wolfram Schulte. 2006. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE Software* 23, 4 (2006), 38–47. https://doi.org/10.1109/MS.2006.117

[50] Elaine J. Weyuker and Filippos I. Vokolos. 2000. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Trans. Software Eng.* 26, 12 (2000), 1147–1156. https://doi.org/10.1109/32.888628

[51] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Bo Zhou. 2015. Automatic, high accuracy prediction of reopened bugs. *Autom. Softw. Eng.* 22, 1 (2015), 75–109. https://doi.org/10.1007/s10515-014-0162-2

[52] PengCheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. 2013. vPerfGuard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013*. 271–282. https://doi.org/10.1145/2479871.2479909

[53] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar (Eds.). ACM, 127–138. https://doi.org/10.1145/3184407.3184416

[54] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A qualitative study on performance bugs. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*. 199–208. https://doi.org/10.1109/MSR.2012.6224281

[55] H. Zhang, S. Wang, T. P. Chen, Y. Zou, and A. E. Hassan. 2019. An Empirical Study of Obsolete Answers on Stack Overflow. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2906315