PerfJIT: Test-level Just-in-time Prediction for Performance Regression Introducing Commits

Jinfu Chen, *Student Member, IEEE,* Weiyi Shang, *Senior Member, IEEE,* Emad Shihab, *Senior Member, IEEE*

Abstract—Performance issues may compromise user experiences, increase the cost resources, and cause field failures. One of the most prevalent performance issues is performance regression. Due to the importance and challenges in performance regression detection, prior research proposes various automated approaches that detect performance regressions. However, the performance regression detection is conducted after the system is built and deployed. Hence, large amounts of resources are still required to locate and fix performance regressions. In our paper, we propose an approach that automatically predicts whether a test would manifest performance regressions given a code commit. In particular, we extract both traditional metrics and performance-related metrics from the code changes that are associated with each test. For each commit, we build random forest classifiers that are trained from all prior commits to predict in this commit whether each test would manifest performance regression. We conduct case studies on three open-source systems (*Hadoop*, *Cassandra* and *OpenJPA*). Our results show that our approach can predict tests that manifest performance regressions in a commit with high AUC values (on average 0.86). Our approach can drastically reduce the testing time needed to detect performance regressions. In addition, we find that our approach could be used to detect the introduction of six out of nine real-life performance issues from the subject systems during our studied period. Finally, we find that traditional metrics that are associated with size and code change histories are the most important factors in our models. Our approach and the study results can be leveraged by practitioners to effectively cope with performance regressions in a timely and proactive manner.

Index Terms—performance regression, software performance, software quality, mining software repositories, empirical software engineering

1 Introduction

Performance assurance activities are an essential step in the development cycle of large software systems [1]. One of the goals of performance assurance activities is to detect performance regressions, i.e., the performance of the same feature in the system is worse than before. Response time degradation and increased CPU usage are typical examples of performance regressions. These performance regressions may directly affect the user experience, increase the resources cost of the system and lead to reputational repercussions. Therefore, detecting and resolving performance regressions is an important task even though the system's performance may meet the requirement. For example, Mozilla has a performance regression policy that requires performance regressions to be reported and resolved as bugs [2].

Although automated techniques are proposed to detect performance regressions [3], [4], [5], [6], [7], challenges in the practice of performance regression detection still exist. First of all, performance regressions detection remains a task that is conducted after the system is developed and built, as almost the last step in the release cycle. Therefore, fixing performance regressions at such a late stage in the development cycle is difficult and sometimes impossible. Second, a significant amount of effort is required to locate the root-cause of the performance regression after detection.

In order to detect performance regressions, prior research studies performance regressions at the code commit level [8]. In particular, various performance regressions are detected by running all the readily available tests for the software systems. Such knowledge may assist in addressing the aforementioned challenges. In particular, developers are notified about the introduction of a performance regression after the code commit, developers may address such regression in a timely manner, avoiding other code changes depending the performance-regression-introducing commit. More importantly, such prediction would ease the developer to locate the root-cause of the performance regression, and further address the performance regression.

Taking a real-life example in *Hadoop*, issue *YARN-4862*[1] is a performance issue with major priority. The performance regression was introduced in a code commit[2] that was performed half a year before the reporting date of the issue. However, if immediately after the code commit, the developer was notified that a performance regression might have been introduced into the source code that is executed by test *TestResourceTrackerService*, this major issue may not be hidden for such a long time.

Unfortunately, running all tests to detect performance regressions is an extremely time-consuming task, especially for every commit. From our study results, on average to detect performance regression by rigorously running all tests that are impacted by the code changes takes hours per commit (c.f. Table 7). Recent work by Oliveira et al. [9] proposed an approach named *Perphecy*, which aims to select

• *Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.*
E-mail: (fu_chen, shang, eshihab)@encs.concordia.ca

1. https://issues.apache.org/jira/browse/YARN-4862
2. https://github.com/apache/hadoop/commit/528b809d. A test in this commit is successfully predicted by our approach (cf. Section 5-RQ4) to manifest performance regression with slower response time.

978-1-5090-1629-7/16/$31.00 © 2019 IEEE

performance benchmark suites that may manifest performance regressions. While shown to be effective, *Perphecy* depends on a short list of boolean indicators whose thresholds are tuned from prior executions. Due to the observation in the study that only a very small number of performance benchmarks in the commits contains performance regressions, the paper proposes a conservative approach for the tuning, i.e., detecting more benchmarks to achieve higher recall [9]. However, our prior study shows that when examining the performance regressions at the test level, a rather large number of performance regressions can be detected. Using conservatively tuned thresholds on such large number of performance regressions at the test level may lead to a large number of false-positive results (as also shown in our results in RQ1 and RQ4). A high false positive rate may lead to a large number of tests that need to run, making the process still a time-consuming task.

Therefore, in this paper, we present an automated approach that builds just-in-time prediction models to predict the tests that manifest performance regressions with a given code commit. We call such tests performance-regression-prone tests. In other words, our approach predicts whether there is regression in a *particular test* of a particular commit (not any test in a commit). To build the prediction model, we first identify performance-regression-prone tests by evaluating all the tests that are impacted by the code changes in prior commits and measuring performance (i.e., response time, CPU and memory usage, I/O read and I/O write) by running each test repetitively. Afterwards, we build five classifiers, one for each performance metric, modeling whether a test would manifest a performance regression. With such prediction models, after developers commit their code changes, our approach can automatically predict which tests manifest performance regressions for each particular performance metric. Developers can prioritize their performance assurance activities by only running the predicted performance-regression-prone tests to address the performance regressions. For the tests that are executed and shown to have regression, the developers may check the covered and changed source code of each test to start their investigation. The newly executed performance evaluation results are then included in the training data to update the classifiers in order to predict the performance-regression-prone tests in the next new commit.

To evaluate our approach, we conduct case studies on three open source systems[3], namely *Hadoop*, *Cassandra* and *OpenJPA*. In particular, we aim to answer five research questions:

RQ1 *How well can we predict performance-regression-prone tests?*
Our random forest classifiers can provide the accurate prediction, outperforming logistic regression, support vector machines and XGBoost, with an average AUC of 0.85, 0.87 and 0.88 for *Hadoop*, *Cassandra* and *OpenJPA*, respectively. Our approach also out-performs the state-of-the-art approach *Perphecy* in all the studied subjects.

RQ2 *How cost-effective is the prioritization of performance-regression-prone tests?*
We consider the time to spent for each test as the cost and build cost-aware models for performance-regression-prone tests. Our models provide high cost-effectiveness prioritization, that is close to the optimal prioritization. By spending only 5% of the cost for executing all tests, our approach can help detect up to 65% of the performance-regression-prone tests.

RQ3 *How much testing time can our approach save?*
Our prediction drastically reduces the testing time needed compared with running only the tests that are impacted by the code changes in a commit (up to 97%).

RQ4 *Can our approach detect the introduction of real-life performance issues?*
Our approach can be used to detect six out of nine real-life performance issues that are found to be introduced during the studied period of our evaluation. The predicted tests by our approach can cover the changed source code that introduces the performance issues.

RQ5 *What are the most important factors in determining performance-regression-prone tests?*
We find that performance-related metrics are not important factors of performance-regression-prone tests, while traditional metrics that are associated with history and size of the code changes are the most important factors.

Based on our case study results, developers can adopt our approach in practice to provide accurate prediction on evaluating performance impact of their code changes in a timely and proactive manner.

The rest of this paper is organized as follows: Section 2 presents prior research related to this paper. Section 3 presents our approach of predicting performance-regression-prone tests. Section 4 presents our subject systems and how to extract performance-regression-prone tests as our ground truth. Section 5 presents the results of our case studies. Section 6 discusses our learned lessons. Section 7 presents the threats to the validity of our study. Finally, Section 8 concludes this paper.

2 Related Work

In this section, we present the related prior research to this paper in three aspects: 1) software defect prediction and 2) empirical studies on software performance and 3) test case prioritization.

2.1 Software defect prediction

Mockus and Weiss [10] are the first ones to use metrics that describes the characteristics of software changes to build linear regression model to predict the probability of failure after code changes. Kamei et al. [11], [12], [13] conduct a series of studies on commit-level defect prediction. Kamei et al. [11] extract 14 change metrics from six open-source systems and five commercial systems and build a logistic regression model to predict whether the commit would introduce software defects. The prediction model built is able to predict defect-inducing changes with an accuracy

3. https://users.encs.concordia.ca/~fu_chen/data

of 0.68 and a recall of 0.64. Their findings also show that diffusion and purpose of the code changes play important roles in defect-inducing changes.

Kamei et al. [13] present an overview in the research field of defect prediction. To build the prediction model, researchers need to extract and select a list of metrics. Zhang et al. [14] use code metrics, process metrics, and context factors to build a universal defect prediction model for a large set of systems. Zhang et al. find that adding context factors can assist in the prediction of software defect of the universal models. Shivaji et al. [15] realize that the more features prediction model learned, the more insufficient performance the model predicts. Hence, Shivaji et al. perform multiple feature selection algorithms to reduce the metrics that predict software defects. He et al. [16] study the feasibility of defect-predictions built with simplified metrics in different scenarios, and offer suggestions on choosing datasets and metrics.

There exist various kinds of prediction models to predict software defects. Tourani and Adams [17] build logistic regression models to study the impact of human discussion metrics on commit-level predicting models. The result shows that there exists a strong correlation between human discussion metrics and defect-prone commits. Tsakiltsidis et al. [18] use four machine learning algorithms to build classifiers to predict performance bugs. The study finds that the most satisfying model is based on logistic regression with all attributes added. Yang et al. [19] use 14 code-change based metrics to build simple unsupervised and supervised models to predict software defect. The results show that many simple unsupervised models perform better than the state-of-the-art supervised models in effort-aware commit-level defect prediction.

Compared to the above papers, we build classifiers to predict tests that manifest performance regressions. We extract traditional metrics that are widely used in prior studies [10], [11] and also include performance-related metrics in our classifiers.

2.2 Empirical studies on software performance

Various empirical studies are conducted to study performance issues. Jin et al. [20] study 109 real-world performance bugs that are derived from five software systems to learn guidance for software practitioners. The study shows that developers need tool support to fix the similar performance issues automatically. The study calls for in-depth research on performance diagnosis, performance testing, performance issue detection. Zaman et al. [21], [22] conduct both qualitative and quantitative studies on 400 performance and non-performance issues. Zaman et al. find that developers spend more time fixing performance issues than non-performance issues. This study also advocates the importance of identifying the root cause of performance issues. Han et al. [23] study 300 bug reports from three large open source projects. The authors find that performance bugs require specific input parameters to expose.

Huang et al. [24] study real-world performance issues and propose a lightweight approach named performance risk analysis (PRA) to improve performance regression testing efficiency. The analysis statically evaluates the risk of introducing performance regression. Pradel et al. [25] present an automatic approach named SpeedGun to detect performance regressions in the case of concurrent classes. This approach intends to generating multi-threaded performance tests to test and report the performance differences between two versions of concurrent classes. Tarvo and Reiss [26] build models to predict the performance of multi-threaded programs or individual program using computer simulation. The authors collect program information using static and dynamic analyses, and simulate each thread using a probabilistic call graph. Luo et al. [3] propose an approach to mine performance regressions inducing code changes. The paper implements a tool named *PerfImpact* to find the inputs that lead to performance regressions and rank the code changes that may be closely responsible to performance regressions. Oliveira et al. [9] present a lightweight tool named *Perphecy* to detect which commit will cause performance regression on which benchmark. Leitner et al. [27] aim to understand the current state of art of performance testing. They conduct a study on 111 open-source java-based systems from GitHub to investigate the use of performance tests across five perspectives. Leitner et al. find that there is a lack of standard guideline to conduct performance tests in an easy and powerful way. This paper argues that the future performance testing should implement more flexible testing framework to support low-friction testing.

Prior research on performance are typically based on either limited issue reports or releases of the software. However, performance regressions may not be defects, and this paper is the first (to the best of our knowledge) to predict performance-regression-prone tests at the commit level. In addition, in this paper, our extracted performance-related metrics are built on top of the findings from the prior studies on performance issues.

2.3 Test case prioritization

The goal of Test Case Prioritization (TCP) is to select the most appropriate tests to execute and enable faster feedback [28]. Various prior research has been proposed to improve test case prioritization [29], [30], [31], [32], [33]. Wang et al. [30] present a quality-aware technique namely *QTEP* to prioritize tests by giving more weight to fault prone source code. This paper shows that *QTEP* can improve existing coverage-based TCP techniques. PerfRanker [31] was proposed to prioritize performance test cases for collection-intensive software. The approach profiles software using dynamic call graph to build performance model and analyzes performance impact introduced by code changes. To address the coverage profiling overhead, Saha et al. [33] present an information retrieval approach namely *REPiR* based on program changes, e.g., identifier names and comments in the code.

Historical information produced by the test can also be used to improve test case prioritization [34], [35], [36], [37], [38]. Kim et al. [34] build models based on the test execution history to prioritize tests. The prioritization models assign a selection probability to each test case in test suite. Anderson et al . [35] investigate the test historical information including test case pass, fail information, code change information. The authors use such historical metrics to build classification model to predict future test case failures. Noor and

Hemmati [36] use a logistic regression model to predict the failing test cases for tests prioritization based on a set of test quality metrics, e.g., change-related metrics. Zhu et al. [37] propose an approach *CODYNAQ* to use the historical co-failure distributions among tests to re-prioritize tests after each test run. Najaf et al . [38] customize and combine multiple test selection and prioritization techniques in a large industrial system based on test execution history information. The authors find that simple approaches are often shown effective in an industrial setting.

Prior research on test case prioritization typically focus on validating functional bugs rather than performance issues. Our paper aims to prioritize tests for a cost-effective detection of performance regressions.

3 Approach

In this section we present our approach of predicting performance-regression prone tests in each commit. In other words, when developers commit their code changes, our approach predicts which *tests* are likely to manifest performance regressions. Developers can prioritize to execute the tests that are predicted to have performance regression. The overview of our approach is shown in Figure 1.

3.1 Extracting metrics

To build classifiers, we first extract metrics from the Git repositories of our subject systems. Prior research on commit-level defect prediction extracts metrics that describe characteristics of the commit. On the other hand, our approach predicts the performance-regression-prone tests in each commit. Hence, our extracted metrics are at the test-level, i.e., the metrics measure the code changes from a commit associated with each test. However, not all the code changes in a commit are associated with all the tests.

For example, there may exist two tests, i.e., Test A and Test B, and a commit has a total code churn of 100 lines of code. Our code churn metric for Test A and Test B for this commit is not simply 100. We check the amount of code churn that are covered by executing Test A (e.g., 60 lines of code) and the amount of code churn that are covered by executing Test B (e.g., 50 lines of code). We note that, these two pieces of code churn can overlap, since some code churn may be covered by both Test A and B. Then for this commit, the code churn metric for Test A is 60 and the code churn metric for Test B is 50.

Therefore, we first need to identify code changes in each commit that are associated with each test. We leverage the mapping between code and test that is created in Section 4.2 to identify the code changes in each commit. Due to the resource needed of generating such mapping, we only generate these mappings once per release. The overview of our extracted metrics is shown in Table 1.

Traditional metrics. Prior studies on commit-level defect prediction leverage various metrics to predict the risk of a code commit [10], [11]. Similarly, we extract 16 traditional metrics from six dimensions, i.e., size of the change, complexity of the changed files, diffusion, development history, developers' experiences and the purpose of the commit. The details of the metrics are shown in Table 1.

Performance-related metrics. We aim to predict performance-regression-prone tests for each commit. Hence, based on findings from prior research on performance issues and performance regressions [8], [20], [24], [47], [48], [49], we extract metrics that describe the code changes in a commit that may related to performance. For example, adding *synchronization* into the source code is one of the root-causes of performance regressions [8], [20]. Considering the findings from prior research, we extract seven groups performance-related metrics. The rationale of extracting each type of entity is shown in Table 1.

All the performance-related metrics are calculated only from the code changes that are associated with each test in a commit. To automatically analyze the code changes, we use *srcML* [50] to convert the source code to XML files representing their abstract syntax trees. We use *lxml* [51] to compare the two xml trees to extract the added, deleted and changed code entities. In particular, the metrics are calculated in the following aspects.

Basic code entity change. We count the added and deleted code entities including *final*, *static*, *try*, *catch*, *throw*, *throws*, *finally*, *break*, *continue*, *label*. Afterwards, we generate two metrics for each type of code entity, i.e., one metric for added entity and one metric for deleted entity. If the code entity can contain expressions, we also generate a metric for the changes to the code entity.

Synchronization. We measure the added and deleted statements that are associated with synchronization. In particular, Java has two kinds of expressions related to synchronization, i.e., *synchronized* statement and *synchronized* specifier. We consider both expressions as synchronization and generate two metrics for added and deleted synchronization expressions, respectively.

Condition. We calculate *condition* metrics from the following statements in Java: *if*, *elseif*, *else*, *switch*, *case* and *assert* statements. We generate two metrics for added and deleted condition, respectively.

Loop. We consider all kinds of loops in Java, such as *for*, *while*, *foreach* and *do while*. Besides generating two metrics for added and deleted loops, we also generated a metric for changed loops, such as changing the expressions in the loop.

Expensive variable change (expVariable). We count the variables that are changed from primitive data type to reference data type for this metric, as expensive variable change proposed by prior research [49].

Expensive parameter change (expParameter). Similar to expensive variable change, we count the method invocation parameters that are changed from primitive type to reference type for this metric.

External function call (externalCall). Function calls that access external resources may introduce performance overhead, leading to performance regression. For example, adding logging statements to print the execution information to a file may introduce performance regressions. In this paper, particularly, we only consider the function calls to the logging library. We generate three metrics for added, deleted and changed external function calls.

3.2 Data preprocessing

Before leveraging the extracted data to build classifiers, we preprocess the data. Prior research shows that multi-

TABLE 1
Summary of extracted metrics. The symbol † indicates metrics that represent multiple metrics.

Dim.	Metric	Definition	Values	Rationale
Diffusion	NS	Number of modified subsystems	Numerical	The more subsystems are changed, the higher risk the change may be [10].
	ND	Number of modified directories	Numerical	Changing more directories may more likely introduce performance regressions [10].
	NF	Number of modified files	Numerical	Changing many source files are more likely to cause performance regression [39].
	NM	Number of modified methods	Numerical	Changes altering many methods are more likely introduce performance regression [40].
	Entropy	Distribution of modified code across files	Numerical	Scattered changes are more possible to be performance-regression-prone [41].
Size	LA	Lines of code added in all the tested methods	Numerical	The more lines of code added, the higher risk that the program will suffer from performance regression [42].
	LD	Lines of code deleted in all the tested methods	Numerical	The more lines of code deleted, the higher risk of performance regression is introduced [42].
	LT	Lines of code before the change in all the tested methods	Numerical	Modifying a large method is more likely to introduce performance regression due to the large method being more complex [40], [43].
	SOL	Lines of code before the change in all the tested classes	Numerical	Large class is more complex than the small class, and may be more performance-regression-prone [43].
Purpose	FN	Whether the code commit fixes a bug	Boolean	Bug fixing changes are more likely to introduce performance regressions [8], [44].
History	NDEV	Number of developers that changed the modified files	Numerical	Changes involving many developers are more possible to cause regression due to the differences between different developers [45].
	AGE	The average time interval between the last and the current change	Numerical	More recent and frequent changes are more likely to introduce performance regression [46].
Experience	EXP	Developer experience	Numerical	Senior programmers may introduce more stable code change than a less experienced developer [10].
	REXP	Recent developer experience	Numerical	Recent developers are more familiar with the program so that it is less likely to introduce performance regression [10].
Complexity	MCC	McCabe Cyclomatic complexity	Numerical	Program with higher complexity is more likely to suffer from performance regression [41].
	FanIn	Number of calling subprograms	Numerical	Large calling subprograms will amplify the regression if there exists performance regression in the called program [39].
Performance-related metrics	†Basic code changes	Number of added or deleted on the basic code entity in method level. (e.g., final_add and final_del of final basic code entity)	Numerical	Changes on the basic code entities may directly increase the complexity of the code and introduce performance regressions.
	†Synchronization	Number of added or deleted synchronization expressions in method level	Numerical	Synchronizations are expensive actions for software performance [47].
	†Condition	Number of added or deleted condition statement in method level	Numerical	Changing condition may cause more operation eventually executed by the software, leading to performance regression [24].
	†Loop	Number of added, deleted or changed loop statement in method level	Numerical	Changes involving the loop may significantly slow down performance [48].
	†ExpVariable	Number of added or deleted expensive variable in method level	Numerical	Some variables are more expensive to be held in memory and need more resources to visit or operate [24], [49].
	†ExpParameter	Number of added or deleted expensive parameter in method level	Numerical	Using more expensive parameter, like reference type other than primitive type, leading to performance regression [8].
	†ExternalCall	Number of added or deleted external function call in method level	Numerical	Some code changes that introduce new functionality and external operations may cause performance regression [8], [20].

collinearity data and redundant metrics may be harmful in interpreting prediction models. In addition, multicollinearity may introduce bias when using the models to explain a phenomenon [52], [53]. To deal with multicollinearity, we first perform a correlation analysis on the metrics in order to remove the most highly correlated metrics. We used Pearson's correlation [54] coefficient among all metrics to find the pair of metrics that have a correlation higher than 0.7. From these two metrics, we remove the metric that has a higher average correlation with all other metrics. We repeat this step until there exists no correlation higher than 0.7 [55]. We then perform redundancy analysis on the metrics. The redundancy analysis would consider a metric redundant if it can be predicted from a combination of all other metrics [56]. We use each metric as a dependent variable and use the rest of the metrics as independent variables to build a regression model. We calculate the R^2 of each model and if the R^2 is larger than a threshold (0.9) [57], the current dependent variable is considered redundant. We then remove the metric with the highest R^2 and repeat the process until no metric can be predicted with R^2 higher than the threshold.

3.3 Building classifiers and predicting performance-regression-prone tests

In this step, we build classifiers to model whether a test in a commit would manifest performance regressions. In particular, we build five classifiers, each predictor for one performance metric (i.e., response time, CPU usage, memory usage, I/O read and I/O write). We use data from prior commits to train the classifiers. The dependent variable of each classifier is whether the test manifests a performance regression with that particular performance metric (see Table 4). The independent variables are based on the metrics that are presented in Section 3.1. All the metrics are preprocessed as described in Section 3.2.

Projects may not have readily available historical performance evaluation results on prior commits to build classifiers. To "cold start" our approach, we require to exercise performance evaluation on the prior 50 commits to make the first set of training data [58]. The details of the performance evaluation are presented in Section 4.2. We choose 50 commits because of the amount of metrics that we have. Fewer commits may result into over-fitting the classifier, while more commits may waste resources on the performance evaluation.

With a new commit, our approach leverages the five built classifiers (each classifier for one performance metric) to predict which tests are likely to manifest performance regressions.

3.4 Exercising tests and updating classifiers

After having the prediction results from our classifiers, developers can choose to only exercise the tests that are predicted to be performance-regression-prone for performance evaluation (c.f., Section 4.2). After the performance evaluation of the tests, the training data for the classifiers are updated by including the results of the newly evaluated tests. Afterwards, the five classifiers are re-built for the prediction of the next commit. Moreover, the practitioners can use the performance evaluation results of the tests assist in locating the root causes of the performance regressions. In particular, developers may use the changed source code that is also executed by the test to assist in locating the root causes of the performance regression.

4 Evaluation Setup

In this section we present the setup of our case study. We present the subject systems that are used and detail how to extract the tests that manifest performance regressions in each commit as our ground truth.

4.1 Subject systems

We choose three open-source systems, *Hadoop*, *Cassandra* and *OpenJPA* as the subject systems of our case study. *Hadoop* [59] is a distributed system infrastructure. *Hadoop* performs data processing in a reliable, efficient, high fault tolerance, low cost and scalable manner. *Cassandra* is an open-source distributed NoSQL database management system. *OpenJPA* is an open source system from the Apache organization that implements the JPA standard. We choose the three subject systems since they are highly concerned with their performance and have been studied in prior research in mining performance data [60], [61]. The overview of the three subject systems is shown in Table 2.

TABLE 2
Overview of our subject systems.

Subjects	#release	#commit	The starting and ending releases	#Total lines of code (K)	#files	#tests
Hadoop	11	1,403	2.6.0 – 2.7.5	970	6,373	1,853
Cassandra	9	902	3.0.7 – 3.0.15	346	1,867	369
OpenJPA	4	726	2.3.0 – 2.4.2	429	4,579	916

4.2 Extracting performance-regressions-tests in each commit

In this subsection, we present how we perform performance evaluation to extract performance-regressions-tests in each commit. Such extracted data is used as our ground truth in the case study. The role of this data is similar to the extracted "bug-fixing commits" in a commit-level bug prediction approach [11]. In addition, to "cold start" projects that do not have performance evaluation results on prior commits, this subsection describes how to obtain initial training data for our approach.

Filtering commits. We start our approach to filter commits by only keeping the commits that have source code changes, i.e., changes to *.java* files. To accomplish one development task, multiple commits, including temporary commits, may be made. We would like to avoid considering such temporary commits. Since all our subject systems use JIRA as their issue tracking systems, we use the issue id mentioned in their commit messages to identify commits that belong to the same issue. If multiple commits are associated with the same issue, we only consider the last commit.

Identifying impacted tests. Our approach considers using all the functional test that exist in the repository. We use these tests since these tests are maintained and typically executed regularly during every build in the release pipeline of software development [62]. Not all the tests are impacted by the code changes in a commit and running those unimpacted tests is not likely to detect performance regressions. To identify all the impacted tests by each commit, we create mappings between source code and tests. We automatically instrument all the methods in every version of the source code of our subject systems by adding invocation to logging libraries. We run all the available tests of each released version of the subject systems. By analyzing the output of our instrumentation, we obtain a list of methods that are executed during the running of each test. Then, we can create mappings between each test and the executed methods of the test. With such a mapping between tests and methods in the source code, for each commit, we can identify the tests that are likely to be impacted by identifying the methods that are changed in the commit, i.e., a method-to-test mapping for each commit. Due to the resources needed for creating such mappings, we only update such mappings for every release of the subject systems.

Dealing with changed tests. Some commits may change both source code and test code. The changes to the test code may bias the performance evaluation. Therefore, we opt to use the test code before the code change, since the new version of the test code may execute new features, which is not the major concern of performance regression. In the cases where old test cases cannot compile or fail, we use the new test code. Finally, if both new and old test cases are failed or not compilable, we do not include this test in the performance evaluation. In total, we have 226 tests in 121 commits that are evaluated with the new tests and 48 tests in 35 commits that are not included in our performance evaluation.

Evaluating performance. Finally, we exercise the selected tests of each pair of current and parent commits to evaluate their performance. Our performance evaluation environment is based on Microsoft Azure node type Standard F8s (8 cores, 16 GB memory). In order to generate statistically rigorous performance results, we adopt the practice of repetitive measurements [63] to evaluate performance. Conservatively, we executed each test 30 times independently, since prior research often only repeat the tests 5 to 20 times [64], [65], [66]. We use a performance monitoring software named *psutil* [67] to collect performance metrics during the execution, i.e., response time, CPU usage, memory usage, I/O read and I/O write.

Statistical analyses for labeling performance evalua-

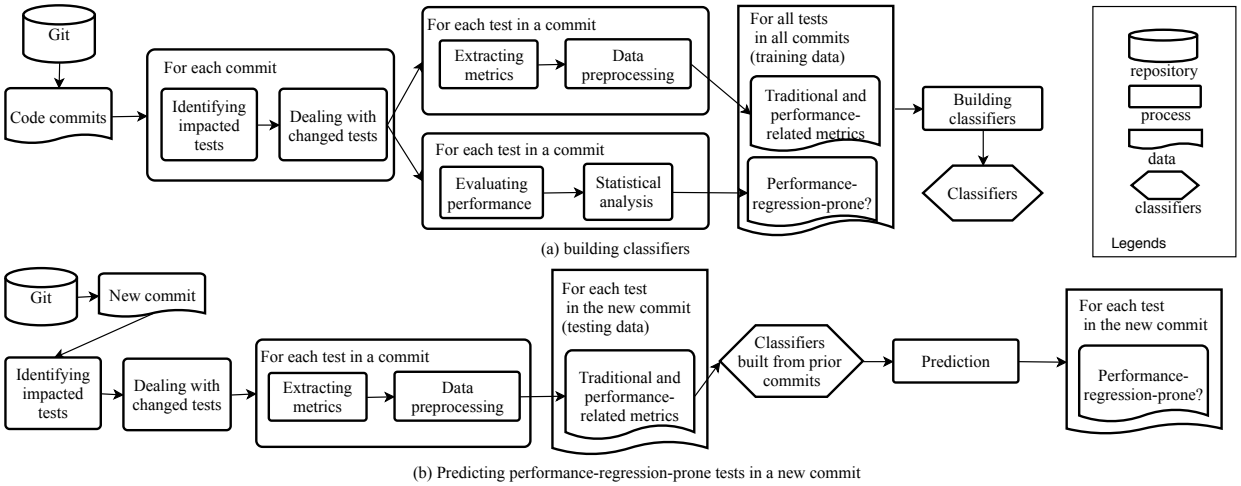


Fig. 1. An overview of our approach.

tion results. We perform statistical analyses on the performance data from each pair of current and parent commits to determine the existence and the magnitude of performance regression in a statistically rigorous manner. We use Mann-Whitney U test [68] to examine if there exist statistically significant differences (i.e., p-value < 0.05). We choose Mann-Whitney U test since it does not have any assumption on the distribution of the data. Researchers have shown that reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, p-value can be small even if the difference is trivial). We use Cliff's delta to quantify the effect sizes [69]. Cliff's delta measures the effect size statistically and has been used in prior engineering studies [70]. We only consider a test manifesting a performance regression if the effect size is medium ($0.33 < \textit{Cliff's delta} \leqslant 0.474$) or large ($0.474 <$*Cliff's delta*).

Based on the statistical analysis results, each commit is labelled with five different performance metrics, i.e., response time, CPU usage, memory usage, I/O read and I/O write, that are collected during the execution of the test. For example, the CPU label indicates whether the test demonstrate performance regression in terms of CPU usage. Therefore, with the same set of data, by considering one label, we can build a classifier that predicts the performance regression only for that metric. For example, if we only take the CPU label, we build a classifier that predicts whether a test in a commit would demonstrate performance regression for CPU usage.

In addition, we calculate the average increase of mean values of each performance metric with and without regression, shown in Table 3. We can see that the ones without regressions have a much smaller increase of values in each performance metric. Such small increase of values may be due to the measurement noise, hence not considered as performance regressions in our experiment. On the other hand, there may exist extreme values as outliers that should not be considered by our approach. Therefore, we use the $median \pm 3 \times median\ absolute\ deviance(MAD)$ as an indicator of outliers. We find that only 1.5% of our data are be impacted and we remove such outliers from our data.

In total, we spend 133 machine days running all the tests. Table 4 shows the amount of identified performance-regression-prone tests in the subject systems. We find that only a small portion of the tests is performance-regression-prone. Taking *Hadoop* as an example, out of the 1,349 tests, only 118 tests have performance regression in response time, which only accounts for 9% of all tests. Therefore, running all the tests to detect these performance regressions is not cost-effective.

TABLE 3
Average increase of mean values of each performance metric by comparing commits without and with regressions.

	Hadoop		Cassandra		OpenJPA	
	With regression	Without regression	With regression	Without regression	With regression	Without regression
Res. Time (s)	9.68	0.06	1.18	0.01	0.44	0.01
CPU (s)	1.38	0.29	0.54	0.01	1.55	0.02
Mem (KB)	67.56	8.47	15.5	1.24	119.13	27.34
I/O read (count)	119.19	3.97	273.95	4.28	614.6	26.46
I/O write (count)	392.82	13.33	233.29	1.43	122.26	13.72

TABLE 4
The number and percentage of identified performance-regression-prone tests w.r.t different performance metrics.

	Total	Response time	CPU	Memory	I/O read	I/O write
Hadoop	1,349	118 (9%)	111 (8%)	92 (7%)	110 (8%)	110 (8%)
Cassandra	985	23 (2%)	44 (4%)	31 (3%)	45 (5%)	28 (3%)
OpenJPA	1,868	154 (8%)	375 (20%)	265 (14%)	385 (21%)	488 (26%)

4.3 Preliminary study

One may consider measuring performance only once while executing the tests for every commit, since running all the tests for each commit is a typical practice in the software development. If one can measure performance while executing the test once and calculate the differences in performance metrics to detect performance regression, our approach may be less useful. For example, by running a test once, a developer may find that the test takes 11 minutes compared to 10 minutes with last commit. In this case, the developer may conclude that the test has 10% ($\frac{11-10}{10}$) regression in response time. Hence, a question that lingers is: ***can performance regression be detected by only running the tests once?***

In order to examine the applicability of such a simple approach, for each test in each commit, we measure the performance metrics of running each test only once and compare the metrics that was measured in the last commit. We calculate the relative difference between performance

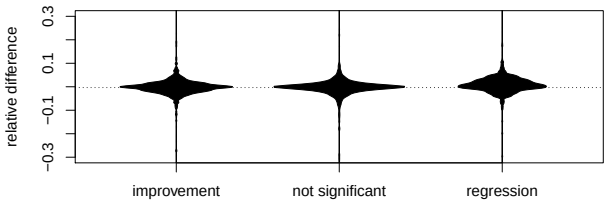


Fig. 2. Distribution of relative difference between performance metrics by running tests only once in each commit.

metrics. Positive relative differences mean that the tests have performance regressions, while negative relative differences mean that the tests have performance improvements. We group the tests based on whether they have performance regressions, improvement or no significant changes based on our performance evaluation in Section 4.2). Finally, we use bean plots to visualize the distribution of the relative differences of performance metrics in each group.

Figure 2 presents the distribution of relative difference between performance metrics by running the test only once. We find that, in all three groups, most of the relative difference are close to zero. None of the groups has the values significantly shift to either the positive or negative side, implying the inapplicability of using the relatives difference to detect performance regressions. For example, from Figure 2, we can see that more than half of the performance-regression-prone tests have their relative performance differences positive. Relying on this result will falsely lead to missing the detection of performance regression. We apply Mann-Whitney U test [68] and Cliff's delta [69] to compare the relative performance differences between each pair of the three groups. We find that the difference is either not statistically significant (p-value > 0.05) or with negligible effect sizes. Such a result shows that one **cannot** trust the performance measurement by only running the test once to determine whether there exists performance regression, confirming the need of our approach that aims to predict tests that are executed rigorously to detect performance regressions.

5 Evaluation Results

Our study aims to answer five research questions. For each research question, we present the motivation for the question, the approach that we use to answer the question and the result for the question.

RQ1: How well can we predict performance-regression-prone tests?

Motivation. We want to provide developers an accurate prediction on the performance-regression-prone tests when developers commit code changes. By evaluating the accuracy of the classifier, we can understand whether developers can depend in practice on our prediction results provided by our approach.

Approach. To answer RQ1, we first build five types of classifiers or models, including logistic regression (LR) [71], support vector machine (SVM) [72], XGBoost (XG) [73] with 50 (XG-50), 100 (XG-100) and 500 (XG-500) iterations, random forest classifiers [74], to model whether a test would manifest performance regressions. In addition, We replicate *Perphecy* by performing the prediction at the test level by considering the source code and code changes that are impacted by each test. We compare our approach to *Perphecy* (baseline) in this research question.

To realistically evaluate our classifiers, for every commit, we use our approach to predict performance-regression-prone tests and update the classifiers for each commit as presented in Section 3.

We utilize four metrics to evaluate our classifiers' performance, including *precision*, *recall*, *F-measure* and *AUC*. *Precision* measures the correctness of our classifier. *Precision* refers to that the number of tests that were correctly labeled as performance-regression-prone divided by the total number of tests that were labeled as performance-regression-prone by the classifier. *Recall* measures the completeness of our classifier. *Recall* is defined as the number of tests that were correctly labeled as performance-regression-prone by the classifier divided by the total number of tests with actual performance regression. *F-measure* is the harmonic mean of *precision* and *recall*, which gives equal weight to *precision* and *recall*. Shown in Table 4, our data is highly skewed since the majority of the tests do not manifest performance regressions. Therefore, we exploit *AUC* which allows us to measure the overall ability of our model to discriminate tests with performance regression and without performance regression. The *AUC* is the area under the ROC curve which indicates the performance of a binary classifier as its discrimination is varied [75]. The value of *AUC* ranges from 0 to 1, and larger value for *AUC* indicates a high discrimination in the prediction model.

Results. The results of using our classifiers to predict whether a test is performance-regression-prone is shown in Table 5. Due to the limited space, we only present AUC of all classifiers while presenting random forest and Perphecy with precision, recall and F-measure (F1 in Table 5). The full details can be found in Appendix A. The results show that the best of our classifiers is random forest, which achieves an average *AUC* of 0.85, 0.87 and 0.88 in *Hadoop*, *Cassandra* and *OpenJPA*, respectively. Random forest classifiers achieve an average precision of 0.32, 0.3, 0.59 and recall of 0.75, 0.75, 0.77 in *Hadoop*, *Cassandra* and *OpenJPA*, respectively. We find that although *Perphecy* achieves a higher average recall than other models in *Hadoop* and *Cassandra*, *Perphecy* only achieves an average precision of 0.08, 0.02, and 0.20 in *Hadoop*, *Cassandra* and *OpenJPA*, respectively. When considering the F-measures, our approach out-performs *Perphecy* in predicting performance regression with all the performance metrics of all subjects.

Our classifiers can accurately predict performance-regression-prone tests despite the fact that they are rare. Table 4 shows low percentages of tests that actually have performance regressions. For example, only 9% of the tests in *Hadoop*, 2% of the tests in *Cassandra* and 8% of the tests in *OpenJPA* have performance regressions in *response time*. However, the precision in predicting performance-regression-prone tests in *response time* is 0.3, 0.5 and 0.29, for *Hadoop*, *Cassandra* and *OpenJPA*, respectively. With such a large improvement over the random classifier in precision, our classifiers still provide a high recall (an average recall of 0.76).

Our classifiers have a similar AUC for all performance

TABLE 5
Results of using our approach to predict performance regressions with different performance metrics, comparing with Perphecy. Bold values highlight the best predictors.

	Hadoop												
	Random Forest				Perphecy				LR	SVM	XG-50	XG-100	XG-500
	Pre.	Recall	F1	AUC	Pre.	Recall	F1	AUC	AUC	AUC	AUC	AUC	AUC
Res. time	0.30	0.78	0.43	**0.87**	0.08	0.94	0.14	0.51	0.69	0.84	0.83	0.83	0.83
Cpu	0.43	0.77	0.55	**0.87**	0.09	0.90	0.16	0.55	0.63	0.72	0.80	0.80	0.79
Memory	0.20	0.84	0.32	**0.89**	0.06	0.88	0.11	0.53	0.58	0.82	0.83	0.82	0.80
I/O Read	0.43	0.67	0.52	**0.82**	0.07	0.86	0.13	0.52	0.62	0.81	0.74	0.75	0.76
I/O Write	0.25	0.71	0.36	**0.81**	0.08	0.95	0.14	0.53	0.60	0.61	0.69	0.68	0.68
Average	0.32	0.75	0.44	0.85	0.08	0.91	0.14	0.53	0.62	0.76	0.78	0.78	0.77
	Cassandra												
	Random Forest				Perphecy				LR	SVM	XG-50	XG-100	XG-500
	Pre.	Recall	F1	AUC	Pre.	Recall	F1	AUC	AUC	AUC	AUC	AUC	AUC
Res. time	0.50	0.63	0.56	0.78	0.03	0.79	0.05	0.54	0.66	0.65	0.78	**0.79**	0.78
Cpu	0.27	0.86	0.41	**0.90**	0.03	0.63	0.07	0.60	0.60	0.72	0.80	0.81	0.82
Memory	0.27	0.84	0.40	**0.95**	0.02	0.56	0.04	0.62	0.70	0.68	0.91	0.90	0.90
I/O Read	0.31	0.68	0.42	**0.86**	0.02	0.38	0.04	0.73	0.62	0.64	0.82	0.81	0.79
I/O Write	0.15	0.76	0.25	**0.87**	0.02	0.65	0.03	0.59	0.60	0.70	0.83	0.83	0.79
Average	0.30	0.75	0.41	0.87	0.02	0.60	0.05	0.62	0.64	0.68	0.83	0.83	0.82
	Openjpa												
	Random Forest				Perphecy				LR	SVM	XG-50	XG-100	XG-500
	Pre.	Recall	F1	AUC	Pre.	Recall	F1	AUC	AUC	AUC	AUC	AUC	AUC
Res. time	0.29	0.85	0.43	**0.92**	0.09	0.80	0.16	0.60	0.67	0.60	0.90	0.90	0.89
Cpu	0.73	0.85	0.78	0.91	0.23	0.70	0.35	0.56	0.81	0.63	**0.93**	**0.93**	**0.93**
Memory	0.48	0.72	0.57	**0.88**	0.10	0.81	0.18	0.51	0.75	0.69	0.84	0.84	0.84
I/O Read	0.72	0.81	0.76	**0.92**	0.23	0.68	0.34	0.54	0.79	0.62	0.89	0.89	0.88
I/O Write	0.71	0.60	0.65	**0.79**	0.34	0.70	0.45	0.57	0.72	0.57	0.77	0.77	0.77
Average	0.59	0.77	0.64	0.88	0.20	0.74	0.30	0.56	0.75	0.62	0.87	0.87	0.86

metrics. By examining the prediction results with different performance metrics, we find that all the performance metrics have a similar AUC. The majority of the AUC of our classifiers are over 0.8 only two classifiers have an AUC lower than 0.8 (i.e., 0.79 in I/O write for *OpenJPA*). In general, *response time* and *CPU* usage always have a high AUC value. Since *response time* and *CPU* usage are widely used performance metrics in practice, such results advocate the usefulness of our approach.

Our random forest classifier can achieve high AUC values when predicting performance-regression-prone tests. The precision and recall of the classifiers also drastically outperform baseline classifiers.

RQ2: How cost-effective is the prioritization of performance-regression-prone tests?

Motivation. In RQ1, our results show that we can accurately predict the performance-regression-prone tests. After the developers are notified by the prediction results, one still needs to actually execute the tests in order to review and address the performance regressions. However, performance evaluation is a time and resource consuming task [63]. A desired prediction result would predict performance-regression-prone tests while minimizing the needed time to execute them. Therefore, in this research question, we want to factor in the cost (performance testing time) that is associated with our prediction results.

Approach. First, we measure the actual performance testing time as a cost factor for every test in every commit of our subject systems. By knowing the actual testing time and whether each test actually manifests performance regression, we create an *optimal model*. In the optimal model, we sort all the actual performance-regression-prone tests by increasing the cost, i.e., the shortest test that manifests performance regression runs first. The optimal model is used as a baseline since it represents the optimal scenario of executing the tests in real-life.

Afterwards, we examine the predicted results by our classifiers from RQ1. We call this model *Non-Cost-AWare random forest model (NCAW)*. We sort all the tests ordered by decreasing predicted probability of being performance-regression-prone tests, without considering the cost of the tests.

Finally, we build a cost-aware prediction model by also considering the cost of the tests. Instead of modelling a binary outcome of whether the test is performance-regression-prone, we use the cost of the test to normalize the output. In particular, similar to Mende [76] and Kamei [11], we also calculate R_d as

$$R_d(x) = \frac{Y(x)}{Cost(x)} \quad (1)$$

where $Y(x)$ is 1 if the test is performance-regression-prone and 0 otherwise. $Cost(x)$ is the running time of the test. We build random forest regression model to predict the cost-aware output R_d. We call this model *Cost-AWare random forest model (CAW)*. We then sort all the tests by decreasing the predicted cost-aware output R_d.

To evaluate the cost-effectiveness of the cost-aware models, we first evaluate the successfully predicted performance regression given a threshold of the cost. Prior research on bug prediction finds that approximately 20% of the files with the highest faults contain at least 83% of the faults [77]. Since the percentage of tests with performance regression range from 2% to 26% (see Table 4), if we simply aim to examine 20% of the files, our approach would detect all the performance regressions. Therefore, to further demonstrate the strength of our approach, we set the threshold at 5% by further limiting the amount of resources that is available to execute the tests. We calculate the coverage as the percentage of predicted performance-regression-prone tests to all performance-regression-prone tests when we just spend 5% of the total running time. Moreover, since setting different threshold values lead to different results, to find the best threshold value, we use the number of detected performance-regression-prone test divided by the effort as a measure. We compute the measure by changing the effort from 5% to 90% in the steps of 5%. The result shows that 5% effort is the most cost-effective in *Cassandra* and *Hadoop*, and 10% of the effort is the most cost-effective in *OpenJPA*.

Finally, we use cumulative lift charts [78] to evaluate the prediction performance of model. An example of the cumulative lift chat is shown in Figure 3. We generate a cumulative lift chart from the optimal model, the non-cost-aware model and the cost-aware model (see Figure 3). The lines in the chart illustrate that by spending more time to execute tests, how much more performance-regression-prone tests can be evaluated. The P_{opt} is calculated by the size of the area under each line (from cost-aware and non-cost-aware models), divided by the area under the optimal line. Therefore, P_{opt} has a range from 0 to 1. The closer the P_{opt} to 1, the better our model is, i.e., closer to the optimal execution prioritization of the tests.

Results. Our cost-aware models have high cost-effectiveness, out-perform *Perphecy* and are close to the optimal model. Table 6 presents the cost-effectiveness of our prediction models. In particular, by spending only 5% of cost for executing all tests, our approach can help detect 12% to 65% of the performance-regression-prone tests. In addition, we observe that all our models have a high P_{opt} value. The average P_{opt} values are always higher than 0.8,

TABLE 6
Summary of non-cost-aware and cost-aware models. The coverage values are calculated when spending 5% of the total cost.

	Hadoop				Cassandra				OpenJPA			
	NCAW		CAW		NCAW		CAW		NCAW		CAW	
	Cov.	*Popt*	Cov.	*Popt*	Cov.	*Popt*	Cov.	*Popt*	Cov.	*Popt*	Cov.	*Popt*
Res. time	0.44	0.87	0.53	0.87	0.63	0.75	0.63	0.79	0.31	0.90	0.45	0.93
CPU	0.45	0.83	0.43	0.86	0.54	0.89	0.54	0.90	0.21	0.93	0.24	0.95
Memory	0.48	0.86	0.53	0.89	0.44	0.94	0.42	0.94	0.26	0.85	0.32	0.88
I/O read	0.52	0.88	0.52	0.82	0.46	0.75	0.62	0.86	0.21	0.91	0.23	0.92
I/O write	0.31	0.81	0.41	0.81	0.59	0.83	0.65	0.88	0.12	0.80	0.17	0.84
Average	0.44	0.85	0.48	0.85	0.53	0.83	0.57	0.87	0.22	0.88	0.28	0.90

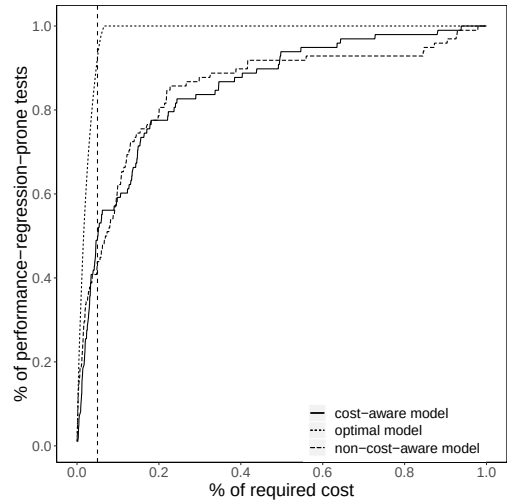


Fig. 3. Cumulative distribution function of optimal model, CAW and NCAW models in the performance counter of *Response time* in *Hadoop*.

indicating a high cost-effectiveness of our models. On the other hand, we find that our cost-aware models out-perform *Perphecy*. In particular, *Perphecy* only has an average P_{opt} value of 0.51 and by spending only 5% of cost, *Perphecy* on average can only detect 15% of the performance-regression-prone tests.

When comparing the non-cost-aware and cost-aware models, we find that our cost-aware models can still provide improvements to the non-cost-aware model, even though it already is very close to the optimal model. P_{opt} values from the cost-aware models are always higher than the non-cost-aware models in all the performance counters for all subject systems, only except one case, i.e., I/O read in *Hadoop*. Similarly, when having a threshold of spending only 5% of cost for executing all tests, the cost-aware models can detect on average more performance-regression-prone tests than the non-cost-aware models. The non-cost-aware models have better results only in the CPU usage for *Hadoop* and in the memory usage for *Cassandra*, when there is only small (2% and 2%) difference between the results of the two models. On the other hand, in all other cases, the cost-aware models typically have an improvement over the non-cost-aware models.

Our classifiers are highly cost-effective. By building a cost-aware model, we can further improve the cost-effectiveness of our classifiers.

RQ3: How much testing time can our approach save?

Motivation. The goal of our approach is to save developer's time from running all the tests to detect performance regressions. Therefore, in this research questions we would like to answer to what extend can developers benefit from our approach regarding to time saving.

Approach. To measure the time saving from our approach, we start by calculating the time needed without our approach. Since not all the tests are impacted by the code changes in each commit, one may opt to only run the tests that are impacted by each commit (c.f., *identifying impacted tests* in Section 4.2). Therefore, we measure the average time needed to run the tests impacted per commit as a baseline.

Afterwards, we measure the average time needed per commit for only running the tests that are predicted by our models in RQ1 to be performance-regression-prone. In particular, we calculate the time needed for each model related to each performance metric (e.g., CPU usage). In practice, developers may decide to run the test if any of the performance metrics are predicted to be performance-regression prone. Therefore, we also calculate the needed time to run the tests if at least one performance metric is predicted to be performance-regression-prone.

Results. Our classifiers can considerably save time of performance tests. We find that even only executing the impacted tests takes hours. Table 7 shows that, running all the impacted tests per commit for *Hadoop* takes on average 324 minutes (5.4 hours). The long execution time of rigorous performance evaluation makes it difficult to be carefully carried out in practice and hence confirms the motivation of our work.

On the other hand, with our approach, the needed time can be reduced from hours down to a matter of minutes. For example, the needed time to run predicted tests per commit in *Cassandra* is as low as 2 to 3 minutes, which make 97% to 95% time reduction comparing with running only the impacted tests. Even if developers choose to run the test if any performance metric is predicted to have regression, the needed test is 8 minutes, i.e., 87% time reduction.

TABLE 7
The average needed performance testing time (in minutes) for each commit.

	Impacted tests	Tests predicted by one model					Tests predicted by any model
		Res. Time	CPU	Memory	I/O read	I/O wirte	
Hadoop	324	26	35	18	23	30	91
Cassandra	61	2	3	3	3	2	8
OpenJPA	313	22	71	35	71	99	114

Our prediction drastically reduces the needed testing time comparing with running only the tests that are impacted by the code changes in a commit.

RQ4: Can our approach detect the introduction of real-life performance issues?

Motivation. In order to demonstrate the practical impact of our approach, we would like to examine whether the prediction results of our approach (cf. RQ1) can be used to assist in detecting the introduction of performance issues.

Approach. The goal of this RQ is to study whether any test in the performance-issue introducing commits are predicted by our approach and whether the predicted tests cover the code changes that introduce the performance issue.

We first collect all the performance issues whose introduction is possible to be during our studied period. In particular, we collect all performance issues that are

reported after the first commit of our study period from our subject systems. To collect the performance issues, we search a list of keywords, such as *performance* and *slow*, on the content (like title, description and discussion) of each issue. To avoid missing related keywords for performance issues, we identify all similar keywords using Word2vec that is trained from Stack Overflow[4]. We manually examine every issue that contains the keywords to ensure the issue is indeed a performance issue and we label each issue with performance metrics (like CPU or memory) to indicate which performance metric can be used to illustrate the performance issue.

However, not all of these performance issues are introduced during our studied period. Therefore, we leverage an SZZ algorithm [79] to detect the inducing commit of the performance issue. Study shows there may exist false positives results from SZZ algorithms, i.e., the commit identified by SZZ is not a true issue inducing commit [80]. Therefore, we manually check the results from SZZ algorithm to collect the list of issue inducing commits and intersect the results with the commits in our studied period to obtain the commits that indeed introduce a fixed performance issue in the future of the studied period. Finally, for each of the performance issue introducing commits, examine our prediction results from RQ1 to know if our approach successfully identifies any test in the commit. In addition, we examine the covered source code of the predicted tests to know if the covered source code is the location where the performance issue was introduced.

Results. Our approach can be used to detect the introduction of real-life performance issues. In total, we identify nine performance issues that are introduced during our studied period. Table 8 presents the fixing commit each performance issue, the inducing commits for each issue and their corresponding performance metrics. Table 8 shows that six out of nine issues have at least one true positive prediction, which shows that our approach could be used to detect the introduction of real-life performance issues. More importantly, we find that the code changes that introduce the performance issues are covered by the predicted tests.

Similar to the findings from RQ1, we find that *Perphecy* although can detect two more real-life bugs, it produces a large number of false-positive detection results. Shown in Table 8, 348 tests are false-positively detected as manifesting performance regressions. Such a large number of false positive results may introduce much overhead in executing the tests and extra effort for the practitioners to manually examine the results.

We manually examine the three issues that are not successfully predicted by our approach, and we find that for YARN-4307, one of the predicted probability of having a run-time performance regression is 0.49, which is almost at the threshold (0.5) of being considered as a positive prediction. The other two issues (YARN-7102 and HDFS-12754) are rather complicated performance issues (like deadlock), which are expected to be difficult to capture using our metrics. For the same bugs, because of the tendency of having false-positive results, *Perphecy* predicts the test with almost all metrics as positive as performance regressions. We also observe a high false positive rate for OPENJPA-2665. We find that almost all the false positives are related to physical performance metrics, such as CPU and Memory, while the description of the issue is only associated with its impact on response time.

4. The complete list of the keywords are listed in our replication package

TABLE 8
Results of using our approach and Perphecy to detect the introduction of real-life performance issues.

Issue ID	Issue fixing commit	Issue inducing commit	Performance metric	Predicted? PerfJIT	#FP PerfJIT	Predicted? Perpechy	#FP Perpechy
Hadoop							
YARN-4307	308d63f	e914220	Resp. time	NO	0	NO	20
		7af5d6b	Resp. time	NO		NO	
YARN-5889	5fb723b	d9281fb	Resp. time	YES	1	YES	7
			CPU	NO		YES	
			I/O Write	YES		YES	
YARN-3388	444b2ea	d9281fb	Resp. time	YES	2	YES	13
			I/O Write	NO		YES	
YARN-7102	ff8378e	528b809	Resp. time	NO	0	YES	3
YARN-4862	352cbaa	528b809	Resp. time	NO	0	YES	2
			CPU	YES		YES	
			Memory	NO		YES	
HDFS-12754	738d1a2	decf8a6	Resp. time	NO	0	YES	3
			CPU	NO		YES	
Cassandra							
CASSANDRA-12763	d73f45b	b32a9e6	Resp. time	YES	0	YES	2
CASSANDRA-13794	f93e6e3	1b36740	Resp. time	YES	0	YES	116
		a7cb009	Resp. time	YES		YES	
		88d2ac4	Resp. time	NO		YES	
OpenJPA							
OPENJPA-2665	f0286a2	9fa9ef4	Resp. time	YES	46	YES	182

Our approach can be used to detect commits that introduce performance issues. In addition, the predicted tests cover the code changes that introduce the issue. Developers in practice cloud use our approach to prevent the introduction of performance issues.

RQ5: What are the most important factors in determining performance-regression-prone tests?

Motivation. The results in RQ1 show that our approach can successfully predict performance-regression-prone tests at a given commit. By understanding the influential factors of such tests, we may further understand the characteristics of performance-regression-prone tests. Such characteristics can be leveraged by practitioners to proactively avoid introducing performance regressions. In particular, developers in practice often conduct code reviews to improve the quality of software, where due to the complexity of performance regressions, it may be challenging to identify the performance regressions based on reviewing source code. With the knowledge of the characteristics of performance-regression-prone tests, one can use such information to prioritize effort during their code review process to identify the potential performance regressions. For example, we may learn that changes to loops provide important influence to the introduction of performance regressions. When doing code reviews, or even during the writing of source code, developers should be aware that changes to loops may consider spending more effort on reviewing such code changes.

Approach. To address this RQ, we use the variable importance value that is calculated with the random forest classifiers. The variable importance value is calculated by permuting the values of the corresponding metric while keeping the values of the other metrics unchanged. The classifier measures the impact of such a permutation based on the classification error rate [81]. We use the function *importance* of the *randomForest* R package to compute the variable importance values.

To avoid bias caused by analyzing just one classifier and to ensure a robust conclusion, for each classifier that is built for each commit, we use bootstrap to resample the training data and build random forest predictors with the bootstrap sample data. We repeat the above process 100 times and collect 100 variable importance values for each change metric in each commit. In particular, we use the function *boot* of the *boot* R package to perform bootstrap resampling.

Afterwards, each metric will have 100 variable importance values in each commit. However, the difference of variable importance values between two metrics may not be statistically significant. To find statistically distinct ranks of metrics, we perform Scott-Knott Effect Size Difference (ESD) test [82] to cluster the metrics based on both their statistical significance and effect sizes. The Scott-Knott ESD test uses hierarchical cluster analysis to partition different metrics into distinct groups. With this analysis, each metric has a rank in the classifier for each commit. Since we build a classifier for every commit, we calculate the average rank of a metric based on its ranks in all the classifiers.

Finally, to understand whether the value of the metric has positive or negative relationship with the existence of performance regressions, we compare the average metric values in the tests that are with and without performance regression by performing a t-test. We consider that the metric has a positive relationship with the existence of performance regressions if the average metric values in the tests with the performance regression is higher than that without performance regression. We only consider the positive or negative relationship if the p-value of the Student T-test [83] is smaller than 0.05.

TABLE 9
Average rank of the top important metrics in our classifiers. The up/down arrows indicates whether the relationship is positive/negative.

Metric	Resp. time	CPU	Memory	I/O read	I/O write
AGE	2.2↓	1.3↓	1.5↓	1.4↓	1.6↓
SOL	2.6↑	2.9↑	2.8↑	2.3↑	2.7↑
LT	3.6↑	3.3↑	2.9↑	2.9↑	3.1↑
REXP	4.0↑	2.5↑	2.4↑	3.0↑	3.7↑
NDEV	3.1↑	4.0↑	4.5↑	3.7↑	4.4↑
Entropy	2.9↑	4.0↑	3.3↑	3.8↑	3.7↑
Complexity	5.2↑	4.4↑	4.2↑	4.3↑	4.6↑
LA	4.6↑	4.0↑	4.4↑	4.7↑	4.4↑
LD	4.7↓	5.9↓	5.5↓	5.4↓	5.2↓
for_chg	6.4↑	9.3↑	8.4↑	5.6↑	6.9↑

Results. The result of average rank of important metrics is shown in Table 9. Due to limited space, we only show the top important metrics for the classifiers and models for each performance metric. Each row in the table presents the average rank of the importance of a metric among the three subject systems. The arrows in the table indicate whether each metric has a positive (up arrow) or negative (down arrow) to the probability of having performance regressions.

Traditional metrics are more important than performance-related metrics in the prediction of performance-regression-prone tests. Surprisingly, despite that our performance-related metrics are derived based on prior research [8], [24], [47], [48], [49] on software performance, we find that most of the top important metrics are traditional metrics. In fact, there exists only one performance-related metrics, i.e., *for_chg* (changes to for loops) in the top metrics. Therefore, we try our approach based with random forest classifiers based on only with transitional metrics and evaluate as RQ1. We find that on average the AUC values of each classifier only decreases 0.01, 0.06 and 0.05 for *Hadoop*, *Cassandra* and *OpenJPA*, respectively.

The *history* and *size* dimensions are more important than other dimensions in the traditional metrics. Metrics *SOL* and *AGE* are the two most important factors in the prediction model. *AGE* has a negative relationship with performance regression, which implies that more recent and frequent changes are more likely to introduce performance regression. Besides *AGE*, We also find that metrics in the *size* (e.g., *SOL*) has a positive relationship with performance regressions, which is also similar to prior research on commit-level defect predictions [11].

Changes to loops (*for_chg*) is the most important factors in the performance-related metrics. This indicates that there is a relationship between changing the operation of *loop* and the performance regressions. For example, in commit #94a9a5d0 of *Hadoop*, developers added a *for* loop into the method *getUsers* in the source files *LeafQueue.java*. The *for* loops adds an item to a queue repetitively, leading to performance regressions in I/O read.

We find that the traditional software metrics dominate the important factors in the prediction of performance-regression-prone tests. On the other hand, changes to loops are the only top important factors in the performance-related metrics.

6 Discussion

In this section, we discuss the learned lessons during the implementation of our approaches, the limitation and future work of our approach and the generalizability of our study.

6.1 Traditional metrics are more important than performance-related metrics

In RQ5, we find that traditional metrics are more important than performance-related metrics, when predicting performance-regression-prone tests. Such a result is unexpected since all our performance-related metrics are derived by prior empirical study findings on performance bugs [8], [20], [47]. In order to understand why such metrics do not have a high importance when predicting performance-regression-prone tests, we manually examine the tests that are not performance-regression-prone, while having a high value in any performance-related metrics. We find the importance of the performance-related metrics often depend on a specific workload, while such a workload may not exist in the tests.

For example, in commit #*94a9a5d0* in *Hadoop* project, developers added a *for* loop into the *class LeafQueue.java*, which produces a performance-related metric *loop* change. In our prediction model, the two corresponding tests, *TestCapacityScheduler.java* and *TestRMWebServicesCapacitySched.java* are predicted as performance-regression-prone tests. However, the actual tests are not performance-regression-prone because the two tests do not execute the *loop* with a large number of times, to demonstrate the regression.

6.2 Our approach out-performs Perphecy

In our evaluation, we compare our approach with *Perphecy* since it is the closest research approach to ours. However, we find that *Perphecy* does not provide an accurate prediction in our evaluation setting. In particular, we find that, *Perphecy* suffers badly from a low precision, e.g., the precision of *Perphecy* in *Hadoop* and *Cassandra* is only 0.06 and 0.02. Although *Perphecy* has a relatively high recall, it may mean that *Perphecy* always tends to predict a test as performance-regression-prone. In fact, we find that, for all the tests in our dataset, 61% to 92% of them are predicted as performance-regression-prone by *Perphecy*. We consider the reason may be that *Perphecy* only treats its metrics as boolean values based on a threshold. In addition, the tuning strategy and the use of disjunctions to combine all metrics may cause more tests being predicted as performance-regression-prone. On the other hand, our approach uses a much larger set of metrics while depending on machine learning classifiers instead of simplifying metric values into a boolean value to make predictions. In addition, our results in RQ5 show that most of the top influential factors are rather traditional product and process software metrics. The information in these top metrics is not captured in the indicators by *Perphecy*. The lack of such information may also cause *Perphecy*'s lower precision in our study.

6.3 Limitation of our approach and future work

The main limitation of our approach is that that our extracted metrics are all based on static code analysis. When applying our approach to detect the introduction of real-file performance issues in RQ4, we find that our approach fails to detect a few performance bugs with the complex root causes such as deadlocks. By examining our extracted metrics and manually studying the details of the performance bugs, we understand that our extracted metrics that are based on static code analysis may not be able to capture the characteristics of these performance bugs. On the other hand, performance bugs like deadlocks often occur with specific execution conditions, where dynamic data that captures the interactions between procedures in the executing the source code is crucial to the detection and prediction of these bugs. Therefore, in order to improve our approach to detect such complex performance bugs, we plan our future study by extracting dynamic information from the test execution and extract more metrics to capture the interactions between procedures in tests.

6.4 Generalizability of our study

Although our approach is not designed in particular for any programming language or any type of project, there still exist some aspects that may influence the generalizability of our study.

Test coverage. Our approach depends on executing small-scale tests that are readily available in the software system to detect performance regressions. If the source code or the code changes are not covered by the tests, our approach cannot help in detecting the associated regressions. We find that by measuring the method coverage by tests, our studied projects may not have high method coverage. In particular, the method coverage of *Hadoop*, *Cassandra* and *OpenJPA* is only 13.64%, 24.82%, and 11.09%, respectively. However, since our approach works for each code commit, only the changed methods need to be covered by the test. We find that for all the changed methods in all commits, 67.97%, 53.3%, and 64.62% are covered by the tests in *Hadoop*, *Cassandra* and *OpenJPA*, respectively. Such high coverage ensures the success of our approach. This also implies that, in order to adopt our approach, practitioners may first evaluate whether the source code that are likely to be changed is covered by tests.

The granularity of commits. Different projects and developers may follow different granularities of making code commits. Some practitioners may stack a large amount of code changes in one commit. In such a scenario, it is easier for our approach to do the prediction while on the other hand, the prediction results may not be as beneficial to developers since they may still end up with many code changes to investigate. On the other hand, one developer may commit very often, leading to commits with very few changes in it. Since performance regression is often a result of a combination of contribution from multiple sources, such small commits may isolate the impact, making our approach not able to see the performance influence in each individual commit. In our three studied projects, the average code churn per commit is 155, 170 and 182, for *Hadoop*, *Cassandra* and *OpenJPA*, respectively. The granularities of the commits among the three projects are rather similar, so as the accuracy of our approach on the three projects. However, future research may perform in-depth investigation on the granularity of commits and the accuracy of our approach.

The quality of the test itself (e.g., test being flaky). Our approach depends on the tests to evaluate the performance of the associated source code. However, if the test itself is written with a sub-optimal quality, the results may be biased. For example, the test failures in the flaky test may introduce noise and requires extra running time to achieve the needed repetition. Recent research [84] discusses the reasons for tests not suitable for performance evaluation, which can be leveraged to know how well another project can adopt our approach.

7 Threats to Validity

External validity. Due to the huge computing resources (133 machine days spent in our case studies) needed for identifying performance regressions, we carry out our study on three subject systems. All our three subject systems are implemented in *Java*. Hence, our findings might not be suitable for other systems. Future studies may especially focus on commercial close source systems with other languages (such as C++).

Internal validity. We use traditional metrics and performance-related metrics based on the findings from prior research. We choose our classifiers, based their widely usage in prior software engineering research [85], [86], and typically provide a high accuracy in the modeling. There may exist other metrics that we do not include in our study and other machine learning classifiers may also achieve an accurate classification. Although our results have shown high (0.86 on average) AUC values of our prediction models,

adding more metrics or employing other classifiers by future research may further improve the models.

We were only able to collect nine real-life performance issues in our RQ4. However, there can be other performance issues that are introduced during our studied period while not yet being fixed or reported. We could not evaluate our approach on the un-reported or un-fixed performance issues. A long-term study by applying our approach in practice may address this threat.

Construct validity. When extracting performance-regression-prone tests, we execute each test 30 times repetitively to address the flakiness and unstableness of the test results. The more repetition, the less that the testing results is biased. Future work may vary the number of repetitions to complement our results. Due to the high cost of tests, we use the release to estimate the mapping between code and test rather than creating such mapping for every commit. Such mapping might not be perfect since code changes in during the release may alter such mapping. Future research can evaluate such an approach or investigate the use of other techniques to generate the mapping between code and test.

The results of our approach are achieved without re-balancing the training data. In order to know the impact of data re-balancing, we leveraged two re-balancing techniques, i.e., up-sampling and SMOTE [87]. However, neither of the two techniques can improve our prediction results. Similar findings have been reported in software defect prediction [53], [88]. In addition, we did not fine-tune all the parameters of classifiers, while only running XGBoost with different number of iterations. Although practitioners find mainly the importance of iterations in XGBoost [89], future work can investigate the optimization of other parameters of the classifiers. When measuring the time saving in RQ3, we did not include the time needed to set up the performance measurement environment and rebuilding models for each commit. By providing automated scripts, the environment set up only takes seconds and only needs to be set up once. In addition, extracting metrics for the newly detected tests and rebuilding models only negligible time (take less than one minute) in all the cases in our study. We also examine the testing time needed for the initial 50 commits for each subject system when cold start our approach without any historical data. The testing time for the initial 50 commits is 2,425 minutes, 2,257 minutes, and 13,099 minutes, respectively. However, such time is only needed once to cold start our approach.

8 Conclusion

In this paper, we propose an approach that automatically predicts whether a test would manifest a performance regression given a code commit. The case study results show that our approach can provide accurate prediction results, drastically outperforming a random classifier and being able to detect the introduction of real-life performance issues. In particular, this paper makes the following contributions:

- To the best of our knowledge, our work is the first to predict performance-regression-prone tests at the commit level.
- Our approach can provide accurate prediction results, and save testing time, easing the adoption of the approach in practice.
- The important factors identified in our case study can be leveraged by developers to proactively avoid introducing performance regressions.

References

[1] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, Dec 2000.

[2] "Mozilla performance regressions policy." [Online]. Available: https://www.mozilla.org/en-US/about/governance/policies/regressions/

[3] Q. Luo, D. Poshyvanyk, and M. Grechanik, "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*, pp. 25–36.

[4] W. Shang, A. E. Hassan, M. N. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31-February 4, 2015*, pp. 15–26.

[5] C. Heger, J. Happe, and R. Farahbod, "Automated root cause isolation of performance regressions during software development," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13. New York, NY, USA: ACM, 2013, pp. 27–38.

[6] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 299–310.

[7] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "An industrial case study of automatically identifying performance regression-causes," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 232–241.

[8] J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 341–352.

[9] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Perphecy: Performance regression test selection made simple but effective," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, 2017, pp. 103–113.

[10] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[11] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.

[12] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[13] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 5. IEEE, 2016, pp. 33–45.

[14] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 182–191.

[15] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 552–569, April 2013.

[16] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Information and Software Technology*, vol. 59, pp. 170–190, 2015.

[17] P. Tourani and B. Adams, "The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 189–200.

[18] S. Tsakiltsidis, A. Miranskyy, and E. Mazzawi, "On automatic detection of performance bugs," in *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 132–139.

[19] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 157–168.

[20] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 77–88.

[21] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.

[22] S. Zaman, B. Adams, and Hassan, "A qualitative study on performance bugs," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 199–208.

[23] X. Han, T. Yu, and D. Lo, "Perflearner: learning from bug reports to understand and generate performance test frames," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 17–28.

[24] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 60–71.

[25] M. Pradel, M. Huggler, and T. R. Gross, "Performance regression testing of concurrent classes," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 13–25.

[26] A. Tarvo and S. P. Reiss, "Using computer simulation to predict the performance of multithreaded programs," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 217–228.

[27] P. Leitner and C.-P. Bezemer, "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*, ser. ICPE '17. New York, NY, USA: ACM, 2017, pp. 373–384.

[28] M. Laali, H. Liu, M. Hamilton, M. Spichkova, and H. W. Schmidt, "Test case prioritization using online fault detection information," in *Ada-Europe International Conference on Reliable Software Technologies*. Springer, 2016, pp. 78–93.

[29] R. Kazmi, D. N. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–32, 2017.

[30] S. Wang, J. Nam, and L. Tan, "Qtep: quality-aware test case prioritization," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 523–534.

[31] S. Mostafa, X. Wang, and T. Xie, "Perfranker: Prioritization of performance regression tests for collection-intensive software," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 23–34.

[32] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396, 2015.

[33] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 268–279.

[34] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 119–129.

[35] J. Anderson, S. Salem, and H. Do, "Striving for failure: an industrial case study about test failure prediction," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 49–58.

[36] T. B. Noor and H. Hemmati, "Studying test case failure prediction for test case prioritization," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2017, pp. 2–11.

[37] Y. Zhu, E. Shihab, and P. C. Rigby, "Test re-prioritization in continuous testing environments," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 69–79.

[38] A. Najafi, W. Shang, and P. C. Rigby, "Improving test effectiveness using test executions history: an industrial experience report," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 213–222.

[39] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461.

[40] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007.*, May 2007, pp. 9–9.

[41] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

[42] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292.

[43] A. G. Koru, D. Zhang, K. E. Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 293–304, March 2009.

[44] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, May 2010, pp. 495–504.

[45] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 18.

[46] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, Jul 2000.

[47] M. M. u. Alam, T. Liu, G. Zeng, and A. Muzahid, "Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 298–313.

[48] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 370–380.

[49] D. Costa, A. Andrzejak, J. Seboek, and D. Lo, "Empirical study of usage and performance of java collections," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: ACM, 2017, pp. 389–400.

[50] "srcml," 2017. [Online]. Available: http://www.srcml.org/

[51] S. Behnel, M. Faassen, and I. Bicking, "lxml: Xml and html with python," 2005.

[52] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on the interpretation of defect models," *IEEE Transactions on Software Engineering*, 2019.

[53] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: Pitfalls and challenges," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 286–295.

[54] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[55] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An Empirical Study of the Impact of Modern Code Review Practices on Software Quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[56] F. E. Harrell, *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. Springer, 2001.

[57] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads," *Automated Software Engg.*, vol. 24, no. 1, pp. 189–231, Mar. 2017.

[58] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock, "Methods and metrics for cold-start recommendations," in *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '02. New York, NY, USA: ACM, 2002, pp. 253–260.

[59] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[60] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads," *Automated Software Engineering*, vol. 24, no. 1, pp. 189–231, 2017.

[61] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1001–1012.

[62] N. Tillmann and W. Schulte, "Unit tests reloaded: Parameterized unit testing with symbolic execution," *IEEE Software*, vol. 23, no. 4, pp. 38–47, 2006.

[63] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 666–677.

[64] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 119–130.

[65] P. Leitner and J. Cito, "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds," *ACM Trans. Internet Technol.*, vol. 16, no. 3, pp. 15:1–15:23, Apr. 2016.

[66] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. how bad is it really?" *Empirical Software Engineering*, vol. 24, no. 4, pp. 2469–2508, 2019.

[67] "psutil," 2017. [Online]. Available: https://github.com/giampaolo/psutil

[68] N. Nachar *et al.*, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in Quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.

[69] L. A. Becker, "Effect size (es)," *Accessed on October*, vol. 12, no. 2006, pp. 155–159, 2000.

[70] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[71] F. E. Harrell, "Regression modeling strategies," vol. 3, no. 3, 2001.

[72] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their Applications*, vol. 13, no. 4, pp. 18–28, July 1998.

[73] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.

[74] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001.

[75] J. M. Lobo, A. Jiménez-Valverde, and R. Real, "Auc: a misleading measure of the performance of predictive distribution models," *Global ecology and Biogeography*, vol. 17, no. 2, pp. 145–151, 2008.

[76] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *2010 14th European Conference on Software Maintenance and Reengineering*, March 2010, pp. 107–116.

[77] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, April 2005.

[78] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, ser. PROMISE '09. New York, NY, USA: ACM, 2009, pp. 7:1–7:10.

[79] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.

[80] C. Williams and J. Spacco, "Szz revisited: Verifying when changes induce fixes," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, ser. DEFECTS '08. New York, NY, USA: ACM, 2008, pp. 32–36.

[81] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, Aug 2017.

[82] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, Jan 2017.

[83] H. von Storch, "Misuses of statistical analysis in climate research," in *Analysis of Climate Variability*, H. von Storch and A. Navarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 11–26.

[84] Z. Ding, J. Chen, and W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet?" 2020.

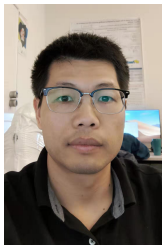
[85] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Software Engineering*, vol. 23, no. 1, pp. 290–333, Feb 2018.

[86] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 812–823.

[87] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

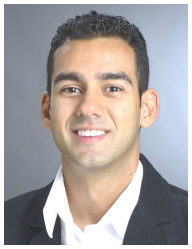
[88] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *CoRR*, vol. abs/1801.10269, 2018.

[89] "Complete guide to parameter tuning in xgboost." [Online]. Available: https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/



Jinfu Chen is a Ph.D. student in the department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his M.Sc. degree from Chinese Academy of Sciences and he obtained B.Eng. from Harbin Institute of Technology. His research interest lies in empirical software engineering, software performance engineering, performance testing, software log mining.

Weiyi Shang is a Concordia University Research Chair at the Department of Computer Science. His research interests include AIOps, big bata software engineering, software log analytics and software performance engineering. His research has been adopted by industrial collaborators (e.g., BlackBerry and Ericsson) to improve the quality and performance of their software systems that are used by millions of users worldwide.



Emad Shihab is an Associate Professor and Concordia University Research Chair in the Department of Computer Science and Software Engineering at Concordia University. His research interests are in Software Engineering, Mining Software Repositories, and Software Analytics. His work has been published in some of the most prestigious SE venues, including ICSE, ESEC/FSE, MSR, ICSME, EMSE, and TSE. He serves on the steering committees of PROMISE, SANER and MSR, three of the leading conferences in the software analytics areas. His work has been done in collaboration with and adopted by some of the biggest software companies, such as Microsoft, Avaya, BlackBerry, Ericsson and National Bank. He is a senior member of the IEEE. His homepage is: http://das.encs.concordia.ca.