# Performance Regression Detection in DevOps

**Jinfu Chen**

**A Thesis**

**in**

**The Department**

**of**

**Computer Science and Software Engineering**

**Presented in Partial Fulfillment of the Requirements**

**for the Degree of**

**Doctor of Philosophy (Software Engineering) at**

**Concordia University**

**Montréal, Québec, Canada**

**November 2020**

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Jinfu Chen**

Entitled: **Performance Regression Detection in DevOps**

and submitted in partial fulfillment of the requirements for the degree of

**Doctor of Philosophy (Software Engineering)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Youmin Zhang

_____ External Examiner
Dr. Andy Zaidman

_____ Examiner
Dr. Nikolaos Tsantalis

_____ Examiner
Dr. Jinqiu Yang

_____ Examiner
Dr. Yan Liu

_____ Supervisor
Dr. Weiyi Shang

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

November 9, 2020 _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

# Abstract

**Performance Regression Detection in DevOps**

**Jinfu Chen, Ph.D.**

**Concordia University, 2020**

Performance is an important aspect of software quality. The goals of performance are typically defined by setting upper and lower bounds for response time and throughput of a system and physical level measurements such as CPU, memory, and I/O. To meet such performance goals, several performance-related activities are needed in development (Dev) and operations (Ops). Large software system failures are often due to performance issues rather than functional bugs. One of the most important performance issues is performance regression. Although performance regressions are not all bugs, they often have a direct impact on users' experience of the system. The process of detection of performance regressions in development and operations is faced with challenges. First, the detection of performance regression is conducted after the fact, i.e., after the system is built and deployed in the field or dedicated performance testing environments. Large amounts of resources are required to detect, locate, understand, and fix performance regressions at such a late stage in the development cycle. Second, even we can detect a performance regression, it is extremely hard to fix it because other changes are applied to the system after the introduction of the regression.

These challenges call for further in-depth analyses of the performance regression. In this thesis, to avoid performance regression slipping into operation, we first perform an exploratory study on the source code changes that introduce performance regressions in order to understand root-causes of performance regression in the source code level. Second, we propose an approach that automatically predicts whether a test would manifest performance regressions in a code commit. Most of the performance issues are related to configurations. Therefore, third, we propose an

approach that predicts whether a configuration option manifests a performance variation issue. To assist practitioners to analyze system performance with operational data, we propose an approach to recovering field-representative workload that can be used to detect performance regression.

# Related Publications

The following publication is related to the materials presented in this thesis:

- **Jinfu Chen** and Weiyi Shang. An Exploratory Study of Performance Re-gression Introducing Code Changes. In2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September17-22, 2017. 341-352.
- **Jinfu Chen**, Weiyi Shang, Ahmed E. Hassan, Yong Wang, and Jiangbin Lin. An Experience Report of Generating Load Tests Using Log-Recovered Workloadsat Varying Granularities of User Behaviour. In34th IEEE/ACM InternationalConference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. IEEE, 669-681.
- **Jinfu Chen**. Performance Regression Detection in DevOps. In the 42nd International Conference on Software Engineering Doctoral Symposium (ICSE DS). 2020.
- **Jinfu Chen**, Weiyi Shang, and Emad Shihab. PerfJIT: Test-level Just-in-time Prediction for Performance Regression Introducing Commits. IEEE Transactions on Software Engineering, 2020.
- **Jinfu Chen**, Mohammed Sayagh, Heng Li, Weiyi Shang, and Bram Adams. An Empirical Study on the Inconsistent Options Performance Variation. Empirical Software Engineering, 2020. [Under Review]

The following publications are not directly related to the material presented in this thesis but were conducted as parallel work to the research presented in this thesis.

- Yi Zeng, **Jinfu Chen**, Weiyi Shang, and Tse-Hsun Chen, Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. Empirical Software Engineering, pp. 1-41, 2019.
- Zishuo Ding, **Jinfu Chen**, and Weiyi Shang. Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet? In the 42nd International Conference on Software Engineering (ICSE). 2020. **ACM SIGSOFT Distinguished Paper Award**.
- Lizhi Liao, **Jinfu Chen**, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Using black-box performance models to detect performance regressions under varying workloads: an empirical study. Empirical Software Engineering, 1-31, 2020
- Lizhi Liao, **Jinfu Chen**, Heng Li, Yi Zeng, Weiyi Shang, Catalin Sporea, Andrei Toma, Sarah Sajedi. Locating Performance Regression Root Causes in the Field for Web-based Systems. IEEE Transactions on Software Engineering, 2020. [Under Review]

# Statement of Originality

I, Jinfu Chen, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

# Dedications

To my family.

# Acknowledgments

First, I would like to express my greatest gratitude to my supervisor Dr. Weiyi (Ian) Shang for all his guidance and unconditional support. This thesis would have been impossible to complete without your aid and support. Ian, thank you for giving me the opportunity to tackle this challenge, for believing in me. You were there every time I hesitated to take the next step, and guided me to success. I have learned a great deal from you beyond as a researcher.

I would also like to show my sincere gratitude to my examination committee members, Dr. Nikolaos Tsantalis, Dr. Emad Shihab, Dr. Jinqiu Yang, Dr. Yan Liu, and Dr. Andy Zaidman, for taking their precious time to consider my work and offer insightful comments, support, and guidance. Many thanks to my examiners for their valuable feedback.

I extend my gratitude to the professors and collaborators with whom I worked the closest for my degree and research, Dr. Tse-Hsun (Peter) Chen, Dr. Nikolaos Tsantalis, Dr. Emad Shihab, Dr. Jinqiu Yang, Dr. Heng Li, Dr. Ahmed E. Hassan, Dr. Mohammed Sayagh, Dr. Bram Adams, and Dr. Jianmei Guo. It was a delight to follow your teachings and guidance.

An immense thank you to all the members of the Software Engineering and System Engineering (SENSE) lab, Zishuo Ding, Kundi Yao, Guilherme Bicalho de Padua, Maxime Lamothe, Mehran Hassani, Muhammad Moiz Arif, Armin Najafi, Zhenhao Li, Joy Zeng, Hetong Dai, Sophia Quach, Lizhi Liao, Haonan Zhang, Roger Xia, Nian Liu. I would also like to thank the members of our peer labs DAS and SPEAR, Dr. Rabe Abdalkareem, Dr. Diego Elias Costa, Suhaib Mujahid, Giancarlo Sierra Monge, Sultan Wehaibi, Xiaowei Chen, Ahmad Abdellatif, Olivier Noury and Zi Peng, Steven Locke, An Ran Chen. I am glad to not only call you my colleagues but my friends.

While pursuing this Ph.D. degree, I received the constant support of my closest friends outside

academy. Thank you for always being there. Last but not least, I would like to thank my family for their unconditional support. A special thank you to my wife, Yuanhui. Despite the distance that sets us apart, I never felt alone on this journey with you by my side. You are the light that keeps me going.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Introduction

The rise of large-scale software systems (e.g., Amazon.com and Google Gmail) has posed an impact on people's daily lives from mobile devices users to space station operators. The increasing importance and complexity of such systems make their quality a critical, yet extremely difficult issue to address. Failures in such systems are more often associated with performance issues, rather than with feature bugs (Dean & Barroso, 2013; Simic & Conklin, 2012; Weyuker & Vokolos, 2000). Therefore, performance is an important aspect of software quality.

The goals of performance are typically defined by setting upper and lower bounds for response time and throughput of a system and physical level measurements such as CPU, memory, and I/O. In order to ensure that the performance of a system meets a requirement, several performance assurance activities are required during development (Dev) and operations (Ops) in the release cycle of large software systems. DevOps is a set of software practices to smooth the deploying release of a software system between development and operations Hüttermann (2012). One of the goals of performance assurance activities in DevOps is to detect performance regressions, i.e., the performance of the same feature in the system is worse than before Brunnert et al. (2015). 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 are important tasks 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 Joel (2017).

Due to the importance of performance regression, extensive prior research has proposed automated techniques to detect performance regressions (Heger, Happe, & Farahbod, 2013; Luo, Poshyvanyk, & Grechanik, 2016; T. H. D. Nguyen, Nagappan, Hassan, Nasser, & Flora, 2014; Shang, Hassan, Nasser, & Flora, 2015). However, challenges in the practices of DevOps of performance regression detection still exist.

- **No existing performance regression data.** There is no existing data of performance regression introducing code changes.
- **Performance regression detection is too late.** The existing performance regression detection is applied after the system is built and deployed in the field. Large amounts of resources are required to detect, locate, and fix performance regressions at such a late stage in the development cycle.
- **Large volume of operational data.** The increasing volume of operational data is beyond the capacity of traditional human-driven practices. It is very time-consuming and error-prone to analyze such large-scale operational logs to understand and detect performance regression.

In this thesis, we propose several techniques to detect performance regression in the context of DevOps J. Chen (2020). To collect performance regression data, we propose a statistically rigorous approach that identifies performance regression introducing code changes J. Chen and Shang (2017). In order to detect performance regression as early as possible, we propose an approach to predicting whether a test would manifest performance regressions in a code commit J. Chen, Shang, and Shihab (2020). We also propose an approach that predicts whether a configuration option manifests a performance variation issue. To assist performance regression detection in operations, we first propose to use a recovery workload to detect performance regression based on field execution logs J. Chen, Shang, Hassan, Wang, and Lin (2019). The goal of DevOps is to combine software development and operations to shorten the delivery cycle. Our approaches can be used to detect

performance regression in the loop of Devs and Ops. In particular, our performance regression prediction during development can help software release and deploy early. And the results of performance regression detection during development can assist operators to monitor the specific performance metrics. For example, if most of the performance regressions are the performance metric CPU, then the operators can monitor the system CPU usage. In our approach, we use the operational data to recover filed-representative workloads. Therefore, during development, the practitioners can design a better load test based on such recovery workload. This means assisting performance regression detection using operational data.

## 1.2 Research hypothesis

The practices of DevOps generate two sources of data about software: development data and field data. Development data, generated during software development, includes a large amount of historical information of software development. A typical example of development data is source code changes. Field data, generated during operations, consists of available and valuable information about how the system operates. System execution logs and performance metrics are two kinds of typical field data. In this dissertation, we plan to propose automated approaches to detecting performance regression by mining a large amount of development and field data. Our research hypothesis is as follows:

> ***Software development historical repositories and field operation logs, which are valuable and readily available resources, can be used to detect performance regression in the context of DevOps.***

We will validate our research hypothesis based on two aspects:

- **Frequent performance assurances during development:** We start off by examining the prevalence and root-cause of performance regression introducing code changes. We then propose an approach that automatically predicts whether a test would manifest performance

regressions given a code commit. Afterwards, we propose an approach to predicting whether a configuration option manifests a performance variation issue.

- **Assisting performance assurances with operational data:** We study the use of recovery workload to detect performance regression using field execution logs.

Therefore, this thesis intends to 1) detect or predict performance regression introducing source code changes in the early stage during development and 2) assist operators to analyze a huge of field execution logs to detect system performance regression during operations.

## 1.3 Thesis Overview

In particular, we further divide the scope of our main content of this thesis into two parts of the research questions. One is frequent performance assurances during development, another is assisting performance assurances with operational data. The first part consists of Chapter 3, Chapter 4, and Chapter 5. The second part consists of Chapter 6.

### 1.3.1 Chapter 2: Background and Literature Review

This chapter first presents an overview of the relevant background to our research. We then present the related works of our research, including empirical studies on software performance, performance regression detection, recovering workload, performance load testing, performance regression prediction, and performance of configuration. We conclude Chapter 2 by mentioning the limitations of our literature review and recapping our findings from it.

### 1.3.2 Chapter 3: What are the prevalence and root-causes of performance regressions introducing code changes?

Software changes during development, i.e. source code changes, may cause a performance regression. If we can detect performance regression as early as possible, the amount of required resources would be significantly reduced if developers were notified whether a code change introduces performance regressions during development. Prior research has conducted empirical studies

on performance bugs (Huang et al., 2014; G. Jin et al., 2012; Zaman et al., 2012, 2011), using the reported performance bugs in issue reports (like JIRA issues). However, there may exist many more performance issues, such as performance regressions, that are not reported as JIRA issues. Moreover, there is no existing data of performance regression introducing code changes. Therefore, in this chapter, we start off by examining how prevalent are performance regression introducing changes and the root-cause of performance regression.

### 1.3.3 Chapter 4: Can we predict tests that manifest performance regressions at the commit level?

Although automated techniques are proposed to detect performance regressions (Heger et al., 2013; Luo et al., 2016; T. H. Nguyen et al., 2012; T. H. D. Nguyen et al., 2014; Shang et al., 2015), challenges in the practice of performance regression detection still exist. First, 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. Therefore, to overcome these challenges, we propose an approach that automatically predicts whether a test would manifest performance regressions given a code commit.

### 1.3.4 Chapter 5: Can we predict whether a configuration option manifests performance variation?

Large systems tend to be highly configurable, which allows users to change the behaviour of these systems by simply altering the values of certain configuration options Sayagh, Kerzazi, Adams, and Petrillo (2018). However, such flexibility comes with a cost. In fact, such software systems suffer throughout their evolution from what we refer to as "Inconsistent Options Performance Variation" (***IoPV***). An *IoPV* indicates, for a given commit, that a performance regression or improvement is inconsistent when altering the values of the same option. For instance, a new change might not suffer from any performance regression under the default configuration (i.e., when all the options are set to their default values), while altering one option's value manifest a hidden

regression. Similarly, when developers improve the performance of their systems, performance regression might be manifested under a subset of the existing configurations. Unfortunately, such hidden regressions are harmful as they can go unseen to the production environment. Therefore, in this chapter, we first quantify how consistent is a performance regression or improvement among the values of an option. We then build a model to predict whether a configuration option manifests performance variation.

### 1.3.5 Chapter 6: Can we generate load tests using log-recovered workloads at varying granularities of user behavior?

Activities of the operation team generate a large amount of operational data, i.e., thousands of performance counters and large-scale execution logs. Such operational data can be used to assist in performance regression detection. If performance regression is hidden during development, such large valuable operational data can be used to assist in performance assurances. The process of operations generates huge execution logs. It is time-consuming and error-prone for the performance analyst to analyze such large-scale logs. One of the goals of analyzing such execution logs is to recover workload to support load testing a system. Prior research often captures the workload as the frequency of user actions (Cohen et al., 2005; Shang et al., 2015; Syer, Shang, Jiang, & Hassan, 2017). However, there exists much valuable information in the context and sequences of user actions. Such richer information would ensure that the load tests that leverage such workloads are more field-representative. In this chapter, we study the use of system execution logs when recovering workloads for use in the load testing of large-scale systems.

## 1.4 Thesis Contributions

The main contributions of this thesis are the following:

- We propose a statistically rigorous approach to identifying performance regression introducing code changes. Further research can adopt our methodology in studying performance regressions (Chapter 3).

- We find six root-causes of performance regressions that are introduced by code changes. 12.5% of the manually examined regressions can be avoided or their performance impact may be reduced (Chapter 3).

- We propose an approach that can 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 (Chapter 4).

- Our findings highlight the importance of considering different configurations when performing performance regression detection, and that leveraging predictive models can mitigate the difficulty of exhaustively consider all configurations of a system during such a process (Chapter 5).

- We introduce an approach for recovering workload using user contextual information and frequent action sequences from system execution logs. Our approach has been adopted in practice to assist in the testing and operation of an ultra-large-scale industrial software system (Chapter 6).

## 1.5 Thesis organization

This rest of this thesis is organized as follows: Chapter 2 provides the background and related work of performance regression detection in DevOps. Motivated by the prior research, Chapter 3 studies the prevalence of performance regression and identifies root-causes of performance regressions introducing code changes. Chapter 4 predicts whether a test would manifest performance regressions given a code commit. Chapter 5 builds a model to predict whether a configuration option manifests performance variation. Chapter 6 studies the use of system execution logs to recovery field-representative workloads to detect performance regression. Finally, Chapter 7 concludes this thesis and presents our future work.

# Chapter 2

# Background and Literature Review

In this chapter, we first present an overview of the relevant background to our research, including DevOps, software performance, performance regression, and performance testing. We then present the related works of our thesis, including empirical studies on software performance, performance regression detection, performance regression prediction, and performance of configuration.

## 2.1 Background

### 2.1.1 DevOps

DevOps is a set of software practices to smooth the deploying release of a software system between development and operations Hüttermann (2012). The integration of Dev and Ops intends to allow a more flexible reaction to changes, i.e., new features or bug fixes Brunnert et al. (2015). Such changes involve development and operation teams, which forces Dev and Ops teams to work closer together. From a technical perspective, the tasks within the process of DevOps include code development, code review, continuous building, continuous testing, application pre-deployment, releasing an application, application performance monitoring Wikipedia (2016). To this end, Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment (CDE) Shahin, Babar, and Zhu (2017) have been introduced.

### 2.1.2 Software performance

Software performance was born at the beginning of the 1990s due to the increasing complexity of software system Cortellessa, Di Marco, and Inverardi (2011). Software performance is an indicator of how well a software system or component meets its objectives for timelines Smith and Williams (2002). The timelines include two factors: the processing time spent by software (e.g. CPU processing time), and the waiting time spent by the software while waiting to access system resources. Performance is an important aspect of software quality. In fact, large software system failures are often due to performance issues rather than functional bugs Weyuker and Vokolos (2000). The goals of performance are typically defined by setting upper and lower bounds for response time and throughput of a system and physical level measurements such as CPU, memory, and I/O.

### 2.1.3 Performance regression

A software performance regression is that the software system performs more slowly or uses more resources Shang et al. (2015). If the new version has worse performance than the old version, then there exists performance regression. Performance regression is a typical example of performance issues. Such regressions may compromise the user experience, increase the operating cost of the system, and cause field failures. Such regressions may also have financial impacts. Page load slowdown is one of the typical examples of performance regression. A report by Amazon shows that one-second page slowdown may cause a 1.6 billion loss of revenue Kit (2010).

### 2.1.4 Performance testing

Performance testing is the process of measuring system performance-related metrics Z. M. Jiang and Hassan (2015). Such performance metrics include response time, throughput, and resource utilization. A typical example of performance testing is shown in Figure 2.1. In practice, developers set up the testing environment and define the workload. Then the same requests are sent to exercise the system and the performance metrics are collected. Finally, operators compare the two performance metrics and identify the performance regression.

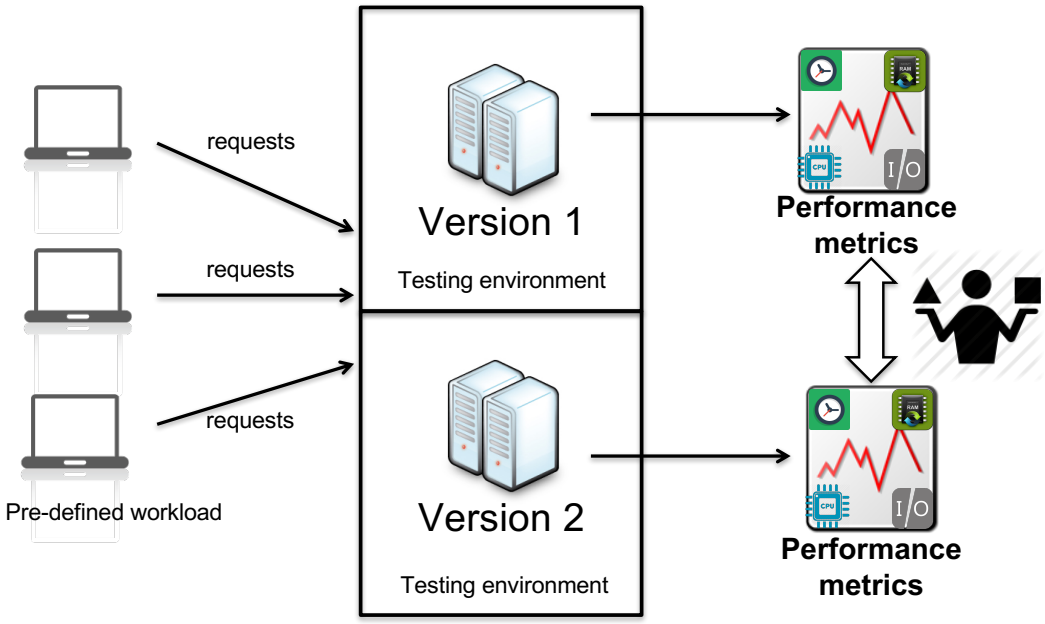



Figure 2.1: An example of performance regression testing

One of the goals of performance assurance activities in DevOps 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. There exists rich software quality research that examines the impact of code changes on software quality during development and operations (Heger et al., 2013; Luo et al., 2016; T. H. Nguyen et al., 2012; T. H. D. Nguyen et al., 2014; Shang et al., 2015); while a majority of prior findings do not use performance regression as a sign of software quality degradation.

## 2.2 Literature review

In this section, we perform a literature review to study state-of-the-art research related to software performance.

### 2.2.1 Paper selection

Software performance regression detection involves many research areas. We divide the related works into three aspects, including understand software performance bug, understand performance

regression detection, and the performance of configuration.

First, we wish to understand the software performance bug. Therefore, we select the papers that conduct an empirical study on software performance bugs. Second, we wish to study the techniques used to detect performance regression. On one hand, some researchers trust that the most reliable approach to detect performance regression is performance testing. And performance testing consists of three steps, including designing a proper workload, executing performance tests, and analyzing performance testing results. On the other hand, some researchers consider that statistical defect prediction techniques can be used to detect performance regression without the need to execute costly performance testing. Therefore, to understand the performance regression detection techniques, we categorize the related works into recovering a proper workload, performance load testing, analyzing performance testing results, and software defect prediction. Finally, all too often a software system performance regression is caused by configuration options. Therefore, we select papers that relate to understanding the performance of configuration.

### 2.2.2 Empirical studies on software performance

Empirical studies are conducted in order to study performance issues (Han, Yu, & Lo, 2018; Huang et al., 2014; G. Jin et al., 2012; Leitner & Bezemer, 2017; Nistor, Jiang, & Tan, 2013; Zaman et al., 2012, 2011). To understand the difference between performance and non-performance bugs, Zaman et al. (2012) conducts a qualitative study on 400 performance and non-performance bugs reports across four dimensions (Impact on the stakeholder, Context of the bug, Bug fix, and Bug fix validation) in two open-source web browsers Mozilla Firefox and Google Chrome. The authors find that performance bugs are more likely due to regression and the performance bugs are more possible to block release than non-performance bugs. Performance bugs are more difficult to solve and need more collaboration than fixing non-performance bugs. However, not all regressions are unexpected since performance regression may not be performance bugs and the performance may still meet the requirement, even though the performance is worse than the previous version. This paper reveals that developers may need to spend more time fixing performance bugs than non-performance bugs.

G. Jin et al. (2012) study 109 real-world performance bugs that are from five software projects (Apache, Chrome, GCC, Mozilla, and MySQL) to provide guidance for software engineer for bug

avoidance, performance testing, bug detection, bug fixing. This paper discovers three common root-cause patterns, uncoordinated functions, skippable function, and synchronization issues. The authors also find that performance bugs are often introduced by workload mismatch, API misunderstanding. Due to the diversity and complexity of software workload, performance bugs are often introduced when developers misunderstand the workload. The study calls in-depth research on performance diagnosis, performance testing, performance bug detection.

Nistor et al. (2013) conduct a comprehensive study to compare performance and non-performance bugs to understand how performance bugs are discovered, reported, and fixed. The authors find that fixing performance bugs is as likely to introduce new functional bugs as fixing non-performance bugs. Fixing performance bugs is more difficult than fixing non-performance bugs. Many performance bugs are found through code reasoning and profiling. Such results call in-deep need too support to address performance bugs.

Leitner and Bezemer (2017) conduct a study on 111 open-source Java-based projects from GitHub and use a combination of quantitative and qualitative research methods to investigate the performance tests. The authors find that developers have not yet understood how to conduct performance tests and often mix performance tests within the functional test suite. There is a lack of standard guidelines to conduct performance tests in an easy and powerful way. This paper argues that future performance testing should implement a more flexible testing framework to support low-friction testing or can combine functional test and performance testing together.

Han et al. (2018) study 300 bug reports from three large open-source projects. The authors find that most of the performance bugs occur for specific combinations of data input and configurations. They also propose a framework named *PerfLearning* to extract such data input and configurations from bug reports to generate test frames. Huang et al. (2014) studied real-world performance issues. They propose an approach to improve the efficiency of performance regression testing by leveraging a static analysis technique to estimate the risk of a given commit in introducing a performance regression.

The vast amounts of research on software performance bugs signify its importance and motivate our work. Prior studies on performance typically are based on either limited performance issue reports or release of the software. However, the limited amount of issue reports and releases of the

software hides the prevalence of performance regressions. In our thesis, we evaluate performance at the commit level. Therefore, we are able to identify more performance regressions and are able to observe the prevalence of performance regression introducing changes in DevOps.

### 2.2.3 Performance regression detection

In order to conduct performance testing, developers first set up the testing environment and pre-define the workload. Second, the same requests are sent to test the system and the performance metrics are collected. Finally, operators compare the two performance metrics and identify the performance regression. Therefore, in this subsection, we categorize the related works into three aspects, recovering workload, performance load testing, and analyzing performance testing results to detect performance regression.

#### Recovering Workload

Workload recovery is an essential part in the performance assurance of large-scale systems. Prior research proposes approaches for recovering workloads to assist in the design of load tests Vögele, van Hoorn, Schulz, Hasselbring, and Krcmar (2018), validating whether load tests are field representative as production Syer et al. (2017), optimizing system performance (Summers, Brecht, Eager, & Gutarin, 2016; H. Xi et al., 2011) and detecting system performance issues (Cohen et al., 2005; Hassan, Martin, Flora, Mansfield, & Dietz, 2008; Shang et al., 2015; Syer et al., 2017). All the above prior work illustrates the value and importance of recovering representative workloads.

Prior approaches for recovering and replaying workloads can be categorized along the granularity of the captured user actions. One may choose a coarse-granularity by recovering only the type of workload from a system, or to the other extreme, considering each individual user and replaying their individual workload one by one. One may anonymize all high-level user behaviors and only consider the physical metrics such as CPU (Cohen et al., 2005; Shang et al., 2015), I/O (Busch et al., 2015; Haghdoost, He, Fredin, & Du, 2017; Seo et al., 2014; Yadwadkar, Bhattacharyya, Gopinath, Niranjan, & Susarla, 2010) and other system resources Cortez et al. (2017). One may choose a finer granularity by building complex models such as Hidden Markov Models Yadwadkar et al. (2010) to capture the details for each user. A pilot study by Cohen et al. (2005) demonstrates that

grouping workloads into a smaller number of clusters outperforms having one unified workload. Intuitively, recovering a workload at a too fine or too coarse grained detail is neither desired. A too coarse-grained approach may miss the important characteristics of user behaviour, leading to a non-representative workload, while a too fine grained approach may lead to a workload that is costly to replay and maintain.

To achieve an optimal granularity of user behavior, prior research often chooses event or action-driven approaches for workload recovery (Hassan et al., 2008; Summers et al., 2016; Syer et al., 2017; Vögele et al., 2018; H. Xi et al., 2011). However, there exists extensive research on execution log analysis that demonstrates the value of considering contextual information and sequence of actions for various software engineering tasks (Barik, DeLine, Drucker, & Fisher, 2016; Beschastnikh, Brun, Ernst, & Krishnamurthy, 2014; Beschastnikh, Brun, Schneider, Sloan, & Ernst, 2011; Fu et al., 2012; S. He et al., 2018; Z. M. Jiang, Hassan, Hamann, & Flora, 2008b, 2009; Lin, Zhang, Lou, Zhang, & Chen, 2016; Oliner, Ganapathi, & Xu, 2012; Shang et al., 2013). Such extensive usage of contextual information and user action sequences in log analysis motivates our research to leverage the similar information to recover richer workloads from execution logs for generating load tests.

In this thesis, our focus is primarily on exploring whether research approaches work in practice. In particular, compared to prior research, our work uses more valuable contextual and action sequence information from execution logs to recover workloads.

### Performance Load Testing

Load testing is a type of performance testing that intends to measuring system's performance under a volume of users performing transactions. Due to the importance of load testing, a large amount of previous research on load testing has been proposed. Z. M. Jiang and Hassan (2015) survey 196 research papers to understand three aspects in load testing, including load testing designing, executing a load test and load test data analysis. The authors summarize the current techniques of the three phases and propose open research problems for load testing. T. Chen et al. (2017) present an industrial experience report on load testing in large-scale systems. This paper presents the challenges of load testing in practice and proposes a general guideline for improving the effectiveness of load testing designing, executing and analyzing.

There exist a volume of previous researchers studying load testing to help performance analyst detect and tackle performance-related problems. Jiang et al. (Z. M. Jiang, 2010; Z. M. Jiang et al., 2008b, 2009) conduct a series of studies on the use of load testing to automatically identify system's functional or performance problems. Z. M. Jiang et al. (2008b) propose an approach to discover normal system behavior and reveal functional anomalies by mining execution logs generated during load testing. Z. M. Jiang et al. (2009) also propose an approach to automatically generate a report to detect and rank potential performance problems by using system execution logs. In their approach, the authors first conduct log abstraction, extracting sequence performance and performance summarization. Then they identify performance problems using statistical analysis on the performance summarization data. Gao and Jiang (2017) evaluate the impact of environment variations on the results of load testing. The authors conduct their case studies in three open-source systems. They find that environment variation has an impact on the system's performance. In addition, the authors propose ensemble performance models to predict system performance in realistic runs. Trubiani, Bran, van Hoorn, Avritzer, and Knoche (2018) present an approach to automatically detecting two performance anti-patterns, circuitous treasure hunt (CTH) and extensive processing (EP), using load testing and profiling data. In addition, the authors apply software refactoring to solve those two performance anti-patterns.

Performance load testing is too complex and time-consuming, i.e., designing a proper load, executing or replaying a load test and analyzing the results of a load test. In this thesis, we focus on generating load tests using log-recovered workloads based on field logs.

### 2.2.4 Analyzing performance testing results to detect performance regression

Extensive prior research has proposed automated techniques to detect performance regression. There are two categories of performance regression detection approaches, measurement-based and model-based.

Measurement-based approach measures performance metrics and compares these performance metrics between two consecutive versions of a system to detect performance regression. Nguyen *et al*. (T. H. Nguyen et al., 2012; T. H. D. Nguyen et al., 2011, 2014) conduct a series of studies on performance regressions. Nguyen *et al*. apply control charts to analyze performance counters

across test runs to detect performance regression automatically. They construct the control chart to detect performance regressions by setting upper and lower bounds of performance counters. Malik, Hemmati, and Hassan (2013) propose four approaches to automatically filter performance counters subset of load testing to assist to detect performance deviations. The authors conduct their approaches on both industrial and open-source systems and find that supervised approach named *WRAPPER* is more efficient. Foo et al. (2010) propose an approach to comparing the test results to performance metrics derived from performance regression testing repositories to detect potential performance regressions.

Model-based approach builds a detected model to detect performance regressions. Cohen et al. (2005) et al. show an implication that it is ineffective and not enough to index and identify performance problems with simple records of raw system metrics. Therefore, the authors present an approach to capture signatures representing system states from a running system and cluster such signatures to detect recurrent or similar performance problems. Bodík, Goldszmidt, and Fox (2008) employ logistic regression with L1 regularization models to construct signatures to improve Cohen *et al.*'s work. Foo et al. (2015) propose an approach to build ensembles of models to detect performance regressions in heterogeneous environments. The authors find that ensembles of associated rules can achieve high precision and recall in the detection of performance regression. Xiong, Pu, Zhu, and Griffith (2013) propose a model-driven framework to diagnose application performance. Such a framework uses linear regression to build the prediction model to automatically diagnose the system performance in a cloud environment and lead to the root cause of performance problems.

Prior research on the detection of performance regressions is all designed to be conducted after the system is built and deployed. Large amounts of resources are required to detect, locate, understand and fix performance regressions at such a late stage in the development cycle. In this thesis, we detect performance regressions at the commit level.

### 2.2.5 Performance regression prediction

To reduce the resources and time in performance regression testing, statistical just-in-time prediction models can be used to predict performance regression. However, there is a lack of research

that studies the performance regression prediction. Typically, prior software defect prediction research focuses on functional bugs rather than performance bugs. We infer that the general software defect prediction techniques are useful in the prediction of the performance regression. Therefore, in this subsection, we select papers that relate to software defect prediction.

Mockus and Weiss (2000) are the first one to use metrics that describes the characteristics of software changes to build a linear regression model to predict the probability of failure after code changes. Kamei et al. (Kamei et al., 2016; Kamei & Shihab, 2016; Kamei et al., 2013) conduct a series of studies on commit-level defect prediction. Kamei et al. (2013) 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 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 and Shihab (2016) present an overview in the research field of defect prediction, which intends to introduce and help researchers and practitioners understand previous studies on defect prediction, and highlight important challenges for future research. To build the prediction model, researchers need to extract and select a list of metrics. F. Zhang, Mockus, Keivanloo, and Zou (2014) 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, Whitehead, Akella, and Kim (2013) realize that the more features prediction model learned, the more the model predicts insufficient performance. Hence, Shivaji et al. perform multiple feature selection algorithms to reduce the metrics that predict software defects. P. He, Li, Liu, Chen, and Ma (2015) 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 (2016) 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, Miranskyy, and Mazzawi (2016) 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. In order to find whether or not unsupervised models perform better than the supervised models in effort-aware just-in-time defect prediction, Yang et al. (2016) 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 at the commit level. Since our study is to predict performance regression, we extract traditional metrics that are widely used in prior studies (Kamei et al., 2013; Mockus & Weiss, 2000) and also include performance-related metrics in our classifiers.

### 2.2.6 Performance of configuration

A large body of research efforts has been conducted on software configuration, which mainly focuses on understanding configuration problems, preventing configuration errors, and debugging configuration errors. Few research efforts consider the performance aspect of software configuration.

#### Understanding configuration

Configuration makes a software system complex Sayagh et al. (2018), which leads to configuration errors that are severe, common, and hard to debug Yin et al. (2011). For instance, D. Jin, Qu, Cohen, and Robinson (2014) find that configuration options add more complexity to the development and testing of highly configurable software systems. Han and Yu (2016) find that configuration options are responsible for 59% of the performance bugs. Gousios, Karakoidas, and Spinellis (2006) observe that the configuration of the garbage collectors has an impact on the performance of server applications. Furthermore, Sayagh et al. (Sayagh & Adams, 2015; Sayagh, Kerzazi, & Adams, 2017) found that the impact of a configuration option can spread to multiple layers of an architectural stack.

A second line of research proposed and evaluated different approaches to identify misconfigured

configuration options. Dong et al.(Dong, Andrzejak, & Shao, 2015; Dong, Ghanavati, & Andrzejak, 2013) leverage the slicing technique to identify the misconfigured option for a given error message or exception. Rabkin and Katz (2011) leverage a data flow analysis technique to identify for each option, which source code lines it might impacts. Attariyan and Flinn (2010) combined dynamic control and data flow analysis to identify misconfigured options. Zhang et al. (S. Zhang, 2013; S. Zhang & Ernst, 2014) compared the trace of a correct execution against the trace of an incorrect execution to identify culprit options. Prior systematic literature review Sayagh et al. (2018), the work of Tianyin and Yuanyuan (2015) and Andrzejak, Friedrich, and Wotawa (2018) further details about the existing configuration debugging approaches.

### The Performance of Configurations

Another line of research considers the identification of the optimal configurations for a software system and the debugging of performance errors that are caused by configuration options. Attariyan, Chow, and Flinn (2012) propose an approach based on dynamic taint analysis technique to identify the option that causes a performance error. Siegmund, Grebhahn, Apel, and Kastner (2015) build mathematical models that describe the impact of a configuration on software performance based on each option's value. Raghavachari, Reimer, and Johnson (2003) proposed an iterative approach to identify an optimal configuration in terms of performance. Their approach consists of selecting for a J2EE web application a first configuration, compare its performance to a second configuration until the optimal configuration is found. Similarly, Diao, Hellerstein, Parekh, and Bigus (2003) proposed an approach that automatically adjusts the values of existing configuration options at run-time to optimize the CPU and memory usage objectives.

Li, Chen, Hassan, Nasser, and Flora (2018) leverage performance monitoring data and execution logs to dynamically optimize the values of performance-related configuration options according to varying workloads in the field. J. Guo, Czarnecki, Apel, Siegmund, and Wasowski (2013) leverage non-linear regression to suggest an optimal configuration. However, collecting a large amount of data for training a model that predicts the performance of a configuration is expensive. Therefore, Sarkar, Guo, Siegmund, Apel, and Czarnecki (2015) evaluate the progressive and projective sampling to train a model that predicts the performance of configuration. For their

initial training sample, they consider data on which each option is enabled at least once. Other efforts identified the optimal configuration options in terms of performance by leveraging existing optimization approaches, i.e., iterative search Lengauer and Mössenböck (2014), multi-objective optimization Singh, Bezemer, Shang, and Hassan (2016), and smart hill climbing B. Xi, Liu, Raghavachari, Xia, and Zhang (2004).

### 2.2.7 Summary

In this chapter, we present a literature review on the state-of-the-art performance regression research. From our literature review, we find that performance is an important aspect of software quality. However, there is a lack of performance testing and few developers create and maintain the performance tests in practice. Performance testing is more complicated than functional testing due to the high number of confounding factors, such as the hardware platform, algorithm design, data structure, program configuration, database access, the scale of workload. What is more, performance testing is too time-consuming and causes a high cost. Motivated by the findings of our literature review, we first study the prevalence and root-cause of performance regressions (Chapter 3). We then propose an approach to predict performance regression at the commit level (Chapter 4). Next, we study how can configuration impacts performance regressions (Chapter 5). Finally, we propose an approach to recovery workload (Chapter 6).

# Chapter 3

# What are the prevalence and root-causes of performance regressions introducing code changes?

Performance is an important aspect of software quality. In fact, large software systems failures are often due to performance issues rather than functional bugs. One of the most important performance issues is performance regression. Examples of performance regressions are response time degradation and increased resource utilization. Although performance regressions are not all bugs, they often have a direct impact on users' experience of the system. Due to the possible large impact of performance regressions, prior research proposes various automated approaches that detect performance regressions. However, the detection of performance regressions is conducted after the fact, i.e., after the system is built and deployed in the field or dedicated performance testing environments. On the other hand, there exists rich software quality research that examines the impact of code changes on software quality; while a majority of prior findings do not use performance regression as a sign of software quality degradation. In this study, we perform an exploratory study on the source code changes that introduce performance regressions. We conduct a statistically rigorous performance evaluation on 1,126 commits from ten releases of *Hadoop* and 135 commits from five releases of *RxJava*. In particular, we repetitively run tests and performance

micro-benchmarks for each commit while measuring response time, CPU usage, Memory usage and I/O traffic. We identify performance regressions in each test or performance micro-benchmark if there exists statistically significant degradation with medium or large effect sizes, in any performance metric. We find that performance regressions widely exist during the development of both subject systems. By manually examining the issue reports that are associated with the identified performance regression introducing commits, we find that the majority of the performance regressions are introduced while fixing other bugs. In addition, we identify six root-causes of performance regressions. 12.5% of the examined performance regressions can be avoided or their impact may be reduced during development. Our findings highlight the need for performance assurance activities during development. Developers should address avoidable performance regressions and be aware of the impact of unavoidable performance regressions.

## 3.1 Introduction

The rise of large-scale software systems (e.g., Amazon.com and Google Gmail) has posed an impact on people's daily lives from mobile devices users to space station operators. The increasing importance and complexity of such systems make their quality a critical, yet extremely difficult issue to address. Failures in such systems are more often associated with performance issues, rather than with feature bugs Weyuker and Vokolos (2000). Therefore, performance assurance activities are an essential step in the release cycle of large software systems.

Performance assurance activities aim to identify and eliminate performance regressions in each newly released version. A software system is considered to have performance regressions when the performance of the system (or a certain feature of the system) is worse than before. Examples of performance regressions are response time degradation and increased resource utilization. Such regressions may compromise the user experience, increase the operating cost of the system, and cause field failures. We note that performance regression may not be performance bugs, since the performance may still meet the requirement, even though the performance is worse than the previous version. However, detecting performance regressions is an important task since such regressions may have a direct impact on the user experience of the software system, leading to significant

financial and reputational repercussions. For example, Mozilla has a strict policy on performance regressions Joel (2017), which clearly states that unnoticed and unresolved performance regressions are not allowed.

Due to the importance of performance regression, extensive prior research has proposed automated techniques to detect performance regressions (Heger et al., 2013; Luo et al., 2016; T. H. Nguyen et al., 2012; T. H. D. Nguyen et al., 2014; Shang et al., 2015). However, detecting performance regressions remains a task that is conducted after the fact, i.e., after the system is built and deployed in the field or dedicated performance testing environments. Large amounts of resources are required to detect, locate, understand and fix performance regressions at such a late stage in the development cycle; while the amount of required resources would be significantly reduced if developers were notified whether a code change introduces performance regressions during development.

On the other hand, prior software quality research typically focuses on functional bugs rather than performance issues. For example, post-release bugs are often used as code quality measurement and are modeled by statistical modeling techniques in order to understand the relationship between different software engineering activities and code quality Hassan (2009). In addition, bug prediction techniques are proposed to prioritize software quality assurance efforts (N. Nagappan & Ball, 2005; N. Nagappan, Ball, & Zeller, 2006; Zimmermann, Premraj, & Zeller, 2007) and assess the risk of code changes Kamei et al. (2013). However, performance regressions are rarely targeted in spite of their importance.

In this work, we perform an exploratory study on the performance regression introducing code changes. We conduct a statistically rigorous performance evaluation on 1,126 commits from ten releases of *Hadoop* and 135 commits from five releases of *RxJava*. In particular, the performance evaluation of each code commit with impacted tests or performance micro-benchmarks is repeated 30 times independently. In total, the performance evaluation lasts for over 2,000 hours. With the performance evaluation results, we identify performance regression introducing changes with statistical tests. To the best of our knowledge, our work is the first **that extensively evaluates and studies performance at the commit level**.

By examining the identified performance regression introducing changes, we find that performance regression introducing changes are prevalent during software development. The identified

performance regressions are often associated with a complex syndrome, i.e., multiple performance metrics have performance regression. Interestingly, we find that performance regression introducing changes also improve performance at the same time. Such results show that developers may not be aware of the existence of performance regressions, even when they are trying to improve performance. By studying the context and root-causes of performance regression introducing changes, we find that performance regressions are mostly introduced while fixing other functional bugs. Our manual examination on the performance regression introducing code changes identifies six code level root-causes of performance regressions, where some root-causes (such as skippable functions) can be avoided. In particular, we find that 12.5% of the examined performance regressions in the two subject systems can be avoided or their impact on performance may be reduced during development.

Our study results shed light on the characteristic of performance regression introducing changes and suggest the lack of awareness and systematic performance regression testing in practice. Based on our findings, performance-aware change impact analysis and designing inexpensive performance tests may help practitioners better mitigate the prevalent performance regression that is introduced during software development.

The rest of this work is organized as follows: Section 3.2 presents our subject systems and our approach to identifying performance regression introducing code changes. Section 3.3 presents the results of our case study. Section 3.4 presents threats to the validity of our study. Section 3.5 presents prior research that related to this work. Finally, Section 3.6 concludes this work.

## 3.2 Case Study Setup

In this section, firstly we discuss our subject systems and our experimental environment. Then we present our approach to identifying performance regression introducing changes.

### 3.2.1 Subject systems

We choose two open-source projects, *Hadoop* and *RxJava* as the subject systems of our case study. *Hadoop* White (2009) is a distributed system infrastructure developed by the Apache Foundation. *Hadoop* performs data processing in a reliable, efficient, high fault tolerance, low cost, and

scalable manner. We choose *Hadoop* since it is highly concerned with its performance and has been studied in prior research in mining performance data Syer et al. (2017). *RxJava* is a library for composing asynchronous and event-based programs by using observable sequences and it carries the JMH benchmarks test options. *RxJava* is a *Java* VM implementation of reactive extensions. *RxJava* provides a slew of performance micro-benchmarks, making it an appropriate subject for our study. We choose the most recent releases of the two subject systems. The overview of the two subject systems is shown in Table 3.1.

Table 3.1: Overview of our subject systems.

| Subjects | Version | Total lines of code (K) | # files | # tests |
|---|---|---|---|---|
| Hadoop | 2.6.0 | 1,496 | 6,086 | 1,664 |
| | 2.6.1 | 1,504 | 6,117 | 1,679 |
| | 2.6.2 | 1,505 | 6,117 | 1,679 |
| | 2.6.3 | 1,506 | 6,120 | 1,681 |
| | 2.6.4 | 1,508 | 6,124 | 1,683 |
| | 2.6.5 | 1,510 | 6,127 | 1,685 |
| | 2.7.0 | 1,552 | 6,413 | 1,771 |
| | 2.7.1 | 1,556 | 6,423 | 1,775 |
| | 2.7.2 | 1,562 | 6,434 | 1,784 |
| | 2.7.3 | 1,568 | 6,439 | 1,786 |
| RxJava | 2.0.0 | 164 | 1,107 | 76 |
| | 2.0.1 | 242 | 1,513 | 76 |
| | 2.0.2 | 243 | 1,524 | 76 |
| | 2.0.3 | 244 | 1,524 | 76 |
| | 2.0.4 | 244 | 1,526 | 76 |

### 3.2.2 Identifying performance regression introducing changes

In this subsection, we present our approach to identifying performance regression introducing changes. In general, we extract every commit from the version control repositories (Git) of our subject systems and identify impacted test cases of each commit. Afterward, we chronologically evaluate the performance of each commit using either the related test cases (for *Hadoop*) or performance micro-benchmarks (for *RxJava*). Finally, we perform statistical analysis on the performance evaluation results to identify performance regression introducing changes. The overview of our approach is shown in Figure 3.1.

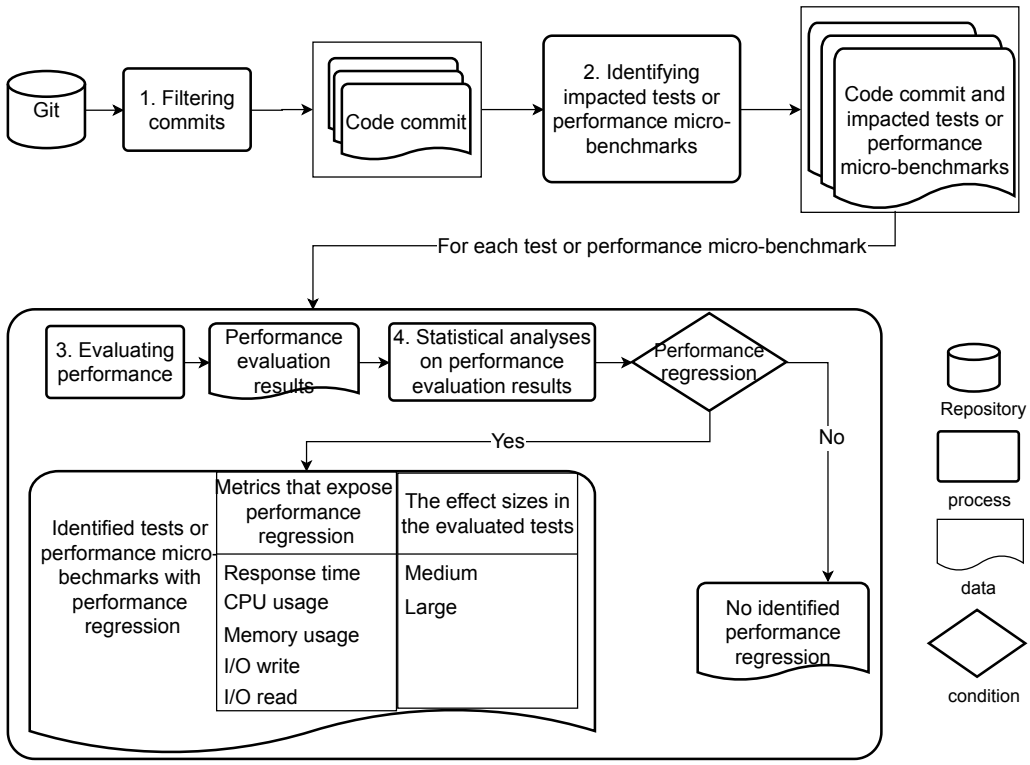



Figure 3.1: An overview of our approach that identifies performance regression introducing changes.

### Filtering commits

As the first step of our approach, we start off by filtering commits in order to focus on commits that are more likely to introduce performance regressions. In particular, we use the *git log* command to list all the files that are changed in each commit. We only extract the commits that have source code changes, i.e., changes to *.java* files.

When we extract the commits related to a task, we find that there may exist multiple commits that are made to accomplish one task, making some of the commits temporary. We would like to avoid considering the performance regressions that are introduced in such temporary commits. Since *Hadoop* uses JIRA as their issue tracking system and *RxJava* uses the internal issue tracking system in Github, we use the issue id that is mentioned in each commit message to identify the task of each commit. If multiple commits are associated with the same issue, we only consider the snapshot of the source code after the last commit.

**Preparing impacted tests**

*Identifying impacted tests.* In order to evaluate the performance of each code commit, we use the tests and performance micro-benchmarks that are readily available in the source code of our subject systems. As mature software projects, each subject system consists of a large amount of test cases. For example, *Hadoop* release 2.7.3 contains 1,786 test cases in total. Exercising all test cases may cause two issues to our performance evaluation: 1) the test cases that are not impacted by the code change would dilute the performance impact from the code changes and introduce noise in the performance evaluation and 2) the large amounts of un-impacted test cases would require extra resources for performance evaluation (e.g., much longer test running time).

Therefore, in this step, we leverage a heuristic to identify impacted tests for each commit. In particular, we find that *Hadoop* test cases follow a naming convention that the name of the test files contains that same name of the source code files being tested. For example, a test file named *TestFSNamesystem.java* tests the functionality of *FSNamesystem.java*. Hence, for each changed source code file in a commit, we automatically identify the test files. If multiple commits are associated with one issue, we consider all the tests that are impacted by these commits, but will later only evaluate performance on the last one of these commits.

*Dealing with changed tests.* Some commits may change source code and test code at the same time. Such changed test cases will bias the performance evaluation if ample testing logic is added, removed, or modified in the test cases. In order to minimize the impact of changed test cases in performance evaluation, we opt to use the test code before the code change. Since the new version of the test cases may include new features of the system, which is not the major concern of performance regression. However, in the cases where old test cases cannot compile or fail, we use the new test cases, since the failure of the compilation or the tests indicates that the old feature may be outdated. Finally, if both new and old test cases are failed or un-compilable, we do not include this test in the performance evaluation. In total, we have 132 tests with 106 commits that use the new tests to evaluate performance and 21 test with 19 commits not included in our performance evaluation. There exist only six commits that are not included because all of their tests are either un-compilable or failed.

*Leveraging micro-benchmarks for* RxJava. Fortunately, *RxJava* provides a slew of micro-benchmarks with the goal of easing performance evaluation. Micro-benchmark is used to evaluate different small units' performance in *RxJava*.We opt to run all 76 micro-benchmarks from *RxJava*. In the rest of this work, we also refer to these micro-benchmarks as test cases to ease the description of our results.

**Evaluating performance**

In this step, we exercise the prepared test cases and the performance micro-benchmarks to evaluate performance of each considered commit. Our performance evaluation environment is based on Azure node type Standard F8s (8 cores, 16 GB memory). We do not create a cluster of machines to evaluate the performance for *Hadoop* since the exercised tests all run on local machines. We do not opt to build *Hadoop*, deploy on a cluster and run *Hadoop* jobs on the cluster to evaluate performance for the following reasons: 1) not all commits can be built into a deployable version of *Hadoop*; 2) running *Hadoop* jobs to evaluate performance may not cover the changed code in each commit; 3) it is challenging to locate the cause of performance regressions from deployed *Hadoop* on clusters. In addition, prior research Singh et al. (2016) has leveraged repeated local tests to evaluate performance.

In order to generate statistically rigorous performance results, we adopt the practice of repetitive measurements T.-H. Chen, Shang, Hassan, Nasser, and Flora (2016) to evaluate performance. In particular, each test or performance micro-benchmark is executed 30 times independently. We collect both domain level and physical level performance metrics during the tests. We measure the response time of each test case as a domain level performance metric. A shorter response time indicates better performance from the users' perspective. Sometimes performance regressions may not cause an impact on response time but rather cause a higher resource utilization. The high resource utilization, may not directly impact user experience, however, it may cause extra cost when deploying, operating and maintaining the system, with lower scalability and reliability. For example, systems that are deployed on cloud providers (like Microsoft Azure) may need to choose virtual machines with higher specification for higher resource utilization. Moreover, a software release with higher memory usage is more prone to crashes from memory leaks. Therefore, physical level performance metrics are also an important measurement for performance regressions. We use

a performance monitoring software named *psutil* (2017) to monitor physical level performance metrics, i.e., the CPU usage, Memory usage, I/O read, and I/O write of the software, during the test.

**Statistical analyses on performance evaluation**

Statistical tests have been used in prior research and in practice to detect whether performance metric values from two tests reveal performance regressions Alghmadi, Syer, Shang, and Hassan (2016). After having the performance evolution results, we perform statistical analyses to determine the existence and the magnitude of performance regression in a statistically rigorous manner. We use Student's t-test to examine if there exists a statistically significant difference (i.e., p-value $<$ 0.05) between the means of the performance metrics. A p-value $<$ 0.05 means that the difference is likely not by chance. A t-test assumes that the population distribution is normally distributed. Our performance measures should be approximately normally distributed given the sample size is large enough according to the central limit theorem T.-H. Chen et al. (2014).

T-test would only tell us if the differences of the mean between the performance metrics from two commits are statistically significant. On the other hand, effect sizes quantify such differences. Researchers have shown that reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, the p-value can be small even if the difference is trivial). We use *Cohen' s d* to quantify the effects Becker (2000). *Cohen' s d* measures the effect size statistically and has been used in prior engineering studies (Kitchenham et al., 2002). *Cohen' s d* is defined as:

$$Cohen's\ d = \frac{mean(x1) - mean(x2)}{s}$$

where *mean(x1)* and *mean(x2)* are the mean of two populations, and s is the pooled standard deviation Hartung, Knapp, and Sinha (2011).

$$effect\ size = \begin{cases} trivial & \text{if } Cohen's\ d \leqslant 0.2 \\ small & \text{if } 0.2 < Cohen's\ d \leqslant 0.5 \\ medium & \text{if } 0.5 < Cohen's\ d \leqslant 0.8 \\ large & \text{if } 0.8 < Cohen's\ d \end{cases}$$

## 3.3 Case Study Result

In this section, we perform an exploratory study on the extracted performance regressions from our subject systems (*Hadoop* and *RxJava*). Our study aims to answer two research questions. For each research question, we present the motivation of the question, the approach that we use to answer the question, the results of the question and we discuss the results.

### RQ1: How prevalent are performance regression introducing changes?

#### Motivation

Prior research has conducted empirical studies on performance bugs (Huang et al., 2014; G. Jin et al., 2012; Zaman et al., 2012, 2011), using the reported performance bugs in issue reports (like JIRA issues). However, there may exist many more performance issues, such as performance regressions, that are not reported as JIRA issues. On the other hand, we evaluate performance on each code commit instead of depending on JIRA issues. Intuitively, we may uncover instances of performance regressions that are not reported, and hence are not be able to be investigated using the approach of prior studies. Therefore, in this research question, we start off by examining how prevalent are detected performance regression introducing changes.

#### Approach

With the approach presented in Section 3.2, we obtain the results of performance evaluation of the impacted tests in every commit of our subject systems. Since one commit may impact multiple tests, where there may exist both performance regression and performance improvement. However, from users' perspective, having worse performance in one feature may result in bad experiences, despite the performance improvement in other features. Therefore, we examine the existence of performance regression by each impacted test separately, instead of considering a commit as a whole.

We use both domain level performance metrics, i.e., response time, and physical level performance metrics, i.e., CPU usage, Memory usage, I/O read and I/O write, as measurements of performance regressions. As explained in Section 3.2, we examine whether a commit would cause any

test case to complete with a statistically significantly longer response time, or utilizing statistically significantly more resources. To ensure the identified performance regressions are not negligible, we only consider a test having performance regression if the effect size is medium or large.

In order to understand whether each performance metric can provide complementary information to others, we also calculate the Pearson correlation between the effect sizes of performance regressions calculated using different metrics. Therefore, we would understand whether we can use a smaller set of metrics to identify performance regressions.

### Results

**Performance regressions are not rare instances and are often with large effects.** We find 243 and 91 commits that contain at least one test with performance regression in at least one performance metric for *Hadoop* and *RxJava*, respectively. In particular, we find 93 commits from *Hadoop* and 91 commits from *RxJava* that contain at least one test case with performance regression in response time. In fact, some commits may have multiple test cases that demonstrate statistically significantly slower response time as performance regression. In a total of 1,270 executed tests from *Hadoop* and 7,600 executed tests from *RxJava*, 129 and 1,410 have statistically significantly slower response time with medium or large effect sizes, respectively. The performance regressions on response time may have a direct impact on users' experiences, making these regressions a higher priority to be examined by developers. When examining the effect sizes of the detected performance regressions, we find that there exist more performance-regression-prone tests with large effect sizes than medium (see Table 3.2). Such results imply that developers may not ignore these performance regressions since they may have a large impact on system performance. In addition, we detect more tests with performance regressions in CPU and Memory usage, than other performance metrics. Since CPU and Memory usage both have a large impact on the capacity of the software systems, these regressions may impact the reliability or financial cost of the software system.

**Physical performance metrics are important complementary indicators of performance regressions.** We use the four physical performance metrics, i.e., CPU usage, Memory usage, I/O read and I/O write to measure performance regression. We find that with the physical performance metrics, we can identify more commits and tests with performance regressions that are not identified

Table 3.2: Results of identifying performance regression introducing changes.

| Number of commits that have at least one test with performance regressions in different metrics. | | | | | | |
|---|---|---|---|---|---|---|
| | Any metric | Response time | CPU | Memory | I/O read | I/O write |
| Hadoop | 243 | 93 | 175 | 138 | 90 | 82 |
| RxJava | 91 | 91 | 91 | 91 | 91 | 90 |

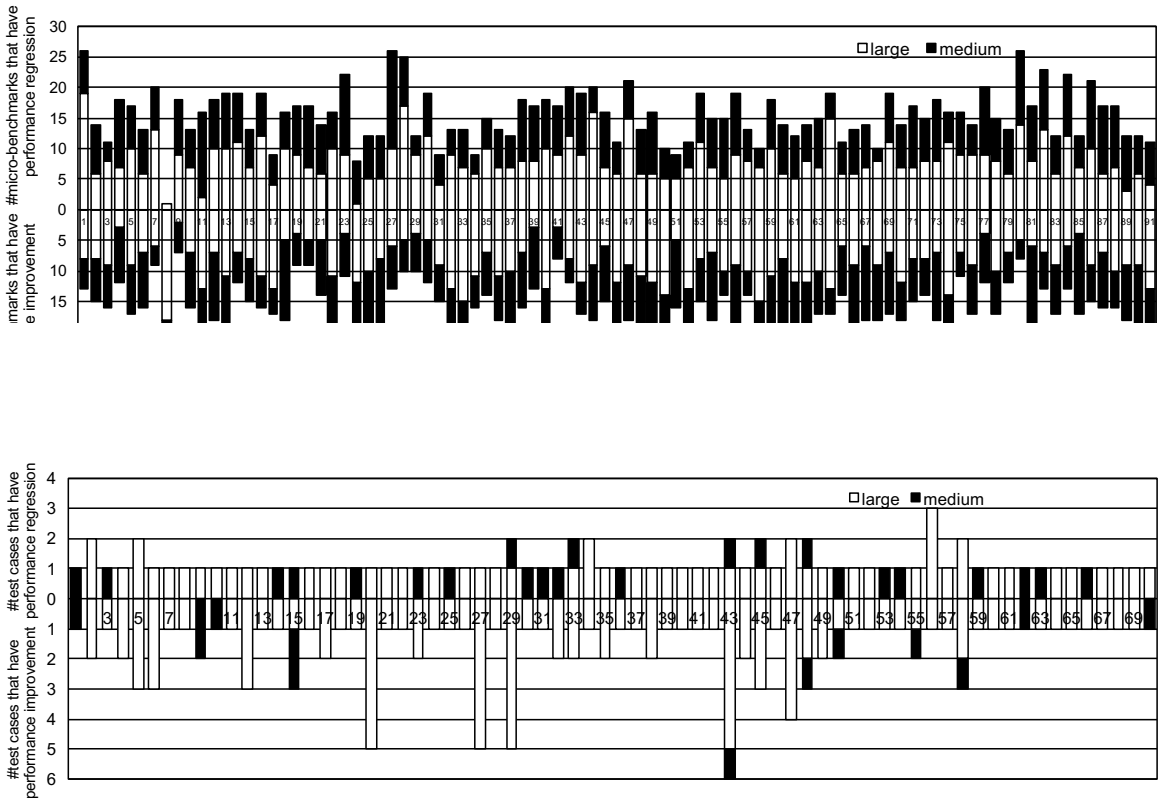
| Total number of tests with performance regressions in different metrics. | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total executed tests | Any metric | Response time | | CPU | | Memory | | I/O read | | I/O write | |
| | | | large effect | medium effect | large effect | medium effect | large effect | medium effect | large effect | medium effect | large effect | medium effect |
| Hadoop | 1,270 | 338 | 87 | 42 | 202 | 97 | 167 | 74 | 75 | 28 | 75 | 17 |
| RxJava | 7,600 | 3,100 | 745 | 665 | 659 | 487 | 919 | 489 | 657 | 449 | 38 | 0 |

with response time. In fact, we find the low correlation between the effect sizes calculated with response time and the physical metrics (see Table 3.3). On the other hand, the effect sizes calculated with the physical metrics may have medium or large correlations with each other. We believe that these correlations are often due to the nature of the software itself. For instance, database accessing software may have its CPU usage highly correlated with I/O. Prior study has also reported that different performance counters often have a high correlation between each other Shang et al. (2015).

### Discussions

**Performance regressions are often with complex syndromes.** One of the approaches in practice of resolving performance regression is to examine syndrome of the regression, i.e., which physical performance metrics contain performance regression, with the consideration of code changes. However, we find that such an approach can be inefficient due to the complexity of performance regressions. In our case study results, we find that 154 commits and 203 tests from *Hadoop*, 91 commits and 777 tests from *RxJava*, have multiple physical performance metrics associated with performance regressions. Moreover, there even exist one test in one commits and one test in three commits of *Hadoop* and *RxJava* having all four physical metrics with performance regressions. Resolving these performance regressions may be challenging and time-consuming.

**Performance regression and performance improvement co-exist.** We find that although many commits contain performance regressions, most of these commits also have performance




(b) Hadoop.

Figure 3.2: Performance regression and improvement measured using response time for each commit from Hadoop and RxJava. The commits are ordered chronologically from left to right.

Table 3.3: Pearson correlation between effect sizes measured using different performance metrics.

| Hadoop | | | | | |
|---|---|---|---|---|---|
| | Response time | CPU | Memory | I/O read | I/O write |
| Response time | 1 | -0.02 | 0.04 | 0.01 | 0.06 |
| CPU | -0.02 | 1 | 0.60 | 0.32 | -0.02 |
| Memory | 0.04 | 0.60 | 1 | 0.27 | 0.04 |
| I/O read | 0.01 | 0.32 | 0.27 | 1 | 0.06 |
| I/O write | 0.06 | -0.02 | 0.04 | 0.06 | 1 |

| RxJava | | | | | |
|---|---|---|---|---|---|
| | Response time | CPU | Memory | I/O read | I/O write |
| Response time | 1 | -0.01 | 0.01 | 0.01 | -0.01 |
| CPU | -0.01 | 1 | 0.39 | 0.46 | 0.06 |
| Memory | 0.01 | 0.39 | 1 | 0.57 | -0.26 |
| I/O read | 0.01 | 0.46 | 0.57 | 1 | 0.06 |
| I/O write | -0.01 | 0.06 | -0.26 | 0.06 | 1 |

improvements at the same time. Figure 3.2 shows the number of tests that have performance regression and improvement for each commit measured using response time[1]. In *Hadoop*, 70 commits contain both performance improvement and regression in different tests. Among these commits, 82 tests executed with these commits have performance regression while 117 tests have performance improvement. In *RxJava*, all 91 performance-regression-prone commits have performance improvement. Among the tests executed for these commits, 1,410 tests have performance regression and 1,545 tests have performance improvement. It may be the case that such performance regressions are side-effect of performance improvement activities. However, our results suggest the possible trade-off between improving performance and introducing performance regression at the same time. Therefore, in-depth analysis on these code changes is needed to further understand the context and reason for introducing performance regressions (see RQ2). In addition, our results illustrate the need for performance regression testing in order to increase the awareness of introducing performance regressions during other development activities.

[1]Due to limited space, we do not show such results for other metrics. Those results will be shared with our data online. https://jinfuchen.github.io/icsmeData

*Finding:* **We find that performance regression introducing changes are prevalent phenomenon with complex syndromes, yet lacking in-depth understanding from the current and prior research. Moreover, these code changes mostly introduce performance regressions while improving performance at the same time.**

*Actionable implication:* **The findings suggest the need for more frequent performance assurance activities (like performance testing) in practice.**

## RQ2: What are the root-causes of performance regressions?

### Motivation

In RQ1, we find that there exist prevalent performance regressions that are introduced by code changes. If we can understand what causes the introduction of these performance regressions, we may provide guidance or automated tooling support for developers to prevent the regressions during code change.

### Approach

We follow two steps in our approach to discover the reasons for introducing performance regressions. First, we investigate the high-level context when these performance regressions are introduced. We use the issue id in a commit message to identify the issue report (JIRA issue report for *Hadoop* and Github issue report for *RxJava*) that is associated with a performance regression introducing code change. We use the type of issues as the context (fixing a bug or developing new features) that are related to the performance regression introducing changes. Sometimes, in *Hadoop*, an issue may be labeled as "subtask". We manually look for the related issue field in the issue report for the issue type. However, there still exist "subtask" issues for which we cannot identify an issue type.

The information in issue reports is about the entire commit rather than the code change that impacts the tests with regression. Therefore, in the second step, we would like to know the code

level root-causes (e.g., which code change) of the performance regressions in each identified test from RQ1. As shown in Table 3.2, there exist 338 and 3,100 tests that have at least one metric with performance regression, in *Hadoop* and *RxJava*, respectively. For *Hadoop*, we manually examine all the code changes that are associated with the corresponding tests where performance regression are identified. For *RxJava*, we take a statistically significant random sample (95% confidence level and 5% confidence interval) from the 3,100 tests from *RxJava*. Our random sample consists of 342 tests in total. We follow an iterative approach to identify the root causes that the code change introduces performance regression, until we could not find any new reasons. Based on our manual study, we also try to examine whether the introduced performance regression can be avoided or whether the performance impact from these regressions may be reduced.

### Results

**A majority of the performance regressions are introduced with bug fixing, rarely with new features.** Figure 3.3 presents the number of performance regression introducing commits that are associated with different issue types. We find that 176 out of 243 (72%) of the commits from *Hadoop* and 48 out of 91 commits (53%) from *RxJava* are associated with issue type *bug*. Such results show that these performance regressions are often introduced during bug fixing tasks. But manually examining all these issues, we find that all such issues are fixing functional bugs. For example, issue HADOOP-11252 [2] is associated with type *bug*. The goal of the issue is to control the timeout of RPC client. While performing functional bug fixing, developers may introduce performance issues at the same time.

We only observe three commits that are associated with having new features. We think the reason is that when having new features, developers typically would create a new test, or modify the existing test to include the new feature. However, in our approach to identifying performance regressions (see Section 3.2), we ensure to use the exact same test cases with prioritizing on using the old test to eliminate the chance of detecting new features as performance regression.

**Performance regressions may be introduced by tasks with typically low impact.** Another interesting finding is that performance regressions can also be introduced during tasks that are not

[2] https://issues.apache.org/jira/browse/HADOOP-11252

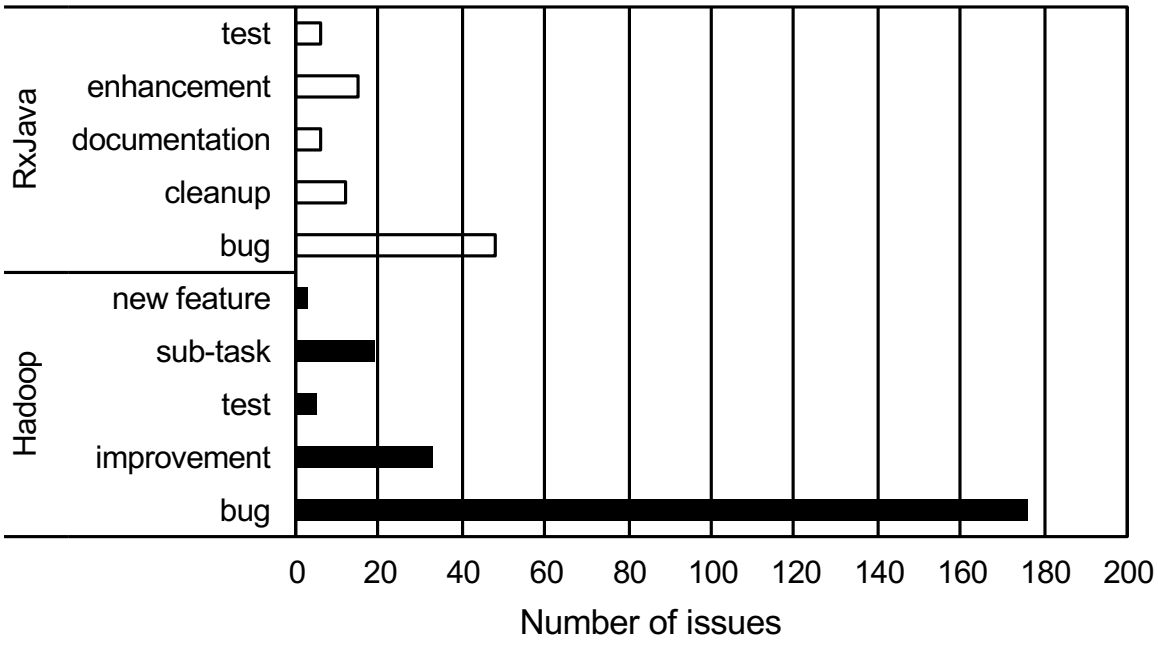



Figure 3.3: Number of performance regression introducing commits associated with different issue types.

generally considered with high impact. For example, 12 commits are associated with issue type *cleanup* and 6 commits are associated with issue type *documentation*. We manually investigate these commits and find that, all too often developers label the issue as *documentation* or *cleanup* while committing code changes for other small tasks (like bug fixes) on the side. In *RxJava*, issue #4987 [3] is labeled as *documentation* but developers fix additional input sources problems on this issue. Issue #4706 [4] is labeled as *cleanup* but fix minor mistakes for operators. Such small tasks may introduce unexpected performance regressions. Therefore, developers should not ignore such commits that are labeled with low-impact task when evaluating performance regressions.

**We identify six code level root-causes of introducing performance regressions.** Based our manual analysis on all commits that are identified with performance regressions, we discover six code level root-causes. The distribution of each root-cause is presented in Table 3.4.

*Changing function calls.* The regression may be introduced by changed function calls in the source code. For example, in class *Mover.java* of commit #c927f938 [5] in *Hadoop*. Developers added

[3] https://github.com/ReactiveX/RxJava/pull/4987
[4] https://github.com/ReactiveX/RxJava/pull/4706
[5] https://github.com/apache/hadoop/commit/c927f938fecf587071d1d07a8077ecf3ab42238a

an API call *shuffle* of *Collections* inside a loop, leading to the performance regression. In particular, we identify four patterns of changing function calls that may introduce performance regression: 1) new functionality, 2) external impact, 3) changing algorithm and 4) skippable function. In particular, Skippable function means that a function call with un-used results. The performance regressions that are introduced by changing algorithm and skippable function should be avoided or resolved by developers. On the other hand, some code changes that introduce new functionality and depend on external resources may not be avoidable. In total, among the 334 identified regressions in this root-cause, we find 36 of them are avoidable. Developers should still be aware of the un-avoidable regression and consider possible alternatives if they have a large impact on users.

*Changing parameters.* The regression may be introduced by changing parameters. For example, in commit #81a445ed[6] in *Hadoop*. The developer added a new parameter *conf* into the function *doPreUpgrade* in file *FSImage.java*. The parameter *conf* contains a large number of variables such that initialization can cause memory overhead when calling function *doPreUpgrade*. Making it worse, the function is called inside a loop, leading to a large performance regression. We identify three patterns of changing parameters that may introduce performance regressions: 1) using more expensive parameter, 2) changing condition parameter and 3) changing configuration parameter. In particular, using more expensive parameter is the case when a parameter is more expensive than before. If all the data in the new parameter is indeed needed, developer may not be able to resolve this regression, while on the other hand, if developers can identify unneeded data in the parameter, the performance impact from the identified regression may be reduced. Similarly, changing condition parameter and changing configuration parameter may both be due to the need of other code change. Among the 100 identified regression in this root-cause, we find 18 of them are avoidable or reducible.

*Changing conditions.* Changing condition can change the code that is executed and may cause more operation eventually executed by the software, leading to performance regressions. For example, in class *AccessControlList.java* of commit #3c1b25b5[7] in *Hadoop*, developers changed the *else* condition to *else if*. So every time the function *isEmpty* inside the condition has to execute. More

[6] https://github.com/apache/hadoop/commit/81a445edf81f42c90a05d764dfebfadfafad622b
[7] https://github.com/apache/hadoop/commit/3c1b25b5fb49b8e9c8c9e1b3367b8bb7e609356d

operations inside the *else if* condition will execute and it will cause more operations. We identified 13 avoidable or reducible regression out of 85 regressions this root-cause.

*Having extra loops.* Changing loops may significantly slow down performance. For example, in commit #94a9a5d0[8] in *Hadoop*, developers added a *for* loop into the file *LeafQueue.java*, which adds an item to a queue repetitively, while such action can be done as a batch process. If the functionality of a loop can be achieved by a batch process, developers may consider implementing batch to minimize the regression. Such solutions can make this regression avoidable or may reduce the performance impact from the loop. All the identified regressions in this root-cause are un-avoidable.

*Using more expensive variables.* Some variables are more expensive to be held in memory and need more resources to visit or operate. For example, it is recommended to use local variables and to avoid *static* variables. If the keyword *static* is used to declare a variable, the lifetime of the variable will be longer, costing more memory. Developers should avoid performance regressions that are introduced by unnecessary expensive variables in the code. We find 9 out of 48 regressions avoidable or reducible in this root-cause.

*Introducing locks and synchronization.* Locks are expensive actions for software performance. Introducing locks and synchronization can suspend threads waiting on a lock until released, causing performance degradation on response time. For example, in commit #2946621a[9] in *Hadoop*, developers added *synchronized* operation to lock the block in class *MetricsSourceAdapter.java*, in order to protest the shared resources used by the two functions inside the block. *synchronized* operation introduces performance regression to the software. Indeed, it is often necessary to have locks in the source code to protect shared resources. On the other hand, developers should always only lock the necessary resource to minimize performance regression. All the identified regressions in this root-cause are un-avoidable.

Table 3.4: Number of tests with different root-causes of performance regressions with the number of avoidable or reducible ones in brackets.

| | Changing function call | | | | Changing parameters | | | Using more expensive variables | Having extra loops | Changing condition | Introducing locks and synchronization | Others |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | New functionality | External impact | Changing algorithm | Skippable function | Expensive parameter | Condition parameter | Configuration parameter | | | | | |
| Hadoop | 125 (11) | 32 (6) | 15 (2) | 6 (6) | 12 (4) | 15 (2) | 22 (5) | 22 (5) | 6 (0) | 37 (8) | 6 (0) | 40 (5) |
| Rxjava | 119 (6) | 18 (3) | 19 (2) | 0 | 22 (3) | 21 (3) | 8 (1) | 26 (4) | 8 (0) | 48 (5) | 0 | 53 (4) |

[8] https://github.com/apache/hadoop/commit/94a9a5d01464e6004f69054bdc308945d40db8e6
[9] https://github.com/apache/hadoop/commit/2946621a531bd71e2408951ab72ecaf5f9fea3f0

**Discussion**

**A considerable amount of performance regressions are avoidable.** Based on our manual study on the root-causes of a performance regression, even though only a few regressions are due to having new features, many regressions cannot be avoided. For example, if new function calls or more data is needed to fix a bug, such performance regressions may not be avoidable. However, there also exist performance regressions that should be entirely avoided by developers, such as having skippable functions. Such performance regressions may eventually become performance bugs. Nevertheless, developers should still be aware of the impact from un-avoidable performance regressions, and minimize the impact on users' experiences by either providing more hardware recourses or considering alternative solutions. Among the total 680 manually examined performance regressions, we find 85 of them are avoidable or their impact may be reduced. With more frequent and thorough performance assurance activity, the impact from these performance regressions can be minimized.

**Functional bugs and performance regressions.** We find that performance regressions are mostly introduced during fixing functional bugs. Such findings may be due to two reasons. First of all, performance regression testing is not enforced during development. After developers fix a functional bug, there is no systematic mechanism to prevent the introduction of performance regression at the same time. Although thorough performance regression testing can help avoid such performance regressions, these tests are often resource-intensive. Designing inexpensive performance testing can help developers better avoid performance regressions in practice. Second, the performance regressions can be ignored due to the pressure or trade-off of fixing the functional bug. Developers may consider the high importance of fixing a functional bug while choose to sacrifice performance for the ease of bug fixes. It is important to make such choices wisely based on the impact of functional bugs and performance regressions. In-depth user studies and field data analysis may help developers minimize the impact from these choices, and avoid performance regression to some extent.

*Finding:* **We find that the majority of performance regressions are introduced with fixing functional bugs, while surprisingly, tasks that are typically considered with low impact also may introduce performance regression. In addition, we identify six root-causes of performance regressions, some of which are avoidable.**

**Actionable implication: Developers should always be aware of the possible performance regression from their code change, in order to address avoidable regressions or minimize the impact from un-avoidable regressions.**

## 3.4 Threats to Validity

### 3.4.1 External Validity

**Generalizing our results.** In our case study, we only focus on fifteen releases from two open-source systems, i.e., *Hadoop* and *RxJava*. Both of the subject systems are mainly written in *Java* languages. Some of the findings might not be generalizable to other systems or other programming languages. Future studies may consider more releases from more systems and even different programming languages (such as C#, C++).

### 3.4.2 Internal Validity

**Subjective bias of manual analysis.** The manual analysis for root-causes of performance regression is subjective by definition, and it is very difficult, if not impossible, to ensure the correctness of all the inferred root-causes. We classified the root-causes into six categories; however, there may be different categorizations. Combining our manual analysis with controlled user studies on these performance regressions can further address this threat.

**Causality between code changes and performance regressions.** By manually examining the code changes in each commit, we identify the root-causes of each performance regression. However, the performance regression may be not caused by the particular code change but due to unknown

factors. Furthermore, the performance regression may not be introduced by one change to the source code but a combination of confounding factors. In order to address this threat, future work can leverage more sophisticated causality analysis based on code mutation that can be leveraged to confirm the root cause of the performance regression.

**Selection of performance metrics.** Our approach requires performance metrics to measuring performance. In particular, we pick one commonly used domain level and four commonly used physical level performance metrics based on the nature of the subject systems. There exist a large number of other performance metrics. However, practitioners may require system-specific expertise to select an appropriate set of performance metrics that are important to their specific software. Future work can include more performance metrics based on the characteristic of the subject systems.

### 3.4.3 Construct Validity

**Monitoring performance of subject systems.** Our study is based on the ability to accurately monitor the performance of our subject systems. This is based on the assumption that the performance monitoring library, i.e. *psutil* can successfully and accurately providing performance metrics. This tool monitoring library is widely used in performance engineering research (Ahmed, Bezemer, Chen, Hassan, & Shang, 2016; T.-H. Chen, Shang, Hassan, et al., 2016). To further validate our findings, other performance monitoring platforms (such as PerfMon Enbody (1999)) can be used.

**Noise in performance monitoring results.** There always exists noise when monitoring performanceMytkowicz, Diwan, Hauswirth, and Sweeney (2009). For example, the CPU usage of the same software under the same load can be different in two executions. In order to minimize such noise, for each test or performance micro-benchmark, we repeat the execution 30 times independently. Then we use a statistically rigorous approach to measuring performance regressions. Further studies may opt to increase the number of repeated executions to further minimize the threat based on their time and resource budget.

**Issue report types.** We depend on the types of issues that are associated with each performance regression introducing commit. The issue report type may not be entirely accurate. For example, developers include extra code changes in issue reports with type *documentation*. Firehouse-style

user studies Murphy-Hill, Zimmermann, Bird, and Nagappan (2013) can be adopted to better understand the context of performance regression introducing changes.

**The effectiveness of the tests.** In our case study, we leverage test cases and performance micro-benchmarks to evaluate the performance of each commit. In particular, for *Hadoop* our heuristic of identifying impacted tests are based on naming conventions between source code files and test files. In addition, we also rely on the readily available performance micro-benchmarks in *RxJava*. Our heuristic and the performance micro-benchmarks both may not cover all the performance impacts from code changes. However, the goal of our study is not to detect all performance regression in the history of our subject systems, but rather collect a sample of performance regression introducing commits for our further investigation. Future work may consider using more sophisticated analysis to identify the impacted tests Qusef, Bavota, Oliveto, De Lucia, and Binkley (2014) or manually adapting the tests to address this threat. Moreover, conducting systematic long-lasting performance tests may minimize this threat, the long-lasting time of these test (often more than eight hours) make it almost impossible for every commit. It is still an open research challenge of how to design in-expensive yet representative performance tests, which our case study signifies the importance of breakthrough in such a research area.

**In-house performance evaluation.** We evaluate the performance of our subject systems with our in-house performance evaluation environment. Although we minimize the noise in the environment to avoid bias, such an environment is not exactly the same as in-field environment of the users. There is a threat that the performance regressions identified in our case study may not be noticeable in the field. To minimize the threat, we only consider the performance regressions that have non-trivial (turn out to be mostly large in our experiment) effect sizes. In addition, with the advancing of DevOps, more operational data will become available for future mining software repository research. Research based on field data from the real users can address this threat.

## 3.5 Related Work

In this section, we present the related work to this work.

### 3.5.1 Performance regression detection

A great amount of research has been proposed to detect performance regression. Ad hoc analysis selects a limited number of target performance counters (e.g., CPU and memory) and performs simple analysis to compare the target counters. Heger et al. (2013) present an approach to support software engineers with root cause analysis of the problems. Their approach combines the concepts of regression testing, bisection and calls tree analysis to detect performance regression root cause analysis as early as possible.

Pair-wise analysis compares and analyzes the performance metrics between two consecutive versions of a system to detect the problem. Nguyen *et al*. (T. H. Nguyen et al., 2012; T. H. D. Nguyen et al., 2011, 2014) conduct a series of studies on performance regressions. Nguyen *et al*. propose an approach to detect performance regression by using a statistical process control technique called control charts. They construct the control chart and apply it to detect performance regressions and examine the violation ratio of the same performance counter. Malik et al. (2013) propose approaches that combine one supervised and three unsupervised algorithms to help performance regression detection. They employ feature selection methods named Principal Component Analysis (PCA) to reduce the dimensionality of the observed performance counter set and validate their approach through a large case study on a real-world industrial software system Malik et al. (2010).

Model-based analysis builds a limited number of detected models for a set of target performance counters (e.g., CPU and memory) and leverages the models to detect performance regressions. Xiong et al. (2013) propose a model-driven framework to diagnose the application performance in cloud condition without manual operation. In the framework, it contains three modules consisting of sensor module, model building module, and model updating module. It can automatically detect the workload changes in cloud environment and lead to the root cause of performance problem. Cohen, Chase, Goldszmidt, Kelly, and Symons (2004) propose an approach that builds a promising class of probabilistic models (Tree-Augmented Bayesian Networks or TANs) to correlate system level counters and systems average-case response time. Cohen et al. (2005) present that performance counters can successfully be used to construct statistical models for system faults and compact signatures of distinct operational problems. Bodík et al. (2008) employ logistic regression with L1

regularization models to construct signatures to improve Cohen *et al.*' s work.

Multi-models based analysis builds multiple models from performance counters and uses the models to detect performance regressions. Foo et al. (2010) propose an approach to detect potential performance regression using association rules. They utilize data mining to extract performance signatures by capturing metrics and employ association rules techniques to collect correlations that are frequently observed in the historical data. Then use the change to the association rules to detect performance anomalies. M. Jiang, Munawar, Reidemeister, and Ward (2009) present two diagnosis algorithms to locate faulty components: RatioScore and SigScore based on component dependencies. They identify the strength of relationships between metric pairs by utilizing an information-theoretic measures and track system state based on in-cluster entropy. A significant change in the in-cluster entropy is considered as a sign of a performance fault. Shang et al. (2015) propose an approach that first clusters performance metrics based on correlation. Each cluster of metrics is used to build a statistical model to detect performance regressions.

The vast amounts of research on performance regression detection signify its importance and motivate our work. Prior research on performance regressions are all designed to be conducted after the system is built and deployed. In this work, we explore performance regressions at the commit level, i.e., when they are introduced.

### 3.5.2 Empirical studies on performance

Empirical studies are conducted in order to study performance issues (Huang et al., 2014; G. Jin et al., 2012; Zaman et al., 2012, 2011). G. Jin et al. (2012) study 109 real-world performance issues that are reported from five open-source software. Based on the studied 109 performance bugs, G. Jin et al. (2012) develop an automated tool to detect performance issues. Zaman et al. (2012, 2011) conducted both qualitative and quantitative studies on performance issues. They find that developers and users face problems in reproducing performance bugs. More time is spent on discussing performance bugs than other kinds of bugs. Huang et al. (2014) studied real-world performance issues and based on the findings. They propose an approach called performance risk analysis (PRA), to improve the efficiency of performance regression testing.

Table 3.5 summarizes the comparison between this work and the four prior studies (Huang et

Table 3.5: Comparing this work with the four prior studies (Huang et al., 2014; G. Jin et al., 2012; Zaman et al., 2012, 2011)

| | G. Jin et al. (2012) | Huang et al. (2014) | Zaman et al. (2012) | This work |
|---|---|---|---|---|
| Data source | Issue reports | Issue reports | Issue reports | Code commits |
| Granularity | Patch | performance issue | performance issue | Performance micro-benchmarks or impacted tests |
| #instances | 109 | 100 | 100 | 3,438<br>680 for manual study |
| Main approach | Dynamic rule-based checker | Static analysis & risk modeling | N/A | Repeated measurement |

al., 2014; G. Jin et al., 2012; Zaman et al., 2012, 2011). In particular, prior studies only study a small amount (around 100) of performance issues often due to the lower number of such issues reported. On the contrary, since our study does not depend on the existence of performance issue reports, we observe a much higher prevalence of performance regressions that are introduced during development. Therefore, our results suggest that maintaining the performance of software may be more challenging. Without an efficient feedback channel from the end users, the developers may overlook possible performance regressions. Our findings suggest the importance of the awareness of possible performance regressions introduced during development.

Luo et al. (2016) propose a recommendation system, called PerfImpact, to automatically identify code changes that may potentially be responsible for performance regression between two releases. Their approach searches for input values that expose performance regressions and compare execution traces between two releases of a software to identify problematic code changes. Hindle (2015) present a general methodology to measure the impact of different software metrics on power consumption. They find the effect of software change on power consumption regressions. Hasan et al. (2016) create energy profiles as a performance measurement for different Java collection classes. They find that energy consumption can have a large difference depending on the operation. Lim et al. (2014) use performance metrics as a symptom of performance issues and leverage historical data to build the Hidden Markov Random Field clustering model. Such a model has been used to detect both reoccurring and unknown performance issues.

Prior studies on performance typically are based on either limited performance issue reports or

release of the software. However, the limited amount of issue reports and releases of the software hides the prevalence of performance regressions. In our work, we evaluate performance at the commit level. Therefore, we are able to identify more performance regressions and are able to observe the prevalence of performance regression introducing changes in development.

## 3.6 Conclusion

In this work, we conduct an empirical study on performance regression introducing changes in two open-source software *Hadoop* and *RxJava*. We evaluate the performance of every commit by executing impacted tests or performance micro-benchmarks. By comparing performance metrics that are measured during the tests or performance micro-benchmarks, we identify and study performance regressions introduced by each commit. In particular, this work makes the following contributions:

- To the best of our knowledge, our work is one of the first to evaluate and to study performance regressions at the commit level.
- We propose a statistically rigorous approach to identifying performance regression introducing code changes. Further research can adopt our methodology in studying performance regressions.
- We find that performance regressions widely exist, and often are introduced after bug fixing.
- We find six root-causes of performance regressions that are introduced by code changes. 12.5% of the manually examined regressions can be avoided or their performance impact may be reduced.

Our findings call for the need for frequent performance assurance activities (like performance testing) during software development, especially after fixing bugs. Although such activities are often conducted before release Z. M. Jiang and Hassan (2015), while developers may find it challenging since many performance issues may be introduced during the release cycle. In addition, developers should resolve performance regressions that are avoidable. For the performance regressions that cannot be avoided, developers should evaluate and be aware of their impact on users. If there exists

a large impact on users, strategies, such as allocating more computing resources, may be considered. Finally, in-depth user studies and automated change impact on performance are future directions of this work.

# Chapter 4

# Can we predict tests that manifest performance regressions at the commit level?

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 this work, 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.

## 4.1 Introduction

Performance assurance activities are an essential step in the development cycle of large software systems Weyuker and Vokolos (2000). 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 Joel (2017).

Although automated techniques are proposed to detect performance regressions (Heger et al., 2013; Luo et al., 2016; T. H. Nguyen et al., 2012; T. H. D. Nguyen et al., 2014; Shang et al., 2015), 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, we propose an approach to identifying performance regressions at the code commit level J. Chen and Shang (2017). 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 on 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 4.7). Recent work by de Oliveira, Fischmeister, Diwan, Hauswirth, and Sweeney (2017) proposed an approach named *Perphecy*, which aims to select 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 de Oliveira et al. (2017). 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 a 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 work, 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

[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 4.5-RQ4) to manifest performance regression with slower response time.

call such tests performance-regression-prone tests. In other words, our approach predicts whether there is a 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 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 of 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?*

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

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 an accurate prediction on evaluating the performance impact of their code changes in a timely and proactive manner.

The rest of this work is organized as follows: Section 4.2 presents prior research related to this work. Section 4.3 presents our approach to predicting performance-regression-prone tests. Section 4.4 presents our subject systems and how to extract performance-regression-prone tests as our ground truth. Section 4.5 presents the results of our case studies. Section 4.6 discusses our learned lessons. Section 4.7 presents threats to the validity of our study. Finally, Section 4.8 concludes this work.

## 4.2 Related Work

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

### 4.2.1 Software defect prediction

Mockus and Weiss (Mockus & Weiss, 2000) are the first ones to use metrics that describes the characteristics of software changes to build a linear regression model to predict the probability of

failure after code changes. Kamei et al. (Kamei et al., 2016; Kamei & Shihab, 2016; Kamei et al., 2013) conduct a series of studies on commit-level defect prediction. Kamei et al. (2013) 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 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 and Shihab (2016) present an overview of the research field of defect prediction. To build the prediction model, researchers need to extract and select a list of metrics. F. Zhang et al. (2014) 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. (2013) 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. P. He et al. (2015) 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 (2016) 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. (2016) 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. (2016) use 14 code based metrics to build simple unsupervised and supervised models to predict software defect. The results show that many simple unsupervised models perform better than 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 (Kamei et al., 2013; Mockus & Weiss, 2000) and also include performance-related metrics in our classifiers.

### 4.2.2 Empirical studies on software performance

Various empirical studies are conducted to study performance issues. G. Jin et al. (2012) 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. (2012, 2011) 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. (2018) 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. (2014) 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, Huggler, and Gross (2014) 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 (2012) 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. (2016) 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 for performance regressions. de Oliveira et al. (2017) present a lightweight tool named *Perphecy* to detect which commit will cause performance regression on which benchmark. Leitner and Bezemer (2017) aim to understand the current state of the 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 guidelines to conduct performance tests in an easy and powerful way.

This paper argues that future performance testing should implement a flexible testing framework to support low-friction testing.

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

### 4.2.3 Test case prioritization

The goal of Test Case Prioritization (TCP) is to select the most appropriate tests to execute and enable faster feedback Laali, Liu, Hamilton, Spichkova, and Schmidt (2016). Various prior research has been proposed to improve TCP (Di Nardo, Alshahwan, Briand, & Labiche, 2015; Kazmi, Jawawi, Mohamad, & Ghani, 2017; Mostafa, Wang, & Xie, 2017; Saha, Zhang, Khurshid, & Perry, 2015; Wang, Nam, & Tan, 2017). Wang et al. (2017) 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. Mostafa et al. (2017) was proposed to prioritize performance test cases for collection-intensive software. The approach profiles software using a dynamic call graph to build a performance model and analyzes performance impact introduced by code changes. To address the coverage profiling overhead, Saha et al. (2015) 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 (Anderson, Salem, & Do, 2015; Kim & Porter, 2002; Najafi, Shang, & Rigby, 2019; Noor & Hemmati, 2017; Zhu, Shihab, & Rigby, 2018). Kim and Porter (2002) 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. (2015) investigate the test historical information including test case pass, fail information, code change information. The authors use such historical metrics to build a model to predict future test case failures. Noor and Hemmati (2017) 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. (2018) propose an approach *CODYNAQ* to use the historical co-failure distributions among tests to re-prioritize tests after each test run. Najafi et al. (2019) 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 focuses on validating functional bugs rather than performance issues. Our work aims to prioritize tests for a cost-effective detection of performance regressions.

## 4.3 Approach

In this section, we present our approach to 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 4.1.

### 4.3.1 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 4.4.2 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 4.1.

**Traditional metrics.** Prior studies on commit-level defect prediction leverage various metrics to predict the risk of a code commit (Kamei et al., 2013; Mockus & Weiss, 2000). 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 4.1.

**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 (Alam et al., 2017; J. Chen & Shang, 2017; Costa et al., 2017; Huang et al., 2014; G. Jin et al., 2012; L. Song & Lu, 2017), we extract metrics that describe the code changes in a commit that may relate to performance. For example, adding *synchronization* into the source code is one of the root-causes of performance regressions (J. Chen & Shang, 2017; G. Jin et al., 2012). 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 4.1.

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* (2017) to convert the source code to XML files representing their abstract syntax trees. We use lxml (2005) 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.

Table 4.1: 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. |
| | ND | Number of modified directories | Numerical | Changing more directories may more likely to introduce performance regressions Mockus and Weiss (2000). |
| | NF | Number of modified files | Numerical | Changing many source files are more likely to cause performance regression N. Nagappan et al. (2006). |
| | NM | Number of modified methods | Numerical | Changes altering many methods are more likely introduce performance regression Zimmermann et al. (2007). |
| | Entropy | Distribution of modified code across files | Numerical | Scattered changes are more possible to be performance-regression-prone Hassan (2009). |
| 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 N. Nagappan and Ball (2005). |
| | 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 N. Nagappan and Ball (2005). |
| | 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 Koru, Zhang, Emam, and Liu (2009); Zimmermann et al. (2007). |
| | 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 Koru et al. (2009). |
| Purpose | FN | Whether the code commit fixes a bug | Boolean | Bug fixing changes are more likely to introduce performance regressions J. Chen and Shang (2017); P. J. Guo, Zimmermann, Nagappan, and Murphy (2010). |
| 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 Matsumoto, Kamei, Monden, Matsumoto, and Nakamura (2010). |
| | 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 Graves, Karr, Marron, and Siy (2000). |
| Experience | EXP | Developer experience | Numerical | Senior programmers may introduce more stable code change than a less experienced developer Mockus and Weiss (2000). |
| | REXP | Recent developer experience | Numerical | Recent developers are more familiar with the program so that it is less likely to introduce performance regression Mockus and Weiss (2000). |
| Complexity | MCC | McCabe Cyclomatic complexity | Numerical | Program with higher complexity is more likely to suffer from performance regression Hassan (2009). |
| | FanIn | Number of calling subprograms | Numerical | Large calling subprograms will amplify the regression if there exists performance regression in the called program N. Nagappan et al. (2006). |
| 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. |
| | †Syn | Number of added or deleted synchronization expressions in method level | Numerical | Synchronizations are expensive actions for software performance Alam, Liu, Zeng, and Muzahid (2017). |
| | †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 Huang et al. (2014). |
| | †Loop | Number of added, deleted or changed loop statement in method level | Numerical | Changes involving the loop may significantly slow down performance L. Song and Lu (2017). |
| | †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 Costa, Andrzejak, Seboek, and Lo (2017); Huang et al. (2014). |
| | †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 J. Chen and Shang (2017). |
| | †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 J. Chen and Shang (2017); G. Jin et al. (2012). |

*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 Costa et al. (2017).

*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 work, particularly, we only consider the function calls to the logging library. We generate three metrics for added, deleted and changed external function calls.

### 4.3.2 Data preprocessing

Before leveraging the extracted data to build classifiers, we preprocess the data. Prior research shows that multicollinearity 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 (Jiarpakdee, Tantithamthavorn, & Hassan, 2019; Tantithamthavorn & Hassan, 2018a). To deal with multicollinearity, we first perform a correlation analysis on the metrics to remove the most highly correlated metrics. We used Pearson's correlation Benesty, Chen, Huang, and Cohen (2009) 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 McIntosh, Kamei, Adams, and Hassan (2016). We then perform a 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 Harrell (2001). 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) Syer et al. (2017), 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.

### 4.3.3 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 4.4). The independent variables are based on the metrics that are presented in Section 4.3.1. All the metrics are pre-processed as described in Section 4.3.2.

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 Schein, Popescul, Ungar, and Pennock (2002). The details of the performance evaluation are presented in Section 4.4.2. We choose 50 commits because of the amount of metrics that we have. Fewer commits may result in 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.

### 4.3.4 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 4.4.2). 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 to 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.

## 4.4 Evaluation Setup

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

### 4.4.1 Subject systems

We choose three open-source systems, *Hadoop*, *Cassandra* and *OpenJPA* as the subject systems of our case study. *Hadoop* White (2009) 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 (T.-H. Chen et al., 2014; Syer et al., 2017). The overview of the three subject systems is shown in Table 4.2.

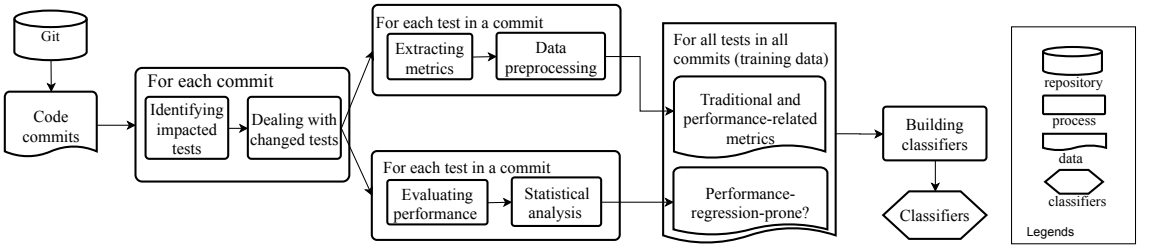
### 4.4.2 Extracting performance-regressions-tests in each commit

In this subsection, we present how we perform performance evaluation to extract performance regression tests in each commit. Such extracted data is used as our ground truth in the case

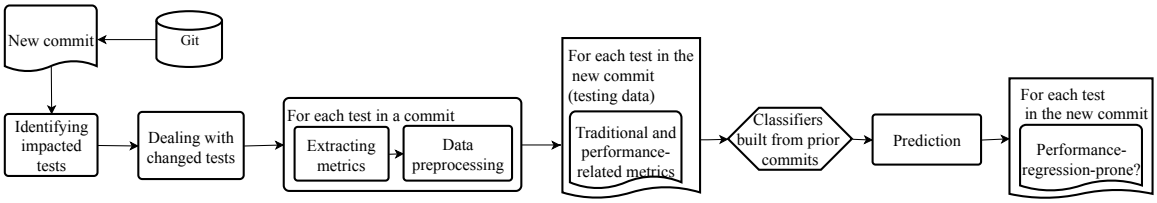
Table 4.2: Overview of our subject systems.

| 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 |

study. The role of this data is similar to the extracted "bug-fixing commits" in a commit-level bug prediction approach Kamei et al. (2013). 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.




(a) building classifiers




(b) Predicting performance-regression-prone tests in a new commit

Figure 4.1: An overview of 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 tests 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 Tillmann and Schulte (2006).

Not all the tests are impacted by the code changes in a commit and running those un-impacted 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 T.-H. Chen, Shang, Hassan, et al. (2016) to evaluate performance. Conservatively, we executed each test 30 times independently, since prior research often only repeat the tests 5 to 20 times (Laaber & Leitner, 2018; Laaber, Scheuner, & Leitner, 2019; Leitner & Cito, 2016). We use a performance monitoring software named *psutil* (2017) 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 evaluation 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 Nachar et al. (2008) 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, the p-value can be small even if the difference is trivial). We use Cliff's delta to quantify the effect sizes Becker (2000). Cliff's delta measures the effect size statistically and has been used in prior engineering studies Kitchenham et al. (2002). We only consider a test manifesting a performance regression if the effect size is medium ($0.33 <$ *Cliff's delta* $\leqslant 0.474$) or large ($0.474 <$*Cliff's delta*).

Based on the statistical analysis results, each commit is labeled 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 demonstrates 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 4.3. We can see that the ones without regressions have a much smaller increase of values in each performance metric. Such a 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 being impacted and we remove such outliers from our data.

In total, we spend 133 machine days running all the tests. Table 4.4 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

Table 4.3: 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 |

for 9% of all tests. Therefore, running all the tests to detect these performance regressions is not cost-effective.

Table 4.4: 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%) |

### 4.4.3 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 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 the 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

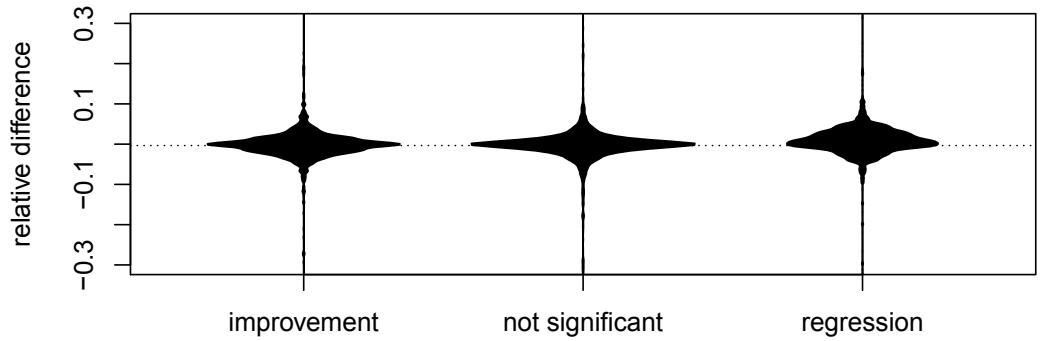



Figure 4.2: Distribution of the relative difference between performance metrics by running tests only once in each commit.

was measured in the last commit. We calculate the relative difference between performance 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 4.4.2). Finally, we use bean plots to visualize the distribution of the relative differences of performance metrics in each group.

Figure 4.2 presents the distribution of the relative difference between performance metrics by running the test only once. We find that, in all three groups, most of the relative differences 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 4.2, 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 Nachar et al. (2008) and Cliff's delta Becker (2000) 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 for our approach that aims to predict tests that are executed rigorously to detect performance regressions.

## 4.5 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) Harrell (2001), support vector machine (SVM) Hearst, Dumais, Osuna, Platt, and Scholkopf (1998), XGBoost (XG) T. Chen and Guestrin (2016) with 50 (XG-50), 100 (XG-100) and 500 (XG-500) iterations, random forest classifiers Breiman (2001), 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, we use our approach to predict performance-regression-prone tests and update the classifiers for each commit as presented in Section 4.3.

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 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 4.4, 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 Lobo, Jiménez-Valverde, and Real (2008). The value of *AUC* ranges from 0 to 1, and a 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 are shown in Table 4.5. 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 4.5). 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 4.4 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 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.

Table 4.5: 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 |

***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 T.-H. Chen, Shang, Hassan, et al. (2016). 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 results of our classifiers from RQ1. We call this model *Non-Cost-AWare random forest model (NCAW)*. We sort all the tests ordered by decreasing the 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 and Koschke (2010) and Kamei et al. (2013), we also calculate $R_d$ as

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

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 Ostrand, Weyuker, and Bell (2005). Since the percentage of tests with performance regression range from 2% to 26% (see Table 4.4), if we simply aim to examine 20% of the files,

Table 4.6: 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 |

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 are 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 Mende and Koschke (2009) to evaluate the prediction performance of the model. An example of the cumulative lift chat is shown in Figure 4.3. We generate a cumulative lift chart from the optimal model, the non-cost-aware model and the cost-aware model (see Figure 4.3). 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 4.6 presents the cost-effectiveness of our prediction models. In particular, by spending only 5% of cost executing all tests, our approach can help detect 12% to 65%

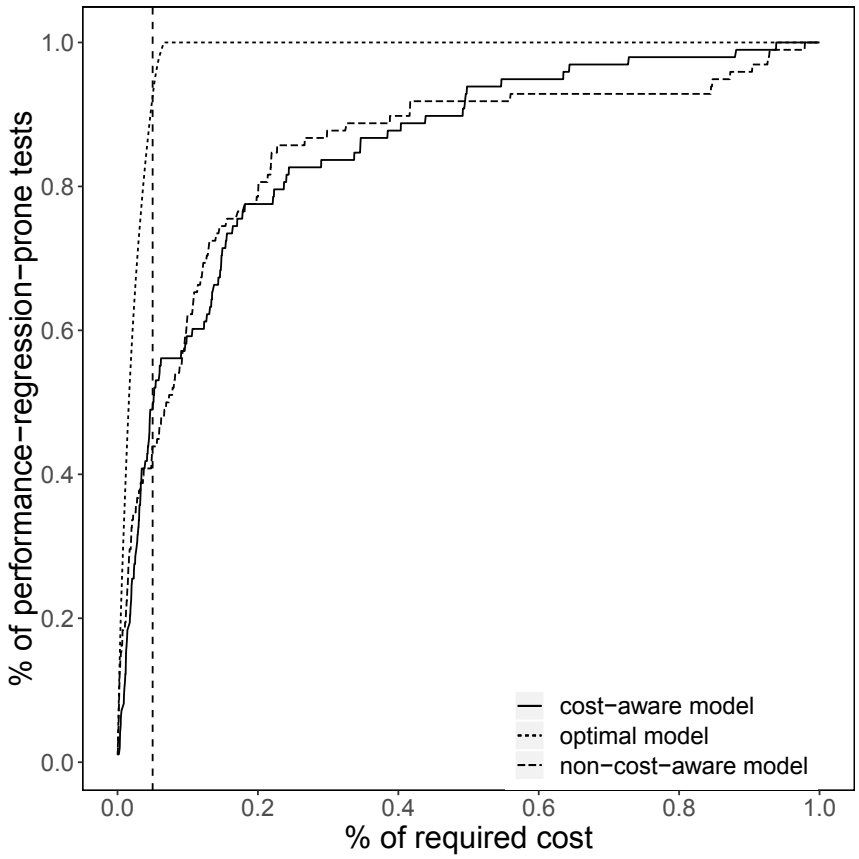



Figure 4.3: Cumulative distribution function of the optimal model, CAW and NCAW models in the performance counter of *Response time* in *Hadoop*.

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, 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 question 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 4.4.2). 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 4.7 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 makes 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 4.7: 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 write | |
| 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 the performance issues are introduced during our studied period. Therefore, we leverage an SZZ algorithm Śliwerski, Zimmermann, and Zeller (2005) 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 Williams and Spacco (2008). 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 4.8 presents the fixing commit each performance issue, the inducing commits for each issue and their corresponding performance metrics. Table 4.8 shows that six out of nine issues have at least one

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

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 4.8, 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.

> ***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

Table 4.8: 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 |

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 an important influence on 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 Li, Shang, Zou, and

E. Hassan (2017). 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 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 detail, 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 Tantithamthavorn, McIntosh, Hassan, and Matsumoto (2017) 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 a 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 von Storch (1999) is smaller than $0.05$.

**Results.** The result of the average rank of important metrics is shown in Table 4.9. 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

Table 4.9: Average rank of the top important metrics in our classifiers. The up/down arrows indicate whether the relationship is positive/negative.

| Metric | Response 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 |

metrics are derived based on prior research (Alam et al., 2017; J. Chen & Shang, 2017; Costa et al., 2017; Huang et al., 2014; L. Song & Lu, 2017) 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 decrease 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 Kamei et al. (2013).

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[5] 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.

[5] https://github.com/apache/hadoop/commit/94a9a5d01464e6004f69054bdc308945d40db8e6

***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.***

## 4.6 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.

### 4.6.1 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 (Alam et al., 2017; J. Chen & Shang, 2017; G. Jin et al., 2012). In order to understand why such metrics do not have 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.

### 4.6.2 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.

### 4.6.3 Limitation of our approach and future work

The main limitation of our approach is 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 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.

### 4.6.4 Generalizability of our study

Although our approach is not designed 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 contributions 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 require extra running time to achieve the needed repetition. Recent research Ding, Chen, and Shang (2020) discusses the reasons for tests not suitable for performance evaluation, which can be leveraged to know how well another project can adopt our approach.

## 4.7 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 on their wide usage in prior software engineering research (Kabinna, Bezemer, Shang, Syer, & Hassan, 2018; Tantithamthavorn, McIntosh, Hassan, Ihara, & Matsumoto, 2015), 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 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. To know the impact of data re-balancing, we leveraged two re-balancing techniques, i.e., up-sampling and SMOTE Chawla, Bowyer, Hall, and Kegelmeyer (2002). However, neither of the two techniques can improve our prediction results. Similar findings have been reported in software defect prediction (Tantithamthavorn, Hassan, & Matsumoto, 2018). In addition, we did not fine-tune the parameters of classifiers, while only running XGBoost with a different number of iterations. Although practitioners find mainly the importance of iterations in XGBoost JAIN (2016), 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.

## 4.8 Conclusion

In this work, 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 study 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.

# Chapter 5

# Can we predict whether a configuration option manifests performance variation?

Maintaining a good performance of a software system is a primordial task when evolving a software system. The performance regression issues are among the dominant problems that large software systems face. In addition, these large systems tend to be highly configurable, which allows users to change the behaviour of these systems by simply altering the values of certain configuration options. However, such flexibility comes with a cost. Such software systems suffer throughout their evolution from what we refer to as "Inconsistent Option Performance Variation" (***IoPV***). An *IoPV* indicates, for a given commit, that the performance regression or improvement of different values of the same configuration option is inconsistent compared to the prior commit. For instance, a new change might not suffer from any performance regression under the default configuration (i.e., when all the options are set to their default values), while altering one option's value manifests a regression, which we refer to as a hidden regression as it is not manifested under the default configuration. Similarly, when developers improve the performance of their systems, performance regression might be manifested under a subset of the existing configurations. Unfortunately, such hidden regressions are harmful as they can go unseen to the production environment. In this study, we first quantify how prevalent (in)consistent performance regression or improvement is among the values of an option. In particular, we study over 803 *Hadoop* and 502 *Cassandra* commits,

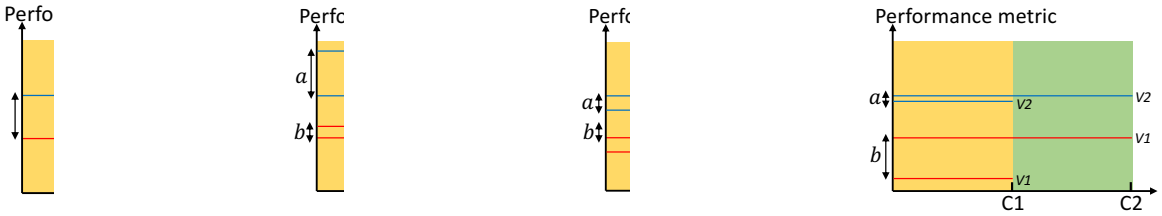
for which we execute a total of 4,902 and 4,197 tests, respectively, amounting to 12,536 machine hours of testing. We observe that *IoPV* is a common problem that is difficult to manually predict. 69% and 93% of the *Hadoop* and *Cassandra* commits have at least one configuration that hides a performance regression. Worse, most of the commits have different options or tests leading to *IoPV* and hiding performance regressions. Therefore, we propose a prediction model that identifies whether a given combination of commit, test, and option (***CTO***) manifests an *IoPV*. Our evaluation for different models show that random forest is the best performing classifier, with a median AUC of 0.91 and 0.82 for *Hadoop* and *Cassandra*, respectively. Our work defines and provides scientific evidence about the *IoPV* problem and its prevalence, which can be explored by future work. In addition, we provide an initial machine learning model for predicting *IoPV*.

## 5.1 Introduction

Software systems are increasingly evolving in their size and complexity. Large scale software systems, such as Amazon, Google, and Facebook tend to deal with a high operation workload (e.g., millions of active users), which is complicated the complexity and the continuous evolution of such systems. Such systems usually report more performance bugs than feature-related ones Weyuker and Vokolos (2000). Performance assurance activities are an essential part of the release cycle of such large-scale software systems.

On the other hand, modern large-scale software systems tend to have a large number of configuration options, which can hide performance issues. These options are used to customize the behaviour of a software system without changing its source code. Although these options add flexibility to a software system, they make testing a software performance a challenging task. For example, in theory, one has to run $2^{10}$ tests for a software system with just 10 boolean configuration options, while a highly configurable software system such as *Hadoop* can have as many as 365 available options Sayagh et al. (2018). While there are constraints between configuration options, bringing down the total number of configurations in practice, this still amounts to a too large set of configurations to test exhaustively, especially for (long-running) performance tests.

A large body of research proposed and evaluated approaches that detect performance issues (Foo




(d)

Figure 5.1: The definition of *IoPV* and how different is it from the traditional way of comparing the performance of two values for the same configuration option: (a) Approaches that do not consider the historical evaluation, (b) An option with an inconsistent performance variation (a-b), (c) An option with a consistent performance variation (a-b), and (d) An option with an inconsistent performance variation (a-b). V1 and V2 are two different values of the same configuration option. C1 and C2 are two revisions. A smaller performance metric value (e.g., CPU usage) indicates a better performance.

et al., 2010, 2015; T. H. Nguyen et al., 2012; T. H. D. Nguyen et al., 2011, 2014). Prior studies also proposed approaches to test the performance of highly configurable software systems (Halin et al., 2019; Saxena et al., 2000; Wu & Wang, 2010). However, existing approaches aimed at a late stage of the software release cycle, i.e., after the build and deployment of new releases. Nevertheless, identifying performance issues at earlier stages, especially at the development stage, can minimize the amount of resources required for identifying and fixing a performance regression. In fact, going through the whole build, testing, and packaging process to find the root cause of a performance regression is time-consuming.

Traditionally, prior work studied the difference in system performance caused by different values of the same option, without considering how the performance impact of an option evolve due to code changes Sayagh et al. (2018). For instance, traditional approaches compare different values of a configuration option based on their raw performances values (Diao et al., 2003; Raghavachari et al., 2003; Siegmund et al., 2015), as illustrated in Figure 5.1a. However, such a comparison is subjective as the option's value *V2* with a worse performance might not necessarily be problematic; it might, as an example, just enables the execution of some extra features. Instead, even if an option's value has a good performance compared to other values, it might be significantly different when compared to the performance of the same option value in the prior commit, as shown in the example in Figure 5.1b. Although the *V2* value has a worse performance, it is much better than the prior commit. Figure 5.1c shows that both values do not have a large inconsistent performance

variation. On the opposite side, an option's value that has a better performance compared to another value of the same option might be facing a large performance regression compared to the prior commit, as shown in the example of Figure 5.1d. *V1* has a better performance compared to *V2*, but *V1* faces a severe performance regression.

Therefore, different values of an option can have an inconsistent variation in terms of performance compared to the prior commit. This happens when one or more values of an option exhibit a significant difference compared to the prior release. We refer to such an issue as **Inconsistent Options Performance Variation (a.k.a, *IoPV*)**. The *IoPV* might be problematic as it can hide a performance regression that is manifested under one configuration option. Similarly, when a default configuration shows an improvement, altering one option's value may indicate no performance improvement or even a performance regression. Such regressions can unfortunately go as unseen to the production environment.

In this work, we perform a case study on two large-scale open-source software systems: *Hadoop* and *Cassandra*. We first conduct a preliminary study to quantify the prevalence of the *IoPV* problem. We observe that 81% of the commits have at least one option manifesting an *IoPV* issue. We also observe that manually identifying such issues is challenging, as commits do not share the same options that manifest an *IoPV*. That motivates us to propose an automated model that predicts if the combination of a Commit, a Test, and an Option (***CTO***) would exhibit an *IoPV* problem. We evaluate our prediction model using the following two research questions:

- **RQ1. How accurately can we predict *IoPV* problems?**

  Our prediction model reaches an AUC up to 0.93 and 0.90 for predicting *IoPV* for *Hadoop* and *Cassandra*, respectively. Besides, we observe that random forest is the most performing model for four and three out of five performance measures (i.e., response time, CPU, memory, I/O Read, and I/O write) for *Cassandra* and *Hadoop*, respectively.

- **RQ2. What are the most important metrics for predicting *IoPV* problems?**

  We observe that all four dimensions of metrics considered in our study, namely the code structure, code change, code token, and configuration options metrics, have a statistically significant impact in predicting *IoPV*. The dimensions that are related to the configuration

Table 5.1: Our definition of configuration, option, and value

| Term | Definition | Example |
|---|---|---|
| **Option** | A typed, configurable item that allows users to set different values. | $A$ |
| **Value** | A specific assignment of a value for an option. | $A = 1$ |
| **Configuration** | An assignment of values to all options by a user. | $A = 1; B = 2$ |

options and the tokens of the changed code are the most important dimensions for both case studies.

**Organization.** The rest of this work is organized as follows: Section 5.2 provides the background information and defines *IoPV*. Section 5.7 provides prior work related to our work. Section 5.3 discusses our approach to conducting experiments and collecting data. Sections 5.4 and 5.5 present our results. Section 5.6 discusses the threats to the validity o four findings. Finally, Section 5.8 concludes the work.

## 5.2 Background

Software configuration is a mechanism used to customize the behaviour of a software system without changing the source code. The configuration **options** are often stored in configuration files as a set of key - value pairs, where the key represents an option's name and the **value** represents a default or user-chosen value for that option. We define a **configuration** as one particular assignment of a value to all existing options. Table 5.1 lists the definition of these terms. For example, *A=1* and *B=2* is one possible configuration for a software system with the two integer options *A* and *B*. Configuration options enable users to adapt the execution of their software systems by simply modifying the values of certain configuration options, without re-compilation. For example, a user can change the directory that stores the cache for *Cassandra* by changing the value of the *saved_caches_directory* configuration option.

Although configuration introduces large flexibility for users, considering all the possible configurations during testing is impossible. A software system with only three boolean configuration

options requires testing $2^3$ configurations. In fact, configuration problems are among the dominant problems in software engineering (D. Jin et al., 2014; Sayagh et al., 2018).

In particular, a software system can suffer from what we refer to as **Inconsistent Options Performance Variation** (a.k.a, *IoPV*). This occurs when, for a given commit *C*, the performance of a subset of an option's values evolved differently relative to their performance in the commit prior to *C*. Considering the example in Figure 5.1, when comparing the raw performance of the two option values *V1* and *V2* (Figure 5.1a), we observe that *V1* shows a better performance than *V2*. However, that might not be problematic as *V2* might just enable an extra feature, such as logging a transaction. In fact, in Figure 5.1b the performance of *V2* is improved compared to the prior commit, while that improvement does not manifest under option value *V1*. Similarly, Figure 5.1d shows that even if *V2* does not show any significant performance variation from the prior commit, *V1* suffers from a performance regression. A performance variation is calculated as the difference between the performance variation of each option's value after and before each commit, which is illustrated in Figure 5.1 by "$a - b$".

## 5.3 Data Collection

In this section, we present our subject systems and our approach to collect performance regressions and configuration data.

### 5.3.1 Subject Systems

In this work, we consider *Hadoop* and *Cassandra* as two subject systems. *Hadoop* White (2009) is a distributed system infrastructure, whereas *Cassandra* is a distributed NoSQL database management system. We choose these two subject systems since their performance is critical for the users and they have been studied in prior research on mining performance data (T.-H. Chen et al., 2014; Syer et al., 2017). In addition, they have a large number of configuration options. The overview of our subject systems is shown in Table 5.2.

Table 5.2: Our studied dataset.

| Subjects | # Studied Releases | Last release | # Commits | # Configuration Options | # Tests |
|---|---|---|---|---|---|
| Hadoop | 7 | 2.7.3 | 803 | 365 | 1,853 |
| Cassandra | 5 | 3.0.15 | 502 | 162 | 369 |

### 5.3.2 Data Gathering

To answer our research questions, we followed the approach that is summarized in Figure 5.2 and discussed below.

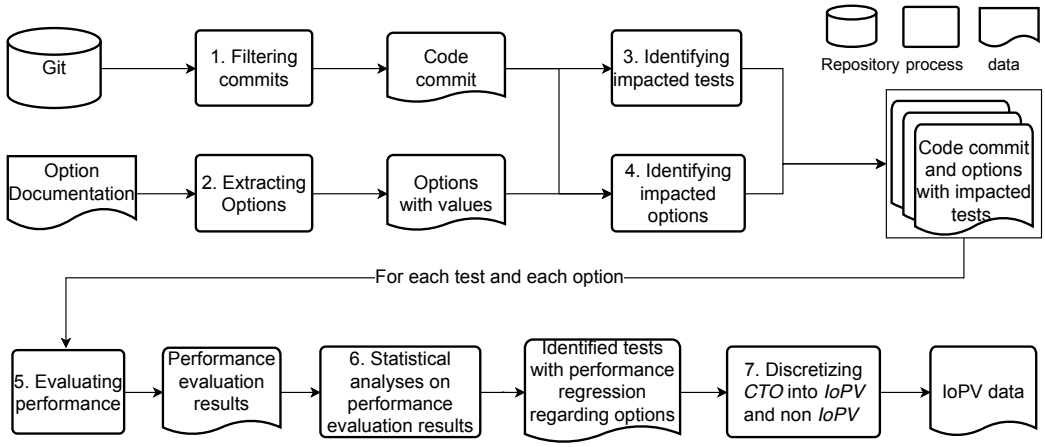



Figure 5.2: An overview of our approach to collect data.

#### Filtering Commits

Since we study performance variation across different versions of a software system, we only consider source code related changes. In particular, we filter out commits without any *java* source code changes. Furthermore, developers can commit multiple changes toward fixing the same issue, which is defined in the issue tracking system. As *Hadoop* and *Cassandra* use JIRA as a tracking system and have an explicit mapping between commits and issues, we use the issue id mentioned in the commit messages to identify the commits that belong to the same issue. If multiple commits are associated with the same issue, we only consider the last commit. This is important as developers can initially introduce a regression but then fix it before releasing the code changes related to the issue.

### Extracting Options

In the second step, we extract configuration options and their corresponding values for each subject system (i.e., 355 and 162 configuration options in the last studied releases of *Hadoop* and *Cassandra*, respectively). We obtain option names and default values by crawling the documentation of *Hadoop*[1] and *Cassandra*[2] and extracting the configuration file that is shipped with the project's releases. Finally, we manually classified the extracted options based on their expected data types (e.g., Boolean when the default value is *TRUE* or *FALSE*).

### Identifying Impacted Tests

We automatically create a mapping between the changed source code in each commit and the existing unit tests. We derive such commit-test mapping based on the automatically generated method-level code coverage results, similarly to our prior study J. Chen et al. (2020). For each commit, we automatically add logging instrumentation to each method, which will print log messages that indicate the execution of the method at runtime[3]. We then run each test for the commit. A test is considered impacted by the commit if any instrumented logging is outputted. Afterwards, we only run the tests that execute the changed source code for a given commit since executing all the existing tests of a software system for each commit and each possible configuration is practically infeasible. In addition, running those tests that are not impacted by the code change of a commit is not likely to detect performance variations (regressions or improvements under some values of an option).

### Identifying Impacted Options

Similarly to identifying which tests to run for a given commit, we also select which configuration options to change when running each of these last tests. To do so, we first identify how *Hadoop* and *Cassandra* access the value of a configuration option by searching for option names in the source

[1] https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml

[2] https://cassandra.apache.org/doc/latest/configuration

[3] Note that the instrumented versions are only used to identify impacted tests and options, and they are not used in our actual performance valuation.

code files. We found that all the options are accessible via *getters* that are defined in one class provider (e.g., *DatabaseDescriptor.java*[4] to access *Cassandra*'s options). Second, we identify the methods that invoke these configuration options' *getters*. Finally, we dynamically identify which of these methods are executed when running a test similar to the approach discussed in Section 5.3.2. If any method that can access an option is executed, the option is considered impacted by the commit.

**Evaluating Performance**

After obtaining which tests and which options are impacted by each commit, we exercise the test on each commit and its parent commit (i.e., the previous commit) to evaluate their respective performances. We first execute each test with all the configuration options set at their default values. Then we alter the value of one configuration option at a time. For the configuration options with boolean values, we alter the configuration option to the value that is not the default. For example, if the default value is *TRUE*, we would alter the value to be *FALSE*. For the numeric type option, we alter the configuration option to the value of the double of the default value and half of the default value. For example, if a configuration option has a default value of 100, we would run the test with altering the value to 200 and then run the same test with altering the value to 50. We alter each of the possible values for the enumerate type options.

Our performance evaluation environment is based on Google Compute Engine [5] with 8GB memory and 16 cores CPU. In order to generate statistically rigorous performance results, we adopt the practice of repetitive measurements T.-H. Chen, Shang, Hassan, et al. (2016) to evaluate performance. Conservatively, we executed each test 30 times independently, which is larger than prior work that repeats a test only 5 to 20 times (Laaber & Leitner, 2018; Laaber et al., 2019; Leitner & Cito, 2016).

To measure the performance that is associated with each test, we use a performance monitoring tool named *psutil* [6] (Python system and process utilities). *Psutil* can capture detailed performance metrics and has been widely used in prior research (J. Chen & Shang, 2017; Yao et al., 2018). We

[4] https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/config/DatabaseDescriptor.java

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

[6] https://github.com/giampaolo/psutil

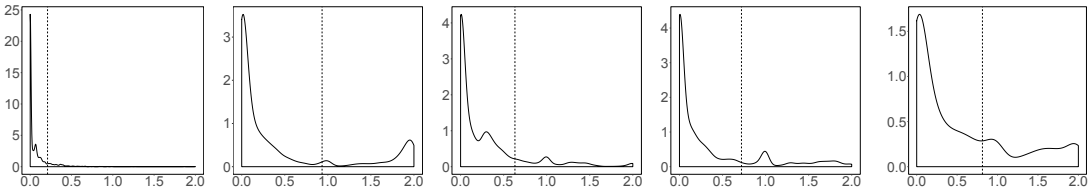
collect both domain level and physical level performance metrics. In our execution, we collect five performance metrics during the execution, including response time, CPU usage, memory usage, I/O read and I/O write. The entire data collection process took around 12,536 machine-hours.

**Statistical Analyses on Performance Evaluation Results**

To identify the *IoPV*, we statistically compare the performance of a given test and a configuration option value before and after each commit using the Mann-Whitney U test Nachar et al. (2008) (i.e., $\alpha = 0.05$) and Cliff's delta Becker (2000) that measures the magnitude of performance regressions. 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 indicate statistical significance even if the difference is trivial). Thus, we also use Cliff's delta to quantify the magnitude of the differences (a.k.a., effect sizes). Cliff's delta measures the effect size statistically and has been used in prior engineering studies (Kitchenham et al., 2002; Li, Chen, Shang, & Hassan, 2018; Liao et al., 2020). Cliff's delta ranges from -1 to +1, where a value of 0 indicates two identical distributions. Therefore, each commit, test, and option's value has a Cliff's delta value. We then calculate the differences between the maximum and minimum Cliff's delta for an option's different values, which we used later on to discretize a commit, test, and option as an *IoPV* or a non-*IoPV*, as discussed in the following paragraph.

We also measure whether a test manifesting a performance regression when the value of the effect size is positive and indicates medium ($0.33 <$ *Cliff's delta* $\leqslant 0.474$) or large ($0.474 <$*Cliff's delta*) magnitude. On the other hand, we consider a test manifesting a performance improvement if the value of the effect size is negative and indicates medium ($-0.33 <$ *Cliff's delta* $\leqslant -0.474$) or large ($-0.474 >$*Cliff's delta*) magnitude.

Note that we consider this statistical analysis for each performance metric (i.e., response time, CPU usage, memory usage, I/O read and I/O write) separately. For example, a commit may show a CPU regression or improvement, but does not show any differences for the response time.

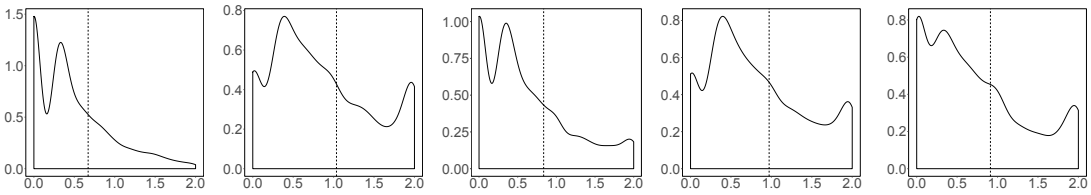


(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.3: The automatically obtained threshold for splitting option variation into *IoPV* and non-*IoPV* groups for *Hadoop*.

**Discretizing *CTO* into *IoPV* and non *IoPV***

In the final step, we categorize each commit, test, and option (*CTO*) into a *IoPV* or a non *IoPV* based on an automatically determined threshold. Our intuition is that the maximum difference values would be concentrated in either small values (i.e., when adjusting an option does not make a difference) or large values (i.e., when adjusting an option does make a difference), which is demonstrated in Figure 5.1. Specifically, we use Ckmeans.1d.dp J. Song (2016), a one-dimensional clustering method, to find a threshold that separates the maximum difference values into two groups, i.e., *IoPV* and a non-*IoPV* (see Figure 5.3 and 5.4). Note that the option variation range between 0 when there is no variation and 2 when the effect size (c.f. Section 5.3.2) is 1 for one option value and -1 for another value of the same option. Most of the automatically obtained thresholds are close to 1. That indicates, as an example, that one option value might show a large performance effect size compared to the prior commit, while another value for the same option does not show any difference from the prior commit (effect size equals to 0). A second example is when one option's value shows a large performance improvement over the prior commit (effect size equals to -1), when another value of the same option does not show any statistically significant difference (effect size equals to 0).



(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.4: The automatically obtained threshold for splitting option variation into *IoPV* and non-*IoPV* groups for *Cassandra*.

## 5.4 Preliminary Study: Quantifying the Prevalence of *IoPV* and the Challenge of Identifying *IoPV*

This preliminary study will shed light on the problem of *IoPV*, which can be further explored by future work. We quantify, through this preliminary study, the existence of the *IoPV* problem in large and highly configurable software systems, as well as how difficult is it to identify the *IoPV*. This preliminary analysis will also motivate the need for an approach that automatically identifies the *IoPV*. In particular, we address the following preliminary research questions:

PQ1: How common are *IoPV* issues?

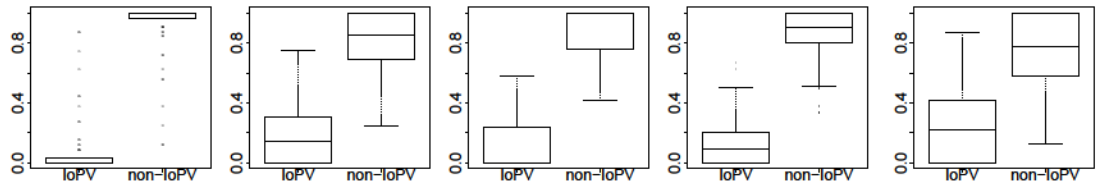
PQ2: How difficult is it to manually identify *IoPV* issues?

### PQ1. How common are *IoPV* issues?

#### Motivation

The goal of this preliminary research question is to quantify and provide scientific evidence on how often a configuration option can suffer from the *IoPV* problem. While a new code change might not show any performance regression under the default configuration, another configuration can hide a performance regression that can go as unseen to the production environment. This is an important problem as performance issues often lead to serious monetary losses[7]. similarly, a configuration improvement might not be manifested under all the configurations.

[7] https://www.eweek.com/networking/it-outages-cause-businesses-26.5-billion-in-lost-revenue-each-year-survey




(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.5: Percentage of *IoPV* for each commit of *Hadoop*.

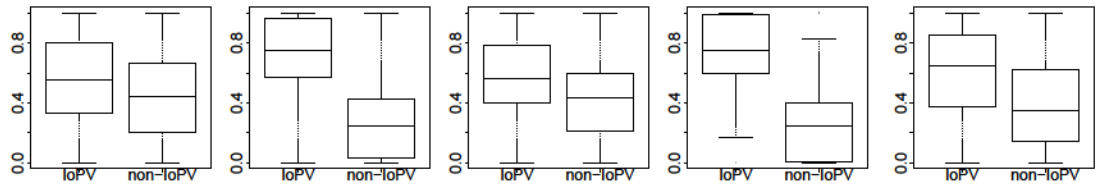
### Approach

To quantify the prevalence of *IoPV*, we followed the approach discussed in Section 5.3 to collect the performance data. In particular, we first collect performance measurements for each commit, test, and option (which we refer to as ***CTO***) and label each *CTO* as *IoPV* or a non *IoPV*. Then, we identify for each commit and unit test the number of configurations under which the performance is statistically significantly worse (aka, performance regression) or better (aka, performance improvement) than the performance of the same test and configuration in the prior commit. Finally, we quantify for each commit the number of tests that show a performance regression or a performance improvement under just a subset of the existing configurations. In the studied *Hadoop* and *Cassandra* releases, there are 4,902 and 4,197 *CTO*, respectively.

### Result

**The *IoPV* is a common problem, as 61% and 91% of our studied *CTO* in *Hadoop* and *Cassandra* suffer from the *IoPV* problem.** In addition, each *Hadoop* and *Cassandra* commit has a median percentage of 43% and 96% of the tests and options that manifest at least one *IoPV*, as further detailed in Figures 5.5 and 5.6. Although only a small percentage of *Hadoop* tests and options suffer from an *IoPV* in terms of response time, the other performance metrics show a large percentage of tests and options that suffer from *IoPV*, as shown in Figure 5.5. On the other hand, the percentage of tests and options that suffer from an *IoPV* is larger than the tests and options that do not face an *IoPV* for *Cassandra* across all the performance metrics, as shown in Figure 5.6.

24% and 54% of the *CTO* in *Hadoop* and *Cassandra*, respectively, show a performance regression on at least one performance measure when the default configuration does not show any




(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.6: Percentage of *IoPV* for each commit of *Cassandra*.

performance regression. 68% and 89% of our studied commits in *Hadoop* and *Cassandra* have at least one *CTO* that shows a performance regression under one non-default configuration while showing no regression under the default configuration, which indicates that having a hidden performance regression is common. For instance, we observe a performance regression in terms of response time on 42 and 1,023 out of 4,902 and 4,197 *Hadoop* and *Cassandra CTO*, respectively, when the default configuration does not show any regression, as shown in Table 5.3. As shown in the same Table, the performance measure that suffers the most from the *IoPV* problem are the I/O write and CPU measures. In addition, these are not minor regression differences, as 78% of the regressions are large based on our effect size analysis.

14% and 20% of the *CTO* in *Hadoop* and *Cassandra* show a performance regression when the default configuration shows a performance improvement. Therefore, improving the performance of a system should consider different configurations. For example, 439 and 594 *Hadoop* and *Cassandra CTO* show a CPU performance regression when the default configuration shows an improvement, as shown in Table 5.4. This problem occurs for all of the studied performance measures. Similarly to our last finding, 58% and 65% of the commits in *Hadoop* and *Cassandra*, and a median of 10% and 14% of *CTO* per commit in *Hadoop* and *Cassandra* are impacted by such a problem.

Note that we also observe cases for which an option manifests a regression under the default value, but a non-regression or even an improvement under other values of the same option, as shown in Table 5.5. 25% and 34% of the *CTO* in *Hadoop* and *Cassandra* have performance regression under default value but non-regression under other values.

Table 5.3: Number of *CTO* with no regression under the default option value but with regression under other option values. Medium (large) means the effect size *Cliff's delta* of performance regression is medium (large).

| subject | #CTO | Response time | | CPU | | Memory | | I/O read | | I/O write | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | large | medium | large | medium | large | medium | large | medium | large | medium |
| Hadoop | 4,902 | 24 | 18 | 517 | 84 | 208 | 214 | 208 | 214 | 528 | 98 |
| Cassandra | 4,197 | 600 | 423 | 1,094 | 352 | 788 | 404 | 1,033 | 363 | 921 | 326 |

Table 5.4: Number of *CTO* with improvement under the default option value but with regression under other option values.

| subject | #CTO | Response time | | CPU | | Memory | | I/O read | | I/O write | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | large | medium | large | medium | large | medium | large | medium | large | medium |
| Hadoop | 4,902 | 4 | 3 | 425 | 14 | 102 | 36 | 170 | 30 | 426 | 46 |
| Cassandra | 4,197 | 122 | 53 | 450 | 95 | 220 | 52 | 412 | 93 | 327 | 74 |

## PQ2. How difficult is it to manually identify *IoPV* issues?

### Motivation

The goal of this preliminary question is to understand how difficult the manual identification of *IoPV* (i.e., identification of *IoPV* without running the tests) is. For instance, the higher the number of options that manifest an *IoPV* in a large number of commits and tests, the more difficult the identification of *IoPV* is, as it indicates that an *IoPV* can occur in an unexpected way and any option can be responsible for such a problem. On the other hand, the lower the number of options that suffer from an *IoPV*, the easiest it is to test all of these *IoPV* responsible options.

### Approach

To investigate the difficulty of identifying an *IoPV*, we calculate the intersection of the <test, option, *IoPV* > triplets between each pair of commits using the Jaccard similarity defined as follows:

$$J(C1, C2) = \frac{|CTO_{C1} \cap CTO_{C2}|}{|CTO_{C1} \cup CTO_{C2}|} \tag{2}$$

where $C1$ and $C2$ refer to every pair of commits (both consecutive and non-consecutive commits). $|CTO_{C1} \cap CTO_{C2}|$ is the number of *CTO* that share the same <test, option, *IoPV* > (i.e., the intersection). $|CTO_{C1} \cup CTO_{C2}|$ is the total number of unique <test, option, *IoPV* > in commits

Table 5.5: Number of *CTO* with regression under the default option value and non-regression/improvement under other values.

| subject | #CTO | Response time | | CPU | | Memory | | I/O read | | I/O write | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | large | medium | large | medium | large | medium | large | medium | large | medium |
| Hadoop | 4,902 | 17 | 9 | 431 | 60 | 128 | 200 | 228 | 64 | 441 | 86 |
| Cassandra | 4,197 | 236 | 222 | 592 | 298 | 314 | 229 | 553 | 264 | 439 | 229 |

Table 5.6: Example of Jaccard similarity between each pair of commits. The two commits share two of the five unique <test, option, *IoPV* > triplets. Thus the Jaccard similarity is 2/5 = 0.4.

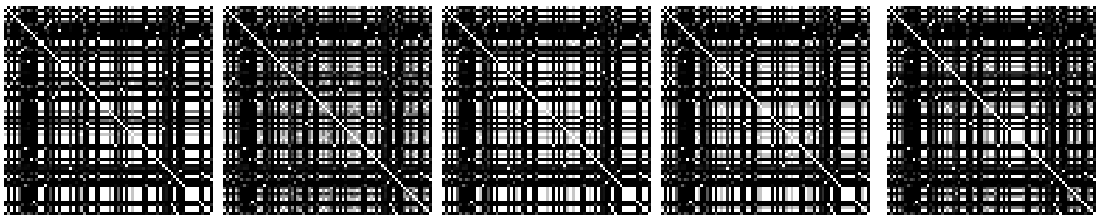
| Commit | Test | Configuration Option | IoPV |
|---|---|---|---|
| commit1 | test 1 | option a | 1 |
| | test 2 | option a | 0 |
| | test 2 | option b | 0 |
| commit2 | test 1 | option a | 1 |
| | test 2 | option a | 0 |
| | test 2 | option b | 1 |
| | test 3 | option a | 0 |

$C1$ and $C2$ (i.e., the union). The Jaccard distance ranges between 0 and 1, where a value of 1 means that the pair of commits share the same <test, option, *IoPV* >, while 0 indicates that the pair of commits does not share any <test, option, *IoPV* > triplet. For example, in Table 5.6, there are three *CTO* in *Commit1* and four *CTO* in *Commit2*. The number of the intersection of $|CTO_{C1} \cap CTO_{C2}|$ is 2, and the union of $|CTO_{C1} \cup CTO_{C2}|$ is 5. Therefore, the $J(C1, C2)$ is 0.4 (2/5) between *Commit1* and *Commit2*.

### Results

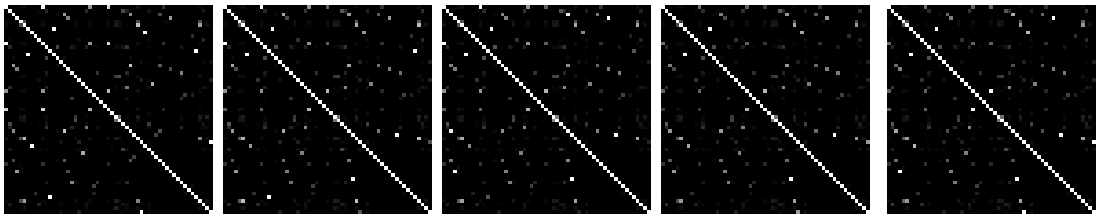
***IoPV* problems are hard to manually predict.** In particular, 81% and 100% of the *Hadoop* and *Cassandra* commits show at least one *CTO* with an *IoPV*. Similarly, all the options of *Hadoop* and *Cassandra* suffer at least once from a *IoPV* through the studied commits. Table 5.7 shows more details about how common are *IoPV* for the studied commits, tests, and options. In summary, our results indicate that the *IoPV* problem is not limited to a small set of commits, tests, or options, which makes it a challenge to predict which *CTO* would have an *IoPV*.

Even if most of the commits show at least one *IoPV*, it is not easy to predict which test and option may suffer from the *IoPV*. Figure 5.7 and Figure 5.8 show the pairwise Jaccard distance between



(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.7: Pairwise Jaccard distance between the <test, option, *IoPV* > triplets of the studied commits of the *Hadoop* system. The $x$-axis and $y$-axis show the studied commits, ordered chronologically from left to right on the $x$-axis and bottom to top on the $y$-axis. Each cell of the Figure refers to the Jaccard distance of any pair of commits: the darker the color is, the larger the distance is.



(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.8: Pairwise Jaccard distance between the <test, option, *IoPV* > triplets of the studied commits of the *Cassandra* system. The $x$-axis and $y$-axis show the studied commits, ordered chronologically from left to right on the $x$-axis and bottom to top on the $y$-axis. Each cell of the Figure refers to the Jaccard distance of any pair of commits: the darker the color is, the larger the distance is.

the <test, option, *IoPV* > triplets of the studied commits in the *Hadoop* and *Cassandra* systems, respectively. The figures indicate that most of the commits do not share any <test, option, *IoPV* > (i.e., with dark cells), especially for the *Cassandra* system (i.e., more dark cells). In other words, different commits are unlikely to have the same tests and options that can lead to *IoPV* problems. Therefore, it is difficult for developers to manually identify which tests and options that they need to run and configure to verify the existence of *IoPV*.

In order to understand why different commits show inconsistent <test, option, *IoPV* > triplets, we manually analyze some commits that show the largest Jaccard distance from other commits (i.e., with dark horizontal/vertical lines in Figure 5.7 and Figure 5.8). We consider the response time measure as an example in our manual analysis. In particular, there are two and six commits with large Jaccard distance ($> 0.8$) to all other commits in *Hadoop* and *Cassandra*, respectively.

For *Hadoop*, we find that most of the options cause *IoPV* in the test named *TestKMS.java* in the *Hadoop-common* sub-project. Then, we pick up one[8] out of the two commits of *Hadoop* to manually examine the configuration option and the commit changes. We find that the options that cause *IoPV* are usually related to connection time, such as the options *dfs.ha.fencing.ssh.connect-timeout* and *fs.s3a.connection.timeout*. By examining the code in the test *TestKMS.java*, we find that *TestKMS.java* loads the connection timeout configuration options. Thus, the commits that impact such connection time related options and the test may lead to *IoPV* problems while other commits may not lead to the same *IoPV*. For *Cassandra*, we select one commit [9] with the largest Jaccard distance to other commits. Our results show that two tests named *EmbeddedCassandraServiceTest* and *DebuggableScheduledThreadPoolExecutorTest* manifest the largest performance regression regarding options *max_hints_file_size_in_mb* and *memtable_heap_space_in_mb*, respectively. By manually examining the commit changes covered by the tests, we find that there exist code changes in the method *start* within the Java file *EmbeddedCassandraService.java*[10]. In particular, 69% of commits in *Hadoop and* 96% of commits in *Cassandra* have a Jaccard distance more than 0.5. Such a change adds function calls to initialize the options and leads to performance regressions. In summary, different commits may lead to different options and tests that exhibit *IoPV* problems.

> **IoPV** ***is a common problem and it is difficult to manually identify*** **IoPV** ***without exhaustively running the tests. Our results suggest the need for an approach that automatically identifies which*** **CTO** ***manifests an*** **IoPV.**

[8] https://github.com/apache/hadoop/commit/b17d365f

[9] https://github.com/apache/cassandra/commit/0fe82be8

[10] https://github.com/apache/cassandra/blob/0fe82be83ccceceb12172d63913388678253413bc/src/java/org/apache/cassandra/service/EmbeddedCassandraService.java

Table 5.7: Number of unique commits, tests, options with *IoPV* problems.

| | | Commit | | Test | | Option | |
|---|---|---|---|---|---|---|---|
| | | Total | IoPV | Total | IoPV | Total | IoPV |
| Hadoop | Res. time | 74 | 27 | 74 | 13 | 122 | 74 |
| | CPU | 74 | 47 | 74 | 62 | 122 | 113 |
| | Memory | 74 | 38 | 74 | 59 | 122 | 113 |
| | I/O read | 74 | 45 | 74 | 53 | 122 | 108 |
| | I/O write | 74 | 47 | 74 | 57 | 122 | 117 |
| | Any metric | 74 | 60 | 74 | 67 | 122 | 117 |
| Cassandra | Res. time | 57 | 55 | 216 | 189 | 54 | 43 |
| | CPU | 57 | 55 | 216 | 204 | 54 | 49 |
| | Memory | 57 | 53 | 216 | 202 | 54 | 43 |
| | I/O read | 57 | 56 | 216 | 202 | 54 | 44 |
| | I/O write | 57 | 53 | 216 | 192 | 54 | 39 |
| | Any metric | 57 | 56 | 216 | 208 | 54 | 50 |

## 5.5 Predicting *IoPV* Problems

### RQ1. How accurately can we predict *IoPV* problems?

#### Motivation

The goal of this research question is to evaluate different classification approaches on predicting for which *CTO* one has to check multiple option's values. In our preliminary study, we observe that the *IoPV* is common and hard to manually predict, which indicates that developers need to test different values for each option. However, as there are typically a large number of configuration options (e.g., *Hadoop* version 2.7.3 has 365 configuration options) with different possible values, exhaustively experimenting with all different options for each test in performance testing is time and resource-consuming. In this RQ, we aim to reduce the effort of conducting configuration-aware performance testing by predicting the need for adjusting a configuration option for a test when a code change is made (i.e., for a *CTO*). Specifically, our approach predicts whether a *CTO* manifests an *IoPV*, such that developers can make an informed decision on whether they should consider different values for that option in their performance testing.

### Approach

In this RQ, we build ML models to predict whether a *CTO* manifests an *IoPV*. Below, we describe the detailed steps involved in our modeling process.

**Step 1. Data preparation.**

*Step 1.1. Defining the target variable.* Our target variable is a binary variable that indicates whether a *CTO* manifests an *IoPV*, which we obtained following the approach discussed in Section 5.3. In particular, after collecting performance measurements, we calculated the performance variation for each option, and discretize each *CTO* into an *IoPV* or a non *IoPV* following the discretization approach of Section 5.3.2.

*Step 1.2. Selecting the features.* We consider four dimensions of software metrics that are related to the likelihood of a configuration option impacting the performance testing of a code commit for each test (i.e., of a *CTO*). Table 5.8 lists the detailed metrics used in our models. We already found in our prior work J. Chen et al. (2020) that code structure, and code change dimensions are important for predicting performance regressions, but we did not consider the impact of different configurations on the manifestation of performance regressions. Therefore, we use the prior dimensions as well as an additional dimension about the configuration options.

**Code change metrics**. This dimension contains metrics that capture the code changes in a commit (e.g., the number of changed lines of code). Some code changes (e.g., big code changes) might be more likely to impact how different values of a configuration option make difference in performance testing.

**Code structure metrics**. This dimension contains metrics that describe the static structure of the source code files (e.g., Cyclomatic complexity). Our intuition is that changing certain code (e.g., complex code) is more likely to alter the performance impact of a configuration option.

**Code token metrics**. This dimension considers the code tokens of the methods that are changed in the commit and the test file. Some code tokens (e.g., "speed") may be more related to the performance than other tokens. We use the *lscp*[11] tool to extract the code tokens from the source code.

[11]Lscp is a lightweight source code preprocessor that can be used to isolate and manipulate the linguistic data (i.e., identifier names, comments, and string literals) from the source code: https://github.com/doofuslarge/lscp

Table 5.8: Overview of our selected metrics.

| Dimension | Metric | Rationale |
|---|---|---|
| Code change | Number of modified subsystems | The more subsystems are changed, the higher risk the change may be Mockus and Weiss (2000). |
| | Number of modified directories | Changing more directories may more likely introduce performance regressions Mockus and Weiss (2000). |
| | Number of modified files | Changing many source files are more likely to cause performance regressions N. Nagappan et al. (2006). |
| | Distribution of modified code across files | Scattered changes are more possible to introduce performance regressions Hassan (2009). |
| | Number of modified methods | Changes altering many methods are more likely introduce performance regressions Zimmermann et al. (2007). |
| | Number of lines SOC in tests | Program with more lines is more likely to suffer from performance regressions Koru et al. (2009). |
| | Lines of code added | The more lines of code added, the higher risk that the program will suffer from performance regressions. |
| | Lines of code deleted | The more lines of code deleted, the higher risk of performance regression is introduced N. Nagappan and Ball (2005). |
| Code structure | Number of methods in impacted test | Program with large number of methods is more likely to suffer from performance regressions. |
| | McCabe Cyclomatic complexity | Program with higher complexity is more likely to suffer from performance regressions Hassan (2009). |
| | Number of called subprograms | Large called subprograms will amplify the regression if there exist performance regressions in the called program N. Nagappan et al. (2006). |
| | Number of calling subprograms | Large calling subprograms will amplify the regressions if there exist performance regressions in the called program N. Nagappan et al. (2006). |
| Code token | Code tokens of the changed source code | Some code tokens may be more related to performance than other tokens. |
| Configuration option | Splited configuration option names | The name of configuration option may be related to a specific performance metric. |

**Configuration option metrics**. This dimension considers the characteristics of the configuration option (e.g., the tokens in the configuration option). We assume that some options (e.g., system resource-related options) are more likely to impact the performance regressions detection results of a commit.

*Step 1.3. Pre-processing the features.* The code token metrics include thousands of unique code tokens. Thus, we need to pre-process such metrics into a numeric representation. We consider three approaches to pre-process the code token metrics.

- Term frequency-inverse document frequency (tf-idf). Tf-idf Ramos et al. (2003) generates a feature for each unique token. The value of a feature for a commit is the term frequency of the corresponding token (i.e., $tf(t, c) = f_{t,c}$, where $f_{t,c}$ is the number of times a token $t$ appears in commit $c$) times the inverse frequency of the commits that contain the token ($idf(t) = \log(N/N_t)$, where $N$ is the total number of commits while $N_t$ is the number of commits containing the token $t$.)

- Principal component analysis (PCA). Using tf-idf generates a larger number of features that may lead to very complex models. Therefore, we apply PCA Wold, Esbensen, and Geladi (1987) on the features resulted from tf-idf to reduce the number of features. Specifically, we only consider the top principal components that contribute to 95% of the variance in the features together.

- Word embeddings. We use word2vec (Mikolov, Chen, Corrado, & Dean, 2013; Mikolov, Sutskever, Chen, Corrado, & Dean, 2013) to code each token into a vector of 128 numerical values. Specifically, we pre-train the embeddings from a large code base[12] then applying the pre-trained embeddings on the tokens in our data. Then we use a mean aggregation of the vectors representing the tokens in a commit.

**Step 2. Model construction.** We build machine learning models to predict whether a configuration option suffers from an *IoPV* on a given *CTO*. For the generalization of our results, we consider five different types of models, including random forest (RF), logistic regression (LR), XGBoost (XG), neural network (NN), and convolutional neural network (CNN). A random forest

[12] https://doi.org/10.5281/zenodo.3801975

is a classifier consisting of a collection of decision tree classifiers and each tree casts a vote for the most popular class for a given input Breiman (2001). Logistic regression is a statistical model that uses a logit function to model a binary variable (the target variable) as a linear combination of the independent variables Hosmer Jr, Lemeshow, and Sturdivant (2013), which is widely used in software analytics (Shang et al., 2015; Tantithamthavorn et al., 2018). XGBoost is an efficient and accurate implementation of the gradient boosting algorithm, which is reported to perform better than other machine learning models in software engineering applications Liao et al. (2020). The neural network model Glorot, Bordes, and Bengio (2011) used in our study consists of four layers and are trained with 100 batch size, and 10 epochs. The CNN model Lawrence, Giles, Tsoi, and Back (1997) in our study consists of five layers, and are trained with 100 batch size, and 10 epochs.

Prior to constructing our models, we check the pairwise correlation between our features using the Pearson correlation test ($\rho$) Benesty et al. (2009). We choose the Pearson correlation method because it is robust to non-normally distributed data. In this work, we choose the correlation value 0.7 as the threshold to remove collinearity. In other words, if the correlation between a pair of features is greater than 0.7 ($|\rho| > 0.7$), we keep one of the two features in the model. We then perform a redundancy analysis on the features. In particular, we use each feature as a dependent variable and use the remaining features as independent variables to build a regression model and calculate the $R^2$ of each model. If the $R^2$ is more than 0.9 Syer et al. (2017), the current dependent variable is considered redundant.

**Step 3. Model evaluation.** We use 10-fold cross-validation to evaluate the performance of our models. In each repetition of the 10-fold cross-validation, the whole data set is randomly partitioned into 10 sets of roughly equal size. One subset is used as the testing set (i.e., the held-out set) and the other nine subsets are used as the training set. We train our models using the training set and evaluate the performance of our models on the held-out set. The process repeats 10 times until all subsets are used as a testing set once.

In each fold of the cross-validation, we use precision, recall, and AUC to measure the performance of our models. Precision measures the ratio of cases when a configuration option actually impacts the performance regressions detection among all the cases that our models predict to adjust a configuration option (i.e., $\frac{\text{true positives}}{\text{true positives}+\text{false positives}}$). Recall measures the ratio of cases

when our models predict to adjust a configuration option among all the cases when a configuration option actually impacts the performance regressions detection (i.e., $\frac{\text{true positives}}{\text{true positives+false negatives}}$). AUC measures our models' ability to discriminate the *CTO* cases into *IoPV* and non *IoPV* cases. Specifically, AUC is the area under the ROC curve which plots the true positive rate against the false positive rate under different classification thresholds. Prior work recommends the use of AUC over threshold-dependent measures (e.g., precision and recall) when measuring the model performance Tantithamthavorn and Hassan (2018b).

**Results**

**Our models can effectively predict when a *CTO* is manifesting an *IoPV* for all of our five studied performance measures,** as shown in Table 5.9 and Table 5.10. Our best models (i.e., as indicated by the ***bold-italic*** values) achieve an AUC of 0.85 to 0.94 on the *Hadoop* project and 0.79 to 0.90 on the *Cassandra* project, for different performance metrics. For the *Hadoop* project, RF is the best model for four out of the five performance metrics, which achieves an AUC of 0.85 to 0.93. Even if XG shows the best AUC performance for the fifth performance metric (i.e., Response time), the difference between RF and XG is only 0.01. For the *Cassandra* project, RF shows the best performance on three out of five performance metrics. NN shows the best performance on also three performance metrics (Memory and I/O read are the same to RF model). The average AUC of the best NN model is 0.83 while the average AUC of the best RF model is 0.82. Note that NN, on the other side, requires a large amount of resources to train and test a model, while the improvements it shows over RF is trivial. CNN shows the best performance on only one performance metric (i.e., with an AUC of 0.79 for the Response time). However, the average AUC of the best CNN model is 0.09 lower than that of RF. In summary, we suggest that developers consider the RF model for predicting when a *CTO* has an *IoPV* problem.

Our best models achieve a better performance for the *Hadoop* system than for the *Cassandra* system. As discussed in RQ1, the different commits show more inconsistent <test, option, *IoPV* > triplets (i.e., more dark cells in Figure 5.8) in the *Cassandra* system than in the *Hadoop* system, thus it is more difficult to predict *IoPV* for the *Cassandra* system, which could explain the reason that our models perform better for the *Hadoop* project.

Table 5.9: *Hadoop*'s results of using different models to predict whether configuration options cause the manifesting of performance regressions. The best results for each performance metric and each model are highlighted in *italic*. The best results for each performance metric across different models are highlighted in ***bold-italic***.

| Hadoop | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | RF with tf-idf | | | RF with PCA | | | RF with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.68 | 0.39 | *0.93* | 0.68 | 0.39 | 0.66 | 0.73 | 0.33 | *0.93* |
| Cpu | 0.70 | 0.51 | 0.90 | 0.55 | 0.02 | 0.71 | 0.77 | 0.60 | ***0.92*** |
| Memory | 0.64 | 0.36 | 0.87 | 0.48 | 0.04 | 0.69 | 0.75 | 0.41 | ***0.91*** |
| I/O Read | 0.68 | 0.54 | 0.91 | 0.58 | 0.02 | 0.76 | 0.79 | 0.56 | ***0.93*** |
| I/O Write | 0.63 | 0.44 | 0.82 | 0.44 | 0.02 | 0.59 | 0.72 | 0.49 | ***0.85*** |
| Average | 0.67 | 0.45 | 0.89 | 0.55 | 0.10 | 0.68 | 0.75 | 0.48 | ***0.91*** |
| | LR with tf-idf | | | LR with PCA | | | LR with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.38 | 0.03 | 0.67 | 0.12 | 0.46 | 0.54 | 0.53 | 0.09 | *0.77* |
| Cpu | 0.66 | 0.06 | 0.73 | 0.27 | 0.29 | 0.61 | 0.48 | 0.14 | *0.76* |
| Memory | 0.49 | 0.04 | 0.71 | 0.16 | 0.40 | 0.55 | 0.48 | 0.10 | *0.73* |
| I/O Read | 0.70 | 0.05 | 0.71 | 0.22 | 0.33 | 0.57 | 0.46 | 0.18 | *0.80* |
| I/O Write | 0.50 | 0.06 | 0.64 | 0.33 | 0.22 | 0.57 | 0.50 | 0.14 | *0.66* |
| Average | 0.55 | 0.05 | 0.69 | 0.22 | 0.34 | 0.57 | 0.49 | 0.13 | *0.74* |
| | XG with tf-idf | | | XG with PCA | | | XG with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.65 | 0.42 | ***0.94*** | 1.00 | 0.05 | 0.60 | 0.71 | 0.31 | 0.93 |
| Cpu | 0.66 | 0.48 | *0.88* | 0.32 | 0.06 | 0.62 | 0.67 | 0.50 | *0.88* |
| Memory | 0.66 | 0.32 | *0.87* | 0.41 | 0.04 | 0.68 | 0.72 | 0.32 | *0.87* |
| I/O Read | 0.66 | 0.49 | *0.91* | 0.49 | 0.08 | 0.73 | 0.73 | 0.50 | *0.91* |
| I/O Write | 0.66 | 0.38 | *0.82* | 0.41 | 0.16 | 0.58 | 0.67 | 0.40 | 0.80 |
| Average | 0.66 | 0.42 | *0.88* | 0.52 | 0.08 | 0.64 | 0.70 | 0.41 | 0.88 |
| | NN with tf-idf | | | NN with PCA | | | NN with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.34 | 0.54 | 0.79 | 0.27 | 0.83 | *0.80* | 0.27 | 0.64 | 0.75 |
| Cpu | 0.53 | 0.30 | 0.72 | 0.63 | 0.41 | *0.73* | 0.39 | 0.33 | 0.65 |
| Memory | 0.43 | 0.27 | *0.67* | 0.52 | 0.34 | *0.67* | 0.31 | 0.42 | 0.66 |
| I/O Read | 0.53 | 0.44 | 0.73 | 0.60 | 0.46 | *0.76* | 0.48 | 0.33 | 0.72 |
| I/O Write | 0.50 | 0.38 | *0.68* | 0.53 | 0.32 | 0.65 | 0.39 | 0.41 | *0.68* |
| Average | 0.47 | 0.39 | *0.72* | 0.51 | 0.47 | *0.72* | 0.37 | 0.43 | 0.69 |
| | CNN with tf-idf | | | CNN with PCA | | | CNN with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.29 | 0.48 | 0.75 | 0.06 | 0.90 | 0.73 | 0.23 | 0.51 | *0.79* |
| Cpu | 0.22 | 0.68 | 0.78 | 0.18 | 0.78 | 0.76 | 0.63 | 0.25 | *0.81* |
| Memory | 0.47 | 0.25 | 0.69 | 0.13 | 0.87 | 0.74 | 0.20 | 0.57 | *0.76* |
| I/O Read | 0.32 | 0.41 | *0.68* | 0.27 | 0.25 | *0.68* | 0.20 | 0.38 | 0.66 |
| I/O Write | 0.27 | 0.31 | 0.64 | 0.14 | 0.64 | 0.65 | 0.19 | 0.60 | *0.67* |
| Average | 0.31 | 0.43 | 0.71 | 0.16 | 0.69 | 0.71 | 0.29 | 0.46 | *0.74* |

Table 5.10: *Cassandra*'s results of using different models to predict whether configuration options cause the manifesting of performance regression. The best results for each performance metric and each model are highlighted in *italic*. The best results for each performance metric across different models are highlighted in ***bold-italic***.

| Cassandra | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | RF with tf-idf | | | RF with PCA | | | RF with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.74 | 0.37 | 0.74 | 0.45 | 0.13 | 0.62 | 0.67 | 0.46 | *0.75* |
| Cpu | 0.68 | 0.39 | 0.76 | 0.46 | 0.15 | 0.61 | 0.73 | 0.59 | *0.82* |
| Memory | 0.71 | 0.37 | 0.78 | 0.35 | 0.04 | 0.61 | 0.71 | 0.58 | ***0.84*** |
| I/O Read | 0.74 | 0.48 | 0.79 | 0.54 | 0.32 | 0.67 | 0.74 | 0.63 | ***0.83*** |
| I/O Write | 0.76 | 0.50 | 0.82 | 0.58 | 0.32 | 0.68 | 0.77 | 0.65 | ***0.86*** |
| Average | 0.73 | 0.42 | 0.78 | 0.47 | 0.19 | 0.64 | 0.72 | 0.58 | *0.82* |
| | LR with tf-idf | | | LR with PCA | | | LR with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.38 | 0.38 | 0.59 | 0.28 | 0.54 | 0.54 | 0.38 | 0.51 | *0.63* |
| Cpu | 0.49 | 0.42 | 0.64 | 0.33 | 0.41 | 0.53 | 0.46 | 0.58 | *0.65* |
| Memory | 0.44 | 0.26 | 0.62 | 0.29 | 0.28 | 0.55 | 0.43 | 0.47 | *0.66* |
| I/O Read | 0.50 | 0.50 | 0.63 | 0.35 | 0.44 | 0.55 | 0.49 | 0.61 | *0.67* |
| I/O Write | 0.53 | 0.51 | *0.69* | 0.36 | 0.37 | 0.54 | 0.47 | 0.63 | 0.68 |
| Average | 0.47 | 0.41 | 0.64 | 0.32 | 0.41 | 0.54 | 0.44 | 0.56 | *0.66* |
| | XG with tf-idf | | | XG with PCA | | | XG with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.66 | 0.38 | *0.75* | 0.37 | 0.20 | 0.60 | 0.63 | 0.44 | 0.74 |
| Cpu | 0.65 | 0.49 | 0.77 | 0.49 | 0.32 | 0.63 | 0.70 | 0.60 | *0.82* |
| Memory | 0.65 | 0.49 | 0.80 | 0.45 | 0.13 | 0.60 | 0.70 | 0.56 | *0.83* |
| I/O Read | 0.69 | 0.55 | 0.79 | 0.46 | 0.35 | 0.61 | 0.72 | 0.62 | *0.81* |
| I/O Write | 0.74 | 0.59 | 0.84 | 0.48 | 0.33 | 0.65 | 0.74 | 0.64 | *0.85* |
| Average | 0.68 | 0.50 | 0.79 | 0.45 | 0.27 | 0.62 | 0.70 | 0.57 | *0.81* |
| | NN with tf-idf | | | NN with PCA | | | NN with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.55 | 0.36 | 0.71 | 0.53 | 0.40 | *0.73* | 0.47 | 0.41 | 0.67 |
| Cpu | 0.27 | 0.94 | 0.88 | 0.31 | 0.96 | ***0.90*** | 0.26 | 0.88 | 0.84 |
| Memory | 0.56 | 0.51 | 0.76 | 0.64 | 0.57 | ***0.84*** | 0.61 | 0.49 | 0.76 |
| I/O Read | 0.66 | 0.46 | 0.75 | 0.67 | 0.57 | ***0.83*** | 0.60 | 0.54 | 0.74 |
| I/O Write | 0.67 | 0.39 | 0.77 | 0.70 | 0.57 | *0.84* | 0.63 | 0.55 | 0.77 |
| Average | 0.54 | 0.53 | 0.77 | 0.57 | 0.62 | ***0.83*** | 0.51 | 0.57 | 0.76 |
| | CNN with tf-idf | | | CNN with PCA | | | CNN with code embedding | | |
| | Pre. | Recall | AUC | Pre. | Recall | AUC | Pre. | Recall | AUC |
| Res. time | 0.30 | 0.28 | 0.75 | 0.37 | 0.35 | ***0.79*** | 0.33 | 0.34 | 0.75 |
| Cpu | 0.29 | 0.37 | *0.76* | 0.25 | 0.67 | 0.75 | 0.11 | 0.96 | *0.76* |
| Memory | 0.37 | 0.21 | *0.77* | 0.37 | 0.21 | *0.77* | 0.33 | 0.30 | 0.74 |
| I/O Read | 0.19 | 0.53 | *0.69* | 0.30 | 0.37 | 0.68 | 0.24 | 0.47 | *0.69* |
| I/O Write | 0.23 | 0.40 | *0.70* | 0.19 | 0.38 | 0.67 | 0.23 | 0.33 | 0.69 |
| Average | 0.28 | 0.36 | *0.73* | 0.30 | 0.40 | *0.73* | 0.25 | 0.48 | *0.73* |

**Using different representations of the code tokens significantly impact the performance of our models.** As shown in Table 5.9 and Table 5.10, for the traditional models (RF, LR, and XG), using code embeddings to represent the code tokens often achieves the best performance, while using PCA usually results in the worst performance. For example, for the *Hadoop* project, the RF model achieves an AUC of 0.85 to 0.93 using code embeddings, 0.82 to 0.93 using tf-idf, and only 0.59 to 0.76 using PCA. The reason for the poor performance of the models using PCA might be that PCA significantly reduced the information in the tokens through dimension reduction, even though we considered the principal components that account for 95% of the variance in the original variables. In contrast, for the deep neural network models (NN and CNN), using PCA to represent the code tokens may achieve better results than the other two representations. For example, for the *Cassandra* project, the CNN model combined with PCA achieves the best AUC for two out of the five performance metrics, across all different models. The reason might be that there are much more options in the deep neural network models, while using PCA could significantly reduce the number of options to be trained.

> ***Our models can effectively predict whether a* CTO *manifests an* IoPV *problem. Random forest based on code embedding shows the best performance on predicting* IoPV *for most of the performance measures.***

## RQ2. What are the most important metrics for predicting *IoPV* problems?

### Motivation

The goal of this research question is to analyze the models (of RQ1) that predict the *IoPV* to understand the factors that play important roles in determining the impact of adjusting a configuration parameter. In particular, we focus on the random forest model with code embeddings, as it shows the best performance in predicting *IoPV*. Our results can help practitioners understand the scenarios where they need to adjusting their configuration parameters in their performance tests.

### Approach

To analyze the most important metrics for predicting *IoPV*, we consider the following experiments:

**Measuring the importance of each dimension of metrics by removing the dimension from the model.** In order to study the importance of each dimension of metrics, we build a model with all the dimensions and compare it to a model with one dropped dimension at a time. That comparison consists of statistically comparing both models' AUC values. The larger the difference is for a dimension, the more important that dimension is. Then, we compare the differences between the model with all dimensions and each of the other models that drop one dimension at a time.

**Measuring the importance of each dimension of metrics by only keeping the dimension in the model.** Since metrics from different dimensions can be correlated, we also consider comparing models that are built using one dimension at a time. For example, some tokens from the code token dimension can be correlated with tokens from the configuration dimension. Therefore, we build a model using one dimension at a time, which results in four models. We compare these models based on their respective AUC values.

### Results

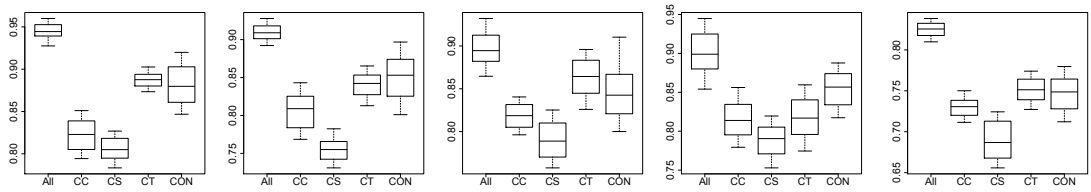
**Every dimension of metrics plays a statistically significant role in predicting *IoPV* cases.** Table 5.11 shows the results of using the Mann-Whitney U test to compare the complete RF model with the RF model that uses only one dimension of metrics or that excludes one dimension of metrics. A p-value that is smaller than 0.05 indicates a statistically significant difference. Table 5.11 shows that, when only keeping one dimension of metrics, all the resulting models show a statistically different (worse) performance. When excluding each dimension of metrics, the resulting models show a statistically different (worse) performance in most of the cases (in 16 out of the 20 combinations of the four metric dimensions and the five performance measures for *Hadoop*, and in 14 out of the 20 combinations for *Cassandra*). Our results highlight that one should consider all the four dimensions of metrics together when building a model to predict which *CTO* manifests an *IoPV*.

Table 5.11: The results (p-values) of using the Mann-Whitney U test to statistically compare the AUC of RF with the complete set of metrics vs. with a subset of metrics.

| Hadoop | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Without CC | Without CS | Without CT | Without CON | Only CC | Only CS | Only CT | Only CON |
| Res. time | <<0.0001 | 0.001 | 0.001 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 |
| Cpu | 0.002 | 0.052 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 |
| Memory | 0.016 | 0.396 | 0.019 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 | 0.001 |
| I/O Read | 0.052 | 0.154 | 0.027 | 0.002 | <<0.0001 | <<0.0001 | <<0.0001 | 0.005 |
| I/O Write | <<0.0001 | 0.001 | 0.005 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 |
| Cassandra | | | | | | | | |
| | Without CC | Without CS | Without CT | Without CON | Only CC | Only CS | Only CT | Only CON |
| Res. time | 0.093 | 0.061 | 0.052 | 0.038 | <<0.0001 | <<0.0001 | 0.019 | <<0.0001 |
| Cpu | <<0.0001 | 0.001 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 | 0.011 | <<0.0001 |
| Memory | <<0.0001 | 0.009 | 0.093 | 0.013 | <<0.0001 | <<0.0001 | 0.002 | 0.011 |
| I/O Read | 0.001 | 0.016 | <<0.0001 | 0.006 | <<0.0001 | <<0.0001 | <<0.0001 | 0.005 |
| I/O Write | 0.001 | 0.312 | <<0.0001 | 0.192 | <<0.0001 | <<0.0001 | <<0.0001 | <<0.0001 |

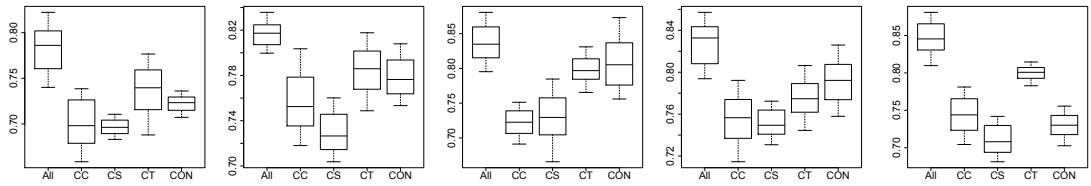
CC is Code Change, CS is Code Structure, CT is Code Token, CON is Configuration.

**The code token and configuration dimensions show the best performance among the four dimensions of metrics.** Figures 5.9 and 5.10 show the results of keeping only one dimension of metrics. For both *Hadoop* and *Cassandra*, for all the performance measures, using only the code token metrics or the configuration metrics in the model achieves a better AUC than using other single dimension of metrics, except that the configuration dimension leads to a relatively worse performance for the I/O write measure of *Cassandra*. The results indicate that the context of the change as well as the goal of configuration options expressed through their tokens are the most important predictors for *IoPV*. However, when excluding each dimension of metrics from the model (Figures 5.11 and 5.12), the differences resulting from excluding each dimension are less significant, and removing the code tokens and the configuration dimensions may not lead to the worst performance. For example, removing the code change dimension from the model for the response time measure of *Hadoop* actually lead to a worse performance than removing the code tokens dimension. This is because the different dimensions of metrics are highly correlated, thus the impact of removing one dimension of metrics may be partially mitigated by other dimensions of metrics.




(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.9: AUC of RF for *Hadoop* when only keeping one dimension of metrics.




(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.10: AUC of RF for *Cassandra* when only keeping one dimension of metrics.

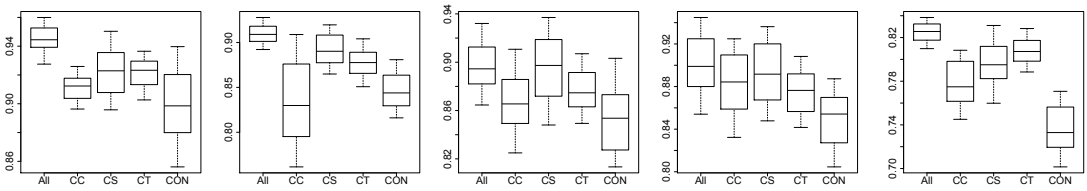
> ***Every dimension of metrics plays a statistically significant role in predicting whether a* CTO *manifests an* IoPV *problem. The most important dimensions are related to code tokens and configurations.***

## 5.6 Threats to Validity

In this section, we discuss the threats to the validity of our study.

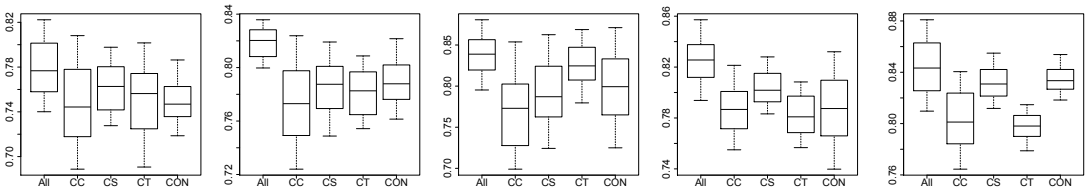
**External validity.** The first external threat to validity concerns the generalizability of our results to other software systems. Due to the expensive computing resources needed (we spent around 12,536 machine hours collecting performance data), we conducted our evaluation on two open-source software systems, i.e., *Hadoop* and *Cassandra*. Our findings may not generalize for other software systems. However, we found motivating results on the prevalence of *IoPV* and the performance of our prediction model, which can be replicated by future studies on other software systems.

**Internal validity.** An internal threat to validity concerns the performance metrics that we consider. In our approach, we collect five popular performance metrics, i.e., Response time, CPU, Memory,




(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.11: AUC of RF for *Hadoop* when removing one dimension of metrics.




(a) Res. time (b) CPU (c) Memory (d) I/O read (e) I/O write

Figure 5.12: AUC of RF for *Cassandra* when removing one dimension of metrics.

I/O read and write, while other performance metrics such as throughput can still be explored by future research.

Similarly, our prediction model does not cover all the possible dimensions of metrics. For example, we do not consider the developers' dimension. However, our model shows a good AUC performance to predict whether a *CTO* manifests an *IoPV*. Future studies can explore more dimensions of metrics to improve the performance of our models.

Finally, our evaluation considers just the traditional models (i.e., logistic regression, random forest, and XGBoost) and neural network models (i.e., general neural network and convolutional neural network). Although we do not cover all the existing models, our study covers the most popular ones that are used in software engineering. Future works are encouraged to explore more models.

**Construct validity.** There might exist environmental performance noise when we use *psutil* to capture the performance data. To minimize the noise, we capture the performance of the corresponding Linux process of the running tests. Furthermore, for each test, we repeat the execution 30 times independently. Finally, we run all of our experiments in environments with the same configurations.

## 5.7 Related Work

In this section, we discuss prior work along two dimensions: software configuration and software performance.

### 5.7.1 Software Configuration

A large body of research efforts has been conducted on software configuration, which mainly focuses on understanding configuration problems, preventing configuration errors, and debugging configuration errors. Few research efforts consider the performance aspect of software configuration.

#### Understanding Configuration Problems

Configuration makes a software system complex Sayagh et al. (2018), which leads to configuration errors that are severe, common, and hard to debug Yin et al. (2011). For instance, D. Jin et al. (2014) found that configuration options add more complexity to the development and test of highly configurable software systems. Han and Yu (2016) found that configuration options are responsible for 59% of the performance bugs. Gousios et al. (2006) observed that the configuration of the garbage collectors have an impact on the performance of server applications. Furthermore, Sayagh *et al.* (Sayagh & Adams, 2015; Sayagh et al., 2017) found that the impact of a configuration option can spread to multiple layers of the LAMP stack.

A second line of research proposed and evaluated different approaches to identify misconfigured configuration options. Dong et al. (2015, 2013) leverage the slicing technique to identify the misconfigured option for a given error message or exception. Rabkin and Katz (2011) leverage a data flow analysis technique to identify for each option, which source code lines it might impacts. Attariyan and Flinn (2010) combined dynamic control and data flow analysis to identify misconfigured options. Zhang *et al.* (S. Zhang, 2013; S. Zhang & Ernst, 2014) compared the trace of a correct execution against the trace of an incorrect execution to identify culprit options. Prior systematic literature review Sayagh et al. (2018), and the work of Tianyin and Yuanyuan (2015) and Andrzejak et al. (2018) further details about the existing configuration debugging approaches.

Our work is different from this line of research since we do not consider debugging configuration errors, but understanding and identifying performance regressions that are caused under certain configurations.

### The Performance of Configuration

Another line of research considers the identification of the optimal configurations for a software system and the debugging of performance errors that are caused by configuration options. Attariyan et al. (2012) proposed an approach based on dynamic taint analysis technique to identify the option that causes a performance error. Siegmund et al. (2015) build mathematical models that describe the impact of a configuration on software performance based on each option's value. Raghavachari et al. (2003) proposed an iterative approach to identify an optimal configuration in terms of performance. Their approach consists of selecting for a J2EE web application a first configuration, compare its performance to a second configuration until the optimal configuration. Similarly, Diao et al. (2003) proposed an approach that automatically adjusts the values of existing configuration options at run-time to optimize the CPU and memory usage objectives. Li, Chen, Hassan, et al. (2018) leveraged performance monitoring data and execution logs to dynamically optimize the values of performance-related configuration options according to varying workloads in the field. J. Guo et al. (2013) leverage non-linear regression to suggest an optimal configuration. However, collecting a large amount of data for training a model that predicts the performance of a configuration is expensive. Therefore, Sarkar et al. (2015) evaluated the progressive and projective sampling to train a model that predicts the performance of configuration. For their initial training sample, they consider data on which each option is enabled at least once. Other efforts identified the optimal configuration options in terms of performance by leveraging existing optimization approaches, i.e., iterative search Lengauer and Mössenböck (2014), multi-objective optimization Singh et al. (2016), and smart hill climbing B. Xi et al. (2004).

The goal of our work is not to identify optimal configuration options or debug an existing performance-related configuration error, but we focus on studying the inconsistent options performance through different commits. In particular, we focus on understanding whether a performance improvement or regression is consistent through all the values of an option. That is important, as

one can improve the performance of his software system or release new changes that do not impact the performance under one configuration when other configurations hide a performance regression.

Furthermore, prior work on this line of research compares the absolute performance between two values for the same option, while this can be subjective as discussed earlier. One option's value can naturally consume performance as it enables the execution of some extra features. However, the execution of the software system under the same option's value can be improved compared to the same option and value prior to that commit. In addition, a better performing option's value can show a regression compared to the prior commit as well.

### 5.7.2 Software Performance

Performance is an important aspect of software quality. Extensive prior research has been conducted to study software performance. In this subsection, we summarize the empirical studies on understanding software performance and the studies on performance regression detection.

#### Empirical Studies on Software Performance

Several empirical studies have been conducted on the performance of software systems (Han et al., 2018; Huang et al., 2014; G. Jin et al., 2012; Leitner & Bezemer, 2017; Zaman et al., 2012, 2011). For instance, G. Jin et al. (2012) studied 109 real-world performance issues that are reported from five open-source software systems and proposed an approach to detect performance issues. Zaman et al. (2012, 2011) conducted qualitative and quantitative studies on performance issues. They found that developers and users face problems in reproducing performance bugs as they spend a lot of time on discussing performance bugs compared to other kinds of bugs (e.g., functional bug). Huang et al. (2014) proposed an approach to improve the efficiency of performance regression testing by leveraging a static analysis technique to estimate the risk of a given commit in introducing a performance regression. Han et al. (2018) studied 300 bug reports from three large open-source projects. The authors found that most of the performance bugs occur for a specific combinations of data input and configurations. They also proposed a framework named *PerfLearning* to extract such data input and configurations from bug reports to generate test frames. Leitner and Bezemer (2017) investigated the state-of-the-practices that are related to performance tests. The authors found that

performance tests form only a small portion of the test suite.

The vast amount of research on software performance signifies its importance and motivate our work. Different from prior research, we evaluate software performance at the commit level and study performance regressions that are manifested under a subset of the possible configurations. In addition, our work is different from this line of research as we consider how to avoid performance regressions that are related to some configurations while being hidden by other configurations, instead of understanding the existing performance-related issues.

**Performance regression detection**

Extensive prior research has proposed automated techniques to detect performance regressions. Such detection techniques can be divided into two categories: measurement-based and model-based detection.

Measurement-based approaches compare performance metrics (e.g., CPU usage) between two consecutive versions to detect performance regressions. For example, Nguyen *et al*. (T. H. Nguyen et al., 2012; T. H. D. Nguyen et al., 2011, 2014) leveraged control charts to identify performance regressions. A control chart has an upper control limit and a lower control limit. A performance regression is detected when a performance metric is above the upper limit or below the lower limit. Foo et al. (2010) proposed an approach that compares a test's performance metrics to historical performance metrics.

Model-based approach builds a machine learning model with a set of performance metrics to detect performance regressions. Cohen et al. (2005) showed an implication that it is ineffective and not enough to index and identify performance problems with simple records of raw system metrics. Cohen et al. used TAN (Tree-Augmented Bayesian Network) models to model the system performance states based on a small subset of metrics. Bodík et al. (2008) leveraged logistic regression model to model system users' behavior to improve Cohen *et al*.'s model. Foo et al. (2015) proposed an approach that uses ensembles of models to detect performance regressions in heterogeneous environments (e.g., different hardware and software configurations). Xiong et al. (2013) proposed a model-driven framework to diagnose application performance and identify the root cause of performance issues.

Our work complements this line of research in the sense that we consider the configuration aspect of highly configurable software systems. For instance, a code change might not show a performance regression on the default configuration, while leading to regressions on other configurations. This work sheds light on the *IoPV* problem by first quantifying the existence of inconsistent performance variations, then proposing a prediction model that identifies the commits, tests, and options that exhibit the *IoPV* problem.

## 5.8 Conclusion

Highly configurable software systems tend to have a large amount of existing options, which makes testing all the possible configurations infeasible. That can, unfortunately, hide performance regression issues, which can go to the production as unseen. Furthermore, a developer might improve the performance of a software system, while the improvement might not be manifested when altering certain options' values. In fact, the performance improvement or regression of a software change might not be equally manifested through all the possible configuration options' values, which we refer to as the Inconsistent Options Performance Variation (*IoPV*). In this work, we observe that *IoPV* is a common problem, which is difficult to manually identify without running exhaustive tests, because most of the commits do not share similar options or tests that may lead to *IoPV* and hide performance regressions. We also observed that predictive models (e.g., RF) can effectively predict the *IoPV* problems using four dimensions of metrics that are related to code changes, code structures, code tokens, and configurations. Our findings highlight the importance of considering different configurations when performing performance regression detection, and that leveraging predictive models can mitigate the difficulty of exhaustively consider all configurations of a system during such a process. We expect that our study inspires a wide spectrum of future studies on configuration-aware performance regression detection.

# Chapter 6

# Can we generate load tests using log-recovered workloads at varying granularities of user behavior?

Designing field-representative load tests is an essential step for the quality assurance of large-scale systems. Practitioners may capture user behaviour at different levels of granularity. A coarse-grained load test may miss detailed user behaviour, leading to a non-representative load test; while an extremely fine-grained load test would simply replay user actions step by step, leading to load tests that are costly to develop, execute and maintain. Workload recovery is at core of these load tests. Prior research often captures the workload as the frequency of user actions. However, there exists much valuable information in the context and sequences of user actions. Such richer information would ensure that the load tests that leverage such workloads are more field-representative. In this study, we study the use of different granularities of user behaviour, i.e., basic user actions, basic user actions with contextual information and user action sequences with contextual information, when recovering workloads for use in the load testing of large-scale systems. We propose three approaches that are based on the three granularities of user behaviour and evaluate our approaches on four subject systems, namely Apache James, OpenMRS, Google Borg, and an ultra-large-scale industrial system (SA) from Alibaba. Our results show that our approach that is based on user action

sequences with contextual information outperforms the other two approaches and can generate more representative load tests with similar throughput and CPU usage to the original field workload (i.e., mostly statistically insignificant or with small/trivial effect sizes). Such representative load tests are generated only based on a small number of clusters of users, leading to a low cost of conducting/-maintaining such tests. Finally, we demonstrate that our approaches can detect injected users in the original field workloads with high precision and recall. Our study demonstrates the importance of user action sequences with contextual information in the workload recovery of large-scale systems.

## 6.1 Introduction

Large-scale software systems (e.g., Amazon AWS and Googles Gmail) have brought a significant influence on the daily lives of billions of users worldwide. For example, Netflix services 150 million subscribers across the globe Fiegerman (2019). As a result, the quality of such systems is extremely important. Failures of such planet-scale systems can result in negative reputational and monetary consequences (Kit, 2010; LINDEN, 2006). Quite often failures in such systems are load and performance-related rather than due to functional bugs Weyuker and Vokolos (2000).

Hence, load tests are widely used in practice to ensure the quality of such systems under load. The goal of a load test is to ensure that a system performs well in production under a realistic field workload. Therefore, one must first recover a workload (Calzarossa, Massari, & Tessera, 2016; Elnaffar & Martin, 2002) then design a load test based on the recovered workload (Andreolini, Colajanni, & Valente, 2005; Krishnamurthy, Rolia, & Majumdar, 2006; Meira, de Almeida, Sunyé, Traon, & Valduriez, 2013; Snellman, Ashraf, & Porres, 2011; Syer et al., 2017).

The recovery of a field-representative workload is a challenging task. In particular, one must achieve a balance between the level of granularity of the workload and the cost to conduct a load test with such a workload. All too often, in practice, the recovered workloads are too coarse, i.e., over simplified workloads. For example, the SPECweb96 Benchmark defines a workload that only specifies the probability of accessing files such as "files less than 1KB account for 35% of all requests" SPEC (2003). Such coarse-grained workloads fail to capture the variance of user behaviour, leading to non-representative load tests.

On the other extreme, a workload can simply replay the exact field workload step by step. Although, such a workload replicates the exact user behaviour, conducting a load test using such a workload and maintaining such a workload are extremely costly. First of all, due to the large amount of users of these systems, replaying the exact workload requires the load tests to simulate every user with a great amount of contextual information and complexity. One would also need to develop simulation code for each specific sequence of events. In addition, since it is almost impossible to observe the exact same workload twice, one would constantly need to update such a detailed workload.

To reach a desirable level of granularity for a workload, prior work often clusters user bahaviour based on important aspects in the workload (Cohen et al., 2005; Shang et al., 2015; Syer et al., 2017). With the clusters of users, instead of maintaining millions or billions of user profiles, a workload is designed based on representative user behaviours from a considerably smaller number of clusters. For example, a recent workload clustering approach clusters users by the frequency of different user actions Syer et al. (2017). However, due to the high variability of users in ultra-large-scale software systems, we argue that solely considering the frequency of actions is too coarse; instead the sequence and the context of user actions can make workloads much more representative to the actual field. Consider the following example: one user repetitively reads small pieces of data from a file then writes each of the small pieces back to the file; while another user interactively reads and writes a large number of small pieces of data to a file. A workload should capture both users differently. However, only considering the frequency of actions (read and write) would not differentiate the workloads of these two users. Adding more detailed information about these user actions would lead to a finer granularity of workload which in turn might be too costly to recover, execute and maintain.

Therefore, in this work, we report on our experience in understanding the impact of adding different levels of details in the recovered workloads for load tests. We first replicate a prior approach that captures the frequency of basic actions Syer et al. (2017) (we refer to this approach as *Action*). Afterwards, we design an approach that enriches the basic user actions with their context values (we refer to this approach as *ActionContext*). Finally, we design an approach that augments *ActionContext* with the frequently-appearing sequences of actions (we refer to this approach as

*ActionSequence*). The three approaches use the frequency of actions, the frequency of enriched actions and the frequency of sequences of enriched actions, respectively, as signatures for each user, then group the users into clusters. Afterwards, we automatically generate load tests based on the signature of the representative (center) user in each cluster.

Our study is performed on two open-source systems: Apache James and OpenMRS, and two commercial systems: Google Borg and an ultra-large-scale software system from Alibaba (we refer to it as SA in the rest of this work). We compare our three approaches by recovering workloads based on the execution logs from the subject systems, generating load tests, and running the automatically generated load tests on the subject systems. In particular, we answer these two research questions:

**RQ1: How field-representative are our generated workloads?**

*ActionSequence* generates the most field-representative workloads. When conducting load tests using *ActionSequence*, the throughput of 10 out of 14 user actions as well as the CPU usage are either statistically insignificant to the original workload or differ with a small/trivial effect size.

**RQ2: How many clusters of users are captured by each of our recovery workload approaches?**

The number of clusters of users is not overwhelming. The most field-representative workload *ActionSequence* is based on eight to 39 clusters of users, which is only three to six clusters more than a less field-representative workload *ActionContext*. The least field-representative workload *Action* is based on as few as two clusters of users.

The rest of the work is organized as follows: Section 6.2 discusses the background and the related work to this work. Section 6.3 describes our approaches in detail. Section 6.4 presents our case study setup. Section 6.5 presents our case study results by answering our two abovementioned research questions. Section 6.6 discusses other usage scenarios for our approaches. Section 6.8 discusses Section 6.7 discusses the challenges that lessons learned from the industrial evaluation. Finally, Section 6.9 concludes the work.

## 6.2 Background and related work

Workload recovery is an essential part in the performance assurance of large-scale systems. Prior research proposes approaches for recovering workloads to assist in the design of load tests Vögele et al. (2018), validating whether load tests are field-representative as production Syer et al. (2017), optimizing system performance (Summers et al., 2016; H. Xi et al., 2011) and detecting system performance issues (T.-H. Chen, Shang, Jiang, et al., 2016; Cohen et al., 2005; Hassan et al., 2008; Shang et al., 2015; Syer et al., 2017). All the above prior work illustrates the value and importance of recovering representative workloads.

### 6.2.1 Recovering workload

Prior approaches for recovering and replaying workloads can be categorized along the granularity of the captured user actions. One may choose a coarse-granularity by recovering only the type of workload from a system, or to the other extreme, considering each individual user and replaying their individual workload one by one. One may anonymize all high-level user behaviours and only consider the physical metrics such as CPU (Cohen et al., 2005; Shang et al., 2015), I/O (Busch et al., 2015; Haghdoost et al., 2017; Seo et al., 2014; Yadwadkar et al., 2010) and other system resources Cortez et al. (2017). One may choose a finer granularity by building complex models such as Hidden Markov ModelsYadwadkar et al. (2010) to capture the details for each user. A pilot study by Cohen et al. (2005) demonstrates that grouping workloads into a smaller number of clusters outperforms having one unified workload. Intuitively, recovering a workload at a too fine or too coarse grained detail is neither desired. A too coarse-grained approach may miss the important characteristics of user behaviour, leading to a non-representative workload, while a too fine grained approach may lead to a workload that is costly to replay and maintain.

To achieve an optimal granularity of user behaviour, prior research often chooses event or action-driven approaches for workload recovery (Hassan et al., 2008; Summers et al., 2016; Syer et al., 2017; Vögele et al., 2018; H. Xi et al., 2011). However, there exists extensive research on execution log analysis that demonstrates the value of considering contextual information and sequence of actions for various software engineering tasks (Barik et al., 2016; Beschastnikh et

al., 2014, 2011; Fu et al., 2012; S. He et al., 2018; Z. M. Jiang et al., 2008b, 2009; Lin et al., 2016; Oliner et al., 2012; Shang et al., 2013). Such extensive usage of contextual information and user action sequences in log analysis motivates our approach to leverage the similar information to recover richer workloads from execution logs for generating load tests. Therefore, compared to prior research, our study uses more valuable contextual and action sequence information from execution logs to recover workload.

### 6.2.2 Software logs analysis

Software logs are widely used in software development to track system execution behaviors Barik et al. (2016). Such logs can be used to analyze system performance (T.-H. Chen, Shang, Hassan, et al., 2016; Syer et al., 2013; Yao et al., 2018). Various studies are conducted to study the use of software logs in performance analysis.

There exists a rich body of research using system execution logs to recovery performance workload. Vögele et al. (2018) propose an approach named *WESSBAS* to automatically extract workload specifications for load testing using session-based logs. Summers et al. (2016) characterize workload of a Netflix streaming video web server by analyzing HTTP request log files to optimize the performance of the servers. H. Xi et al. (2011) analyze real query logs collected from web search engines to generate a synthetic query and replay such synthetic query onto a search engine to evaluate performance.

There also exists a fair amount of studies analyzing system logs to identify performance issues. W. Xu, Huang, Fox, Patterson, and Jordan (2009) present an approach to mine console logs in order to ease the detection of system runtime problems for operators. Tan, Kavulya, Gandhi, and Narasimhan (2010) develop a state-machine views from the native logs of Hadoop system to observe the system behavior and debug performance problems. Syer et al. (2013) propose an automatic approach that combines execution logs and performance counters to diagnose memory-related issues in load tests. The authors evaluate the approach of two software systems to diagnose three types of memory-related issues, memory bloat, memory leak and memory spike. The results show that the approach can flag the corresponding log lines responsible for memory-related issue to performance analysts with high accuracy. T.-H. Chen, Shang, Hassan, et al. (2016) propose a lightweight

approach named *CacheOptimizer* to help developers decide cache configuration in web application by analyzing available web logs. S. He et al. (2018) propose a cluster-based approach to identify impactful system problems, i.e., request latency and service availability, by analyzing log sequences.

As an experience report, our focus is primarily on exploring whether research approaches work in practice. Based on our industrial experience, this was not the case. Hence we had to propose two novel approaches. In particular compared to prior research, our work uses more valuable contextual and action sequence information from execution logs to recover workloads. The next section presents our three approaches to cluster user actions, contextual information and user action sequences from execution logs for recovering a workload for load testing.

## 6.3 Our approaches to recovering a workload for load testing

In this section, we present our approaches to automatically recover workloads using system execution logs. An overview of our recovery process is shown in Figure 6.1. In total, our recovery process consists of six steps: A) extracting user actions, B) enriching user actions with context, C) identifying frequent action sequences, D) grouping similar frequent action sequences, E) grouping users into clusters and F) generating load tests. Our *Action* workloads are generated by steps A, E and F of our recovering process. Our *ActionContext* workloads are generated by steps A, B, E and F. Our *ActionSequence* workloads are generated by all the steps.

### 6.3.1 Extracting user actions

User actions are a typical source of information to recover workload Syer et al. (2017). Therefore, in this step, we extract user actions from execution logs that are generated during the execution of a software system. Execution logs record system events at runtime to enable monitoring, remote issue resolution and system understanding Z. M. Jiang, Hassan, Hamann, and Flora (2008a). Each line of an execution log contains a corresponding action and its contextual information (e.g., user names and data sizes). In this step, we first parse the execution logs by identifying the actions and their contextual information. For example, in Table 6.1, we extract four types of actions from the logs that are generated by OpenMRS. We also extract the *Timestamp*, the *User* and the *Byte* values

Table 6.1: Our running example of execution log lines with extracted user actions and context values.

| Timestamp | User | Log line | Action | Byte |
|---|---|---|---|---|
| 00:00.00 | Alice | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 7204 |
| 00:00.92 | Dan | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 7216 |
| 00:01.52 | Bob | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 1249 |
| 00:01.54 | Alice | DELETE //192.168.165.219:8080/openmrs/ws/rest/v1/person | Delete | 0 |
| 00:02.04 | Alice | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 7227 |
| 00:02.26 | Bob | DELETE //192.168.165.219:8080/openmrs/ws/rest/v1/person | Delete | 0 |
| 00:02.41 | Bob | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/addPerson | Add | 2008 |
| 00:02.58 | Dan | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/addPerson | Add | 8109 |
| 00:03.42 | Bob | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 1247 |
| 00:03.78 | Alice | DELETE //192.168.165.219:8080/openmrs/ws/rest/v1/person | Delete | 0 |
| 00:04.13 | Bob | DELETE //192.168.165.219:8080/openmrs/ws/rest/v1/person | Delete | 0 |
| 00:04.38 | Bob | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/addPerson | Add | 2010 |
| 00:05.69 | Dan | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 7213 |
| 00:06.06 | Alice | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 7231 |
| 00:07.31 | Dan | DELETE //192.168.165.219:8080/openmrs/ws/rest/v1/person | Delete | 0 |
| 00:07.41 | Dan | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 7221 |
| 00:07.81 | Alice | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/addPerson | Add | 2006 |
| 00:09.08 | Dan | DELETE //192.168.165.219:8080/openmrs/ws/rest/v1/person | Delete | 0 |
| 00:10.18 | Bob | GET //192.168.165.219:8080/openmrs/ws/rest/v1/person | Search | 1251 |
| 00:10.32 | Alice | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/addPerson | Add | 8121 |
| 00:10.52 | Dan | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/addPerson | Add | 2012 |
| 00:11.02 | Bob | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/editPerson | Edit | 868 |
| 00:11.47 | Dan | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/editPerson | Edit | 881 |
| 00:12.12 | Alice | POST //192.168.165.219:8080/openmrs/ws/rest/v1/person/editPerson | Edit | 877 |

as the contextual values of each action. After extracting user actions, the workload signature of each user can be represented by one point in an $n$-dimensional space (where $n$ is the total number of extracted actions), i.e., the $n$-dimension vector for each user records the number of occurrence of each action with each action type mapped to one dimension. Such vectors are directly fed into step E to recover our *Action* workload. Table 6.2 shows the vectors of the frequency of user actions from our running example.

Table 6.2: Frequency of actions for users in our running example for the *Action* workloads.

| | User actions | | | |
|---|---|---|---|---|
| | Search | Add | Edit | Delete |
| Alice | 3 | 2 | 1 | 2 |
| Bob | 3 | 2 | 1 | 2 |
| Dan | 3 | 2 | 1 | 2 |

Table 6.3: Frequency of enriched actions (with context) for users in our running example for the *ActionContext* workloads.

| | Enriched actions | | | | | |
|---|---|---|---|---|---|---|
| | Search1 | Search2 | Add1 | Add2 | Edit1 | Delete1 |
| Alice | 0 | 3 | 1 | 1 | 1 | 2 |
| Bob | 3 | 0 | 2 | 0 | 1 | 2 |
| Dan | 0 | 3 | 1 | 1 | 1 | 2 |

### 6.3.2 Enriching user actions with context

Each instance of a user action is associated with a context, which may contain useful information to represent workload. For example, a disk read event is often associated with the size of the read. However, a disk read event with a large size versus one with a small size of data may correspond to different user behaviours. A small disk read may correspond to a user reading a data index while a large disk read may correspond to the actual data reading. Therefore, in this step, we enrich the recovered user actions by considering their context values. In particular, we split each action into multiple ones by categorizing the context values. For example, a disk read action may become two different actions, i.e., a large_read_disk and a small_read_disk. In particular, we use Jenks Natural Breaks Jenks (1967) on the context values of each action. Jenks Natural Breaks is a one-dimension number clustering algorithm, which minimizes each class's average deviation from the class mean, while maximizing each class's deviation from the means of the other clusters. In our running example, we consider the *Byte* value of each action as its context and for the *Search* action, we split it into two actions: *Search1* (1,247 bytes to 1,251 bytes) and *Search2* (7,204 bytes to 7,231 bytes).

After this step, the workload signature of each user becomes a vector where each dimension is the number of occurrence of each enriched user action. Such vectors are directly fed into step E to recover our *ActionContext* workload. Table 6.3 shows the vectors of enriched user actions from our running example.

### 6.3.3 Identifying frequent action sequences

Users often perform multiple actions frequently together. In order to capture these actions, we identify action sequences that frequently appear during system execution.

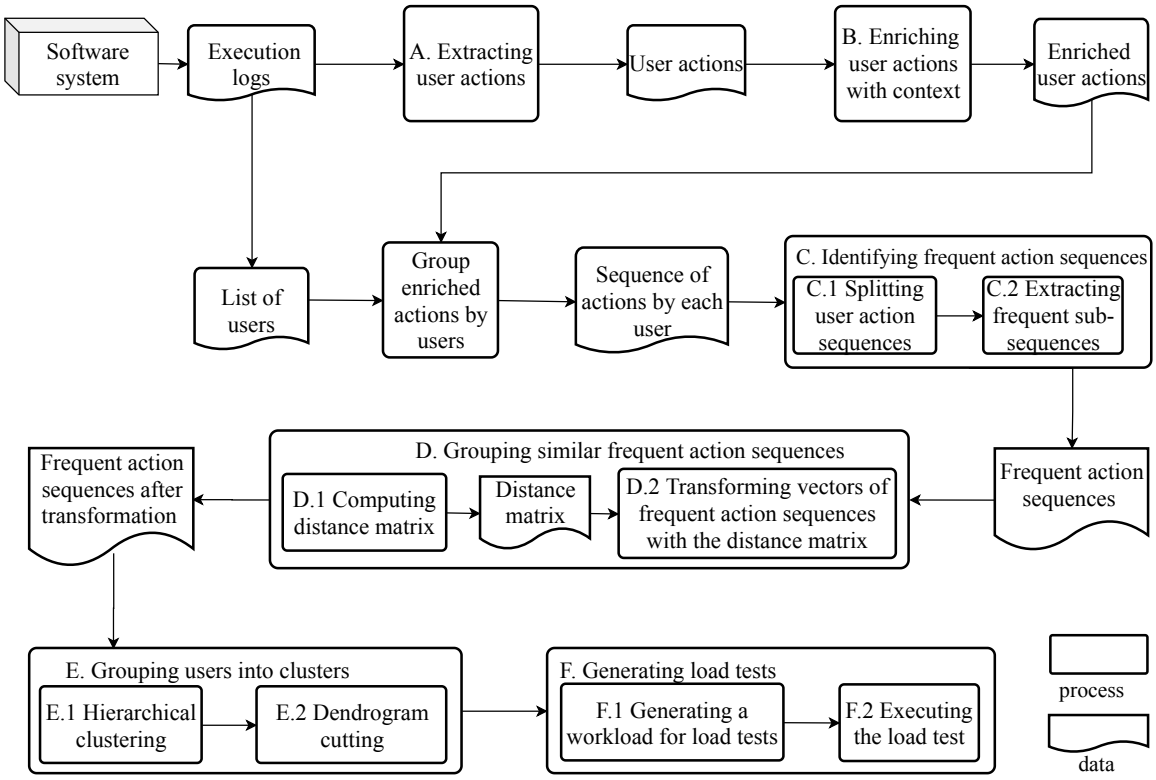



Figure 6.1: An overview of our workload recovery process.

**Splitting user action sequences**

We first group all user actions by each user and sort the actions by their timestamp, in order to generate a sequence of actions for each user. Such sequences are often very long, consisting of thousands of actions; while each user may not perform all the action at once, i.e., a user may perform two series of actions with a long period of idling time in between. Therefore, we wish to split the long user action sequences into smaller ones. One naive approach to split long user action sequences is to consider the time interval between two actions, i.e., if there is a long time interval between two actions, the sequence is split into two actions. However, some actions may actually require a long time to finish, leading to the long wait between two actions. In such cases, the naive approach may incorrectly split the sequences. Therefore, we wish to identify the actions, where the long running time is due to the idling between two septate actions. We leverage a heuristic that considers the relationship between context values and the time interval of each user action as an indicator of the type of wait.

For example, a data read action with a large data size may take longer time than a small size. Based on this intuition, we build a linear regression model using the associated context values of each action as independent variables and its time interval to the next action as the dependent variable. With the linear regression model, if an action has a time interval higher than the predicted value with a residual that is greater than 50%, our process considers that the user has idled after the action. In our running example, we split user Dan's events into three sequences: *Search2→Delete1→Search2→Delete1*, *Search2→Add2* and *Add1→Edit1*.

**Extracting frequent sub-sequences of actions**

We aim to mine frequent sub-sequences in order to represent the series of actions that a user may often perform. An example frequent sub-sequence can be a user first reading small data (to locate the data using an index) before repetitively reading large data (reading the actual data), i.e., *Small_Read→Large_Read→Large_Read→Large_Read*. We apply a suffix array structure and the Longest Common Prefix (LCP) algorithm to mine frequent sub-sequences for each sequence of user actions. Such an approach has been used in prior research to uncover usage patterns from execution logs M. Nagappan, Wu, and Vouk (2009). We re-implemented the same algorithm as prior research M. Nagappan et al. (2009). Due to the limited space, our detailed implementation can be found in our replication package.[1]. In our running example, one of the frequent sub-sequences that we extract is *Search2→Delete1*, which is identified originally from the sequence *Search2→Delete1→Search2→Delete1*.

Since some sub-sequences may be trivial (too short) or do not frequently appear (too rare), we rank the extracted sub-sequences based on the frequency of their occurrence and the frequency of events in the actual sub-sequence, as follows

$$rank = \alpha \times \#occurrence + (1 - \alpha) \times \#events \tag{3}$$

where $\#occurrence$ is the frequency of a sub-sequence's occurrence and $\#events$ is the number of events in the sub-sequence. $\alpha$ is a weight factor for the number of sub-sequence's occurrence.

[1] https://github.com/senseconcordia/ASE2019Data.

Table 6.4: Frequency of frequent action sequences in our running example for the *ActionSequence* workloads.

| | Frequent action sequences | | | | | | |
|---|---|---|---|---|---|---|---|
| | Add1→ Edit1 | Add2→ Edit1 | Search1→ Delete1→ Add1 | Search1→ Edit1 | Search2→ Add1 | Search2→ Add2 | Search2→ Delete1 |
| Alice | 0 | 1 | 0 | 0 | 1 | 0 | 2 |
| Bob | 0 | 0 | 2 | 1 | 0 | 0 | 0 |
| Dan | 1 | 0 | 0 | 0 | 0 | 1 | 2 |

We determine $\alpha$ as 0.5 since we consider the $\#occurrence$ and $\#events$ to be equally important. We use the $rank$ value to keep the top sub-sequences such that the kept sub-sequences cover more than 90% of all actions. We call these kept sub-sequences as frequent action sequences.

After extracting the frequent action sequences, the workload of each user is represented by one point in an $n$-dimensional space (where $n$ is the total number of identified frequent action sequences), i.e., a vector for each user where each dimension is the frequency of each frequent action sequence. Table 6.4 shows the result of frequent action sequences in our example.

### 6.3.4 Grouping similar frequent action sequences

The extracted frequent action sequences are not independent from each other. Intuitively, for example, two frequent action sequences *Read1→Read2→Read2→Read1* and *Read1→Read2→Read1* are similar. One user may only have *Read1→Read2→Read2→Read1* and another user may only have *Read1→Read2→Read1*. The two users may be considered completely different if we do not consider the similarities between the two frequent action sequences.

#### Computing distance matrix

For all the frequent action sequences, we calculate the distance between each pair of them using the normalized Levenshtein distance Mednis and Aurich (2012). For example, the normalized Levenshtein distance between *Read1→Read2→Read2→Read1* and *Read1→Read2→Read1* is 0.75.

Table 6.5: Result of frequent action sequences after transformation based on distance matrix for *ActionSequence* workload.

| | Frequent action sequences | | | | | | |
|---|---|---|---|---|---|---|---|
| | Add1→ Edit1 | Add2→ Edit1 | Search1→ Delete1→ Add1 | Search1→ Edit1 | Search2→ Add1 | Search2→ Add2 | Search2→ Delete1 |
| Alice | 0.5 | 1 | 1 | 0.5 | 2 | 1.5 | 2.5 |
| Bob | 0.5 | 0.5 | 2.33 | 1.67 | 0.67 | 0 | 0.67 |
| Dan | 1 | 0.5 | 0.67 | 0.5 | 1.5 | 2.0 | 2.5 |

**Transforming vectors of frequent action sequences with the distance matrix**

In order to address the similarities between frequent action sequences, we apply a vector transformation based on the distance matrix as follows:

$$vector^u = \left\langle \; s_1^u, \quad \dots \quad , s_n^u \; \right\rangle \begin{bmatrix} d_1^1, & d_1^2 & \dots & d_1^n \\ d_2^1, & d_2^2 & \dots & d_2^n \\ \vdots & & & \\ d_n^1, & d_n^2 & \dots & d_n^n \end{bmatrix}$$

where $vector^u$ is the final vector for user $u$, $s_n^u$ is the frequency of frequent action sequence $n$ for user $u$ and $d_n^1$ is the normalized Levenshtein distance between frequent action sequence 1 and frequent action sequence $n$. We perform a vector transformation in our running example and the result is shown in Table 6.5.

### 6.3.5 Grouping users into clusters

In order to reach a desirable level of granularity for a workload, in this step, we apply a clustering algorithm to group users into clusters.

**Hierarchical clustering**

We apply a hierarchical clustering to cluster users based on the Pearson distance. We choose hierarchical clustering since it is suitable for data with arbitrary shape and there is no need to determine a specific number of clusters beforehand D. Xu and Tian (2015). In addition, hierarchical clustering performs well in prior research of workload recovery (Shang et al., 2015; Syer et al., 2017). In our approach, hierarchical clustering first considers each user as an individual cluster.

Afterwards, we merge the most neighbouring clusters into a new cluster and recalculate the Pearson distance matrix between each two clusters based on average linkage.

#### Dendrogram cutting

Hierarchical clustering can be visualized using a dendrogram. Such a dendrogram must be cut with a horizontal line at a particular height to create our clusters. In practice, one may choose a desired level of granularity in a recovered workload by cutting the dendrogram at different heights, in order to retrieve a different number of clusters. In order to avoid any subjective bias, we use the Calinski-Harabasz stopping rule Caliński and Harabasz (1974) to perform our dendrogram cutting. Prior research notes that the Calinski-Harabasz stopping rule outperforms other rules when clustering workload signatures Syer et al. (2014). The Calinski-Harabasz index measures the dissimilarity of the intra-cluster variance and the similarity of inter-cluster variance. In our running example, Alice, Bob and Dan are all grouped into one cluster for the *Action* workload. Users Alice and Dan are grouped into one cluster while Bob is in another cluster for the *ActionSequence* and *ActionContext* workloads.

### 6.3.6 Generating load tests

In the final step of our process, we generate the load tests as the final outcome of our approach.

#### Generating a workload for load tests

We identify a representative user in each cluster of users to generate a workload for load testing. We apply the Partitioning Around Medoids Kaufman and Rousseeuw (2009) (PAM) algorithm to identify the representative point of each cluster. PAM is based on the $k$ representative medoids among the instances of the clustering data. PAM is an iterative process of replacing representative instances by other instances until the quality of the resulting clustering cannot be improved. The quality is measured by the medoids with the smallest average dissimilarity to all other points. In our case, we set the k as 1 since we only choose one user to represent each cluster. We then iterate each user inside the cluster to find the best representative user based on PAM.

After obtaining a representative user for each cluster, we obtain a vector $\left\langle \ s_1^u, \ \ldots \ , s_n^u \ \right\rangle$ for that user where $s_n^u$ is the number of occurrences of frequent action sequences $n$ from user $u$, for the *ActionSequence* workload. We use the frequency of occurrences of each frequent action sequences from the representative user to calculate a probability of occurrence of that frequent action sequence. Then, we generate the synthesized workload based on such probability of frequent action sequences. In our running example, the center of the cluster that consists of users Alice and Dan is user Dan. Then to generate a workload for user Alice in the load test, we replace the corresponding actions into *Search2→Delete1* with a probability of 50%, *Search2→Add1* with a probability of 25% and *Add2→Edit1* with a probability of 25%, because each of them has a frequency of two, one and one, respectively (see Table 6.4).

For the *Action* and the *ActionContext* workload, this step is similar to above but instead a probability of having an action or enriched action is calculated. For the *Action* workload, since all the users are in one cluster, the load test is generated based on the probability of having each action shown in Table 6.2, i.e., the *Search*, *Add*, *Edit* and *Delete* actions have a probability of 37.5%, 25%, 12.5% and 25%, respectively. For the *ActionContext* workload, users Alice and Dan are in one group with exactly the same distribution frequency of actions with context values. Therefore, the generated load test where the *Search2* action has a probability of 37.5%, the *Add1*, *Add2*, and *Edit1* actions each have a probability of 12.5%, and the *Delete1* action has a probability of 25%.

#### Executing load tests

Finally, our approach executes the load tests based on FIO Axboe (2019) and JMeter JMeter (1998). For software systems that cannot be directly driven by FIO and JMeter, our approach outputs simulated execution logs. Such systems can generate the load test by directly replaying the workload based on our simulated execution logs line by line.

## 6.4 Case study setup

In this section, we present the setup of our case study.

Table 6.6: Overview of our subject systems.

| Subjects | Version | SLOC (K) | # Users | # lines of logs (K) |
|---|---|---|---|---|
| Apache James | 2.3.2.1 | 37.6 | 2,000 | 134 |
| OpenMRS | 2.0.5 | 67.3 | 655 | 231 |
| Google Borg | May 2011 | N/A | 4,895 | 450 |
| SA | 2018 | N/A | $\gg$5,000 | $\gg$1,500 |

### 6.4.1 Subject systems

We choose two open-source systems including Apache James and OpenMRS, as well as two industrial systems including Google Borg, and one large software system (SA) as our subject systems. Apache James is a Java-based mail server developed by the Apache Foundation. OpenMRS is an open-source health care system to develop to support customized medical records. OpenMRS is a web system developed using the Model-View-Controller (MVC) architecture. Google Borg is a large-scale cluster management system that is used to manage internal workloads and schedule machines, jobs and tasks. SA is an ultra-large-scale cloud computing service application that is deployed to support business worldwide and used by a hundred of millions of users. Due to a Non-Disclosure Agreement (NDA), we cannot reveal additional details about the system. We do note that the SA system is one of the largest in the world in its domain with a long development history. All our subject systems cover different domains and are studied in prior research (Ahmed et al., 2016; T.-H. Chen, Shang, Hassan, et al., 2016; Gao, Jiang, Barna, & Litoiu, 2016; Verma et al., 2015). The overview of the four subject systems is shown in Table 6.6.

### 6.4.2 Data collection

In this subsection, we present how we collect execution logs in order to study the use of our different workload approaches. In particular, for the two open-source systems (Apache James and OpenMRS), we deployed the systems in our experimental environment and conducted load tests to exercise the systems. We then collected execution logs that are generated during the load tests. We also collected the CPU usage of both systems by monitoring the particular process of the system with a performance monitoring tool named *Pidstat* Godard (2019) for every five seconds. The data from the two industrial systems are from real end users. The production deployment of SA provides

the CPU usage of the system, while Google Borg does not provide the CPU usage (hence, we do not evaluate that aspect for this system). We discuss the details of our data collection for each of our subject systems. The details of our data collection can be found in our replication package.

**Apache James** We use JMeter to create load tests that exercise Apache James. We replicate a similar workload to prior research Gao et al. (2016). In particular, we simulated 2,000 email users who send, receive and read different sizes of emails with and without different sizes of attachment. Users may only read the email header or load the entire email.

We deploy Apache James on a server machine with an Intel Core i7-8700K CPU (3.70GHz), 16 GB memory on a 3TB SATA hard drive. We run JMeter on a machine with Intel Core i5-2400 CPU (3.10GHz), 8 GB memory and 320GB SATA hard drive to generate a two-hours workload.

**OpenMRS** We used the default OpenMRS demo database in our load tests. The demo database contains data for over 5K patients and 476K observations. OpenMRS contains four typical scenarios: adding, deleting, searching and editing operations. We designed the load tests that are composed of various searches of patients, concepts, encounters, and observations, and addition, deletion and editing of patient information. To simulate a more realistic workload, we added random controllers in JMeter to vary the workload.

We deployed the OpenMRS on two machines, each with Intel Core i5-2400 CPU (3.10GHz), 8 GB memory, 512GB SATA hard drive. One machine is deployed as application server and the other machine as a MySQL database server. OpenMRS provides a web-based interface and RESTFul services. We used the RESTFul API from OpenMRS and ran JMeter on another machine with the same specification to simulate users in the client side with an eight-hour workload.

**Google Borg** We used a publicly-available open dataset from the Google Borg system Wilkes (2011). The data is published by Google with a goal of workload related research. Due to the large size of Google Borg data, we only picked the first part of data to analyze, which consists of 83 minutes of data from the entire Google Borg cluster. Due to the inaccessibility of the Google Borg system, we could not run load tests directly on the system. In the data from Google Borg, there exists no information about users. However, the workload is described as *jobs*. Therefore, we

considered each job as a user when applying our approach on the Google Borg data.

**SA** We retrieved the execution logs and the corresponding CPU usage from SA that is deployed in production and is used by real users. The SA is deployed on a cluster with more than a thousand machines. Due to the NDA, we cannot mention the detailed information of the hardware environment and the usage scenarios of SA.

### 6.4.3 Preliminary analysis: clustering tendency

Before we answer the research questions for our case study, we first conduct a preliminary analysis on the clustering tendency of the data from our subject systems. If the users from our subject systems have random behaviour and do not appear to have inherent groups of similar behaviours, the data from our subject systems would be unsuitable for our study.

Therefore, we calculated Hopkins Statistic to assess the cluster tendency of our data. Hopkins Statistic is a statistical hypothesis test that can be used to accept or reject the random position hypothesis Banerjee and Dave (2004). The value of the Hopkins Statistic ranges from 0 to 1. A value of 1 means that the data has a high cluster-tendentious, a value of 0 indicates the data is uniformly distributed (not cluster-tendentious). Similar to previous research Banerjee and Dave (2004), we used 0.5 as the threshold to reject the alternative hypothesis. If the value of the Hopkins Statistic is higher than 0.5, we consider that the data has a high cluster-tendentious. We used the function *hopkins* of the *clustertend* package in R to calculate the Hopkins Statistic. We observe that our data has a high cluster-tendentious with Hopkins Statistic values that range between 0.80 to 0.99 with an average of 0.92 across all of our subject systems. Since the Hopkins Statistic values are higher than 0.5, we reject the alternative hypothesis and confirm that our data is suitable for our study.

## 6.5 Case study results

In this section, we present our case study results by answering two research questions.

### RQ1: How field-representative are our generated workloads?

**Motivation.** In order to illustrate a practical impact, we wish to first examine whether the generated workloads can lead to similar system behaviour and performance as the original system workload. If the system behaviour and performance that are produced by the generated workloads are drastically different from the original workload, such automatically generated workloads are not field-representative and hence would not be useful in practice.

**Approach.** We use all three workload approaches (*Action*, *ActionContext* and *ActionSequence*) to generate load tests based on the execution logs of the subject systems. The execution logs from Google Borg do not contain any context values, hence we only use the *Action* and *ActionSequence* approaches to generate load tests for Google Borg. When running the generated load tests, we monitor the behaviour and the performance of the systems. For the behaviour of the system, we measure the throughput of each type of action for every minute during the execution of the load tests. For the system performance, we measure the CPU usage of the particular process of the system. In the load tests, we use a performance monitoring tool named *Pidstat* Godard (2019) to collect the physical performance for every five seconds. For our subjects Apache James, OpenMRS and SA, we are able to run the generated load tests directly on the system to measure system behaviour and performance. However, for the subject Google Borg, we cannot directly run the load tests since we do not have access to the system. Therefore, we cannot measure the system performance. Since we can generate simulated execution logs for load tests, we use the simulated execution logs to compute the throughput of each type of user action.

We perform statistical analysis to examine the existence of significant differences between the generated workloads and the original workload, in terms of the throughput of each action and the CPU usage. We use the Mann-Whitney U test Nachar et al. (2008) to determine if there exists a statistically significant difference (i.e., p-value $< 0.05$). We choose the Mann-Whitney U test because it does not enforce any assumptions on the distribution of the data. 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). Hence, we use Cliff's delta to quantify the effect size Becker (2000). The smaller the effect sizes, the more similar the workload is to the original

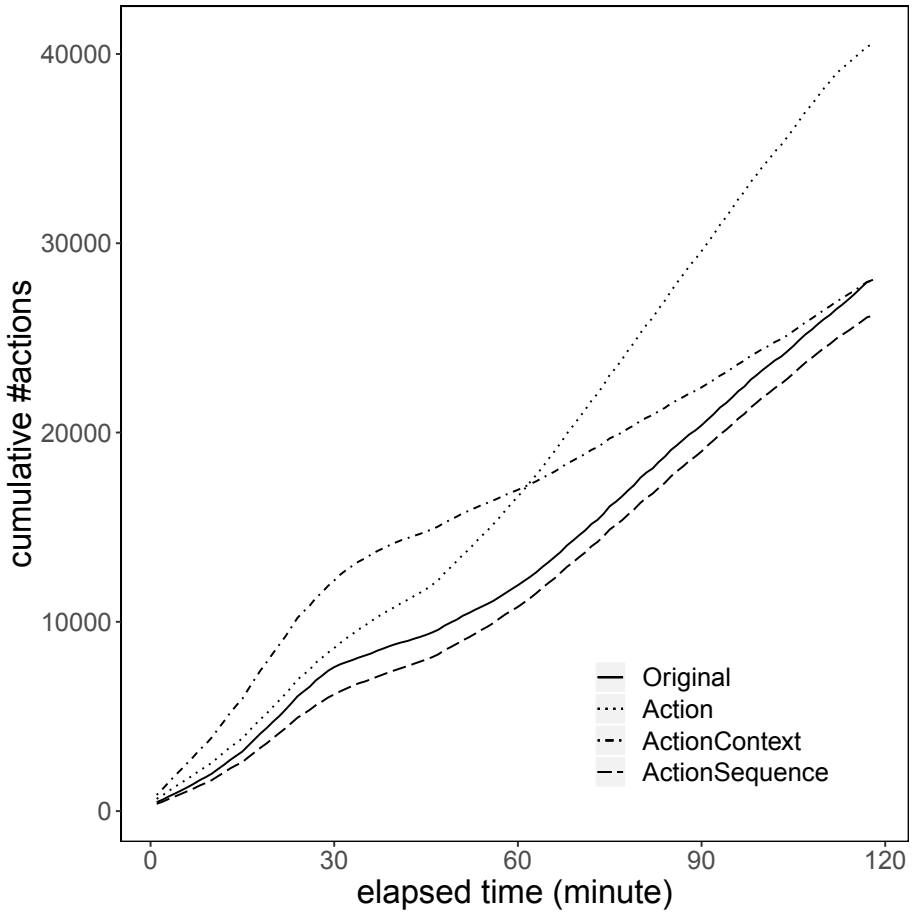



Figure 6.2: Cumulative density plot of the number of user actions from the original workload and the generated load tests for the *Receive* action in Apache James.

workload. Since statistical tests do not consider the trend of the actions, we visualize the differences between the number of each type of actions during execution from the load tests and the original workload using cumulative density graphs.

**Results. The load tests from our recovered workloads have similar system behaviour to the original workload.** The results of the throughput of actions between the original workload and the generated load tests are shown in Table 6.7. In 11 out of 14 user actions in all the subject systems, at least one workload is field-representative (bold in Table 6.7). Even the most coarse-grained workload *Action* has a similar system behaviour in five out of 14 user actions; while the most fine-grained workload *ActionSequence* has a similar system behaviour in 10 out of 14 user actions.

Table 6.7: Comparing the throughput between the original workload and the generated workloads.

| OpenMRS | | | | | | |
|---|---|---|---|---|---|---|
| Action | Add | | | Delete | | |
| | Throughput (per minute) | Comparing with original | | Throughput (per minute) | Comparing with original | |
| | | p-value | effect size | | p-value | effect size |
| Original | 143 | N/A | N/A | 91 | N/A | N/A |
| *Action* | 173 | $\ll$0.0001 | 0.52 (large) | **87** | $\ll$**0.0001** | **0.14 (small)** |
| *ActionContext* | 105 | $\ll$0.0001 | 0.96 (large) | 69 | $\ll$0.0001 | 0.83 (large) |
| *ActionSequence* | **155** | $\ll$**0.0001** | **0.23 (small)** | 83 | $\ll$**0.0001** | **0.26 (small)** |
| Action | Edit | | | Search | | |
| | Throughput (per second) | Comparing with original | | Throughput (per second) | Comparing with original | |
| | | p-value | effect size | | p-value | effect size |
| Original | 92 | N/A | N/A | 131 | N/A | N/A |
| *Action* | **88** | $\ll$**0.0001** | **0.15 (small)** | 110 | $\ll$0.0001 | 0.48 (large) |
| *ActionContext* | 119 | $\ll$0.0001 | 0.87 (large) | 165 | $\ll$0.0001 | 0.88 (large) |
| *ActionSequence* | 71 | $\ll$0.0001 | 0.45 (medium) | 149 | $\ll$0.0001 | 0.43 (medium) |

| Apache James | | | | | | |
|---|---|---|---|---|---|---|
| Action | Send | | | Receive | | |
| | Throughput (per minute) | Comparing with original | | Throughput (per minute) | Comparing with original | |
| | | p-value | effect size | | p-value | effect size |
| Original | 339 | N/A | N/A | 127 | N/A | N/A |
| *Action* | 253 | $\ll$0.0001 | 0.39 (medium) | 213 | $\ll$0.0001 | 0.68 (large) |
| *ActionContext* | **318** | $\ll$**0.0001** | **0.30 (small)** | 149 | $\ll$0.0001 | 0.44 (medium) |
| *ActionSequence* | **338** | **0.002** | **0.18 (small)** | **129** | $\ll$**0.0001** | **0.13 (small)** |

| SA | | | | | | |
|---|---|---|---|---|---|---|
| Action | Action A | | | Action B | | |
| | Throughput (per minute) | Comparing with original | | Throughput (per minute) | Comparing with original | |
| | | p-value | effect size | | p-value | effect size |
| Original | — | N/A | N/A | — | N/A | N/A |
| *Action* | — | 0.58 | 0.89 (large) | — | $\ll$0.0001 | 0.79(large) |
| *ActionContext* | — | **0.58** | **0.06 (trivial)** | — | $\ll$0.0001 | 0.99(large) |
| *ActionSequence* | — | **0.89** | **0.01 (trivial)** | — | **0.49** | **0.17(small)** |

| Google Borg | | | | | | |
|---|---|---|---|---|---|---|
| Action | Submit | | | Schedule | | |
| | Throughput (per minute) | Comparing with original | | Throughput (per minute) | Comparing with original | |
| | | p-value | effect size | | p-value | effect size |
| Original | 4,022 | N/A | N/A | 3,840 | N/A | N/A |
| *Action* | **3,985** | **0.82** | **0.02 (trivial)** | **3,971** | **0.84** | **0.02 (trivial)** |
| *ActionSequence* | **4,192** | **0.51** | **0.06 (trivial)** | **3,849** | **0.98** | **0.003 (trivial)** |
| Action | Fail | | | Finish | | |
| | Throughput (per minute) | Comparing with original | | Throughput (per minute) | Comparing with original | |
| | | p-value | effect size | | p-value | effect size |
| Original | 613 | N/A | N/A | 2,236 | N/A | N/A |
| *Action* | **647** | **0.006** | **0.25 (small)** | 2,679 | 0.007 | 0.42 (medium) |
| *ActionSequence* | **608** | **0.68** | **0.04 (trivial)** | 1,793 | 0.007 | 0.45 (medium) |
| Action | Evict | | | Kill | | |
| | Throughput (per minute) | Comparing with original | | Throughput (per minute) | Comparing with original | |
| | | p-value | effect size | | p-value | effect size |
| Original | 933 | N/A | N/A | 217 | N/A | N/A |
| *Action* | 428 | 0.06 | 0.44(medium) | 151 | $\ll$0.0001 | 0.34 (medium) |
| *ActionSequence* | 1,258 | 0.05 | 0.31(medium) | **162** | **0.37** | **0.08 (trivial)** |

Note: Bold font indicates that the load tests from the corresponding approaches are representative to the original workload (p-value $<$0.05, or effect sizes trivial or small).

Table 6.8: Comparing CPU usage between original workload and the load tests generated by our workload approaches.

| | Comparing with original | | | | | |
|---|---|---|---|---|---|---|
| | *Action* | | *ActionContext* | | *ActionSequence* | |
| Subjects | p-value | effect size | p-value | effect size | p-value | effect size |
| Apache James | $\ll$0.0001 | 0.63 (large) | $\ll$0.0001 | 0.98 (large) | $\ll$**0.0001** | **0.18 (small)** |
| OpenMRS | $\ll$0.0001 | 0.78 (large) | $\ll$0.0001 | 0.61 (large) | $\ll$**0.0001** | **0.26 (small)** |
| SA | $\ll$ 0.0001 | 0.55 (large) | $\ll$0.0001 | 0.47 (medium) | $\ll$**0.0001** | **0.29 (small)** |

Note: The bold fond indicates the most field-representative workload.

**The load tests generated by *ActionSequence* outperform the ones from *Action* and *ActionContext*.** We observe that in 10 out of 14 user actions, *ActionSequence* outperforms *Action* and *ActionContext* when comparing the throughput of the user actions. For example, for the *Send* action in Apache James, the load test from *ActionSequence* generates 337 actions per minute which is much closer to the original workload (339 actions per minute) than *Action* (253 actions per minute) and *ActionContext* (318 actions per minute). We also use cumulative density graph to evaluate the trend of each action between the original and the load tests that are generated from our different workloads. The x-axis of the graphs is the elapsed time of the load tests and the y-axis of the graph is the total number of actions. Due to space limitation, we only present the cumulative density graph for one action of Apache James (see Figure 6.2). The detailed cumulative density graph for each user action can be found in our replication package. The graphs show that the trend of the *Receive* action from the *ActionSequence* workload is much closer to the original workload than the *Action* and *ActionContext* workloads. In particular, Figure 6.2 shows that the total of the number of user actions from the *Action* workload is far from the original.

**The *ActionSequence* workloads produce system performance closer to the original workload than the *Action* and *ActionContext* workloads**. Shown in Table 6.8, the effect sizes are always trivial or small between the CPU usage during the load tests that are generated from the *ActionSequence* workloads and the original workload. On the other hand, the effect sizes of CPU usage differences are medium to large when comparing the original workload with the *Action* and *ActionContext* workloads.

### RQ2: How many clusters of users are captured by each of our recovery workload approaches?

**Motivation.** Our workloads may contain a large number of clusters of users, leading to a too-fine of a granularity for load tests. If our workloads consist of an overwhelming amount of clusters, they would not be useful for practitioners due to the large overhead of developing the infrastructure to execute such workloads, as well as executing and maintaining them.

**Approach.** We use all the workloads of each of our approaches to generate load tests for the four subject systems. We compare the number of clusters of users that are recovered by each workload. Afterwards, we manually examine each cluster of users to understand the differences between our different workload recovery approaches.

**Results. Our approaches do not generate an overwhelming number of clusters.** The numbers of generated user clusters are shown in Table 6.9. Although including richer user information in our workloads, we do not generate an overwhelming number of clusters. In particular, Google Borg and SA are both large-scale systems with a large amount of end users, while our process only generates a maximum of 25 and 13 clusters for the *ActionSequence* approach for Google Borg and SA, respectively. However, the granularity of the *Action* workloads is very coarse. In particular, for the three subject systems, i.e., Apache Jame, OpenMRS and SA, the *Action* workload only consists of two to three clusters of users. Such a small number of clusters helps explain the results from RQ1 where the *Action* workload is not able to generate field-representative load tests. On the other hand, the most field-representative workload from RQ1, i.e., the *ActionSequence* workloads, consists of only three to six more clusters than the *ActionContext* workloads. Such a small number of clusters of users make it possible to manually examine each cluster to qualitatively understand the field workload.

We manually examine each cluster of users and aim to understand the difference between the recovered clusters for the *ActionSequence* and *ActionContext* workload recovery approaches. We identify three typical scenarios that cause the differences. 1) Orders of actions. Some users have the same distribution of actions but they are ordered differently. Such differences are not captured by *ActionContext* workloads. 2) Similar sequences of actions but different distribution. User may

Table 6.9: Number of user clusters for each of our workload approaches.

| Subjects | *Action* | *ActionContext* | *ActionSequence* |
|---|---|---|---|
| Apache James | 2 | 35 | 39 |
| OpenMRS | 3 | 2 | 8 |
| Google Borg | 20 | 20 | 25 |
| SA | 2 | 10 | 13 |

have the same action sequences but the general distribution of each sequence of actions is different. *ActionContext* workloads would consider the users into different clusters while *ActionSequence* workloads group the users together. 3) Varying frequency of actions. Some users have the same distribution of action sequences although their frequencies are varying differently. *ActionContext* workloads would consider the users in the different clusters while *ActionSequence* workloads consider them the same. Such scenarios show that the *ActionSequence* considers a different levels of granularities of user behaviours which may explain the differences in clusters.

## 6.6 Discussion

In this section, we discuss the use of our three workload recovery approaches to detect unseen workloads and sensitivity analysis.

### 6.6.1 Detecting unseen workload

One of the challenges of designing load tests is keeping the load tests field-representative as the user workloads evolve (T. Chen et al., 2017; Cunha & e Silva, 2012). When there exist users with unseen workload, practitioners should be informed in order to act accordingly. For example, if the unseen workload is due to new user behaviours, one may wish to update the load tests. On one hand, if the recovered workloads from our approaches are too fine-grained, our approaches would report false-positively unseen workloads, leading to additional wasted costs to practitioners. On the other hand, if our recovered workloads are too coarse-grained, we may miss the reporting of unseen workloads, leading to load tests that are not field-representative. Therefore, we study the use of our workload recovery process to detect unseen workloads by injecting users with unseen workloads into our existing data. We injected four types of unseen workloads, that are typically used in prior

research (Cherkasova, Ozonat, Mi, Symons, & Smirni, 2008; Cunha & e Silva, 2012).

- *Extra actions.* We randomly pick one action that is not the most frequent action. Then we replace all occurrences of the picked action by the most frequent action.
- *Removing actions.* We randomly pick an action type and remove all occurrences of the action from a user.
- *Double context values.* For every action with a context value, we change the context value by doubling it.
- *Reordering actions.* For all the actions that are performed by a user, we randomly reorder the actions.

For every type of unseen workload, we randomly select one user and alter the data from the user to inject the unseen workload. We apply each of our approaches to recover workloads from the data with only one user injected with an unseen workload. In particular, if one user is located in one cluster without any other users, we consider that the user has an unseen workload (the user is not similar to any other one). If the user is indeed injected with an unseen workload, we consider it as a true-positive detection. Any normal user located in a one-user cluster will be considered as a false-positive detection. We repeat this process 10 times, with every time injecting a random user with an unseen workload, for every type of unseen workload. In total, for our four subject systems, we have 160 sets of data, each of them having one user injected with an unseen workload. We generate 160 workloads to detect those users. We define precision as the number of the true-positive detection divided by the total number of users that are in a one-user cluster; recall as the number of the true-positive detection divided by the total number of users with injected unseen workloads.

***ActionContext* and *ActionSequence* workloads can accurately detect the injected users with an unseen workload.** The results of detecting injected users with an unseen workload is shown in Table 6.10. The results show that *ActionContext* workloads have a precision between 0.75 to 1 with an average of 0.86 and a recall between 0.2 to 0.9 with an average of 0.58. The *ActionSequence* workloads achieve a precision between 0.75 to 1 with an average of 0.87 and a recall between 0.3 to 1 with an average of 0.79. However, the *Action* workloads have both low average precision (0.47) and recall (0.29). The precision and recall of the *ActionSequence* workloads are generally

Table 6.10: Results of precision and recall in detecting injected unseen workloads.

| Apache James | | | | | | |
|---|---|---|---|---|---|---|
| Unseen workload type | *Action* | | *ActionContext* | | *ActionSequence* | |
| | precision | recall | precision | recall | precision | recall |
| Extra actions | **1.00** | 0.30 | 0.80 | **0.40** | 0.80 | **0.40** |
| Removing actions | **1.00** | **0.40** | 0.75 | 0.30 | 0.75 | 0.30 |
| Double context values | 0 | 0 | **0.90** | **0.90** | **0.90** | **0.90** |
| Reordering actions | 0 | 0 | 0.75 | 0.30 | **0.89** | **0.80** |
| Average | 0.50 | 0.18 | 0.80 | 0.48 | **0.84** | **0.60** |

| OpenMRS | | | | | | |
|---|---|---|---|---|---|---|
| Unseen workload type | *Action* | | *ActionContext* | | *ActionSequence* | |
| | precision | recall | precision | recall | precision | recall |
| Extra actions | 0.86 | 0.6 | 0.86 | 0.60 | **0.89** | **0.80** |
| Removing actions | **0.88** | 0.70 | 0.73 | 0.80 | 0.69 | **0.90** |
| Double context values | 0 | 0 | **0.75** | **0.60** | **0.75** | **0.60** |
| Reordering actions | 0 | 0 | **1.00** | 0.30 | 0.91 | **1.00** |
| Average | 0.44 | 0.33 | **0.84** | 0.58 | 0.81 | **0.83** |

| Google Borg | | | | |
|---|---|---|---|---|
| Unseen workload type | *Action* | | *ActionSequence* | |
| | precision | recall | precision | recall |
| Extra actions | **0.90** | **0.90** | **0.90** | **0.90** |
| Removing actions | 0.83 | 0.50 | **0.89** | **0.80** |
| Double context values | **1.00** | 0.10 | 0.90 | **0.90** |
| Reordering actions | **0.91** | 0.50 | 0.90 | **0.87** |
| Average | **0.91** | 0.50 | 0.90 | **0.87** |

| SA | | | | | | |
|---|---|---|---|---|---|---|
| Unseen workload type | *Action* | | *ActionContext* | | *ActionSequence* | |
| | precision | recall | precision | recall | precision | recall |
| Extra actions | **1.00** | 0.80 | 0.89 | 0.80 | 0.90 | **0.90** |
| Removing actions | 0.88 | 0.7 | **0.89** | **0.80** | **0.89** | **0.80** |
| Double context values | 0 | 0 | **1.00** | **0.90** | **1.00** | **0.90** |
| Reordering actions | 0 | 0 | **1.00** | 0.20 | 0.90 | **0.90** |
| Average | 0.47 | 0.38 | **0.95** | 0.68 | 0.92 | **0.88** |

| | *Action* | | *ActionContext* | | *ActionSequence* | |
|---|---|---|---|---|---|---|
| | precision | recall | precision | recall | precision | recall |
| All average | 0.47 | 0.29 | 0.86 | 0.58 | **0.87** | **0.79** |

Note: Values in bold font indicate the best performing approach for each setting.

consistently high across all subject systems for all types of injected unseen workloads. The only exception is Apache Jame with a lower recall in detecting *Extra actions* and *Removing actions* workloads. We find that the Apache James mail server has a small number of action types (cf. Section 6.4), while adding extra actions and removing actions may still generate a user with a high similarity to a previously recovered cluster of users.

***ActionSequence* workloads have a similar precision but much higher recall than *ActionContext* workloads.** Table 6.10 shows that on average *ActionSequence* achieves a similar precision (0.87) as *ActionContext* (0.86) while the recall of *ActionSequence* (0.79) is much higher than that of *ActionContext* (0.56). *ActionSequence* considers finer-grained information than *ActionContext*, hence intuitively, providing a higher ability to uncover more unseen workloads. In particular, in all the cases, *ActionSequence* has a higher or similar recall as *ActionContext*. On the other hand, even though in some cases when *ActionContext* has a higher precision, the difference is rather small.

### 6.6.2 Sensitivity analysis

Our workload recovery process leverages several techniques, such as hierarchical clustering, which may be replaced by other similar techniques. Our process also leverages threshold values. For example, the residual value for the linear regression prediction that is used to split user action sequences is set to 0.5 in our process. The $\alpha$ value to rank frequent action sequences is also set to 0.5. In order to better understand the sensitivity of our workloads to these thresholds, we individually increased each threshold value to 0.75 and decreased the threshold value to 0.25. We also change the clustering algorithm to Mean shift Comaniciu and Meer (2002). We choose Mean shift, since similar to hierarchical clustering, Mean shift does not require us to pre-specify the number of clusters. We re-generated the *ActionSequence* workload and calculated the throughput of each action for each subject system. We used the *ActionSequence* workload in this analysis because *ActionSequence* is the best performing workload shown in our evaluation results (c.f. Section 6.5) and the *ActionSequence* covers all the steps of all our three workload recovery approaches.

We compared the throughput of each action with the default threshold and the clustering algorithm using the Mann-Whitney U test and Cliff's delta. We observed that our approaches are insensitive to both the residual and $\alpha$ thresholds. In addition, the effect size between the hierarchical

Table 6.11: Comparing the results of choosing different threshold values and clustering algorithm for Apache James.

| Send action | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Changed | residual=0.25 | | residual=0.75 | | $\alpha$=0.25 | | $\alpha$=0.75 | | with Mean shift | |
| Default | p-value | effect size | p-value | effect size | p-value | effect size | p-value | effect size | p-value | effect size |
| residual=0.5, $\alpha$=0.5, hierarchical clustering | 0.9 | 0.01 (trivial) | 0.91 | 0.01 (trivial) | 0.001 | 0.19 (small) | 0.99 | 0.0001 (trivial) | 0.0001 | 0.22 (small) |
| Receive action | | | | | | | | | | |
| Changed | residual=0.25 | | residual=0.75 | | $\alpha$=0.25 | | $\alpha$=0.75 | | with Mean shift | |
| Default | p-value | effect size | p-value | effect size | p-value | effect size | p-value | effect size | p-value | effect size |
| residual=0.5, $\alpha$=0.5, hierarchical clustering | 0.9 | 0.01 (trivial) | 0.9 | 0.01 (trivial) | 0.005 | 0.15 (small) | 0.99 | 0.0001 (trivial) | 0.25 | 0.07 (trivial) |

clustering and Mean shift is trivial or small. Table 6.11 only shows the results of such a comparison for Apache James which is the most peculiar workload in our subject systems. Other comparison results are in our replication package.

## 6.7 Challenges and lessons learned from the industrial evaluation of our approaches.

In this section, we discuss the learned lessons and faced challenges during the implementation and evaluation of our approaches in industry. In particular, I was embedded on site with the industrial team for over half a year to enable a faster feedback loop from practitioner – ensure the smooth adoption of our approaches in a large-scale complex industrial setting. Our documented challenges and lessons can assist researchers and practitioners who would like to integrate their research in complex industrial settings.

### 6.7.1 Domain knowledge is crucial for the successful transfer of research to practice.

Our approaches depend on the availability and quality of the important knowledge that resides in system execution logs. Due to the large scale and complexity of the logs in System A, we often faced the challenge that we may not fully understand the information that is communicated in the logs, making it challenging for us to determine the important contextual information to include and analyze by our approaches.

*How to address:* At a first attempt, we naïvely included all log information for our analysis. We ended up observing that such an attempt introduced noise which negatively impacted our results.

Hence, the first author flew down to spend six months on site at Alibaba, where he held several in-person scrum meetings with developers and operators of System A, in order to better understand the information that is communicated in System As logs. These meetings helped the academic team and the industrial team get a better understanding of the problem at hand as well as the strengths and limitations of the research solutions. Being on site helped the academic team create a strong relation with the industrial team as well. Such a relation enabled a much faster and more open feedback loop.

*Lessons learned:* Good domain knowledge is crucial in log analysis and workload recovery. Blindly applying log analysis techniques on large-scale complex logs may not achieve the expected goal. One should work closely with practitioners to leverage their valuable knowledge about their logs.

### 6.7.2 Team support is crucial for the successful transfer of research to practice.

#### Customization of tooling

We started our research working on open-source systems. Such open-source systems commonly use standard load driver tools such as JMeter. However, industrial systems are commonly tested using various in-house custom tools. Such tools hinder us from demonstrating and evaluating our work. It is often costly, time-consuming and sometimes impossible to design and implement a specific load replay tool for each system.

*How to address:* Working closely with the practitioners of Alibaba, we gained a deeper understanding of their in-house load replay tools. We observed that many of them support the direct reading of logs and the replaying of the exact workload based on such logs. Therefore, we provided an option in our toolset for the direct driving of a load test using widely used tools (like JMeter), or the transformation of our generated load test into logs, which can be fed to the customized load replay tools from Alibaba.

*Lessons learned:* There exists a strong need for future research in load replay using more flexible frameworks in order to avoid practitioners having to implement customized toolsets.

**Addition of needed log lines when not all information is available**

Not all the needed information was available in the logs when we started our research on Alibabas system. In particular, the log data was not perfectly designed for our approach and important information was missing.

*How to address:* Working closely with the practitioners, we requested the addition of new logging probes. In order to demonstrate the values of our requests, we conducted several additional analyses.

*Lessons learned:* The technical support and quick turn-around from the industrial team were extremely crucial in addressing this challenge. However, even with all the logging probes in place, we had to wait till the builds with such probes were deployed long enough in the field for us to have sufficient data for our approaches.

**Setting up a realistic industrial environment**

The context of the load testing environment has an important influence on the performed load tests. For example, there should be a large amount of realistic data in a database before one can test a database-driven application. Using open source systems, it is relatively easy to create a load testing environment that is similar to widely adopted benchmarks. However, setting up such an environment in an ultra-large-scale industrial environment is extremely challenging.

*How to address:* To make our automatically generated load tests run successfully, we had the luxury of being strongly supported by the infrastructure team of Alibaba. In particular, we were provided with a testing environment that is a replica of the field environment from which we collected the analyzed logs. However, such a solution is not optimal and may not be cost-effective, especially for practitioners that do not have direct access to infrastructures that are similar to their deployed systems.

*Lessons learned:* Preparing the load testing environment is an important, challenging and yet open problem for load testing practices and research. Technical and infrastructural support is crucial in overcoming this challenge.

### 6.7.3 Coping with the large scale industrial data

Our experiments on open-source systems are conducted with a limited scale of data. When adopting our approach using the industry scale data of Alibaba, our approach did not scale well. The statistical analyses and clustering techniques often suffered from poor scalability.

*How to address:* In order to ease the adoption of our approach on industry scale data, we optimized our solution by learning some threshold values by pre-processing the logs and caching the thresholds across runs. For example, learning the best threshold values to categorize contextual values is time consuming. We can save the learned thresholds and use them directly to generate the categorized contextual values in logs since such thresholds rarely change within a specific context.

*Lessons learned:* One should not assume that a successful research tool can directly scale to industrial data. Optimization for the particular industrial setting is important.

## 6.8 Threats to validity

**External validity.** Our evaluation is conducted on data from two open source and two industrial systems. Although our subject systems cover different domains and sizes, our evaluation results may still not generalize to other systems. Given the automated nature of our process, others can study its effectiveness on their own data sets using our replication script.

**Internal validity.** Our approaches depend on the availability and quality of system execution logs. If a system records limited information about user actions in the execution logs, our approaches may not perform as expected. The evaluation of approaches uses the system CPU usage that is recorded by *Pidstat*. The quality and the frequency of recorded CPU usage can impact the internal validity of our study. Currently, our approach only categorizes numerical contextual values due to the characteristics of the logs in our subject systems. Future work can complement our approach by considering categorizing string literals. Our approach depends on various statistical analyses. Therefore, for small systems with a small amount of data, our approach may not perform well due to the nature of statistical analyses.

**Construct validity.** There exists other aspect of system bahaviour and performance. We focus on the throughput and CPU usage due to special need of SA from our industrial collaborator. Future

study may investigate the impact on other system aspects to complement our findings. Due to inaccessibility of the subject system, the workloads on Google Borg is based on simulated execution logs, instead of actually running the load tests on Google Borg. Therefore, the evaluation results may be different if we were able to run the load tests on the actual Google Borg cluster. In the evaluation of our approaches to detect users with unseen workloads, we only injected four types of unseen workloads. Similar evaluation approaches based on mutation techniques have been often used in prior research (Svajlenko & Roy, 2015; Svajlenko, Roy, & Cordy, 2013). However, these unseen workloads may not be the same as in real life. In addition, there may exist other ways to inject unseen workload to complement our results.

## 6.9 Conclusions

Workload recovery from end users is an essential task for the load testing of large-scale systems. In this work, we conduct a study on recovering workloads at different levels of granularity of user behaviours to automatically generate load tests. We design three approaches that are based on user actions, user actions with contextual information and user action sequences with contextual information, to generate load tests on two open source systems and two industrial systems. We find that our richest approach which uses user action sequences with contextual information outperforms the other two approaches. In most cases, the throughput and CPU usage from the load tests that are generated from the user action sequence based workload outperform the other two, and are statistically insignificant relative to the original workload or with small or trivial effect sizes. Such a field-representative workload is generated only using a small number of clusters of users. In addition, we find that the recovered workloads from our approaches can also be used to detect injected users with unseen workloads with a high precision and recall.

This work has the following contributions:

- To the best of our knowledge, our approach is the first large-scale study on the use of different granularity of recovered user details for load testing.
- To the best of our knowledge, our approaches are the first ones in the field to leverage user contextual information and frequent action sequences in workload recovery.

- Our approaches have been adopted in practice to assist in the testing and operation of an ultra-large-scale industrial software system.

# Chapter 7

# Summary, Contributions, and Future Work

This chapter summarizes the main ideas presented in this thesis. In addition, we propose future work related to software performance.

## 7.1 Summary

This thesis focused on working towards the performance regression detection in DevOps. First, we focus on frequent performance assurances during development. The performance regression might slip from software development to operation. Therefore, second, we focus on assisting performance assurances with operational data. To detect or predict performance regression introducing source code changes in the early stage during development, we first present an approach to identity performance regression introducing source code changes during development. Second, we propose an approach to automatically predict whether a test would manifest performance regressions in a code commit. Most of the performance regressions are caused by mistaken configuration. Therefore, we also propose an approach to predict whether a configuration option manifests performance regression. Our results and approaches are valuable for software performance engineering practitioners.

From the perspective of operations, we use operational data to analyze system performance

and detect performance regression to assist developers to improve their system performance. In particular, we propose an approach to recover field-representative workloads to generate load tests based on execution logs to help operators to perform their load tests.

## 7.2 Thesis contributions

The main contributions of this thesis are the following:

- We propose a statistically rigorous approach to identifying performance regression introducing code changes. Further research can adopt our methodology in studying performance regressions (Chapter 3).
- We find six root-causes of performance regressions that are introduced by code changes. 12.5% of the manually examined regressions can be avoided or their performance impact may be reduced (Chapter 3).
- We propose an approach that can 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 (Chapter 4).
- Our findings highlight the importance of considering different configurations when performing performance regression detection, and that leveraging predictive models can mitigate the difficulty of exhaustively consider all configurations of a system during such a process (Chapter 5).
- We introduce an approach for recovering workload using user contextual information and frequent action sequences from system execution logs. Our approach has been adopted in practice to assist in the testing and operation of an ultra-large-scale industrial software system (Chapter 6).

## 7.3 Future Work

Software performance involves many research areas like program analysis, software testing, software log analysis. There are many open challenges and opportunities still in the field of performance regression. We highlight some avenues for future work.

### 7.3.1 Performance data repository

There exist many systematical and mature repositories, such as code repository Git and issue tracking system JIRA. However, there exists no repository for performance data. And we already know that performance testing is time-consuming and costly. If we have a systematical repository for performance data, it would be valuable for both researchers and practitioners. Therefore, designing a repository to collect, store, manage performance data is an open research question. To do so, we plan to collect the existing performance data as much as possible and try to propose an efficient data structure to store the performance data. We also plan to investigate the approaches to systematically store such performance data in a repository.

### 7.3.2 More empirical studies on the impact of software activities on performance

There is a lack of research on the impact of software activities on the performance of software system. For instance, architecture reengineering, cloning codes and some types of refactoring activities might have negative impacts on performance. To fill this gap, future work can conduct more empirical studies on the impact of software activities on performance.

### 7.3.3 Domain-specific language for performance data analysis

Performance data are often analyzed using script languages, such as python and shell. However, such analysis scripts are difficult to scale and maintain. Since performance testing often produces tons of unstructured data. Reusing performance testing result analysis techniques and scaling such techniques in large-scale performance data is challenging. Therefore, future research should explore the design of domain-specific manipulation languages for performance data to ease the reuse and mitigation of performance data analysis techniques.

### 7.3.4 Reduce the length of performance testing

In practice, performance testing often lasts hours or days, or even weeks. Such length of executing performance test requires a large of system resources. Prior research has reported that performance tests often repeats the same workload multiple time and produce repeated performance data. Such repeated performance data contain invaluable information. Therefore, future work can explore the techniques to not only reduce the length of performance testing but also keep the same system behavior.

### 7.3.5 The usage of unit test to detect performance regression in the load test

Load testing a system is a required testing procedure. However, first, load test is too complex and time-consuming, i.e., executing a load test and analyzing the results of a load test. Second, load tests cannot represent the field workload due to the frequent workload fluctuation. Finally, replaying field workload is difficult due to the need for a dedicated testing environment and tooling support. Therefore, if we can use the field data generated in operations to predict performance regression without the need to execute load tests, large amounts of resources can be reduced. We plan to design an approach to mining the relationship between unit tests' logs and load tests' logs, and study the use of readily available execution logs of unit tests to predict performance regression in load tests.

In this appendix, we present the detailed prediction results from RQ1 of Perf-JIT (Chapter 4) in Table .1. The detailed results including precision, recall, AUC and their corresponding improvement over the *Perphecy* models are shown in Table .1. We also present the detailed results of the average ranks of the important metrics from RQ5 of Perf-JIT (Chapter 4) in Table .2.

Table .1: Detailed results of predicting performance regression with different performance counters. Improvement are calculated by comparing with a random classifier. Bold values highlight the best predictors.

| Hadoop | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Random Forest | | | | | | | Perphecy | | | | LR | | | | SVM | | | | XG-50 | | | | XG-100 | | | | XG-500 | | | |
| | Pre. | Improv. | Recall | Improv. | F-1 | Improv. | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC |
| Res. time | 0.30 | 275% | 0.78 | -17% | 0.43 | 207% | **0.87** | 0.08 | 0.94 | 0.14 | 0.08 | 0.23 | 0.54 | 0.32 | 0.69 | 0.20 | 0.82 | 0.32 | 0.84 | 0.34 | 0.64 | 0.44 | 0.83 | 0.35 | 0.64 | 0.45 | 0.83 | 0.30 | 0.68 | 0.41 | 0.83 |
| Cpu | 0.43 | 378% | 0.77 | -14% | 0.55 | 244% | **0.87** | 0.09 | 0.90 | 0.16 | 0.09 | 0.15 | 0.54 | 0.23 | 0.63 | 0.18 | 0.51 | 0.26 | 0.72 | 0.46 | 0.58 | 0.51 | 0.80 | 0.50 | 0.57 | 0.53 | 0.80 | 0.46 | 0.57 | 0.51 | 0.79 |
| Memory | 0.20 | 233% | 0.84 | -5% | 0.32 | 191% | **0.89** | 0.06 | 0.88 | 0.11 | 0.06 | 0.27 | 0.42 | 0.33 | 0.58 | 0.33 | 0.73 | 0.45 | 0.82 | 0.32 | 0.63 | 0.43 | 0.83 | 0.34 | 0.62 | 0.43 | 0.82 | 0.39 | 0.60 | 0.48 | 0.80 |
| I/O Read | 0.43 | 514% | 0.67 | -22% | 0.52 | 300% | **0.82** | 0.07 | 0.86 | 0.13 | 0.07 | 0.29 | 0.39 | 0.33 | 0.62 | 0.13 | 0.99 | 0.23 | 0.81 | **0.7** | 0.49 | 0.58 | 0.74 | 0.70 | 0.49 | 0.58 | 0.75 | 0.69 | 0.49 | 0.57 | 0.76 |
| I/O Write | 0.25 | 213% | 0.71 | -25% | 0.36 | 157% | **0.81** | 0.08 | 0.95 | 0.14 | 0.08 | 0.19 | 0.30 | 0.24 | 0.60 | 0.10 | 0.96 | 0.18 | 0.61 | 0.16 | 0.58 | 0.25 | 0.69 | 0.15 | 0.57 | 0.24 | 0.68 | 0.15 | 0.58 | 0.24 | 0.68 |
| Average | 0.32 | 324% | 0.75 | -17% | 0.44 | 221% | 0.85 | 0.08 | 0.91 | 0.14 | 0.08 | 0.23 | 0.44 | 0.29 | 0.62 | 0.19 | 0.80 | 0.29 | 0.76 | 0.40 | 0.58 | 0.44 | 0.78 | 0.41 | 0.58 | 0.45 | 0.78 | 0.40 | 0.58 | 0.44 | 0.77 |
| Cassandra | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | Random Forest | | | | | | | Perphecy | | | | LR | | | | SVM | | | | XG-50 | | | | XG-100 | | | | XG-500 | | | |
| | Pre. | Improv. | Recall | Improv. | F-1 | Improv. | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC |
| Res. time | 0.50 | 1567% | 0.63 | -20% | 0.56 | 1020% | 0.78 | 0.03 | 0.79 | 0.05 | 0.54 | 0.04 | 0.79 | 0.08 | 0.66 | 0.04 | 0.84 | 0.07 | 0.65 | 0.28 | 0.58 | 0.38 | 0.78 | 0.16 | 0.68 | 0.25 | **0.79** | 0.16 | 0.68 | 0.26 | 0.78 |
| Cpu | 0.27 | 800% | 0.86 | 37% | 0.41 | 486% | **0.90** | 0.03 | 0.63 | 0.07 | 0.60 | 0.06 | 0.69 | 0.12 | 0.60 | 0.15 | 0.46 | 0.23 | 0.72 | 0.12 | 0.83 | 0.21 | 0.80 | 0.14 | 0.83 | 0.24 | 0.81 | 0.12 | 0.83 | 0.21 | 0.82 |
| Memory | 0.27 | 1250% | 0.84 | 50% | 0.40 | 900% | **0.95** | 0.02 | 0.56 | 0.04 | 0.62 | 0.08 | 0.76 | 0.14 | 0.70 | 0.13 | 0.56 | 0.21 | 0.68 | 0.19 | 0.84 | 0.30 | 0.91 | 0.14 | 0.88 | 0.25 | 0.90 | 0.20 | 0.72 | 0.31 | 0.90 |
| I/O Read | 0.31 | 1450% | 0.68 | 79% | 0.42 | 950% | **0.86** | 0.02 | 0.38 | 0.04 | 0.73 | 0.11 | 0.59 | 0.18 | 0.62 | 0.08 | 0.76 | 0.13 | 0.64 | 0.18 | 0.70 | 0.28 | 0.82 | 0.17 | 0.70 | 0.27 | 0.81 | 0.29 | 0.57 | 0.39 | 0.79 |
| I/O Write | 0.15 | 650% | 0.76 | 17% | 0.25 | 733% | **0.87** | 0.02 | 0.65 | 0.03 | 0.59 | 0.06 | 0.47 | 0.10 | 0.60 | 0.05 | 0.65 | 0.09 | 0.70 | 0.16 | 0.71 | 0.26 | 0.83 | 0.14 | 0.71 | 0.23 | 0.83 | 0.08 | 0.71 | 0.14 | 0.79 |
| Average | 0.30 | 1150% | 0.75 | 25% | 0.41 | 787% | 0.87 | 0.02 | 0.60 | 0.05 | 0.62 | 0.07 | 0.66 | 0.12 | 0.64 | 0.09 | 0.65 | 0.15 | 0.68 | 0.18 | 0.73 | 0.29 | 0.83 | 0.15 | 0.76 | 0.25 | 0.83 | 0.17 | 0.70 | 0.26 | 0.82 |
| Openjpa | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | Random Forest | | | | | | | Perphecy | | | | LR | | | | SVM | | | | XG-50 | | | | XG-100 | | | | XG-500 | | | |
| | Pre. | Improv. | Recall | Improv. | F-1 | Improv. | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC | Pre. | Recall | F-1 | AUC |
| Res. time | 0.29 | 222% | 0.85 | 6% | 0.43 | 169% | **0.92** | 0.09 | 0.80 | 0.16 | 0.60 | 0.29 | 0.59 | 0.39 | 0.67 | 0.10 | 0.95 | 0.17 | 0.60 | 0.24 | 0.79 | 0.36 | 0.90 | 0.23 | 0.79 | 0.35 | 0.90 | 0.19 | 0.86 | 0.31 | 0.89 |
| Cpu | 0.73 | 217% | 0.85 | 21% | 0.78 | 123% | **0.91** | 0.23 | 0.70 | 0.35 | 0.56 | 0.49 | 0.75 | 0.59 | 0.81 | 0.34 | 0.50 | 0.41 | 0.63 | 0.67 | 0.87 | 0.75 | 0.93 | 0.73 | 0.81 | 0.77 | 0.93 | 0.61 | 0.86 | 0.71 | 0.93 |
| Memory | 0.48 | 380% | 0.72 | -11% | 0.57 | 217% | **0.88** | 0.10 | 0.81 | 0.18 | 0.51 | 0.49 | 0.60 | 0.54 | 0.75 | 0.23 | 0.54 | 0.32 | 0.69 | 0.36 | 0.67 | 0.47 | 0.84 | 0.31 | 0.70 | 0.43 | 0.84 | 0.37 | 0.66 | 0.48 | 0.84 |
| I/O Read | 0.72 | 213% | 0.81 | 19% | 0.76 | 124% | **0.92** | 0.23 | 0.68 | 0.34 | 0.54 | 0.66 | 0.60 | 0.63 | 0.79 | 0.54 | 0.27 | 0.36 | 0.62 | 0.60 | 0.79 | 0.68 | 0.89 | 0.58 | 0.80 | 0.67 | 0.89 | 0.59 | 0.78 | 0.67 | 0.88 |
| I/O Write | 0.71 | 109% | 0.60 | -14% | 0.65 | 44% | **0.79** | 0.34 | 0.70 | 0.45 | 0.57 | 0.67 | 0.51 | 0.58 | 0.72 | 0.40 | 0.36 | 0.38 | 0.57 | 0.66 | 0.59 | 0.63 | 0.77 | 0.75 | 0.55 | 0.63 | 0.77 | 0.70 | 0.55 | 0.61 | 0.77 |
| Average | 0.59 | 195% | 0.77 | 4% | 0.64 | 113% | 0.88 | 0.20 | 0.74 | 0.30 | 0.56 | 0.52 | 0.61 | 0.55 | 0.75 | 0.32 | 0.52 | 0.33 | 0.62 | 0.51 | 0.74 | 0.58 | 0.87 | 0.52 | 0.73 | 0.57 | 0.87 | 0.49 | 0.74 | 0.56 | 0.86 |

Table .2: Average rank of the important metrics in our classifiers

| | 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 |
| EXP | 7.5 | 5.0 | 5.6 | 6.5 | 6.1 |
| if_chg | 7.5 | 6.9 | 7.3 | 6.9 | 6.3 |
| NTM | 6.1 | 6.7 | 8.2 | 7.5 | 6.6 |
| expVariable_add | 7.5 | 7.0 | 8.7 | 8.2 | 7.8 |
| expVariable_del | 8.2 | 9.7 | 8.4 | 8.6 | 8.9 |
| if_add | 8.8 | 8.1 | 9.1 | 8.6 | 9.1 |
| NF | 9.6 | 9.7 | 9.7 | 8.9 | 8.3 |
| externalCall_add | 7.7 | 10.2 | 9.4 | 9.6 | 11.4 |
| FN | 10.4 | 8.8 | 9.5 | 9.7 | 9.9 |
| while_chg | 9.6 | 11.0 | 12.2 | 9.8 | 10.2 |
| try_chg | 9.9 | 10.7 | 11.4 | 9.9 | 10.7 |
| ND | 10.7 | 11.3 | 12.3 | 10.9 | 9.9 |
| static_add | 17.8 | 15.8 | 10.9 | 11.2 | 17.9 |
| for_del | 14.7 | 12.0 | 12.0 | 11.3 | 13.0 |
| if_del | 8.9 | 11.8 | 10.0 | 11.8 | 10.8 |
| NS | 12.2 | 12.8 | 10.2 | 11.9 | 11.5 |
| assert_del | 14.4 | 14.6 | 12.4 | 12.4 | 6.6 |
| switch_chg | 17.8 | 15.1 | 11.8 | 12.9 | 18.0 |
| try_del | 15.8 | 16.6 | 13.7 | 13.9 | 11.2 |
| synchronized_chg | 14.8 | 14.6 | 14.8 | 14.0 | 12.4 |
| externalCall_del | 14.6 | 13.5 | 13.8 | 14.1 | 15.3 |
| elseif_add | 19.3 | 16.1 | 14.2 | 14.3 | 14.5 |
| try_add | 11.7 | 15.7 | 14.9 | 14.4 | 15.6 |
| expParameter_add | 13.1 | 14.6 | 15.2 | 15.3 | 14.3 |
| case_add | 17.4 | 17.2 | 14.0 | 15.3 | 15.3 |
| synchronized_add | 16.6 | 15.6 | 18.0 | 15.5 | 16.5 |
| final_add | 17.9 | 16.6 | 17.4 | 15.6 | 15.9 |
| throw_del | 17.7 | 19.1 | 15.2 | 15.7 | 18.2 |
| return_add | 12.6 | 14.4 | 15.4 | 16.1 | 13.9 |
| return_del | 13.0 | 15.3 | 15.8 | 16.1 | 15.7 |
| expParameter_del | 15.8 | 16.3 | 16.6 | 16.8 | 16.5 |
| synchronized_del | 19.0 | 16.0 | 18.7 | 16.9 | 17.4 |
| else_add | 15.1 | 18.0 | 17.7 | 17.0 | 15.2 |
| static_del | 21.0 | 19.6 | 18.7 | 17.3 | 20.4 |

# References


Ahmed, T. M., Bezemer, C.-P., Chen, T.-H., Hassan, A. E., & Shang, W. (2016). Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report. In *Proceedings of the 13th international conference on mining software repositories* (pp. 1–12). New York, NY, USA: ACM.

Alam, M. M. u., Liu, T., Zeng, G., & Muzahid, A. (2017). Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the twelfth european conference on computer systems* (pp. 298–313). New York, NY, USA: ACM.

Alghmadi, H. M., Syer, M. D., Shang, W., & Hassan, A. E. (2016, Oct). An automated approach for recommending when to stop performance tests. In *2016 ieee international conference on software maintenance and evolution (icsme)* (p. 279-289). doi: 10.1109/ICSME.2016.46

Anderson, J., Salem, S., & Do, H. (2015). Striving for failure: an industrial case study about test failure prediction. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 2, pp. 49–58).

Andreolini, M., Colajanni, M., & Valente, P. (2005). Design and testing of scalable web-based systems with performance constraints. In *Firb-perf workshop on techniques, methodologies and tools for performance evaluation of complex systems (firb-perf 2005), 19 september 2005, torino, italy* (pp. 15–25).

Andrzejak, A., Friedrich, G., & Wotawa, F. (2018). Software configuration diagnosis - A survey of existing methods and open challenges. In A. Felfernig, J. Tiihonen, L. Hotz, & M. Stettinger (Eds.), *Proceedings of the 20th configuration workshop, graz, austria, september 27-28, 2018*

(Vol. 2220, pp. 85–92). CEUR-WS.org.

Attariyan, M., Chow, M., & Flinn, J. (2012). X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th usenix symposium on operating systems design and implementation (osdi)* (pp. 307–320).

Attariyan, M., & Flinn, J. (2010). Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th usenix symposium on operating systems design and implementation (osdi)* (pp. 1–14).

Axboe. (2019, Mar). *axboe/fio.* Retrieved from https://github.com/axboe/fio

Banerjee, A., & Dave, R. N. (2004). Validating clusters using the hopkins statistic. In *2004 ieee international conference on fuzzy systems (ieee cat. no. 04ch37542)* (Vol. 1, pp. 149–153).

Barik, T., DeLine, R., Drucker, S., & Fisher, D. (2016, May). The bones of the system: A case study of logging and telemetry at microsoft. In *2016 ieee/acm 38th international conference on software engineering companion (icse-c)* (p. 92-101).

Becker, L. A. (2000). Effect size (es). *Accessed on October*, *12*(2006), 155–159.

Benesty, J., Chen, J., Huang, Y., & Cohen, I. (2009). Pearson correlation coefficient. In *Noise reduction in speech processing* (pp. 1–4). Springer.

Beschastnikh, I., Brun, Y., Ernst, M. D., & Krishnamurthy, A. (2014). Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th international conference on software engineering* (pp. 468–479). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2568225.2568246 doi: 10.1145/2568225.2568246

Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., & Ernst, M. D. (2011). Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (pp. 267–277). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2025113.2025151 doi: 10.1145/2025113.2025151

Bodík, P., Goldszmidt, M., & Fox, A. (2008). Hilighter: Automatically building robust signatures of performance behavior for small-and large-scale systems. In *Sysml.*

Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5–32.

Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., ... Wert, A. (2015). Performance-oriented devops: A research agenda. *CoRR*, *abs/1508.04752*.

Busch, A., Noorshams, Q., Kounev, S., Koziolek, A., Reussner, R. H., & Amrehn, E. (2015). Automated workload characterization for I/O performance analysis in virtualized environments. In *Proceedings of the 6th ACM/SPEC international conference on performance engineering, austin, tx, usa, january 31 - february 4, 2015* (pp. 265–276).

Caliński, T., & Harabasz, J. (1974). A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods*, *3*(1), 1–27.

Calzarossa, M. C., Massari, L., & Tessera, D. (2016). Workload characterization: A survey revisited. *ACM Comput. Surv.*, *48*(3), 48:1–48:43.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, *16*, 321–357.

Chen, J. (2020). Performance regression detection in devops. In G. Rothermel & D. Bae (Eds.), *ICSE '20: 42nd international conference on software engineering, companion volume, seoul, south korea, 27 june - 19 july, 2020* (pp. 206–209). ACM.

Chen, J., & Shang, W. (2017, Sept). An exploratory study of performance regression introducing code changes. In *2017 ieee international conference on software maintenance and evolution (icsme)* (p. 341-352).

Chen, J., Shang, W., Hassan, A. E., Wang, Y., & Lin, J. (2019). An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour. In *34th IEEE/ACM international conference on automated software engineering, ASE 2019, san diego, ca, usa, november 11-15, 2019* (pp. 669–681). IEEE.

Chen, J., Shang, W., & Shihab, E. (2020). Perfjit: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Transactions on Software Engineering (TSE)*, To Appear.

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (pp. 785–794). New York, NY, USA: ACM.

Chen, T., Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M. N., & Flora, P. (2017).

Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *39th IEEE/ACM international conference on software engineering: Software engineering in practice track, ICSE-SEIP 2017, buenos aires, argentina, may 20-28, 2017* (pp. 243–252).

Chen, T.-H., Shang, W., Hassan, A. E., Nasser, M., & Flora, P. (2016). 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* (pp. 666–677). New York, NY, USA: ACM.

Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2014). Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th international conference on software engineering* (pp. 1001–1012). New York, NY, USA: ACM.

Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2016, December). Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Trans. Softw. Eng.*, *42*(12), 1148–1161.

Cherkasova, L., Ozonat, K., Mi, N., Symons, J., & Smirni, E. (2008). Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *2008 ieee international conference on dependable systems and networks with ftcs and dcc (dsn)* (pp. 452–461).

Cohen, I., Chase, J. S., Goldszmidt, M., Kelly, T., & Symons, J. (2004). Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Osdi* (Vol. 4, pp. 16–16).

Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., & Fox, A. (2005). Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the twentieth acm symposium on operating systems principles* (pp. 105–118). New York, NY, USA: ACM.

Comaniciu, D., & Meer, P. (2002, May). Mean shift: A robust approach toward feature space analysis. *IEEE Trans. Pattern Anal. Mach. Intell.*, *24*(5), 603–619.

Cortellessa, V., Di Marco, A., & Inverardi, P. (2011). *Model-based software performance analysis*.

Springer Science & Business Media.

Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., & Bianchini, R. (2017). Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th symposium on operating systems principles, shanghai, china, october 28-31, 2017* (pp. 153–167).

Costa, D., Andrzejak, A., Seboek, J., & Lo, D. (2017). Empirical study of usage and performance of java collections. In *Proceedings of the 8th acm/spec on international conference on performance engineering* (pp. 389–400). New York, NY, USA: ACM. doi: 10.1145/3030207.3030221

Cunha, C. A., & e Silva, L. M. (2012). Separating performance anomalies from workload-explained failures in streaming servers. In *2012 12th ieee/acm international symposium on cluster, cloud and grid computing (ccgrid 2012)* (pp. 292–299).

Dean, J., & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, *56*(2), 74–80.

de Oliveira, A. B., Fischmeister, S., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2017). 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* (pp. 103–113).

Diao, Y., Hellerstein, J. L., Parekh, S., & Bigus, J. P. (2003). Managing web server performance with autotune agents. *IBM Systems Journal*, *42*(1), 136–149.

Di Nardo, D., Alshahwan, N., Briand, L., & Labiche, Y. (2015). Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability*, *25*(4), 371–396.

Ding, Z., Chen, J., & Shang, W. (2020). Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet? In G. Rothermel & D. Bae (Eds.), *ICSE '20: 42nd international conference on software engineering, seoul, south korea, 27 june - 19 july, 2020* (pp. 1435–1446). ACM.

Dong, Z., Andrzejak, A., & Shao, K. (2015). Practical and accurate pinpointing of configuration errors using static analysis. In *Proceedings of the 31st international conference on software maintenance and evolution* (pp. 171–180).

Dong, Z., Ghanavati, M., & Andrzejak, A. (2013). Automated diagnosis of software misconfigurations based on static analysis. In *Proceedings of the international symposium on software reliability engineering workshops (issrew'13)* (pp. 162–168).

Elnaffar, S., & Martin, P. (2002). Characterizing computer systems workloads. *Submitted to ACM Computing Surveys Journal*.

Enbody, R. (1999). Perfmon: Performance monitoring tool. *Michigan State University*.

Fiegerman, S. (2019, January). *Netflix adds 9 million paying subscribers, but stock falls.* https://www.cnn.com/2019/01/17/media/netflix-earnings-q4/index.html. ((Accessed on 03/29/2019))

Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., & Flora, P. (2010). Mining performance regression testing repositories for automated performance analysis. In *Quality software (qsic), 2010 10th international conference on* (pp. 32–41).

Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., & Flora, P. (2015). An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 2, pp. 159–168).

Fu, X., Ren, R., Zhan, J., Zhou, W., Jia, Z., & Lu, G. (2012). Logmaster: Mining event correlations in logs of large-scale cluster systems. In *Proceedings of the 2012 ieee 31st symposium on reliable distributed systems* (pp. 71–80). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/SRDS.2012.40 doi: 10.1109/SRDS.2012.40

Gao, R., Jiang, Z. M., Barna, C., & Litoiu, M. (2016). A framework to evaluate the effectiveness of different load testing analysis techniques. In *2016 IEEE international conference on software testing, verification and validation, ICST 2016, chicago, il, usa, april 11-15, 2016* (pp. 22–32).

Gao, R., & Jiang, Z. M. J. (2017). An exploratory study on assessing the impact of environment variations on the results of load tests. In *Proceedings of the 14th international conference on mining software repositories, MSR 2017, buenos aires, argentina, may 20-28, 2017* (pp. 379–390).

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In G. J. Gordon, D. B. Dunson, & M. Dudík (Eds.), *Proceedings of the fourteenth international conference on artificial intelligence and statistics, AISTATS 2011, fort lauderdale, usa, april 11-13, 2011* (Vol. 15, pp. 315–323). JMLR.org.

Godard, S. (2019). *pidstat(1): Report statistics for tasks - linux man page.* https://linux.die.net/man/1/pidstat. ((Accessed on 04/06/2019))

Gousios, G., Karakoidas, V., & Spinellis, D. (2006). Tuning javas memory manager for high performance server applications. *memory*, *11*(22), 7–15.

Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000, Jul). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, *26*(7), 653-661. doi: 10.1109/32.859533

Guo, J., Czarnecki, K., Apel, S., Siegmund, N., & Wasowski, A. (2013). Variability-aware performance prediction: A statistical learning approach. In E. Denney, T. Bultan, & A. Zeller (Eds.), *2013 28th IEEE/ACM international conference on automated software engineering, ASE 2013, silicon valley, ca, usa, november 11-15, 2013* (pp. 301–311). IEEE.

Guo, P. J., Zimmermann, T., Nagappan, N., & Murphy, B. (2010, May). 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, p. 495-504).

Haghdoost, A., He, W., Fredin, J., & Du, D. H. C. (2017). On the accuracy and scalability of intensive I/O workload replay. In *15th USENIX conference on file and storage technologies, FAST 2017, santa clara, ca, usa, february 27 - march 2, 2017* (pp. 315–328).

Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., & Baudry, B. (2019). Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. *Empir. Softw. Eng.*, *24*(2), 674–717.

Han, X., & Yu, T. (2016). An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th international symposium on empirical software engineering and measurement.*

Han, X., Yu, T., & Lo, D. (2018). 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* (pp. 17–28).

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

Hartung, J., Knapp, G., & Sinha, B. K. (2011). *Statistical meta-analysis with applications* (Vol. 738). John Wiley & Sons.

Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., & Hindle, A. (2016). Energy profiles of java collections classes. In *Proceedings of the 38th international conference on software engineering* (pp. 225–236). New York, NY, USA: ACM. doi: 10.1145/2884781.2884869

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the 31st international conference on software engineering* (pp. 78–88). Washington, DC, USA: IEEE Computer Society.

Hassan, A. E., Martin, D. J., Flora, P., Mansfield, P., & Dietz, D. (2008). An industrial case study of customizing operational profiles using log compression. In *Proceedings of the 30th international conference on software engineering* (pp. 713–723). New York, NY, USA: ACM.

He, P., Li, B., Liu, X., Chen, J., & Ma, Y. (2015). An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, *59*, 170–190.

He, S., Lin, Q., Lou, J., Zhang, H., Lyu, M. R., & Zhang, D. (2018). Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/SIGSOFT FSE 2018, lake buena vista, fl, usa, november 04-09, 2018* (pp. 60–70).

Hearst, M. A., Dumais, S. T., Osuna, E., Platt, J., & Scholkopf, B. (1998, July). Support vector machines. *IEEE Intelligent Systems and their Applications*, *13*(4), 18-28.

Heger, C., Happe, J., & Farahbod, R. (2013). Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th acm/spec international conference on performance engineering* (pp. 27–38). New York, NY, USA: ACM.

Hindle, A. (2015, Apr 01). Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, *20*(2), 374–409.

Hosmer Jr, D. W., Lemeshow, S., & Sturdivant, R. X. (2013). *Applied logistic regression* (Vol. 398). John Wiley & Sons.

Huang, P., Ma, X., Shen, D., & Zhou, Y. (2014). Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th international conference on software engineering* (pp. 60–71). New York, NY, USA: ACM. doi: 10.1145/2568225.2568232

Hüttermann, M. (2012). *Devops for developers*. Apress.

JAIN, A. (2016). *Complete guide to parameter tuning in xgboost with codes in python*. https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python.

Jenks, G. F. (1967). The data model concept in statistical mapping. *International yearbook of cartography*, *7*, 186–190.

Jiang, M., Munawar, M. A., Reidemeister, T., & Ward, P. A. (2009). Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *Dependable systems & networks, 2009. dsn'09. ieee/ifip international conference on* (pp. 285–294).

Jiang, Z. M. (2010). Automated analysis of load testing results. In *Proceedings of the nineteenth international symposium on software testing and analysis, ISSTA 2010, trento, italy, july 12-16, 2010* (pp. 143–146).

Jiang, Z. M., & Hassan, A. E. (2015). A survey on load testing of large-scale software systems. *IEEE Trans. Software Eng.*, *41*(11), 1091–1118.

Jiang, Z. M., Hassan, A. E., Hamann, G., & Flora, P. (2008a). An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution: Research and Practice*, *20*(4), 249–267.

Jiang, Z. M., Hassan, A. E., Hamann, G., & Flora, P. (2008b). Automatic identification of load testing problems. In *24th IEEE international conference on software maintenance (ICSM 2008), september 28 - october 4, 2008, beijing, china* (pp. 307–316).

Jiang, Z. M., Hassan, A. E., Hamann, G., & Flora, P. (2009). Automated performance analysis of load tests. In *25th IEEE international conference on software maintenance (ICSM 2009), september 20-26, 2009, edmonton, alberta, canada* (pp. 125–134).

Jiarpakdee, J., Tantithamthavorn, C., & Hassan, A. E. (2019). The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering*.

Jin, D., Qu, X., Cohen, M. B., & Robinson, B. (2014). Configurations everywhere: implications for testing and debugging in practice. In P. Jalote, L. C. Briand, & A. van der Hoek (Eds.), *36th international conference on software engineering, ICSE '14, companion proceedings, hyderabad, india, may 31 - june 07, 2014* (pp. 215–224). ACM.

Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012). Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd acm sigplan conference on programming language design and implementation* (pp. 77–88). New York, NY, USA: ACM. doi: 10.1145/2254064.2254075

JMeter. (1998). *Apache jmeter - apache jmete.* https://jmeter.apache.org/. ((Accessed on 03/29/2019))

Joel, M. (2017). *Mozilla performance regressions policy.* https://www.mozilla.org/en-US/about/governance/policies/regressions. ((Accessed on 01/12/2019))

Kabinna, S., Bezemer, C.-P., Shang, W., Syer, M. D., & Hassan, A. E. (2018, Feb 01). Examining the stability of logging statements. *Empirical Software Engineering*, *23*(1), 290–333.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., & Hassan, A. E. (2016). Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, *21*(5), 2072–2106.

Kamei, Y., & Shihab, E. (2016). Defect prediction: Accomplishments and future challenges. In *Software analysis, evolution, and reengineering (saner), 2016 ieee 23rd international conference on* (Vol. 5, pp. 33–45).

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013, June). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, *39*(6), 757-773.

Kaufman, L., & Rousseeuw, P. J. (2009). *Finding groups in data: an introduction to cluster analysis* (Vol. 344). John Wiley & Sons.

Kazmi, R., Jawawi, D. N., Mohamad, R., & Ghani, I. (2017). Effective regression test case selection: A systematic literature review. *ACM Computing Surveys (CSUR)*, *50*(2), 1–32.

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

Kit, E. (2010). *How one second could cost amazon $1.6 billion in sales.* https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales. ((Accessed on 09/06/2019))

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., & Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, *28*(8), 721–734.

Koru, A. G., Zhang, D., Emam, K. E., & Liu, H. (2009, March). An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, *35*(2), 293-304. doi: 10.1109/TSE.2008.90

Krishnamurthy, D., Rolia, J. A., & Majumdar, S. (2006). A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. Software Eng.*, *32*(11), 868–882.

Laaber, C., & Leitner, P. (2018). An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th international conference on mining software repositories* (pp. 119–130). New York, NY, USA: ACM. doi: 10.1145/3196398.3196407

Laaber, C., Scheuner, J., & Leitner, P. (2019). Software microbenchmarking in the cloud. how bad is it really? *Empirical Software Engineering*, *24*(4), 2469–2508.

Laali, M., Liu, H., Hamilton, M., Spichkova, M., & Schmidt, H. W. (2016). Test case prioritization using online fault detection information. In *Ada-europe international conference on reliable software technologies* (pp. 78–93).

Lawrence, S., Giles, C. L., Tsoi, A. C., & Back, A. D. (1997). Face recognition: a convolutional neural-network approach. *IEEE Trans. Neural Networks*, *8*(1), 98–113.

Leitner, P., & Bezemer, C.-P. (2017). An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th acm/spec on international conference on performance engineering* (pp. 373–384). New York, NY, USA: ACM.

Leitner, P., & Cito, J. (2016, April). Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, *16*(3), 15:1–15:23. doi: 10.1145/2885497

Lengauer, P., & Mössenböck, H. (2014). The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors. In K. Lange, J. Murphy, W. Binder, & J. Merseguer (Eds.), *ACM/SPEC international conference on performance engineering, icpe'14, dublin, ireland, march 22-26, 2014* (pp. 111–122). ACM.

Li, H., Chen, T.-H. P., Hassan, A. E., Nasser, M., & Flora, P. (2018). Adopting autonomic computing capabilities in existing large-scale systems: An industrial experience report. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice* (p. 110).

Li, H., Chen, T. P., Shang, W., & Hassan, A. E. (2018). Studying software logging using topic models. *Empir. Softw. Eng.*, *23*(5), 2655–2694.

Li, H., Shang, W., Zou, Y., & E. Hassan, A. (2017, Aug 01). Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, *22*(4), 1831–1865.

Liao, L., Chen, J., Li, H., Zeng, Y., Shang, W., Guo, J., ... Sajedi, S. (2020). Using black-box performance models to detect performance regressions under varying workloads: an empirical study. *Empir. Softw. Eng.*, *25*(5), 4130–4160.

Lim, M., Lou, J., Zhang, H., Fu, Q., Teoh, A. B. J., Lin, Q., ... Zhang, D. (2014). Identifying recurrent and unknown performance issues. In R. Kumar, H. Toivonen, J. Pei, J. Z. Huang, & X. Wu (Eds.), *2014 IEEE international conference on data mining, ICDM 2014, shenzhen, china, december 14-17, 2014* (pp. 320–329). IEEE Computer Society.

Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y., & Chen, X. (2016). Log clustering based problem identification for online service systems. In *Proceedings of the 38th international conference on software engineering companion* (pp. 102–111). New York, NY, USA: ACM.

LINDEN, G. (2006). *Geeking with greg: Marissa mayer at web 2.0.* http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. ((Accessed on 04/01/2019))

Lobo, J. M., Jiménez-Valverde, A., & Real, R. (2008). Auc: a misleading measure of the

performance of predictive distribution models. *Global ecology and Biogeography*, *17*(2), 145–151.

Luo, Q., Poshyvanyk, D., & Grechanik, M. (2016). Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th international conference on mining software repositories* (pp. 25–36). New York, NY, USA: ACM. doi: 10.1145/2901739.2901765

lxml. (2005). *Parsing xml and html with lxml.* https://lxml.de/3.1/parsing.html.

Malik, H., Hemmati, H., & Hassan, A. E. (2013). Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 international conference on software engineering* (pp. 1012–1021). Piscataway, NJ, USA: IEEE Press.

Malik, H., Jiang, Z. M., Adams, B., Hassan, A. E., Flora, P., & Hamann, G. (2010). Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *Software maintenance and reengineering (csmr), 2010 14th european conference on* (pp. 222–231).

Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K.-i., & Nakamura, M. (2010). An analysis of developer metrics for fault prediction. In *Proceedings of the 6th international conference on predictive models in software engineering* (p. 18).

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2016). An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering*, *21*(5), 2146-2189.

Mednis, M., & Aurich, M. (2012, 01). Application of string similarity ratio and edit distance in automatic metabolite reconciliation comparing reconstructions and models. *Biosystems and Information technology*, *1*, 14-18.

Meira, J. A., de Almeida, E. C., Sunyé, G., Traon, Y. L., & Valduriez, P. (2013). Stress testing of transactional database systems. *JIDM*, *4*(3), 279–294.

Mende, T., & Koschke, R. (2009). Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th international conference on predictor models in software engineering* (pp. 7:1–7:10). New York, NY, USA: ACM.

Mende, T., & Koschke, R. (2010, March). Effort-aware defect prediction models. In *2010 14th*

*european conference on software maintenance and reengineering* (p. 107-116).

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. In Y. Bengio & Y. LeCun (Eds.), *1st international conference on learning representations, ICLR 2013, scottsdale, arizona, usa, may 2-4, 2013, workshop track proceedings.*

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th international conference on neural information processing systems - volume 2* (pp. 3111–3119). USA: Curran Associates Inc.

Mockus, A., & Weiss, D. M. (2000). Predicting risk of software changes. *Bell Labs Technical Journal*, *5*(2), 169–180.

Mostafa, S., Wang, X., & Xie, T. (2017). Perfranker: Prioritization of performance regression tests for collection-intensive software. In *Proceedings of the 26th acm sigsoft international symposium on software testing and analysis* (pp. 23–34). New York, NY, USA: ACM. doi: 10.1145/3092703.3092725

Murphy-Hill, E., Zimmermann, T., Bird, C., & Nagappan, N. (2013). The design of bug fixes. In *Proceedings of the 2013 international conference on software engineering* (pp. 332–341). Piscataway, NJ, USA: IEEE Press.

Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009, March). Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, *44*(3), 265–276.

Nachar, N., et al. (2008). The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology*, *4*(1), 13–20.

Nagappan, M., Wu, K., & Vouk, M. A. (2009). Efficiently extracting operational profiles from execution logs using suffix arrays. In *ISSRE 2009, 20th international symposium on software reliability engineering, mysuru, karnataka, india, 16-19 november 2009* (pp. 41–50).

Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on software engineering* (pp. 284–292). New York, NY, USA: ACM.

Nagappan, N., Ball, T., & Zeller, A. (2006). Mining metrics to predict component failures. In *Proceedings of the 28th international conference on software engineering* (pp. 452–461). New York, NY, USA: ACM.

Najafi, A., Shang, W., & Rigby, P. C. (2019). 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)* (pp. 213–222).

Nguyen, T. H., Adams, B., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2012). Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd acm/spec international conference on performance engineering* (pp. 299–310). New York, NY, USA: ACM.

Nguyen, T. H. D., Adams, B., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2011, Dec). Automated verification of load tests using control charts. In *2011 18th asia-pacific software engineering conference* (p. 282-289). doi: 10.1109/APSEC.2011.59

Nguyen, T. H. D., Nagappan, M., Hassan, A. E., Nasser, M., & Flora, P. (2014). An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th working conference on mining software repositories* (pp. 232–241). New York, NY, USA: ACM.

Nistor, A., Jiang, T., & Tan, L. (2013). Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (msr)* (pp. 237–246).

Noor, T. B., & Hemmati, H. (2017). 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* (pp. 2–11).

Oliner, A., Ganapathi, A., & Xu, W. (2012, February). Advances and challenges in log analysis. *Commun. ACM*, *55*(2), 55–61.

Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005, April). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, *31*(4), 340-355. doi: 10.1109/TSE.2005.49

Pradel, M., Huggler, M., & Gross, T. R. (2014). Performance regression testing of concurrent classes. In *Proceedings of the 2014 international symposium on software testing and analysis*

(pp. 13–25). New York, NY, USA: ACM. doi: 10.1145/2610384.2610393

*psutil.* (2017). Retrieved 2017-2-11, from https://github.com/giampaolo/psutil

Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., & Binkley, D. (2014, February). Recovering test-to-code traceability using slicing and textual analysis. *J. Syst. Softw.*, *88*, 147–168. doi: 10.1016/j.jss.2013.10.019

Rabkin, A., & Katz, R. H. (2011). Precomputing possible configuration error diagnoses. In P. Alexander, C. S. Pasareanu, & J. G. Hosking (Eds.), *26th IEEE/ACM international conference on automated software engineering (ASE 2011), lawrence, ks, usa, november 6-10, 2011* (pp. 193–202). IEEE Computer Society.

Raghavachari, M., Reimer, D., & Johnson, R. D. (2003). The deployer's problem: configuring application servers for performance and reliability. In *Proceedings of the 25th international conference on software engineering* (pp. 484–489).

Ramos, J., et al. (2003). Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning* (Vol. 242, pp. 133–142).

Saha, R. K., Zhang, L., Khurshid, S., & Perry, D. E. (2015). 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, pp. 268–279).

Sarkar, A., Guo, J., Siegmund, N., Apel, S., & Czarnecki, K. (2015). Cost-efficient sampling for performance prediction of configurable systems (T). In M. B. Cohen, L. Grunske, & M. Whalen (Eds.), *30th IEEE/ACM international conference on automated software engineering, ASE 2015, lincoln, ne, usa, november 9-13, 2015* (pp. 342–352). IEEE Computer Society.

Saxena, N. R., Fernández-Gomez, S., Huang, W., Mitra, S., Yu, S., & McCluskey, E. J. (2000). Dependable computing and online testing in adaptive and configurable systems. *IEEE Des. Test Comput.*, *17*(1), 29–41.

Sayagh, M., & Adams, B. (2015). Multi-layer software configuration: Empirical study on wordpress. In M. W. Godfrey, D. Lo, & F. Khomh (Eds.), *15th IEEE international working conference on source code analysis and manipulation, SCAM 2015, bremen, germany,*

*september 27-28, 2015* (pp. 31–40). IEEE Computer Society.

Sayagh, M., Kerzazi, N., & Adams, B. (2017). On cross-stack configuration errors. In S. Uchitel, A. Orso, & M. P. Robillard (Eds.), *Proceedings of the 39th international conference on software engineering, ICSE 2017, buenos aires, argentina, may 20-28, 2017* (pp. 255–265). IEEE / ACM.

Sayagh, M., Kerzazi, N., Adams, B., & Petrillo, F. (2018). Software configuration engineering in practice: Interviews, survey, and systematic literature review. *IEEE Transactions on Software Engineering*.

Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. In *Proceedings of the 25th annual international acm sigir conference on research and development in information retrieval* (pp. 253–260). New York, NY, USA: ACM. doi: 10.1145/564376.564421

Seo, B., Kang, S., Choi, J., Cha, J., Won, Y., & Yoon, S. (2014). IO workload characterization revisited: A data-mining approach. *IEEE Trans. Computers*, *63*(12), 3026–3038.

Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. In (Vol. 5, pp. 3909–3943). IEEE.

Shang, W., Hassan, A. E., Nasser, M. N., & Flora, P. (2015). Automated detection of performance regressions using regression models on clustered performance counters. In L. K. John, C. U. Smith, K. Sachs, & C. M. Lladó (Eds.), *Proceedings of the 6th ACM/SPEC international conference on performance engineering, austin, tx, usa, january 31 - february 4, 2015* (pp. 15–26). ACM.

Shang, W., Jiang, Z. M., Hemmati, H., Adams, B., Hassan, A. E., & Martin, P. (2013). Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 international conference on software engineering* (pp. 402–411). Piscataway, NJ, USA: IEEE Press.

Shivaji, S., Whitehead, E. J., Akella, R., & Kim, S. (2013, April). Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, *39*(4), 552-569.

Siegmund, N., Grebhahn, A., Apel, S., & Kastner, C. (2015). Performance-influence models

for highly configurable systems. In *Proceedings of the 10th joint meeting of the european software engineering conference and the acm sigsoft symposium on the foundations of software engineering (esec/fse'15)* (pp. 284–294).

Simic, B., & Conklin, S. (2012). *Improving the usability of apm data: Essential capabilities and benefits.* https://assets.extrahop.com/uploads/trac-market-insight-usability-of-apm-data.pdf. ((Accessed on 09/01/2020))

Singh, R., Bezemer, C.-P., Shang, W., & Hassan, A. E. (2016). Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm. In *Proceedings of the 7th acm/spec on international conference on performance engineering* (pp. 309–320).

Śliwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes? In *Proceedings of the 2005 international workshop on mining software repositories* (pp. 1–5). New York, NY, USA: ACM. doi: 10.1145/1082983.1083147

Smith, C. U., & Williams, L. G. (2002). *Performance solutions: a practical guide to creating responsive, scalable software* (Vol. 23). Addison-Wesley Reading.

Snellman, N., Ashraf, A., & Porres, I. (2011). Towards automatic performance and scalability testing of rich internet applications in the cloud. In (pp. 161–169).

Song, J. (2016). *Ckmeans.1d.dp function — r documentation.* https://www.rdocumentation.org/packages/Ckmeans.1d.dp/versions/3.4.0-1/topics/Ckmeans.1d.dp. ((Accessed on 01/02/2020))

Song, L., & Lu, S. (2017). Performance diagnosis for inefficient loops. In *Proceedings of the 39th international conference on software engineering* (pp. 370–380).

SPEC. (2003). *The workload for the specweb96 benchmark.* https://www.spec.org/web96/workload.html.

*srcml.* (2017). Retrieved from http://www.srcml.org/

Summers, J., Brecht, T., Eager, D., & Gutarin, A. (2016). Characterizing the workload of a netflix streaming video server. In *2016 ieee international symposium on workload characterization (iiswc)* (pp. 1–12).

Svajlenko, J., & Roy, C. K. (2015). Evaluating clone detection tools with bigclonebench. In *2015*

*ieee international conference on software maintenance and evolution (icsme)* (pp. 131–140).

Svajlenko, J., Roy, C. K., & Cordy, J. R. (2013). A mutation analysis based benchmarking framework for clone detectors. In *2013 7th international workshop on software clones (iwsc)* (pp. 8–9).

Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M., & Flora, P. (2014). Continuous validation of load test suites. In *Proceedings of the 5th acm/spec international conference on performance engineering* (pp. 259–270).

Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M. N., & Flora, P. (2013). Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *2013 IEEE international conference on software maintenance, eindhoven, the netherlands, september 22-28, 2013* (pp. 110–119).

Syer, M. D., Shang, W., Jiang, Z. M., & Hassan, A. E. (2017). Continuous validation of performance test workloads. *Automated Software Engineering*, *24*(1), 189–231.

Tan, J., Kavulya, S., Gandhi, R., & Narasimhan, P. (2010). Visual, log-based causal tracing for performance debugging of mapreduce systems. In *2010 international conference on distributed computing systems, ICDCS 2010, genova, italy, june 21-25, 2010* (pp. 795–806).

Tantithamthavorn, C., & Hassan, A. E. (2018a). 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* (pp. 286–295). New York, NY, USA: ACM.

Tantithamthavorn, C., & Hassan, A. E. (2018b). 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* (pp. 286–295).

Tantithamthavorn, C., Hassan, A. E., & Matsumoto, K. (2018). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., Ihara, A., & Matsumoto, K. (2015, May). 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,

p. 812-823).

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2017, Jan). An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, *43*(1), 1-18.

Tarvo, A., & Reiss, S. P. (2012). Using computer simulation to predict the performance of multithreaded programs. In *Proceedings of the 3rd acm/spec international conference on performance engineering* (pp. 217–228). New York, NY, USA: ACM.

Tianyin, X., & Yuanyuan, Z. (2015). Systems approaches to tackling configuration errors: a survey. *ACM Computing Surveys*, *47*(4), 1–41.

Tillmann, N., & Schulte, W. (2006). Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, *23*(4), 38–47.

Tourani, P., & Adams, B. (2016). The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse. In *IEEE 23rd international conference on software analysis, evolution, and reengineering, SANER 2016, suita, osaka, japan, march 14-18, 2016 - volume 1* (pp. 189–200). IEEE Computer Society.

Trubiani, C., Bran, A., van Hoorn, A., Avritzer, A., & Knoche, H. (2018). Exploiting load testing and profiling for performance antipattern detection. *Information & Software Technology*, *95*, 329–345.

Tsakiltsidis, S., Miranskyy, A., & Mazzawi, E. (2016). On automatic detection of performance bugs. In *Software reliability engineering workshops (issrew), 2016 ieee international symposium on* (pp. 132–139).

Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In *Proceedings of the european conference on computer systems (eurosys).* Bordeaux, France.

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

von Storch, H. (1999). Misuses of statistical analysis in climate research. In *Analysis of climate*

*variability* (pp. 11–26). Berlin, Heidelberg: Springer Berlin Heidelberg.

Wang, S., Nam, J., & Tan, L. (2017). Qtep: quality-aware test case prioritization. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 523–534).

Weyuker, E., & Vokolos, F. (2000, Dec). Experience with performance testing of software systems: issues, an approach, and case study. *Transactions on Software Engineering*, *26*(12), 1147-1156.

White, T. (2009). *Hadoop: The definitive guide* (1st ed.). O'Reilly Media, Inc.

Wikipedia. (2016). *Devops.* https://en.wikipedia.org/wiki/DevOps. ((Accessed on 09/06/2019))

Wilkes, J. (2011). *Google ai blog: More google cluster data.* https://ai.googleblog.com/2011/11/more-google-cluster-data.html. ((Accessed on 04/04/2019))

Williams, C., & Spacco, J. (2008). Szz revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 workshop on defects in large software systems* (pp. 32–36). New York, NY, USA: ACM.

Wold, S., Esbensen, K., & Geladi, P. (1987). Principal component analysis. *Chemometrics and intelligent laboratory systems*, *2*(1-3), 37–52.

Wu, Q., & Wang, Y. (2010). Performance testing and optimization of j2ee-based web applications. In *2010 second international workshop on education technology and computer science* (Vol. 2, pp. 681–683).

Xi, B., Liu, Z., Raghavachari, M., Xia, C. H., & Zhang, L. (2004). A smart hill-climbing algorithm for application server configuration. In *Proceedings of the thirteenth international world wide web conference* (pp. 287–296).

Xi, H., Zhan, J., Jia, Z., Hong, X., Wang, L., Zhang, L., ... Lu, G. (2011). Characterization of real workloads of web search engines. In *2011 ieee international symposium on workload characterization (iiswc)* (pp. 15–25).

Xiong, P., Pu, C., Zhu, X., & Griffith, R. (2013). vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *Proceedings of the 4th acm/spec international conference on performance engineering* (pp. 271–282). New York, NY, USA: ACM.

Xu, D., & Tian, Y. (2015). A comprehensive survey of clustering algorithms. *Annals of Data Science*, *2*(2), 165–193.

Xu, W., Huang, L., Fox, A., Patterson, D. A., & Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM symposium on operating systems principles 2009, SOSP 2009, big sky, montana, usa, october 11-14, 2009* (pp. 117–132).

Yadwadkar, N. J., Bhattacharyya, C., Gopinath, K., Niranjan, T., & Susarla, S. (2010). Discovery of application workloads from network file traces. In *8th USENIX conference on file and storage technologies, san jose, ca, usa, february 23-26, 2010* (pp. 183–196).

Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., ... Leung, H. (2016). 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* (pp. 157–168). New York, NY, USA: ACM.

Yao, K., de Pádua, G. B., Shang, W., Sporea, S., Toma, A., & Sajedi, S. (2018). Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In K. Wolter, W. J. Knottenbelt, A. van Hoorn, & M. Nambiar (Eds.), *Proceedings of the 2018 ACM/SPEC international conference on performance engineering, ICPE 2018, berlin, germany, april 09-13, 2018* (pp. 127–138). ACM. doi: 10.1145/3184407.3184416

Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., & Pasupathy, S. (2011). An empirical study on configuration errors in commercial and open source systems. In T. Wobber & P. Druschel (Eds.), *Proceedings of the 23rd ACM symposium on operating systems principles 2011, SOSP 2011, cascais, portugal, october 23-26, 2011* (pp. 159–172). ACM.

Zaman, S., Adams, B., & Hassan. (2012). A qualitative study on performance bugs. In *Proceedings of the 9th ieee working conference on mining software repositories* (pp. 199–208). Piscataway, NJ, USA: IEEE Press.

Zaman, S., Adams, B., & Hassan, A. E. (2011). Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories* (pp. 93–102).

Zhang, F., Mockus, A., Keivanloo, I., & Zou, Y. (2014). Towards building a universal defect

prediction model. In *Proceedings of the 11th working conference on mining software repositories* (pp. 182–191). New York, NY, USA: ACM.

Zhang, S. (2013). Confdiagnoser: An automated configuration error diagnosis tool for java software. In *Proceedings of the 35th international conference on software engineering (icse'13)* (pp. 1438–1440).

Zhang, S., & Ernst, M. D. (2014). Which configuration option should i change? In *Proceedings of the 36th international conference on software engineering (icse'14)* (pp. 152–163).

Zhu, Y., Shihab, E., & Rigby, P. C. (2018). Test re-prioritization in continuous testing environments. In *2018 ieee international conference on software maintenance and evolution (icsme)* (pp. 69–79).

Zimmermann, T., Premraj, R., & Zeller, A. (2007, May). Predicting defects for eclipse. In *Predictor models in software engineering, 2007. promise'07: Icse workshops 2007.* (p. 9-9).