Addressing Performance Regressions in DevOps: Can We Escape from System Performance Testing?

by

Lizhi Liao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

© Lizhi Liao 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:
Federica Sarro
Professor
Department of Computer Science
University College London

Supervisor(s):
Weiyi Shang
Associate Professor
Department of Electrical and Computer Engineering
University of Waterloo

Internal Member:
Arie Gurfinkel
Professor
Department of Electrical and Computer Engineering
University of Waterloo

Jialu Zhang
Assistant Professor
Department of Electrical and Computer Engineering
University of Waterloo

Internal-External Member:
Shane McIntosh
Associate Professor
David R. Cheriton School of Computer Science
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. 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.

Statement of Contributions

In all chapters and related publications of the thesis, my contributions include proposing the initial research idea, investigating background knowledge and related literature, proposing and implementing research methods, conducting experiments and analyzing experimental results, and writing and polishing the manuscript. My co-authors supported me in refining the initial research idea, providing feedback on research methods and experimental results, and offering advice for polishing the manuscript.

Earlier versions of the work in the thesis were published as listed below:

- Using Black-Box Performance Models to Detect Performance Regressions under Varying Workloads: An Empirical Study (This work is described in Chapter 3)
  Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi, Empirical Software Engineering (EMSE), 2020, Volume 25, Pages 4130–4160.

- Locating Performance Regression Root Causes in the Field Operations of Web-based Systems: An Experience Report (This work is described in Chapter 4)
  Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi, IEEE Transactions on Software Engineering (TSE), 2021, Volume 48, Issue 12, Pages 4986-5006.

- Adapting Performance Analytic Techniques in a Real-World Database-Centric System: An Industrial Experience Report (This work is described in Chapter 5)
  Lizhi Liao, Heng Li, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi, the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2023, San Francisco, CA, USA, Pages 1855-1866.

- Early Detection of Performance Regressions by Bridging Local Performance Data and Architectural Models (This work is described in Chapter 6)
  Lizhi Liao, Simon Eismann, Heng Li, Cor-Paul Bezemer, Diego Elias Costa, André van Hoorn, and Weiyi Shang, the 47th IEEE/ACM International Conference on Software Engineering (ICSE), 2025, Ottawa, ON, Canada, Accepted.

Abstract

Performance regression is an important type of performance issue in software systems. It indicates that the performance of the same features in the new version of the system becomes worse than that of previous versions, such as increased response time or higher resource utilization. In order to prevent performance regressions, current practices often rely on conducting extensive system performance testing before releasing the system into production based on the testing results.

However, faced with a great demand for resources and time to perform system performance testing, it is often challenging to adopt such approaches to software systems, especially large-scale systems, that employ the practice of fast-paced development and release cycles, e.g., DevOps. In addition, software performance regressions are often addressed late in the software development and release cycle, for instance, after the system is built, integrated, or even released. Not only does this make it laborious for developers to detect, locate, and fix performance regressions, but it can also lead to potential adverse effects on users and companies.

To tackle these key challenges, in this thesis, we focus on addressing software performance regressions in DevOps without relying on expensive system performance tests. In particular, in the first part of this thesis, we propose a series of automated approaches to helping developers and operators first detect performance regressions and then locate performance regression root causes directly based on analyzing field operational data, i.e., without the need for system performance testing. Finally, we adapt the existing performance analytic techniques in a real-life database-centric system. In the second part of this thesis, we leverage small-scale performance testing and architectural models to predict the impact of changes in small software components on the end-to-end performance of the entire system in order to detect performance regressions as early as in the software development phase.

Through various case studies on open-source projects and successful adoptions by our industrial research collaborator, we anticipate that our work can offer helpful insights to researchers and practitioners who are interested in addressing software performance regressions in DevOps without expensive system performance testing.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Dr. Weiyi Shang, for his invaluable guidance, mentorship, and unwavering support throughout my doctoral journey. Being one of his students is a really fortunate thing for me. His trust and support have instilled in me the confidence to overcome challenges and make progress. I have learned so much from him, not only how to be an excellent researcher, but also how to be a reliable friend, a good family member, and a person who embraces life with passion.

I also want to extend my sincere appreciation to the members of my dissertation committee, Dr. Arie Gurfinkel, Dr. Jialu Zhang, Dr. Shane McIntosh, and Dr. Federica Sarro, for dedicating their precious time to consider my work and providing insightful feedback and valuable guidance that greatly enriched the quality of this thesis.

I am so lucky to have worked and collaborated with some of the brightest researchers during my Ph.D. journey. I would like to thank all of my collaborators, Dr. Heng Li, Dr. Lei Ma, Dr. Tse-Hsun (Peter) Chen, Dr. Simon Eismann, Dr. Diego Elias Costa, Dr. Cor-Paul Bezemer, Dr. André van Hoorn, and all of my mates from the Software Engineering and System Engineering (SENSE) lab, Dr. Jinfu Chen, Dr. Maxime Lamothe, Dr. Zhenhao Li, Dr. Zishuo Ding, Yi Zeng, Yuanjie Xia, Haonan Zhang, Hetong Dai, Jianchen Zhao, Xiaoyu Cheng, Yixi Zhao, An Chen, and Hanying Shao, for the many inspiring discussions and fruitful collaborations. I learned so much from you all and enjoyed the moments we worked together.

In addition, I would like to thank ERA Environmental and all the research team members, Catalin Sporea, Andrei Toma, and Sarah Sajedi. I could not evaluate the impact and practical value of my thesis without the industrial environment and thoughtful feedback generously provided by the ERA Environmental team.

A special thanks to my family for their enduring encouragement and understanding during the demanding phases of my academic pursuit. Also, my deepest appreciation goes to my beloved wife, Yating, whose unwavering support, love, and encouragement sustained me through the highs and lows of this challenging endeavor.

Dedication

To my parents who gave me life and strength,
and to my wife who makes my life more beautiful and meaningful.

Table of Contents

Examining Committee Membership ii

Author's Declaration iii

Statement of Contributions iv

Abstract v

Acknowledgments vi

Dedication vii

List of Figures xiv

List of Tables xv

1 Introduction 1

1.1 Introduction 1

1.2 Research Hypothesis 2

1.3 Thesis Overview 3

1.4 Thesis Contributions 5

2 Literature Review **7**

2.1 Software Evolution and Performance 7

2.2 Performance Regression Detection 8

2.3 Locating Performance Regression Root Causes 11

2.3.1 Techniques based on performance testing 11

2.3.2 Techniques based on unit testing 12

2.3.3 Techniques based on field data 13

I Addressing Performance Regressions with Field Operational Data **14**

3 Detecting Performance Regressions in the Field Operations **15**

3.1 Introduction 16

3.2 Background: Black-Box Performance Models 18

3.3 Approaches 19

3.3.1 Extracting metrics 19

3.3.2 Building black-box performance models 23

3.3.3 Detecting performance regressions 24

3.4 Case Study Setup 26

3.4.1 Subject systems 27

3.4.2 Subject workload design 29

3.4.3 Subject models 32

3.5 Case Study Results 33

3.5.1 RQ1: How well can we model system performance under varying workloads? 33

3.5.2 RQ2: Can our approach detect performance regressions under varying workloads? 38

3.6 An Industrial Experience Report 42

3.6.1 Modeling system performance . . . 42
3.6.2 Detecting performance regressions . . . 43
3.7 Challenges and Lessons Learned . . . 43
3.7.1 C1: Determining the sampling frequency of performance metrics . . . 44
3.7.2 C2: Reducing the time cost of performance regression detection . . . 44
3.7.3 C3: Early detection of performance regressions . . . 45
3.8 Threats to Validity . . . 46
3.8.1 External validity. . . . 46
3.8.2 Internal validity. . . . 46
3.8.3 Construct validity. . . . 47
3.9 Conclusion . . . 48

4 Locating Performance Regression Root Causes in the Field Operations 49
4.1 Introduction . . . 50
4.2 Motivational Background from Industrial Partner . . . 52
4.3 Challenges . . . 54
4.3.1 Challenge 1: Understanding the relationship between the performance of a system and its runtime activities . . . 56
4.3.2 Challenge 2: Coping with the collinearity and redundancy among system runtime activities . . . 56
4.3.3 Challenge 3: Identifying problematic runtime activities related to performance regressions . . . 57
4.3.4 Challenge 4: Linking problematic runtime activities to code changes 58
4.3.5 Challenge 5: Increasing the ease of adoption . . . 59
4.4 Approach . . . 60
4.5 Evaluation . . . 65
4.5.1 Open-source systems and their workloads . . . 65
4.5.2 Injected performance regressions . . . 67
4.5.3 Experiment setup . . . 68

4.5.4 Evaluation results of open-source systems . . . 69
4.6 A Success Story from an Industrial Deployment . . . 76
4.7 Lessons Learned . . . 79
4.8 Threats to Validity . . . 86
4.8.1 External validity. . . . 86
4.8.2 Construct validity. . . . 86
4.8.3 Internal validity. . . . 87
4.9 Conclusion . . . 88

5 Adapting Existing Performance Analytic Techniques in a Real-Life Database-Centric System 89
5.1 Introduction . . . 90
5.2 Background . . . 92
5.3 Challenges and Lessons Learned . . . 94
5.3.1 C1: Exposing the runtime measurement of the database for performance analytics . . . 95
5.3.2 C2: Exposing the source code from the database for performance root-cause analysis . . . 97
5.4 Implemented Analysis . . . 100
5.4.1 Performance anomaly analysis within one release (PAA) . . . 100
5.4.2 Performance regression analysis between two releases (PRA) . . . 102
5.4.3 Root cause analysis (RCA) . . . 103
5.5 Detected Real-World Cases . . . 105
5.5.1 Detected Issue 1 (DI-1) . . . 107
5.5.2 Detected Issue 2 (DI-2) . . . 107
5.5.3 Detected Issue 3 (DI-3) . . . 108
5.5.4 Detected Issue 4 (DI-4) . . . 108
5.5.5 Detected Issue 5 (DI-5) . . . 109
5.5.6 Detected Improvement 6 (DI-6) . . . 109

5.6 Threats to Validity . . . 110
5.6.1 Construct validity. . . . 110
5.6.2 External validity. . . . 110
5.6.3 Internal validity. . . . 110
5.7 Conclusion . . . 111

II Early Detection of Performance Regressions during the Development Phase 112

6 Early Detection of Performance Regressions by Bridging Local Performance Data and Architectural Models 113

6.1 Introduction . . . 114
6.2 Background . . . 116
6.2.1 Software performance testing . . . 116
6.2.2 Model-based performance analysis . . . 117
6.3 Approach . . . 118
6.3.1 Collecting local performance data . . . 120
6.3.2 Identifying local performance deviations . . . 121
6.3.3 Propagating local performance deviations to architectural model . . . 121
6.3.4 Detecting system performance regressions . . . 124
6.4 Evaluation Setup . . . 125
6.4.1 Subject systems . . . 125
6.4.2 Evaluation environment . . . 126
6.4.3 Performance regressions . . . 126
6.4.4 Component-level and system-level performance test design . . . 127
6.4.5 Architectural models . . . 128
6.5 Evaluation Design and Results . . . 128
6.5.1 Evaluation design . . . 128

6.5.2 Data analysis . . . 129
6.5.3 Evaluation results . . . 130
6.6 Threats to Validity . . . 135
6.6.1 External validity. . . . 135
6.6.2 Internal validity. . . . 136
6.6.3 Construct validity. . . . 136
6.7 Conclusion . . . 136

7 Conclusion and Future Work **138**
7.1 Conclusion . . . 138
7.2 Future Research . . . 139
7.2.1 Proactive Software Performance Assurance . . . 139
7.2.2 Reduce the Length of Performance Testing with Production Data . . . 139
7.2.3 Performance Assurance of AI-Centric Software . . . 139

References **141**

List of Figures

1.1 The overall process of addressing performance regressions during the software development and release cycle 3

3.1 An overview of our studied approaches to detecting performance regressions. 22

4.1 An overall process of developing our approach in an industrial setting . . . 55

4.2 An overview of our approach to locating performance regression root causes 61

4.3 Screenshot of our Elastic Stack unified management platform 84

5.1 An overall engineering process of adapting performance analytic techniques in *EA* 94

6.1 An example software system with two subsystems and its corresponding architectural model 118

6.2 An overview of our approach for system performance regression detection by bridging local performance data and architectural models 119

6.3 An illustrative example of propagating local performance deviations to architectural model (for subfigures (a), (b), and (c), the red boxes indicate components with performance deviations, while for subfigure (d), the red box indicates updated performance parameter in the architectural model) . 123

List of Tables

2.1 Summary of prior studies on performance regression detection and localization 9

2.1 Summary of prior studies on performance regression detection and localization (cont.) 10

3.1 An illustrative example of log metrics and performance metric data from OpenMRS 21

3.2 Overview of the open-source subject systems and the industrial system. . . 27

3.3 Summary of the test actions with increased appearances (the ones with •) in different load drivers. 28

3.4 Performance regressions in the studied open-source systems 32

3.5 Summary of average CPU utilization of each version in our case studies. . . 32

3.6 Prediction error details for OpenMRS and Apache James under different workloads. 36

3.7 Performance regression detection results for OpenMRS and Apache James. 40

3.8 The earliest time for our approaches to detect regressions in the two open-source systems. 41

4.1 An overview of the open-source subject systems 66

4.2 Results of locating performance regression root causes for TeaStore, OpenMRS, and CloudStore 72

4.3 Model fit (i.e., R^2) of the linear regression models built to model the relationship between the appearances of web requests and the performance modeling errors for TeaStore, OpenMRS, and CloudStore 73

4.4 Statistical test results (i.e., p-value) of comparing the gap between the **effect** with regressions and the highest **effect** without regressions from different open-source systems and different performance regressions 74

4.5 Precision of the baseline approach of locating performance regression root causes for TeaStore, OpenMRS, and CloudStore 75

4.6 Results of locating performance regression root causes for the industrial system (i.e., *ES*). 77

4.7 An example of the influence of collinearity and redundancy among system runtime activities from TeaStore 81

5.1 The summary of real-world cases detected in the target industrial setting (i.e., *EA*) 106

6.1 An overview of our subject systems 125

6.2 Overall results of detecting end-to-end system performance regressions in the **fixed-workload** scenario 131

6.3 Overall results of detecting end-to-end system performance regressions in the **various-workload** scenario 134

Chapter 1

Introduction

1.1 Introduction

Performance is an essential quality attribute of software systems. It measures how effective a software system is with various metrics such as response time, throughput, or CPU utilization. A prior study [223] indicates that compared to functional issues, performance issues are more likely to cause field failures in large-scale software systems. As a common and important type of performance issue, performance regressions are the circumstances where the new version of a system still functions correctly, but offers a worse user experience (e.g., higher response time) and/or consumes extra resources (e.g., memory leak) compared to previous versions. Performance regressions may decrease user satisfaction, increase operating costs, and cause field failures [101]. This is especially critical for large-scale software systems, such as Amazon or Netflix, since they need to continuously provide high-quality services to millions or even billions of clients across the globe, and performance regressions in the system can cause serious financial and reputation losses [202, 49]. In addition, more and more software systems adopt DevOps practices that integrate development (Dev) and operations (Ops) for continuous and fast delivery (e.g., a new version is released every few hours [248]). However, such practices also pose major challenges in addressing performance regressions prior to the new release of software systems, since performance assurance activities often require a considerable amount of time and resources to conduct. Therefore, how to address performance regressions in DevOps, including detecting the existence of performance regressions and locating their corresponding root causes, is an important yet challenging task to prevent the new version of the software system from performing worse than previous versions.

Current practices that address performance regressions often rely on running system performance tests in the in-house testing environment prior to the deployment of every new version of a software system. There has been extensive prior research [208, 177, 159, 32, 176, 157, 195] that proposes various techniques, such as control charts, associations rules, and statistical models, that compare and analyze the performance testing results (e.g., performance metrics) in order to detect performance regressions and locate the corresponding root causes. However, system performance testing is often expensive in resources and time, making it hard to adopt in the DevOps process. It also requires a similar or even the same testing workload between performance testing different software versions, while the real-world workload is constantly varying and may even be completely different over time. Furthermore, to avoid the expensive performance testing on the entire system, unit testing [111, 155, 58] and field data [34, 233, 170, 120] are also leveraged in prior studies to address performance regressions. Nevertheless, unit testing-based approaches can only be aware of the performance of individual small-scale software components, while existing field data-based approaches often yield too coarse-grained results (e.g., only at the service level) to provide developers with enough information in fixing the regressions.

Different from traditional practices that often rely on expensive system-level performance testing to address software performance regressions, in this thesis, we aim to provide support to developers and operators by analyzing the data collected during software development and field operations without relying on system-level performance testing which requires extensive resources and a long time.

1.2 Research Hypothesis

Faced with the challenges of traditional practices that rely on running time- and resource-consuming system performance tests, prior research and our industrial experience lead to the following research hypothesis.

> **Research hypothesis:** By leveraging the abundant information during software development and field operations, we hypothesize that we can provide effective and timely support to developers and operators in addressing performance regressions in DevOps without expensive system performance testing.

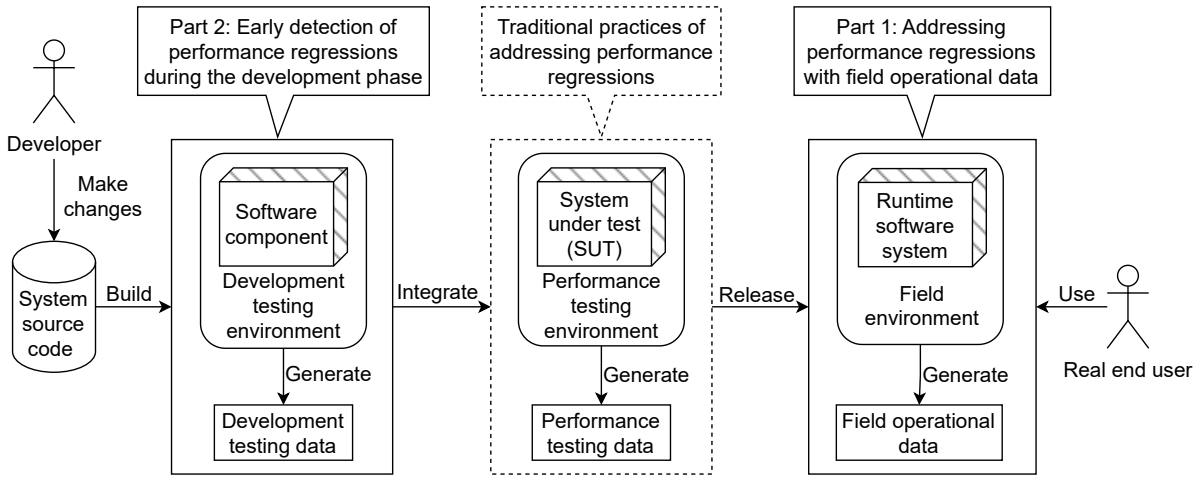


Figure 1.1: The overall process of addressing performance regressions during the software development and release cycle

We validate our research hypothesis through two parts. In the first part, we attempt to address software performance regressions directly based on the operational data when the system is deployed in the field. In the second part, we intend to detect software performance regressions early by analyzing the data collected during the software development phase.

1.3 Thesis Overview

As summarized in Figure 1.1, addressing performance regressions may be related to different phases of the software development and release cycle. Unlike traditional practices that rely on expensive system-level performance testing, in this thesis, we aim to provide support to developers and operators in addressing performance regressions without relying on system-level performance testing in two parts: 1) by analyzing the operational data when the system is deployed in the field (cf. Part 1 in Figure 1.1) and 2) by analyzing the data collected during the software development phase (cf. Part 2 in Figure 1.1).

In Chapter 2, we present a literature review on addressing software performance regressions. We survey and compare the literature along the following three dimensions: 1) software evolution and performance, 2) performance regression detection, and 3) locating performance regression root causes.

Part I of this thesis focuses on addressing performance regressions with field operational

data. This part consists of Chapter 3, Chapter 4, and Chapter 5.

In Chapter 3, inspired by some existing software testing techniques, such as A/B testing or canary releasing, which rely on the end user's data in the field for software quality assurance, we directly detect performance regressions based on the field operational data while a software system is deployed and running in the field. Our approaches leverage black-box models to capture the relationship between the performance of a software system (e.g., CPU usage) under varying workloads and the runtime activities that are recorded in the readily available logs. Then, our approaches compare the black-box models derived from the current software version with an earlier version to detect performance regressions between these two versions. Our experimental results show that such black-box models can effectively and timely detect real performance regressions and injected ones under varying workloads that are unseen when training these models.

In Chapter 4, we propose an approach that can automatically locate the root causes of performance regressions while the software systems are deployed and running in the field. First, our approach uses black-box performance models to capture the relationship between the performance of a system and its runtime activities. Then, our approach analyzes the performance models and uses statistical techniques to suggest the problematic system runtime activities (i.e., the root causes) that are related to a performance regression. From our experiment results, we find that our approach can successfully locate both the injected synthetic and real-world performance regressions. In addition, we share the challenges that we encountered during the industrial adoption, how we addressed those challenges, and the lessons that we learned during the process to benefit practitioners and researchers who wish to leverage the field operation data of a software system to conduct performance assurance activities.

In Chapter 5, we target to adapt performance analytic techniques in a large-scale database-centric system from our industrial collaborator. Our adapted performance analysis pays special attention to the database and the interactions between the database and the application with minimal reliance on expert knowledge and manual effort. Moreover, we document our encountered challenges and how they are addressed during the development and adoption of our solution in the industrial setting as well as the corresponding lessons learned. We also discuss the real-world performance issues detected by applying our analysis to the target database-centric system. We anticipate that our solution and the reported experience can be helpful for practitioners and researchers who would like to ensure and improve the performance of database-centric systems.

Part II of this thesis focuses on early detection of performance regressions as early as in the development phase. This part consists of Chapter 6.

In Chapter 6, we propose a novel approach to the early detection of performance regressions by bridging the local performance data generated by component-level testing and the system-level architectural models. Our approach uses local performance data to identify deviations at the component level, and then propagate these deviations to the architectural model. We then use the architectural model to predict regressions in the performance of the overall system. In an evaluation of our approach on two representative open-source benchmark systems, we show that it can effectively detect end-to-end system performance regressions from local performance deviations with different intensities and under various system workloads. More importantly, our approach can detect regressions as early as in the development phase, in contrast to existing approaches that require the system to be fully built and deployed. Our approach is lightweight and can complement traditional system performance testing when testing resources are scarce.

Finally, in Chapter 7, we conclude the thesis and discuss the directions for future research.

1.4 Thesis Contributions

In this thesis, we demonstrate that by leveraging the abundant information during software development and field operations, we can provide effective and timely support to developers and operators in addressing performance regressions in DevOps without expensive system performance testing.

In particular, the main contributions of this thesis are as follows:

- We demonstrate the effectiveness of using black-box machine learning models to model the performance of a system and detect performance regressions in the field where workloads are constantly evolving (i.e., two releases of the same system are never running the same workloads).
- We propose an approach that can automatically assist developers in identifying and fixing the root causes of performance regressions without the need for resource- and time-consuming system performance testing.
- We provide an industrial experience report that discusses the encountered challenges, proposed solutions, and the learned lessons when adapting performance analytic techniques in a real-world database-centric system.

- We develop a novel approach to the early detection of performance regressions by bridging the local performance testing data and the system architectural models.

Chapter 2

Literature Review

This chapter presents a literature review on addressing software performance regressions. We survey and compare the literature along the following three dimensions: 1) software evolution and performance, 2) performance regression detection, and 3) locating performance regression root causes. In particular, we summarize the literature on performance regression detection and localization in Table 2.1, including the category of the approach proposed in prior work, the analyzed data type, a brief description, and their major limitations.

2.1 Software Evolution and Performance

Software performance issues are essentially related to software evolution, emerging as a consequence of incremental changes, updates, and expansions made to the software over time [30, 246, 31, 145, 49, 47, 60]. These issues may also cause serious financial and reputational losses [202, 49] to both users and companies. Therefore, understanding how software performance evolves across various software revisions is of key importance.

In an effort to achieve this, Bergel et al [47] propose three visualizations, including the structural blueprint, the behavioral blueprint, and the evolutionary blueprint to compare the runtime performance information during the execution of different versions of a software system. In addition, Alcocer and Bergel [30] conduct an empirical study on the performance evolution of 19 applications. Their findings show that every three code revisions introduce a performance variation and the performance variations can be classified into some patterns. Based on these patterns, Zhou et al. [246] propose an approach called DeepTLE that can

predict the performance of submitted source code before executing the system, based on the semantic and structural features of the submitted source code.

While existing source-code level techniques aim to identify performance issues before running performance tests, we believe that the performance of the whole system is not just a simple combination of each code snippet's performance. Instead, the performance of the whole system needed to be measured by performance testing or during the field operations by real end uses. In this thesis, we aim to propose various automated approaches to identifying software performance regressions and the corresponding root causes when the system is under development or running in the field, which can complement or even replace traditional system performance testing.

2.2 Performance Regression Detection

Prior research designs various types of approaches to detecting software performance regressions. The most straightforward approaches to detecting performance regressions are based on directly comparing performance metric values in two versions [173, 176, 157, 177, 58]. This type of research adopts statistical techniques, such as control charts [173, 176] and statistical tests [157, 58] to compare and analyze performance metrics to detect performance regressions. For instance, Nguyen et al. [173] leverage use control charts to analyze performance metrics across test runs to detect performance regressions. Malik et al. [157] propose to use principle component analysis (PCA) to reduce the large number of performance metrics to a smaller set, in order to reduce the efforts of performance analysts.

Prior research also investigates the relationship among performance metrics and/or system logs and uses the deviation of such relationship as indicators of software performance regressions [208, 110, 241, 90]. For example, Foo et al. [90] build association rules between multiple performance metrics to detect performance regressions. Syeret al. [208] also propose to combine execution logs and performance metrics to diagnose memory-related performance issues.

In addition, data-mining-based and statistical techniques like clustering [208, 110] and linear regression models [241] are also leveraged to mine such a relationship between execution logs and performance metrics. For example, He et al. [110] correlate the clusters of log sequences with system performance metrics to identify impactful system problems (e.g., request latency and service availability).

Performance models (e.g., queue models [41], rule-based models [173, 176], and machine learning models [233, 195, 83]) are also widely leveraged as vehicles to capture performance

Table 2.1: Summary of prior studies on performance regression detection and localization

Approach category	Prior work	Analyzed data type						Description	Limitation						
		Performance metrics	Logs	Source code	Unit tests and results	Profiling data	Injected trace data		Cannot locate root cause	Coarse-grained root cause	Target specific performance issue patterns	Expensive testing	Not aware of system-level performance	Not consider varying workloads	Expect system knowledge
Based on performance testing	[173]	X						Using control charts to compare and analyze performance metrics to identify performance regressions	X	n/a		X		X	
	[41]	X						Modeling the software system using a two-layer queuing model to find the performance bottlenecks in the system	X	n/a		X		X	X
	[176]	X						Comparing performance counters across test runs using control charts to detect performance regressions	X	n/a		X		X	
	[157]	X	X					Using machine learning techniques (e.g., K-Means, PCA) to find the best subset of performance counters and build models for performance deviation prediction	X	n/a		X		X	
	[208]	X	X					Adopting hierarchical clustering on time-slices data consists of events and performance counters and analyzing the influence of outliers in clusters to locate anomalous events		X	X	X		X	
	[177]	X						Mining regression root cause repository and using classification techniques to match the behavior of a new version to prior tests to locate performance regression root causes		X		X		X	
	[90]	X						Adopting association rule to capture performance patterns and ensemble learning to detect performance regressions in heterogeneous environments	X	n/a		X		X	
	[195]		X					Clustering various performance counters into groups and build regression models on the clusters to detect performance regressions	X	n/a		X		X	
	[159]	X				X		Proposing a subsuming approach that aggregates performance costs repeated patterns of method calls to identify the patterns that cause performance regressions			X	X		X	
	[32]	X				X		Proposing a sampling technique called horizontal profiling that assigns a cost to each source code change based on the run-time performance testing history				X		X	
Based on unit testing	[111]			X	X			Analyzing the unit testing results during the development phase and utilizing dynamic code analysis technique to extract the call tree information from the source code revision history to identify the commits that introduce the performance regressions					X	X	
	[155]			X	X	X		Using a combination of search-based input profiling and change impact analysis techniques across software versions to identify code changes that are potentially responsible for performance regressions					X	X	

Note: The value "n/a" of the "Coarse-grained root cause" under the "Limitations" column indicates that if the prior work cannot locate the root cause, then it is not applicable to the "Coarse-grained root cause" limitation.

Table 2.1: Summary of prior studies on performance regression detection and localization (cont.)

Approach category	Prior work	Analyzed data type						Description	Limitation						
		Performance metrics	Logs	Source code	Unit tests and results	Profiling data	Injected trace data		Cannot locate root cause	Coarse-grained root cause	Target specific performance issue patterns	Expensive testing	Not aware of system-level performance	Not consider varying workloads	Expect system knowledge
Based on unit testing	[58]			X	X			Adopting unit tests and performance micro-benchmarks to locate performance regression introducing code changes					X	X	
	[183]			X	X			Selecting unit tests that are associated with code changes by static code analysis and measuring the performance of such tests in different versions to locate performance regression root causes					X	X	
	[84]			X	X			Utilizing the tests from the release pipeline to identify performance issue root causes during the rapid development process					X	X	
Based on field data	[34]		X					Abstracting the states of idleness observed in a running program based on a set of rules defined declaratively by an expert to diagnosis the root cause of idle time in applications			X				X
	[233]	X	X					Constructing performance models capturing the relationship between application performance and system resources to locate possible performance bottleneck locations		X					
	[170]	X	X					Proposing an machine learning based approach that utilize unsupervised learning algorithms, i.e., affinity propagation clustering, on the monitored performance counters and system execution logs to locate anomalies on cloud-hosted web applications		X					
	[120]	X					X	Injecting metadata to events at each level of PaaS stack and adopting statistical approaches (e.g., change point analysis, relative importance) to identify the workload change or the cloud service that is responsible for the performance regressions		X					
	[153]		X					Analyzing the Spark execution log by extracting the features related to system runtime performance (e.g., execution time, garbage collection) and utilizing the weighted factors to decide the probability of the root causes		X					
	[110]	X	X					Proposing a cascading clustering approach to cluster log sequences and correlating clusters with performance metrics to identify performance problems		X					
	[160]	X						Combining machine learning with graph centrality algorithms to locate the nodes spreading the performance anomalies		X					
	Our thesis	X	X	X	X			Propose approaches consider both software development data and runtime field data to effectively assist developers and operators in identifying the existence and root causes of performance regressions without expensive system performance testing							

Note: The value "n/a" of the "Coarse-grained root cause" under the "Limitations" column indicates that if the prior work cannot locate the root cause, then it is not applicable to the "Coarse-grained root cause" limitation.

regressions. For example, Shang et al. [195] propose to cluster various types of performance counters into groups and build regression models using machine learning techniques on the clusters to detect performance regressions.

Recent research advocates the use of smaller-scale tests, such as micro-performance benchmarks and even functional tests, to detect performance regressions [58, 206, 183, 84]. For example, Reichelt et al. [183] select unit tests that are associated with code changes by static code analysis and measure the performance of such tests in different versions to locate software performance regressions.

To conclude, we find that most of the prior research on the detection of performance regressions is designed to be conducted on the data generated from time and resource-consuming system performance testing with predefined workloads, or depend on small-scale performance testing which can only be aware of the performance of individual components rather than the entire software system. However, this thesis considers the problem of the real workloads in the field that are continuously varying and relies on the data that is generated in the field from end users or during the development phase to detect the performance regressions of the entire software system without the need for expensive system performance testing.

2.3 Locating Performance Regression Root Causes

Depending on different types of data sources and application contexts, the automated approaches for locating performance regression root causes can be broken down into three categories: 1) techniques based on performance testing; 2) techniques based on unit testing; and 3) techniques based on field data.

2.3.1 Techniques based on performance testing

Various prior research for locating the root causes of performance regressions is based on performance testing the target software system [208, 177, 159, 32]. Such approaches compare the performance testing results (e.g., performance metrics) from different versions of the software system to locate performance regression root causes.

For example, Nguyen et al. [177] propose to mine the regression root cause repository that stores the past performance testing results and performance regressions. They use classification techniques (e.g., Bayes Classifier [126]) to match the behavior of a new version to the behavior of prior tests to locate the performance regression root causes.

However, these approaches suffer some common limitations: 1) in order to compare the performance testing results from different versions, a similar or even the same testing workload is needed. However, it may be difficult to design load test cases that represent the field workloads. In particular, there may exist special real-world cases where the field workload is completely different from other workloads (e.g., when a popular site like Reddit or Twitter directs to a small site, the throughput of that small site would be much greater than the average daily workload, a phenomenon also known as the Slashdot effect [26]). The impact of different workloads between the old and new versions of the system is not considered in these approaches; 2) in practice, performance testing is time and resource-consuming, which requires expensive computing resources in the testing environment and excessive time to execute the tests (e.g., from hours to even days).

2.3.2 Techniques based on unit testing

In order to avoid the need for expensive performance testing on the entire software system, prior work adopts unit tests and combines the testing results with static or dynamic source code analysis techniques to pinpoint performance regression root causes [111, 155, 58, 183, 84].

For instance, Ding et al. [84] propose to study the feasibility of using the JUnits tests that are readily available in the release pipeline as small-scale performance testing. Based on their results, they also identify some important metrics that hinder the unit test's capability to test system performance. Heger et al. [111] analyze the unit testing results during the development phase and utilize dynamic code analysis techniques to extract the call tree information from the source code revision history to identify the commits that introduce the performance regressions. Similarly, Chen et al. [58] utilize unit tests and performance micro-benchmarks on every commit to collect multiple performance metrics, such as response time and CPU utilization, to identify the code changes that may introduce performance regressions.

However, due to the nature that these approaches are based on small-scale testing, they cannot capture the performance of the overall software system and are only aware of the performance of individual software components. Similar to performance testing-based approaches, they do not take the impact of the continuously varying system workloads into consideration either.

2.3.3 Techniques based on field data

Prior research also proposes to use system runtime information (e.g., execution logs and performance metrics) collected directly from the field to locate performance regression root causes [61, 34, 233, 170, 120, 153, 110, 160, 128].

For example, Nair et al. [170] propose a machine learning-based approach that utilizes unsupervised learning algorithms, i.e., affinity propagation clustering, constructed on the monitored runtime performance metrics and system execution logs to locate anomalies in cloud-hosted web applications. Lu et al. [153] provide an approach that analyzes the Spark execution logs to extract the features related to system runtime performance (e.g., execution time, memory usage, and garbage collection) and utilize a weighted combination of certain specific cause related factors to determine the probability of the root causes.

Such field data-based approaches benefit from saving the effort and resources for performance testing, and they also consider the system-level performance under the impact of varying workloads. However, these approaches may suffer from the limitations that the located root causes are too coarse-grained (e.g., only at service level) [34, 233, 170, 120, 153, 110], which can only assist IT operators (instead of developers) to locate the high-level service or modules with performance problems but may not be sufficient to support developers to locate the specific root causes (e.g., at class or method level), or only target specific performance regression patterns [34].

Different from prior research, in this thesis, we aim to propose a collection of approaches that leverage both software development data (e.g., local unit testing data) and field runtime data (e.g., web-access logs and performance metrics) while just requiring minimum knowledge about the internal behaviors of the system to effectively assist developers and operators in identifying and fixing the root causes of performance regressions.

Part I

Addressing Performance Regressions with Field Operational Data

Chapter 3

Detecting Performance Regressions in the Field Operations

Performance regressions of large-scale software systems often lead to both financial and reputational losses. In order to detect performance regressions, performance tests are typically conducted in an in-house (non-production) environment using test suites with predefined workloads. Then, performance analysis is performed to check whether a software version has a performance regression against an earlier version. However, the real workloads in the field are constantly changing, making it unrealistic to resemble the field workloads in predefined test suites. More importantly, performance testing is usually very expensive as it requires extensive resources and lasts for an extended period.

In this chapter, we leverage black-box machine learning models to automatically detect performance regressions in the field operations of large-scale software systems. Practitioners can leverage our approaches to complement or replace resource-demanding performance tests that may not even be realistic in a fast-paced environment. Our approaches use black-box models to capture the relationship between the performance of a software system (e.g., CPU usage) under varying workloads and the runtime activities that are recorded in the readily available logs. Then, our approaches compare the black-box models derived from the current software version with an earlier version to detect performance regressions between these two versions. We performed empirical experiments on two open-source systems and applied our approaches to a large-scale industrial system. Our results show that such black-box models can effectively and timely detect real performance regressions and injected ones under varying workloads that are unseen when training these models. Our approaches have been adopted in practice to detect performance regressions of a large-scale industry system on a daily basis.

3.1 Introduction

Many large-scale software systems (e.g., Amazon, Google, Facebook) provide services to millions or even billions of users every day. Performance regressions in such systems usually lead to both reputational and monetary losses. For example, a recent incident of performance regressions at Salesforce affected eight data centers, resulting in a negative impact (i.e., slowdown and failures of the provided services) on the experience of clients across many organizations [54]. It is reported that 59% of Fortune 500 companies experience at least 1.6 hours of downtime per week [211], while most of the field failures are due to performance issues rather than feature issues [223]. To avoid such performance-related problems, practitioners perform in-house performance testing to detect performance regressions of large-scale software systems [122].

For in-house performance testing, load generators (e.g., JMeter) are used to mimic thousands or millions of users (i.e., the *workload*) accessing the system under test (SUT) at the same time. Performance data (e.g., CPU usage) is recorded during the performance testing for later performance analysis. To determine whether a performance regression exists, the performance data of the current software version is compared to the performance data from running an earlier version on the same workload, using various analysis techniques [93] (e.g., control charts [176]). However, such in-house performance testing suffers from two important challenges. First, the real workloads in the field are constantly changing and difficult to predict [209], making it unrealistic to test such dynamic workloads in a predefined manner. Second, performance testing is usually very expensive as it requires extensive resources (e.g., computing power) and lasts for a long time (e.g., hours or even weeks). For software systems that follow fast release cycles (e.g., Facebook is released every few hours), conducting performance testing in a short period of time becomes particularly difficult.

There exist software testing techniques such as A/B testing [235] and canary releasing [194] that depend on the end user's data in the field for software quality assurance. Similarly, in this work, we propose to directly detect performance regressions based on the end users' data of large-scale software systems when they are deployed in the field. Such an automated detection can complement or even replace typical in-house performance testing when testing resources are limited (e.g., in an agile environment) and/or the field workloads are challenging to reproduce in the testing environment. In particular, we leverage sparsely sampled performance metrics and readily available execution logs from the field to detect performance regressions, which only add negligible performance overhead to SUT.

Inspired by prior research [240, 88], we use black-box machine learning and deep learning techniques (i.e., CNN, RNN, LSTM, Linear Regression, Random Forest, and XGBoost) to

capture the relationship between the runtime activities that are recorded in the readily-available logs of a software system (i.e., the independent variables) and its performance under such activities (i.e., the dependent variables). We use the black-box model which describes the current version of a software system and the black-box model which describes an earlier version of the same software system, to determine the existence of performance regressions between these two versions.

Prior research that builds black-box models to detect performance regressions [240, 88] typically depends on the data from in-house performance testing, where the workload is consistent between two releases. However, when using the field data from the end users, one may not assume that the workloads from the two releases are the same. In order to study whether the use of black-box performance models under such inconsistent workloads may still successfully detect performance regressions, we performed empirical experiments on two open-source systems (i.e., OpenMRS and Apache James) and applied our approaches to a large-scale industrial system. In particular, our study aims to answer two research questions (RQs):

RQ1: *How well can we model system performance under varying workloads?*
In order to capture the performance of a system under varying workloads, we built six machine learning models (including three traditional models and three deep neural networks). We found that simple traditional models can effectively capture the relationship between the performance of a system and its dynamic activities that are recorded in logs. In addition, such models can equivalently capture the performance of a system under new workloads that are unseen when building the models.

RQ2: *Can our approaches detect performance regressions under varying workloads?*
Our black-box model-based approaches can effectively detect both the real performance regressions and injected ones under varying workloads (i.e., when the workloads are unseen when training the black-box models). Besides, our approaches can detect performance regressions with the data from a short period of operations, enabling early detection of performance regressions in the field.

Our work makes the following key contributions:

- Our study demonstrates the effectiveness of using black-box machine learning models to model the performance of a system when the workloads are evolving (i.e., the model trained on a previous workload can capture the system performance under new unseen workloads).

- Our experimental results show that we can use black-box machine learning models to detect performance regressions in the field where workloads are constantly evolving (i.e., two releases of the same system are never running the same workloads).
- Our approaches can complement or even replace traditional in-house performance testing when testing resources are limited.
- We shared the challenges and the lessons that we learned from the successful adoption of our approaches in the industry, which can provide insights for researchers and practitioners who are interested in detecting performance regressions in the field.

The remainder of the chapter is organized as follows. Section 3.2 introduces the background of black-box performance modeling. Section 3.3 outlines our approaches for detecting performance regressions in the field. Section 3.4 and Section 3.5 present the setup and the results of our case study on two open-source subject systems, respectively. Section 3.6 shares our experience of applying our approaches on a large-scale industrial system. The challenges and the lessons that we learned from the successful adoption of our approaches are discussed in Section 3.7. Section 3.8 discusses the threats to the validity of our findings. Finally, Section 3.9 concludes the chapter.

3.2 Background: Black-Box Performance Models

Performance models are built to model system operations in terms of resources consumed. Performance models can be used to predict the performance of systems in the purpose of workload design [138, 236, 210], resources control [96, 74], configuration selection [103, 215, 104], and anomaly detection [118, 88, 94]. Performance models can be divided into two categories [83]. One is white box performance models, which are also called analytical models. White box performance models predict performance based on the assumption of system contexts, i.e., workload intensity and service demand.

Another type of performance model is the black-box performance model. Black-box performance models employ statistical modeling or machine learning techniques to predict system performance by taking the system as a black box. Black-box performance models typically require no knowledge about the system's internal behavior [83, 93]. Such models apply various machine learning algorithms to model a system's performance behavior taking system execution logs or performance metrics as input. The main strength of using black-box modeling techniques is to model the system performance when the system is under field operations. While white-box performance models typically require knowledge

about the system's internal behavior, such knowledge is often not available when the system is deployed in the production environment. Thus, under the production environment, black-box approaches appear to be more effective than white-box modeling approaches.

Typical examples of black-box performance models are linear regression models [195, 240]. For example, prior studies use linear regression models to capture the relationship between a target performance metric (e.g., CPU usage) and the system operations [240] or other performance metrics (e.g., memory utilization) [195]:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n \tag{3.1}$$

where the response variable y is the target performance metric, while the explanatory variables $(x_1, x_2, ..., x_n)$ are the frequency of each operation of the system [240] or each of the other performance metrics [195]. The coefficients $\beta_1, \beta_2, ..., \beta_n$ describe the relationship between the target performance metric and the corresponding explanatory variables.

Linear regression requires that the input data satisfies a normal distribution. With the benefit of being applicable to data from any distribution, regression tree [233] has been used to model system performance. Regression tree models a system's performance behavior by using a tree-like structure. Quantile regression [79] can model a system's different phases of performance behavior.

The prior success of black-box performance modeling supports our study to detect performance regressions under variable workloads from the field.

3.3 Approaches

In this section, we present our three approaches that automatically detect performance regression in a new version of a software system based on the logs and performance data that are collected under varying workloads. Our approaches contain three main steps: 1) extracting metrics, 2) building black-box performance models, and 3) detecting performance regressions. The three approaches share the first two steps while being different in the third step. The overview of our studied approaches is shown in Figure 3.1.

3.3.1 Extracting metrics

We aim to identify whether there is performance regression in the new version of the system based on modeling the relationship between the system performance (e.g., CPU usage) and

the corresponding logs that are generated during system execution. Specifically, we collect system access logs that are readily generated during the execution of web servers, such as the Jetty, Tomcat, and IIS (Internet Information Services). Those logs represent the workload of the system during a period of execution. The performance data is sparsely sampled at a fixed frequency (e.g., every 30 seconds).

In this step, we extract log metrics and performance metrics from the system access logs and the recorded performance data, respectively.

Splitting logs and performance data into time periods

We would like to establish the relationship between the execution of the system and the performance of the system during run time. Since both performance data and logs are generated during system runtime and are not synchronized, i.e., there is no corresponding record of performance metrics for each line of logs, we would first align the logs and records of performance data by splitting them into time periods. For example, one may split a two-hour dataset into 120 time periods where each time period is one minute in the data. Each log line and each record of performance data are allocated into their corresponding time period.

Extracting log metrics and performance metrics

We parse the collected logs into events and their corresponding time stamps. For example, a line of web log "*[2019-09-27 22:43:13] GET /openmrs/ws/rest/v1/person/ HTTP/1.1 200*" will be parsed into the corresponding web request or URL "*GET /openmrs/ws/rest/v1/person/*" and time stamp "*2019-09-27 22:43:13*". Afterward, each value of a log metric is the number of times that each log event executes during the period. For example, if the log event "*GET /openmrs/ws/rest/v1/person/*" is executed 10 times during a 30-second time period, the corresponding log metric for "*GET /openmrs/ws/rest/v1/person/*"'s value is 10 for that period. Then, we consider the aggregation (e.g., taking the average) of the records of performance data as the value of the performance metric of the time period, similar to prior research [89]. Table 3.1 shows an illustrative example of log metrics and performance metrics in five 30-second time periods.

These log metrics together with the performance metrics in the corresponding time periods are used to construct our performance models. Each data point of a log metric is based on a time period (e.g., 30 seconds) instead of a log entry. For example, if we run the subject system for eight hours and we take each 30 seconds as a time period, there will

be 960 data instances in total: 8 (hours) * 60 (minutes per hour) * 2 (time periods per minute).

Table 3.1: An illustrative example of log metrics and performance metric data from OpenMRS

	Log metrics*		Performance metric
Time slice	GET person	GET observation	CPU
1 sec. - 30 sec.	17	18	74.19
31 sec. - 60 sec.	10	2	52.25
61 sec. - 90 sec.	5	19	64.61
91 sec. - 120 sec.	6	6	56.26
121 sec. - 150 sec.	0	3	55.57

* In this example, we show the log metrics that are used in the linear regression, random forest, and XGBoost models. For CNN, RNN, and LSTM, we sort the logs in each time period by their corresponding time stamp to create a sequence of log events.

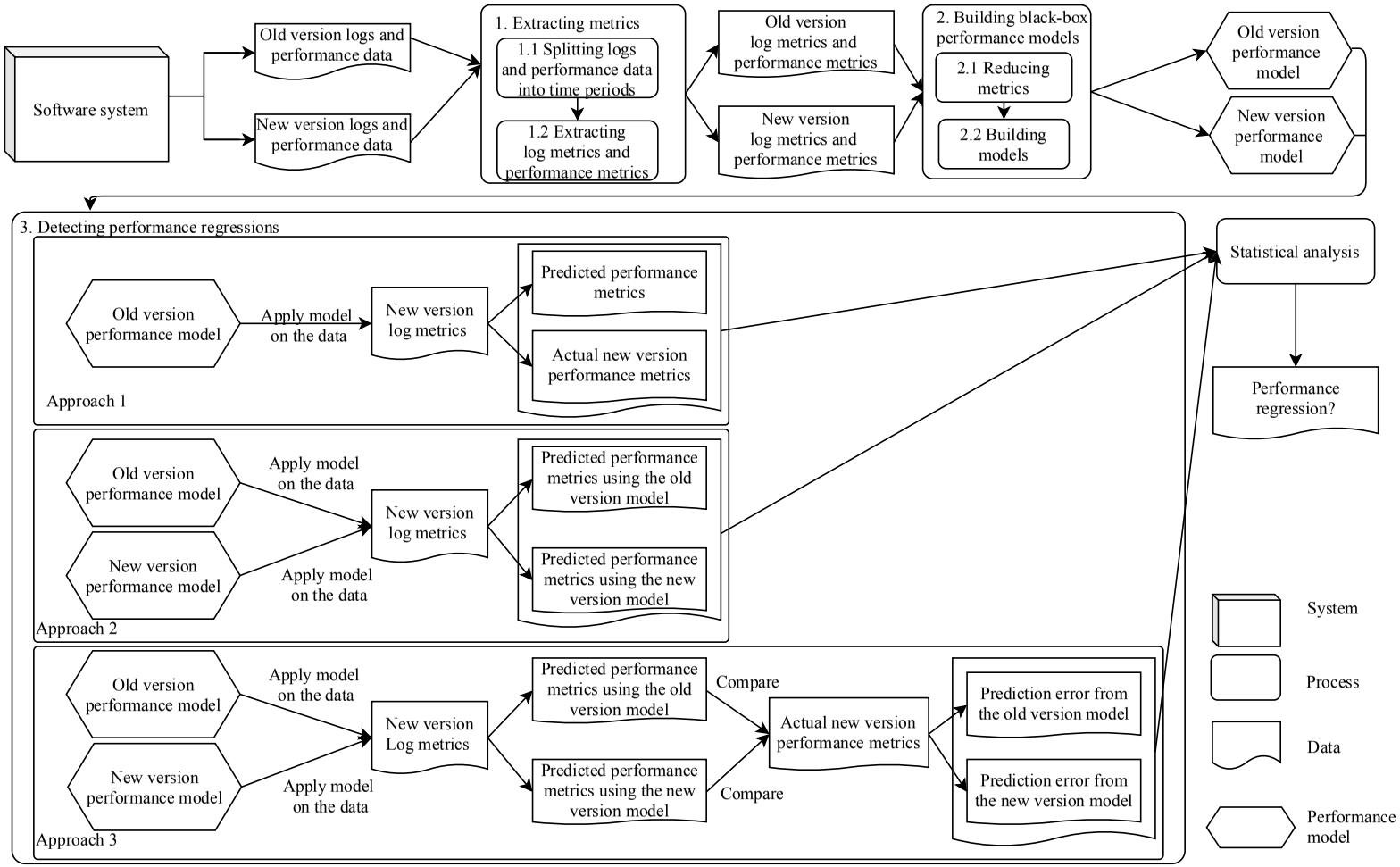


Figure 3.1: An overview of our studied approaches to detecting performance regressions.

3.3.2 Building black-box performance models

In this step, we use the log metrics and performance metrics extracted from the last step to build black-box performance models. The log metrics and the performance metrics (e.g., CPU usage) are used as the independent variables and dependent variables of the models, respectively.

Reducing metrics

The frequency of some log events (e.g., periodical events) may not change over time. The constant appearance of such events may not provide information about the changes in system workloads. Therefore, after we calculate the log metrics of each log event, we reduce log metrics by removing redundant log metrics or log metrics with constant values in both previous and current versions. We first remove log metrics that have zero variance in both versions of the performance tests.

Different log events may always appear at the same time, e.g., the user logging in and checking the user's privilege, and providing repetitive information for the workloads. To avoid bias from such repetitive information, we then perform correlation analyses and redundancy analyses on the log metrics. We calculate Pearson's correlation coefficient [46] between each pair of log metrics (i.e., in total N*N/2 times of correlation calculations where N refers to the number of log metrics). If a pair of log metrics has a correlation higher than 0.7, we remove the one that has a higher average correlation with all other metrics. We repeat the process until there exists no correlation higher than 0.7. The redundancy analysis would consider a log metric redundant if it can be predicted from a combination of other log metrics. We use each log metric as a dependent variable and use the rest of the log metrics as independent variables to build a regression model. We calculate the R^2 of each model. If the R^2 is larger than a threshold (e.g., 0.9), the current dependent variable (i.e., the log metric) is considered redundant. We then remove the log metric with the highest R^2 and repeat the process until no log metrics can be predicted with an R^2 higher than the threshold.

We only apply this step when using traditional statistical models or machine learning models (like linear regression or random forest), while if a deep neural network (like convolutional neural network or recurrent neural network) is adopted to build the black-box performance models, we skip this step.

Building models

We build models that capture the relationship between a certain workload that is represented by the logs and the system performance. In particular, the independent variables are the log metrics from the last step and the dependent variable is the target performance metric (e.g., CPU usage). One may choose different types of statistical, machine learning, or deep learning models, as our approach is agnostic to the choice of models. However, the results of using different types of models may vary (cf. RQ1).

3.3.3 Detecting performance regressions

The goal of building performance models is to detect performance regressions. Therefore, in this step, we use the black-box performance models that are built from an old version of the system to predict the expected system performance of a new version. Then, we use statistical analysis to determine whether there exists performance deviance based on prediction errors of the models. The input of this step is the black-box performance models built from the old version and the new version; the output of this step is the determination of whether there exists performance deviance between the old version and the new version of the system.

Intuitively, one may use the model that is built from the old version of the system to predict the performance metrics from running the new version of the system. By measuring the prediction error, one may be able to determine whether there exists performance deviance [70, 176]. However, such a naive approach may be biased by the choice of thresholds that are used to determine whether there is performance deviance. For example, a well-built performance model may only have less than 5% average prediction error; while another less fit performance model may have 15% average prediction error. In these cases, it is challenging to determine whether an average prediction error of 10% on the new version of the system should be considered as a performance regression. Therefore, statistical analyses are used to detect performance regressions in a systematic manner [93, 195, 90].

In particular, we leverage three approaches to detect performance regressions: *Approach 1*) by comparing the predicted performance metrics (using the model built from the old version) and the actual performance metrics of the new version, *Approach 2*) by comparing the predicted performance metrics of the new version using the model built from the old version and the model built from the new version, and *Approach 3*) by comparing the prediction errors of the performance metrics on the new version using the model built from the old version and the model built from the new version. We describe each approach in detail in the rest of this subsection.

Approach 1: *comparing the predicted performance metrics (using the model built from the old version) and the actual performance metrics of the new version.* The most intuitive way of detecting performance regression is to compare the predicted value and the actual value of performance metrics. Since the model is built from the old version of the system, if the actual value of the performance metrics from the new version of the system is higher than the predicted value and the difference between them is large, we may consider the existence of performance regressions. In particular, we use the data from the old version of the system, i.e., $Data_{old}$ to build a black-box performance model $Model_{old}$ (cf. Section 3.3.2). Afterward, we apply the model $Model_{old}$ on the data from the new version of the system, i.e., $Data_{new}$. Then we compare the predicted and the actual values of the performance metrics.

Approach 2: *comparing the predicted performance metrics of the new version using the model built from the old version and the model built from the new version.* Since our approach aims to be applied to varying workloads, the performance regression may only impact a small number of time periods, while the source code with performance regressions may not be executed in other time periods. Therefore, only the time periods that are impacted by the performance regressions may contain large prediction errors. To address such an issue, we also built a performance model $Model_{new}$ using the data from the new version of the system ($Data_{new}$). This way, $Model_{new}$ is built using the data with potential performance regressions. Afterward, we use $Model_{old}$ and $Model_{new}$ to predict the performance metrics in $Data_{new}$.

Since $Model_{new}$ is built from $Data_{new}$, there exists a bias when applying $Model_{new}$ on $Data_{new}$, as the training data and testing data are exactly the same (a model without any generalizability can just remember all the instances in the training data and make perfect predictions). To avoid bias, instead of building one $Model_{new}$, we build n models, where n is the number of data points that exist in $Data_{new}$. In particular, for each data point in $Data_{new}$, we build a performance model $Model^n_{new}$ by excluding that data point and applying the model $Model^n_{new}$ on the excluded data point. Therefore, for n data points from $Data_{new}$, we end up having n models and n predicted values.

Finally, we compare the predicted values using $Model_{old}$ on $Data_{new}$, and the predicted values by applying each $Model^n_{new}$ on each of the n data points in $Data_{new}$. Similar to approach 1, if the predicted values by applying each $Model^n_{new}$ on each of the n data points in $Data_{new}$ are higher than predicted values using $Model_{old}$ on $Data_{new}$ and the difference is large, we may consider the existence of performance regressions.

Approach 3: *comparing the prediction errors on the new version using the model built from the old version and the model built from the new version.* The final approach to

detecting performance regression is similar to the previous one (Approach 2), where instead of directly comparing the predicted performance metrics, we compare the distribution of the prediction errors by applying $Model_{old}$ on $Data_{new}$, and the predicted errors by applying each $Model^{n}_{new}$ on each of the n data points in $Data_{new}$. The intuition is that, when $Model_{old}$ has a larger prediction error than $Model_{new}$, it may be an indication of performance deviance between the two versions. We note that this approach would only be able to determine whether there exists performance deviance, which may actually be improvement instead of regression.

Statistical analysis. All three approaches generate two distributions of either actual/predicted performance metric values, or prediction errors. we compare the two distributions to determine if there are performance regressions between the two software versions. However, the differences between the data distributions may be due to experimental noises rather than the systematic difference between the two versions. Therefore, similar to previous studies [63], we use statistical methods (i.e., statistical tests and effect sizes) to compare the data distributions of the two versions while taking into consideration the experimental noises.

In particular, we use the Mann-Whitney U test since it is non-parametric and it does not assume a normal distribution of the compared data. we run the test at the 5% level of significance, i.e., if the p-value of the test is not greater than 0.05, we would reject the *null hypothesis* in favor of the alternative hypothesis, i.e., there exists a statistically significant difference between the performance of old version and new version systems. In order to study the magnitude of the performance deviation without being biased by the size of the data, we further adopt the effect sizes as a complement to the statistical significance test. Considering the non-normality of our data, we utilize *Cliff's Delta* [69] and thresholds provided in prior research [188]. For approaches 1 and 2, we can use the direction (i.e., positive or negative) of the effect size to further tell whether the deviation indicates a performance regression or improvement.

3.4 Case Study Setup

To study the effectiveness of our approaches for detecting performance regressions under varying workloads, we perform case studies on two open-source systems. In this section, we first present the subject systems. Then, we present the workloads applied to the systems, the experimental environment and performance issues. Finally, we present the choices of machine learning models that are used in our approaches.

3.4.1 Subject systems

We evaluate our approaches on two open-source systems, namely OpenMRS and Apache James. We also discuss the successful application of our approach in System X, a large-scale industrial system that provides government-regulation-related reporting services (in Section 3.6). Apache James is a Java-based mail server. OpenMRS is an open-source healthcare system that supports customized medical records and it is widely used in developing countries. The two open-source subject systems and the industrial system are all studied in prior research [240, 93] and cover different domains, which ensures that our findings are not limited to a specific domain. The details of the two open-source subject systems and the industrial system are shown in Table 3.2.

Both the two open-source subject systems and the industrial system are CPU-intensive. Besides, CPU is usually the main contributor to server costs [100]. Performance regression in terms of CPU would result in the need for more CPU resources to provide the same quality of service, thereby significantly increasing the cost of system operations. Therefore, in this study, we use CPU as the performance metric for detecting performance regressions.

Table 3.2: Overview of the open-source subject systems and the industrial system.

Subjects	Versions	Domains	SLOC (K)
OpenMRS	2.1.4	Medical	67
Apache James	2.3.2, 3.0M1, 3.0M2	Mail Server	37
System X*	10 releases in 2019	Commercial	>2,000

* We share our experience of applying our approaches on System X in Section 3.6.

Table 3.3: Summary of the test actions with increased appearances (the ones with •) in different load drivers.

OpenMRS					
Test actions	Load driver 1	Load driver 2	Load driver 3	Load driver 4	Load driver 5
Creation of patients	•		•		•
Deletion of patients	•		•		•
Searching for patients	•	•			
Editing patients				•	•
Searching for concepts			•	•	•
Searching for encounters		•	•		
Searching for observations			•	•	•
Searching for types of encounters		•		•	•

Apache James					
Test actions	Load driver 1	Load driver 2	Load driver 3	Load driver 4	Load driver 5
Sending mails with short messages and without attachments				•	
Sending mails with long messages and without attachments	•	•	•		•
Sending mails with short messages and with small attachments			•	•	
Sending mails with short messages and with large attachments		•			
Sending mails with long messages and with small attachments			•		
Sending mails with long messages and with large attachments	•	•			
Retrieving entire mails					•
Retrieving only the header of emails	•			•	•

3.4.2 Subject workload design

OpenMRS

OpenMRS provides a web-based interface and RESTFul services. We used the default OpenMRS demo database in our performance 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 different performance tests that are composed of eight various test actions, including 1) creation of patients, 2) deletion of patients, 3) searching for patients, 4) editing patients, 5) searching for concepts, 6) searching for encounters, 7) searching for observations, and 8) searching for types of encounters. Those performance tests are different in the ratio between the actions. In order to simulate a more realistic workload in the field, we added random controllers and random order controllers in JMeter to vary the workload. Moreover, we also simulated the variety of the number of users and activities in the field by setting random gaps between the repetitions of each user's activities, randomizing the order of the user activities, and setting a different number of maximum concurrent users for different workloads at different times.

Furthermore, we designed a total of five JMeter-based performance tests, each of which consists of a mixture of random workloads. In particular, to drive different workloads, for each of the load drivers, we pick several different test actions and put them in an extra JMeter loop controller that iterates a random number of times to increase their appearances in the workload. Table 3.3 shows the detailed choice of the looped actions of each load driver. To make sure that the two versions of the subject systems have different workloads, for one version of the system (e.g., the old version), we run four JMeter tests together to exercise the system. For the other version (e.g., the new version) of the system, we run one additional JMeter test, i.e., a total of five JMeter tests, to ensure that the two versions have undergone different workloads.

Despite the variation in the workloads, we keep our systems not saturated and running in normal conditions. The CPU saturation can be qualified as the average CPU load increases to a very large value (e.g., 90%) and remains for a long period of time [137]. When the system is saturated, the services of the systems are not provided normally and the relationship between the performance of a software system (e.g., CPU usage) and the runtime activities may change. Besides, real software systems are usually not running under saturated conditions.

For the OpenMRS system, we do not have any evidence showing performance regressions of specific versions. Therefore, we manually inject several performance regressions

into the system. The injected performance regressions are shown in Table 3.4. The original version, i.e., v0 of OpenMRS does not have any injected or known performance regressions. We separately injected four performance regressions (heavier DB request, additional I/O, constant delay, and additional calculation) on the basis of version v0. These four versions are called v1, v2, v3, and v4, respectively. The details of the injected performance regressions of OpenMRS are explained below:

- **v1: Injected heavier DB request.** We increased the number of DB requests in the code responsible for accessing the database in OpenMRS. This performance issue will increase the CPU usage of the MySQL server.
- **v2: Added additional I/O access.** Since accessing I/O storage devices (e.g., hard drives) is usually slower than accessing memory, we added redundant logging statements to the source code that is frequently executed. The execution of the logging statements will introduce performance regression.
- **v3: Created constant delay.** We added the delay bug in the frequently executed code, which creates a blocking performance issue.
- **v4: Injected additional calculation.** We added additional calculations to the source code of OpenMRS that is frequently executed during the performance test.

We deployed the OpenMRS on two machines, each with a configuration of Intel Core i5-2400 CPU (3.10GHz), 8 GB memory, and 512GB SATA hard drive. One machine is deployed as the application server and the other machine is deployed as the MySQL database server. We ran JMeter using the RESTFul API of OpenMRS on five extra machines with the same specification to simulate user workloads on the client side for five hours. Each machine hosts one JMeter instance with one type of workload. We used *Pidstat* [20] to monitor the CPU usage that is used as the performance metrics of this study. To minimize the noise from the system warm-up and cool-down periods, we filtered out the data from the first and last half hour of running each workload. Thus, we only kept four hours of data from each performance test.

Apache James

Apache James is an open-source enterprise mail server. It contains two main actions: sending and retrieving mail. Those two actions can be further divided into many smaller actions, e.g., sending mail with or without attachments and retrieving an entire mail or

only the header of the mail. In total, we built eight actions for the performance test, including 1) sending mails with short messages and without attachments, 2) sending mails with long messages and without attachments, 3) sending mails with short messages and with small attachments, 4) sending mails with short messages and with large attachments, 5) sending mails with long messages and with small attachments, 6) sending mails with long messages and with large attachments, 7) retrieving entire emails, and 8) retrieving only the header of emails. Similar to OpenMRS, we also created a total of five different workloads, where each workload consists of a mixture of eight test actions with different ratios. We conduct performance tests with different versions of Apache James with varying workloads (i.e., 4 tests for one version and 5 tests for the other version). We used JMeter to create performance tests that exercise Apache James. Different from OpenMRS, in which the performance regressions are manually injected, we performance-tested on three different versions of systems with known real-world performance issues for Apache James. Apart from the last stable version 2.3.2, there are multiple Milestone Releases for version 3.0. After checking the release notes on the Apache James website, we picked 3.0M1 and 3.0M2 to use in our study. These three selected versions have many bug fixes and performance improvements [4]. The same subjects are studied in prior research on performance testing [93]. Table 3.4 summarizes the details of performance improvements of three selected versions.

We deployed Apache James on a server machine with an Intel Core i7-8700K CPU (3.70GHz), a 16 GB memory, and a 3TB SATA hard drive. We ran JMeter on five extra machines with a configuration of Intel Core i5-2400 CPU (3.10GHz), 8 GB memory, and 320GB SATA hard drive. These JMeter instances generate multiple workloads for five hours. Similar to OpenMRS, we use *Pidstat* [20] to monitor CPU usage as the performance metrics of this study. We filtered out the data from the first and last half hour of the tests to minimize the system noises.

Generated workloads

Our generated workloads are different across different software versions. By conducting chi-square tests of independence between the number of different actions and the versions, we find that in both OpenMRS and Apache James, the number of different actions and the versions are not statistically independent (p-value $\ll 0.05$), which means that the distribution of the actions is statistically different across versions. For example, for the OpenMRS system, the total number of all actions ranges from 10,673 to 22,392 across versions. In particular, the number of the *deletion of patients* action has a range from 293 to 1,861 across all versions. For another example from Apache James, the number of

Table 3.4: Performance regressions in the studied open-source systems

System	Versions	Performance regressions
OpenMRS	v0	Original version
	v1	Injected heavier DB request
	v2	Added additional I/O access
	v3	Created constant delay
	v4	Injected additional calculation
Apache James	2.3.2	Stable release version
	3.0M1	Improved ActiveMQ spool efficiency [4]
	3.0M2	Improved large attachment handling efficiency [4]

the *retrieving only the header of mails* action has a range from 411 to 25,532 across all versions.

Table 3.5 summarizes the average CPU utilization of each version of the systems under our generated dynamic workloads. We would like to note that when we observe a higher CPU usage, it may not correspond to a performance regression; instead, it may be just due to the system having a higher workload. The influence of the dynamic workload on CPU usage is one of the reasons that we leverage performance modeling to detect performance regressions in the field.

Table 3.5: Summary of average CPU utilization of each version in our case studies.

OpenMRS					
v0 (4w)	v0 (5w)	v1	v2	v3	v4
52.03	52.24	55.55	53.76	53.05	55.11

Apache James			
3.0M2 (4w)	3.0M2 (5w)	3.0M1	2.3.2
0.48	0.51	3.54	4.89

3.4.3 Subject models

Our approach presented in Section 3.3 is not designed strictly for any particular type of machine learning model. In fact, practitioners may choose their preferred models. In this study, we study the use of six different types of models, namely linear regression,

random forest, XGBoost, convolutional neural network (CNN), recurrent neural network (RNN), and long short-term memory (LSTM). In particular, for linear regression, random forest, and XGBoost, the inputs of the models are vectors whose values are the number of appearances of each log event in a time period. For CNN, RNN, and LSTM, we sort the logs in each time period by their corresponding time stamp to create a sequence as the input of the neural networks. Our XGBoost model is fine-tuned using the GridSearchCV [6] method. Our CNN, RNN, and LSTM have three, four, and four layers, respectively, and they are trained with five, five, and ten epochs, respectively. The number of layers and epochs are manually fine-tuned to avoid overfitting.

3.5 Case Study Results

In this section, we evaluate our approaches by answering two research questions.

3.5.1 RQ1: How well can we model system performance under varying workloads?

Motivation

In order to use black-box machine learning models to detect performance regressions under varying workloads, we first need to understand whether such black-box models could accurately model the performance of a software system under varying workloads. Although prior research demonstrates promising results of using black-box models to capture the relationship between system performance and logs [240, 88], these models are built on predefined in-house workloads instead of varying field workloads. If such black-box models are sensitive to the variance in the workloads, they may not be suitable for modeling the performance of software systems under the field workloads from real end users.

Approach

In order to understand the black-box models' ability to model system performance under varying workloads, we train the models on one set of workloads and evaluate the performance of the models on a different set of workloads (i.e., *unseen* workloads).

Performance modeling. For each open-source system (i.e., OpenMRS and Apache James), we select the version of the system that is without performance regressions. We

first run the system with four different concurrent workloads (i.e., *4W*) and collect the logs and performance metrics. In order to ensure having new workloads to the system, we conduct another run by having an additional concurrent workload, i.e., having five different concurrent workloads (i.e., *5W*). We build the performance models using the data that is generated by running the *4W* workloads (*a.k.a.* the training set) and apply the model on the data that is generated by running the *5W* workloads (*a.k.a.* the testing set). The training and testing sets are derived from independent system runs. We evaluate the prediction performance of the black-box models on the 5W workloads by comparing the predicted performance and the measured performance.

Analysis of modeling results. We calculate the prediction errors of the models on the new workloads (i.e., the *5W* workloads for the open-source systems). In order to understand the magnitude of the prediction errors, we use the prediction errors of the models on the old workloads (i.e., the *4W* workloads for the open-source systems) as baselines. The baseline prediction errors are calculated using 10-fold cross-validation to avoid the bias of having the same training and testing data.

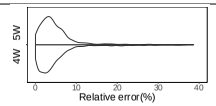
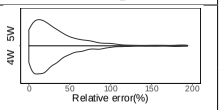
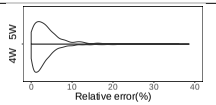
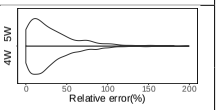
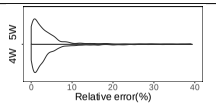
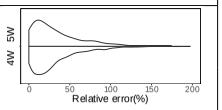
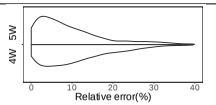
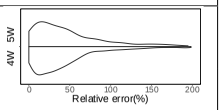
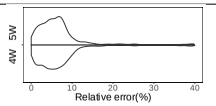
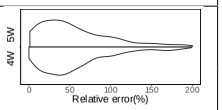
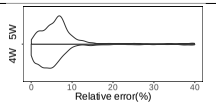
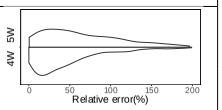
- **Median relative error**. The difference between the predicted performance and the measured performance is normalized by the measured performance.
- **p-value** (Mann-Whitney U). In order to understand whether the models trained on the old workloads can equivalently capture the system performance under the new workloads, we use the Mann-Whitney U test [169] to determine whether there exists a statistically significant difference (i.e., p-value < 0.05) between the prediction errors on the new workloads and the prediction errors on the old workloads.
- **Effect size** (Cliff's delta). Reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, the p-value can be very small even if the difference is trivial) [207]. Therefore, we apply Cliff's Delta [69] to quantify the effect size of the difference between the prediction errors on the old workloads and the prediction errors on the new workloads.

Results

Table 3.6 shows the detailed results of using six machine-learning techniques to model the performance of the studied open-source systems (i.e., OpenMRS and Apache James) under varying workloads. The column "MRE" shows the median relative errors of applying the models (trained on the *4W* workloads) on the *4W* workloads and the *5W* workloads, respectively. The columns "p-value" and "effect size" show the statistical significance and

the effect size of the difference between the prediction errors of the models on the *4W* workloads and the prediction errors of the models on the *5W* workloads, respectively. The "violin plot" column shows the distribution of the relative prediction errors under the *4W* and *5W* workloads.

Table 3.6: Prediction error details for OpenMRS and Apache James under different workloads.

Model	OpenMRS					Apache James				
	MRE		P-value	Effect size	Violin plot	MRE		P-value	Effect size	Violin plot
Linear Regression	4W	3.12%	<0.01	-0.08 (negligible)		4W	23.24%	0.41	N/A	
	5W	3.58%				5W	23.76%			
Random Forest	4W	2.36%	<0.01	-0.09 (negligible)		4W	22.99%	0.42	N/A	
	5W	2.90%				5W	23.08%			
XGBoost	4W	2.16%	0.14	N/A		4W	24.29%	0.44	N/A	
	5W	2.11%				5W	24.42%			
CNN	4W	9.56%	<0.01	0.12 (negligible)		4W	33.61%	0.11	N/A	
	5W	7.47%				5W	37.51%			
RNN	4W	5.63%	0.32	N/A		4W	51.34%	0.01	0.07 (negligible)	
	5W	5.73%				5W	47.22%			
LSTM	4W	4.53%	<0.01	-0.25 (small)		4W	34.88%	<0.01	-0.27 (small)	
	5W	6.42%				5W	57.11%			

Note: The column "MRE" presents the median relative errors.

Our black-box models can effectively model the performance of the studied systems using the dynamic runtime activities that are recorded in the logs. As shown in Table 3.6, the XGBoost model achieves the best results for modeling the performance of the OpenMRS system, with a median relative error of 2.11% on the *5W* workloads (i.e., the new workloads) and a median relative error of 2.16% on the *4W* workloads (i.e., the baseline workload). The random forest model achieves the best results for Apache James, reaching a median relative error of 22.99% and 23.08% for the *4W* workloads and *5W* workloads, respectively. All the machine learning models achieve better results for the OpenMRS system than the results for the Apache James system. The less-promising results for the Apache James system might be explained by the latency between the actual activities of the mail server system and the recorded logs. For example, the system can take an extended period of time to process an email with a large attachment, while a log about the successful processing of the email is only printed after the processing period.

The traditional models (e.g., linear regression and random forest) outperform the deep neural networks (e.g., CNN and RNN) for modeling the performance of the studied systems. As shown in Table 3.6, for both the OpenMRS and the Apache James systems, the three traditional models achieve better results than the three deep neural networks for modeling the system performance. These results indicate that the relationship between the system performance and the runtime activities recorded in the logs can be effectively captured by the simple traditional models. Such results also agree with a recent study [78] that compares deep neural networks and traditional models for the application of automated recommendations.

Our black-box models can equivalently explain the performance of a system under new workloads that are unseen when constructing the models. Table 3.6 shows the statistical significance (i.e. the p-value) and the effect size of the difference between the prediction errors of applying the old models (i.e., trained from the *4W* workloads) on the new workloads (i.e., the *5W* workloads) and applying the old models on the old workloads (i.e., the *4W* workloads). Table 3.6 also compares the distributions of the prediction errors for the *4W* and *5W* workloads. The prediction errors of most of the models (except LSTM) have either statistically insignificant or negligible differences between the *4W* and the *5W* workloads, indicating that the models trained from the old workloads can equivalently model the system performance under new workloads. The LSTM model results in a *small* difference in the prediction errors between the *4W* and the *5W* workloads. We suspect that the complex LSTM model is likely to over-fit the training workloads.

Simple traditional models (e.g., linear regression and random forest) can effectively capture the relationship between the performance of a system and its dynamic activities that are recorded in logs, with varying workloads.

3.5.2 RQ2: Can our approach detect performance regressions under varying workloads?

Motivation

In traditional performance testing, in order to detect performance regressions, performance analysts compare the performance data of two versions of a software system that is generated by running the same workloads from the same performance test suites. However, in a field environment, as the workloads of the systems are constantly changing, it is almost impossible to run two software versions on the same workloads to detect performance regressions. The results of RQ1 show that our black-box models can accurately capture the performance of a software system even under new workloads that are unseen when training the models. Therefore, in this research question, we would like to leverage such black-box models to detect performance regressions when the workloads of the two versions of a system are not consistent.

Running the systems for hours or days before discovering performance regressions incurs a high cost. As the systems are already running in the field, any delay in detecting the performance regressions may have a huge impact on the end users. Hence, the desired approach in practice should be able to detect performance regression in a timely manner. Therefore, we also want to study how fast our approaches can detect performance regressions.

Approach

The results from RQ1 show that random forest and XGBoost have the lowest prediction errors when modeling performance (cf. Table 3.6). Since XGBoost requires resource-heavy fine-tuning, we opt to use random forest in this research question. For the open-source systems, we first build the performance models from running the systems without performance regressions under the *4W* workloads (i.e., a combination of four different concurrent workloads). We then run the systems without performance regressions under

the *5W* workloads (i.e., a combination of five different concurrent workloads). Ideally, our approach should not detect performance regressions from these runs. We use such results as a baseline to evaluate the effectiveness of our approach for detecting performance regressions under the new workloads (i.e., the *5W* workloads). Afterward, we run the systems with the performance regressions (cf. Table 3.4) under the *5W* workloads. Our approach should be able to detect performance regressions from these runs.

Finally, to study how fast our approaches can detect performance regressions, for the new versions of the systems that have performance regressions, we only use the first 15-minute data and apply our approach to detect the performance regressions. Then, we follow an iterative approach to add another 5-minute data to the existing data, until our approach can detect the performance regressions (i.e., with a medium or large effect size that is higher than the baseline).

Results

Our black-box-based approaches can effectively detect performance regressions under varying workloads. Table 3.7 shows the results of our three approaches of performance regression detection (cf. Section 3.3.3) on OpenMRS and Apache James. We find that with all three approaches when there are known performance regressions between two versions, the statistical analysis always shows a significant difference between the two versions with medium or large effect sizes. In addition, the effect sizes from Approaches 1 and 2 are negative, confirming the existence of performance regressions (negative values indicate performance regression and positive values indicate performance improvement).

When we compare the effect sizes with the baseline, i.e., running our approaches with systems without performance regressions but under two different workloads, we find that the baseline effect sizes are always smaller than the corresponding ones with performance regressions, except when detecting the regression in v3 of OpenMRS. We consider the reason to be the nature of the regression in v3, i.e., an injected delay. Since our considered performance metric is CPU usage and such a delay may not have a large impact on the CPU usage, it is difficult for our approaches to detect such a regression.

Table 3.7: Performance regression detection results for OpenMRS and Apache James.

OpenMRS							
Versions		Approach 1		Approach 2		Approach 3	
Old version	New version	P-value	Effect size	P-value	Effect size	P-value	Effect size
v0	v0	$\ll$0.001	0.39 (medium)	$\ll$0.001	0.36 (medium)	$\ll$0.001	0.17(small)
v0	v1	$\ll$0.001	-0.59 (large)	$\ll$0.001	-0.69 (large)	$\ll$0.001	0.38 (medium)
v0	v2	$\ll$0.001	-0.44 (medium)	$\ll$0.001	-0.63 (large)	$\ll$0.001	0.37 (medium)
v0	v3	$\ll$0.001	-0.36 (medium)	$\ll$0.001	-0.42 (medium)	$\ll$0.001	0.53 (large)
v0	v4	$\ll$0.001	-0.69 (large)	$\ll$0.001	-0.76 (large)	$\ll$0.001	0.51 (large)
Apache James							
Versions		Approach 1		Approach 2		Approach 3	
Old version	New version	P-value	Effect size	P-value	Effect size	P-value	Effect size
3.0m2	3.0m2	0.008	0.09 (negligible)	$\ll$0.001	-0.12 (negligible)	$\ll$0.001	-0.03 (negligible)
3.0m2	3.0m1	$\ll$0.001	-0.65 (large)	$\ll$0.001	-0.76 (large)	$\ll$0.001	0.41 (medium)
3.0m2	2.3.2	$\ll$0.001	-0.90 (large)	$\ll$0.001	-0.93 (large)	$\ll$0.001	0.82 (large)

Note: For all the old versions, we use four concurrent workloads and for all the new versions with and without regressions, we use five concurrent workloads (one extra workload).

Comparing the prediction errors is more effective than comparing the prediction values when detecting performance regressions between two versions. We observe that for OpenMRS, the differences between the prediction values using Approach 1 and 2 can still be medium (0.39 and 0.36) even for the baseline (i.e., without regressions). On the other hand, when comparing the prediction errors instead of the prediction values, i.e., using Approach 3, the baseline without regressions has only a small effect size (0.17). Such a smaller baseline effect size makes Approach 3 easier to adopt in practice, i.e., without the need to spend efforts searching for an optimal threshold on the effect size to detect performance regressions. However, Approach 3 only shows the deviance of the prediction errors without showing the direction of the performance deviance, thus it cannot distinguish a performance regression from a performance improvement. Hence, Approach 3 may be used first to flag the performance deviance and then be combined with other approaches in practice to determine whether the performance deviation is a performance regression or a performance improvement.

Our approaches can detect performance regressions as early as 15 minutes after running a new version. Table 3.8 shows the earliest time that our approach can detect performance regressions in the studied open-source systems. We find that all the performance regressions in the open-source systems can be detected by at least one approach with less than 20 minutes of data from the new version. In particular, the regressions from both versions of Apache James and three versions of OpenMRS can even be detected using only the first 15-minute data. The ability of early detection eases the adoption of our approaches in the practices of testing in the field, where performance regressions are detected directly based on the field data, instead of using dedicated performance testing. In our future work, we plan to investigate on further shortening the needed time for detecting performance regressions in the field to ease practitioners in adopting our approach.

Table 3.8: The earliest time for our approaches to detect regressions in the two open-source systems.

	OpenMRS				Apache James	
	v1	v2	v3	v4	3.0m1	2.3.2
Approach 1	60 mins	160 mins	15 mins	15 mins	15 mins	15 mins
Approach 2	20 mins	50 mins	15 mins	15 mins	15 mins	15 mins
Approach 3	45 mins	15 mins	15 mins	15 mins	15 mins	15 mins

All three approaches can successfully detect performance regressions under varying workloads, requiring data from a very short period of time (down to 15 minutes). Comparing the prediction errors is more effective than comparing the prediction values for detecting performance regressions between two versions.

3.6 An Industrial Experience Report

System X is a commercial software system that provides government-regulation-related reporting services. The service is widely used as the market leader in the domain. System X has over ten years of history with more than two million lines of code that are based on Microsoft .Net. System X is deployed in an internal production environment and is used by external enterprise customers worldwide. System X is a CPU-intensive system. Thus, CPU usage is the main concern of our industrial collaborator in performance regression detection. Due to a Non-Disclosure Agreement (NDA), we cannot reveal additional details about the hardware environment and the usage scenarios of System X.

3.6.1 Modeling system performance

We directly use the field data that is generated by workloads from the real end users to model the performance of System X. For each release cycle with n days, we use the data from the first $n/2$ days to build the models and apply them on the data from the second $n/2$ days to evaluate the prediction performance of the models. We would like to note that there exists no control on the end users for applying any particular workload and that all the data is directly retrieved from the field with no interference on the behavior of the end users. We studied 10 releases of System X.

We find that black-box models can accurately capture the relationship between the system performance and the system activities recorded in the logs, with a prediction error between 10% to 20%. In particular, the models trained from the first half-release data can effectively predict the system performance of the second half-release.

3.6.2 Detecting performance regressions

For every new release, we use our approaches to compare the field data from the new release and the previous release to determine whether there are performance regressions. Since there are no injected or pre-known performance regressions, we present the detected performance regressions to the developers of the system and manually study the code changes to understand whether the detection results are correct. For the releases that are detected as not having performance regressions, we cannot guarantee that these releases are free of performance regressions. However, we also present the results to the developers to confirm whether there exist any users who report performance-related issues for these releases. If so, our detection results may be considered false.

Within all the 10 studied releases of System X, our approaches detected performance regressions from one release. All three approaches detected performance regressions from the release with large effect sizes when compared with the previous release. None of the three approaches false-positively detect performance regressions from the other nine releases (i.e., with either statistically insignificant differences or negligible effect sizes when compared with the previous release). By further investigating the release with detected performance regressions, we observed that developers added a synchronized operation to lock the resources that are responsible for generating a report, in order to protect the shared resources under the multi-thread situation. However, the reporting process is rather resource-heavy, resulting in significant overhead for each thread to wait and acquire the resources. Thus, the newly added lock causes the performance regression. After we discussed with the developers who are responsible for this module, we confirmed that this synchronized operation introduced the performance regression to the software. In addition, for all the nine releases from which our approaches did not detect performance regressions, the developers of System X have not yet received any reported performance issues from the end users till the day of writing this work.

Our approaches have been adopted by our industrial collaborator to detect performance regressions of System X on a daily basis.

3.7 Challenges and Lessons Learned

In this section, we discuss our faced challenges and learned lessons while applying our approach to the production environment of an industrial setting where a large number of customers worldwide access the system on a daily basis.

3.7.1 C1: Determining the sampling frequency of performance metrics

Challenge. Our approach uses both logs and performance metrics as the input data for our black-box models. We use the logs that are automatically generated by the web servers, such as the Jetty, Tomcat, and IIS (Internet Information Services) web servers. The performance metrics (e.g., CPU, I/O) of the systems are collected using tools (e.g., *Pidstat*). A higher sampling frequency of the performance metrics can capture the system performance more accurately. However, a higher sampling frequency would also introduce more performance overhead. Since we need to apply our approach to the production environment, it is necessary to produce as low performance overhead as possible.

Solution. At the first attempt, we intuitively chose 10 seconds as the sampling interval of the performance metrics. After we deployed our approach in production, we found that there was 0.5%-0.8% CPU overhead each time when our approach collected the performance metrics, and the overhead happened six times in 1 minute. Such a monitoring overhead cannot be ignored, especially when the system is serving heavy workloads. After working closely with the IT staff from our industrial partner, we gained a deeper understanding of how the sampling frequency of performance metrics impacts the monitoring overhead. Finally, we agreed that collecting the performance metrics every 30 seconds would be a good balance between reducing the monitoring overhead and ensuring accurate measurement of the system performance.

Lessons learned. Monitoring a system usually comes with the monitoring overhead. A higher monitoring frequency can provide better monitoring accuracy at the cost of a larger monitoring overhead, which is usually undesirable when the system is serving large workloads. Finding a good balance between the monitoring accuracy and the overhead is crucial for the successful adoption of similar approaches in practice.

3.7.2 C2: Reducing the time cost of performance regression detection

Challenge. We choose random forest as our final black-box model to detect performance regressions, as random forest achieves the best results for modeling the performance of the studied systems (see RQ1). A random forest model contains a configurable multitude of decision trees and takes the average output of the individual decision trees as the final output. At first, we started with the default number of trees (i.e., 500 trees) [52]. It took 5-6 hours to detect performance regressions between two releases of the industry system,

including training and testing the random forest models and performing statistical tests. The industrial practitioners usually have an early need to check if there is performance regression between the current version and multiple historical versions, which makes our approaches difficult to adopt in a fast-paced development environment (e.g., an agile environment).

Solution. A larger number of trees usually result in a more accurate random forest model, at the cost of longer training and prediction time. In order to reduce the time cost of performance regression detection, we gradually decreased the number of trees in our random forest model while ensuring the model performance is not significantly impaired. In the end, we kept 100 decision trees in our random forest models. It took less than 2 hours to detect performance regressions between two releases of the industry system (i.e., training and testing the random forest models and performing statistical tests). Such a lighter model also enables us to detect performance regressions between the current version and multiple historical versions (e.g., taking less than 10 hours when comparing the current version with five historical versions). We compared the prediction results of the 100-tree model with the original 500-tree models. We found that the 100-tree model only has a slightly higher median relative error (around 0.1%) than the previous 500-tree models, which is negligible in practice.

Lessons learned. In addition to the model performance, the time cost of training and applying a black-box model is also a major concern in the practice of performance regression detection in the field. Seeking an appropriate trade-off between the model performance and the time cost is essential for the successful adoption of performance regression detection approaches in the field.

3.7.3 C3: Early detection of performance regressions

Challenge. In an in-house performance testing process, performance engineers usually wait until all the tests are finished before they analyze the testing results. However, in a field environment, any performance regressions can directly impact users' experience. We cannot usually wait for a long time to gather plenty of data before performing performance analysis. If our approach cannot detect performance regressions in a timely manner, the performance regressions would already have brought a non-negligible negative impact to users. This is a unique challenge facing the detection of performance regressions in the field.

Solution. In order to detect field performance regressions in a timely manner, we continuously apply our approach to detect performance regressions using the currently available

data. For example, after a new version of the system has been up and running for one hour, we use only the one-hour data as our new workloads to determine the existence of performance regressions in the new versions. Using the data generated in a short time period also allows the analysis part of our approach (i.e., model training and predictions) to be processed faster. As discussed in RQ2, our approach can effectively detect performance regressions when the system has run for a very short time (e.g., down to 15 minutes for Apache James). In other words, our approach can detect early performance regressions in the field.

Lessons Learned. Different from performance regression detection in an in-house testing environment, performance regressions in the field need to be detected in a timely manner, in order to avoid notable performance impact to the users. Continuously detecting performance regressions using the currently available data can help detect early performance regressions in the field.

3.8 Threats to Validity

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

3.8.1 External validity.

Our study is performed on two open-source systems (i.e., OpenMRS and Apache James) and one industry system (i.e., System X) that are from different domains (e.g., health care system and mail server). All our studied systems are mature systems with years of history and have been studied in prior performance engineering research. Nevertheless, more case studies on other software systems in other domains can benefit the evaluation of our approach. All our studied systems are developed in either Java or .Net. Our approach may not be directly applicable to systems developed in other programming languages, especially dynamic languages (e.g., JavaScript and Python). Future work may investigate approaches to minimize the uncertainty in the performance characterization of systems developed in dynamic languages.

3.8.2 Internal validity.

Our approach builds machine learning models to capture the relationship between the runtime activities that are recorded in logs and the measured performance of a system.

However, there might be some runtime activities that are not recorded in logs and that also impact system performance. In our approach, we use logs that capture the high-level workloads of the system. Our experiments on our studied systems demonstrated that such logs can predict system performance with high accuracy. Nevertheless, the correlation between the runtime activities recorded in the logs and the measured system performance does not necessarily suggest a causal relationship between them.

Our approach relies on comparing performance models that are constructed based on software behaviors that are recorded in logs. We admit that our approach is not applicable in situations where the old version and the new version have fundamentally different features. If a new version has many new features that do not exist in an old version, the model constructed on the old version would not have the knowledge about the performance impact of the new features, thus comparing performance models between these two versions would not be practical. However, we would like to emphasize that the goal of our approach is to verify whether the black-box performance models can work in scenarios where the workloads have high variations (e.g., variation in the ratio between different types of activities).

Our approach relies on non-parametric statistical analysis (i.e., Mann-Whitney U test and Cliff's delta) to compare the black-box behaviors of two software versions to detect performance regressions. Our assumption is that statistically different behaviors between two software versions would suggest performance regressions. In practice, however, determining whether there is a performance regression usually depends on the subjective judgment of the performance analysts. Therefore, our approach enables performance analysts to adjust the threshold of the statistics (e.g., the effect size) to detect performance regressions in their specific scenarios.

3.8.3 Construct validity.

In this work, instead of using modern performance monitoring tools (e.g., application performance monitoring (APM)), we use traditional system monitoring tools (e.g., Pidstat) to collect the performance data (e.g., CPU usage) when running the systems. The use of APM to collect more information may complement our approach.

We use CPU usage as our performance metric to detect performance regressions. There exist other performance metrics, such as memory utilization and response time that can be considered to detect performance regressions. Considering other performance metrics would benefit our approach. However, monitoring more performance metrics would introduce more performance overhead to the monitored system. In the case of our industrial

system, CPU usage is the main concern in performance regression detection. Besides, the three studied approaches are not limited to the performance metric of CPU usage. Practitioners can leverage our approach to consider other performance metrics that are appropriate in their context.

We use JMeter to send workloads to the open-source systems. We are not experts for these open-source software systems and the actual field users' data for these open-source systems is not available. Therefore, the portions of each type of action are not pre-chosen but just the results of our randomized workload drivers, and we do not have any empirical evidence to support such numbers. However, our goal is just to make different workloads across different versions, instead of having chosen specific numbers. Similarly, in the real world, it may be difficult to anticipate the number of users of a system and their usage patterns.

3.9 Conclusion

In this chapter, we propose to leverage automated approaches that can effectively detect early performance regressions in the field. We study three approaches that use black-box performance models to capture the relationship between the activities of a software system and its performance under such activities, and then compare the black-box models derived from the current version of a software system and an earlier version of the same software system, to determine the existence of performance regressions between these two versions. By performing empirical experiments on two open-source systems (OpenMRS and Apache James) and applying our approaches to a large-scale industrial system, we found that simple black-box models (e.g., random forest) can accurately capture the relationship between the performance of a system under varying workloads and its dynamic activities that are recorded in logs. We also found that these black-box models can effectively detect real performance regressions and injected performance regressions under varying workloads, requiring data from only a short period of operations. Our approaches can complement or even replace typical in-house performance testing when testing resources are limited (e.g., in an agile environment). The challenges and the lessons that we learned from the successful adoption of our approach also provide insights for practitioners who are interested in performance regression detection in the field.

Chapter 4

Locating Performance Regression Root Causes in the Field Operations

Software developers usually rely on in-house performance testing to detect performance regressions and locate their root causes. Such performance testing is typically resource and time-consuming, making it impractical to conduct when the software is delivered in fast-paced release cycles. On the other hand, the operational data generated in the field environment provides rich information about the performance of a software system and its runtime activities. Therefore, this work explores the idea of leveraging the readily available field operational data to locate the root causes of performance regression instead of running expensive performance tests. However, due to the ever-changing workloads from the end users and the noise from the field, directly analyzing performance metrics such as the response time of the system may not be able to help locate the root causes of performance regressions.

In this chapter, we report our experience of designing and adopting an approach that automatically locates the root causes of performance regressions while the software systems are deployed and running in the field. First, our approach uses black-box performance models to capture the relationship between the performance of a system and its runtime activities. Then, our approach analyzes the performance models and uses statistical techniques to suggest the problematic system runtime activities (i.e., the root causes) that are related to a performance regression. Our evaluation considered three open-source projects and one industrial product. In the three open-source systems, we find that our approach can successfully locate the root causes of all arbitrarily injected synthetic performance regressions. Our approach has successfully detected and located the root causes of three performance regressions in an industry system and it has been adopted by our industrial

partner and used in practice on a daily basis over a 12-month period. In addition, we share the challenges that we encountered during the design and adoption of our approach, how we addressed those challenges, and the lessons that we learned during the process. We believe that our novel approach together with our documented experience can benefit practitioners and researchers who wish to leverage the field-operation data of a software system to conduct performance assurance activities.

4.1 Introduction

Large-scale software systems, such as web-based systems like Facebook and Google, have become an indispensable part of people's daily lives. Driven by the need to provide uninterrupted services to millions or even billions of users around the world, such systems usually need to meet stringent performance requirements. Performance-related problems in such systems often result in financial losses [5, 202, 49, 101]. For example, it is estimated that adding merely one more second to load an Amazon.com page could cost the company as much as $1.6 billion in its annual revenue [1]. Therefore, performance assurance activities (e.g., performance testing) are a crucial step to avoid performance regressions (i.e., when a new version of the system has worse performance than the previous versions). For example, Mozilla has a policy that requires all performance regressions to be reported and resolved [16].

Software practitioners typically conduct in-house performance testing prior to the deployment of every new release of a software system, in order to ensure that potential performance regressions are detected [122, 230]. If a performance regression is detected, the results from the performance testing can provide useful information to guide developers' efforts in identifying and fixing the root causes [111]. However, performance testing is usually very costly, requiring expensive resources, complex environment configuration, and excessive execution time [62, 122]. More and more software systems are delivered along fast-paced release cycles (e.g., a new version is released every few hours) [117, 129]. Hence, it is often challenging, if not infeasible, to detect and locate the root causes of performance regressions through traditional in-house performance testing.

Prior research in software performance often utilizes statistical techniques (e.g., control charts) to compare and analyze the load testing results (e.g., performance metrics) to detect performance regressions and locate the corresponding root causes [208, 177, 159, 32]. In addition to load tests, unit tests are also leveraged in prior studies [111, 155, 58] to locate performance regression root causes. However, load testing usually requires extensive resources and a long time to execute, while unit tests cannot take the impact of large and

varying workloads into consideration. On the other hand, there is also prior research that directly uses field data. Nevertheless, their located root causes are often coarse-grained (e.g., only at service level) [34, 233, 170, 120], or only target specific performance regression patterns [34].

In this chapter, we share our experience of leveraging field-operation data to automatically locate the root causes of performance regressions in a large-scale industrial software system. In particular, instead of running the costly and time-consuming performance tests in a short release cycle (e.g., two weeks), the performance of the system is monitored and directly evaluated during the field operation of the system (i.e., when end users are using the service delivered by the system). Such an automated approach can complement or even replace typical in-house performance testing when testing resources are limited (e.g., in an agile environment) or when field workloads are difficult to reproduce in the testing environment. Performance regressions are detected by comparing the performance of the new release with the old release (e.g., in A/B testing [235] or canary releasing [2] manner). Specifically, when a new version is released, we build performance models that capture the relationship between the performance of the system and the system's runtime activities that are recorded in the web-access logs of the system. We then analyze the difference between the new performance model and the performance model from the previous release, to identify the system activities that contribute to such difference. The identified system activities are considered the root causes of the performance regressions.

The evaluation of our approach is conducted on both open-source and industrial projects. We apply our approach to three open-source projects (i.e., *TeaStore*, *OpenMRS*, and *CloudStore*), with arbitrarily injecting synthetic performance regressions to different locations in these projects. Our results show that our approach can effectively locate the root causes of the injected ones in the open-source systems. In addition, our approach has been adopted by an industrial software system (i.e., *ES*), and used on a daily basis while successfully detecting and locating the root causes of real-life performance regressions. We discuss the challenges that we encountered during the design and adoption of our approach in an industrial environment. For each challenge, we share our solution to address the challenge and what we learned in the process. We believe that our experience and learned lessons can assist software practitioners and researchers in leveraging field-operation data in performance assurance activities.

The main contributions of this chapter are:

- We propose an approach that can automatically locate the root causes of performance regressions without the need for resource- and time-consuming performance testing.

- Our experimental results show that we can adopt black-box machine learning models to assist developers in identifying and fixing the root causes of performance regressions.
- We share the challenges and lessons learned from the successful adoption of our approach in the industry, which can provide insights for researchers and practitioners who are interested in locating the root causes of performance regressions using the field-operation data.

The remainder of the chapter is organized as follows. Section 4.2 introduces the background of locating performance regression root causes for an industrial system. The challenges and corresponding solutions are discussed in Section 4.3. Section 4.4 outlines our approach for locating performance regression root causes in the field. Section 4.5 and Section 4.6 respectively present our results of applying our approach to three open-source subject systems and one large-scale industrial system. Section 4.7 summarizes the lessons that we learned from addressing the faced challenges and the successful adoption of our approach. Section 4.8 discusses the threats to the validity of our findings. Finally, Section 4.8 concludes the chapter.

4.2 Motivational Background from Industrial Partner

The *ES* system. *ES*[1] is a commercial software system that provides government regulation (e.g., Regulations 29 CFR 1910[2]) related reporting services. *ES* is the market leader in its domain and its service is widely used by enterprises around the world. In addition, the system has over ten years of history with more than two million lines of code that are based on Microsoft .Net. It is developed and operated by a global company with a development team of 20 to 30 software engineers. *ES* is deployed on Microsoft Azure and operated by the internal operation team within the company. Due to a Non-Disclosure Agreement (NDA), we cannot reveal additional details about the hardware environment and the usage scenarios of ES.

Due to the importance of the service, the stakeholders of *ES* take its performance as a critical matter. In particular, the stakeholders of *ES* would like to be aware of any performance regressions that are introduced in a new release of the system. On the other

[1]ES is a codename for the system and has no practical meaning. Due to the NDA, we cannot disclose the real name of the system.

[2]https://www.osha.gov/laws-regs/regulations/standardnumber/1910

hand, the development and maintenance of *ES* follow an agile process where the software system has a new release every two weeks. Such a short release cycle of *ES* brings challenges in the detection of performance regressions and the localization of their root causes.

In order to adapt to the fast release pace of *ES*, the developers do not conduct any in-house performance testing before a release, due to the high demand for resources and the long duration. Instead, *ES*'s performance is tested in a field *A/B testing* manner, i.e., the performance of a new release is actively monitored and compared with an existing stable release, in order to quickly identify potential performance regressions based on the field performance data collected during operation by the end users.

Existing (baseline) approach for locating performance regression root causes. Since *ES*'s performance is evaluated in the field with end users, one cannot put extensive instrumentation to monitor the resource cost for each activity of the system. Instead, one may leverage system-level performance metrics, such as CPU usage, to detect performance regressions. However, such detection results cannot reveal the root causes of performance regressions in the source code. Since *ES* is a web-based system, by default, the web-access logs are produced during system execution. Therefore, the existing (baseline) approach leverages the response time that is recorded in each web-access log to locate the performance regressions to the associated web requests.

In particular, the existing (baseline) approach adopts a pair-wise comparison approach (similar to prior research [195]) that compares the response time of the same types of web requests in the new and old releases. The baseline approach parses the collected web-access logs into timestamps, types, and response times. For example, a line of web-access log "[2020-03-27 20:23:07] GET /openmrs/ws/rest/v1/person/ HTTP/1.1 200 53" will be parsed into timestamp "2020-03-27 20:23:07", the corresponding web request type "GET /openmrs/ws/rest/v1/person/", and the response time "53 milliseconds". The approach leverages the Mann-Whitney U test [169] to determine whether there exists a statistically significant difference between the response time of the same types of web requests in the new and old releases. In addition, the existing (baseline) approach uses effect sizes, i.e., Cliff's Delta [69], to quantify the magnitude of the difference and decide whether the difference in response time corresponds to a performance regression or an improvement. In particular, if the effect size of the difference is large ($|d| > 0.474$) and the new version's average response time is larger than the old version, the corresponding web request would be located as the root cause of the performance regression.

Limitation of the existing (baseline) approach. However, as also found by prior research [195], such pair-wise comparison approaches often lead to false-positive results due to the environmental and workload noises. The situation is even worse for *ES*, since

the data is from the field where the workloads of the new and old versions of *ES* are often inconsistent (i.e., varying number of active end users and usage scenarios). Comparing the response time of the new and old versions under inconsistent workloads can lead to even more false-positive results. *In fact, with the existing (baseline) approach, almost every new release of* ES *has web requests that are incorrectly deemed to be the root causes of performance regressions. Developers wasted much effort on examining these false-positive results.*

Our new approach for locating performance regression root causes. Over the last year, we have explored the challenges of locating performance regression root causes using the end user's data from the field (cf. Section 4.3). To address these challenges, we designed and developed an approach that automatically detects and locates performance regression root causes for *ES* (cf. Section 4.4). Our approach has been deployed in the production environment of our industrial partner and used to locate performance regression root causes for *ES* on a daily basis.

4.3 Challenges

In this section, we provide detailed discussions on our faced challenges when we aim to automatically locate the root causes of performance regressions based on the field data, in the context of an industrial software system, i.e., *ES*. We also describe our corresponding solution to each challenge.

Figure 4.1 outlines the challenges and their associated steps in the overall process of developing our approach for locating the root causes of performance regressions on *ES*. We divide the overall process into five parts, and each deals with an important challenge that is described below. The details of our approach are described in Section 4.4 and the lessons that we learned from addressing these challenges are summarized in Section 4.7.

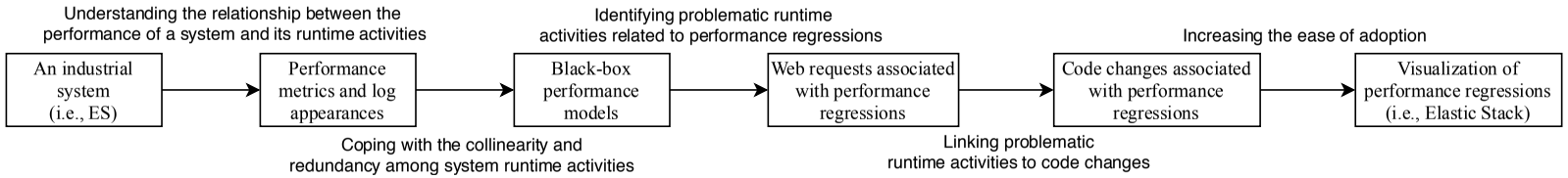


Figure 4.1: An overall process of developing our approach in an industrial setting

4.3.1 Challenge 1: Understanding the relationship between the performance of a system and its runtime activities

Challenge. In order to locate the root causes of performance regressions, we first need to understand the relationship between the performance of a system and its runtime activities (i.e., how system activities impact system performance). However, prior research [48] finds that few practitioners construct performance models (e.g., queuing networks [144]) for performance management during their development process. Thus, after the system is deployed in the field, the performance of most software systems, like *ES*, is usually viewed as a black box, leading to difficulties in describing such a relationship [83, 93]. Moreover, the workloads of a large-scale system are impacted by thousands or millions of users interacting with the system [62]. Such workloads are typically dynamic and vary in many aspects, such as the load intensity, and the order and the ratio among different user operations. Such dynamic workloads further increase the difficulty of capturing the relationship between the performance of a system and its runtime activities.

Solution. We build performance models to understand the relationship between the runtime activities of a system and its performance under such activities. There are two types of performance models: white-box models and black-box models [83]. White-box performance models typically require knowledge about the system's internal behavior and such knowledge is often not available when the system is deployed in the production environment [83, 93]. Therefore, we leverage black-box performance models that do not require knowledge about the internal behavior of a system.

Black-box models typically use machine learning techniques to model a system's performance against its runtime activities that are recorded in the execution logs. In our case, we build black-box performance models that use the appearances of each type of web-access log as the independent variables and performance metrics as the dependent variable. We find that using a simple black-box model (i.e., a random forest regression model) can already achieve a high modeling accuracy, which demonstrates the effectiveness of using black-box models to capture the relationship between the system runtime activities and its performance.

4.3.2 Challenge 2: Coping with the collinearity and redundancy among system runtime activities

Challenge. We observe that some system runtime activities may get entangled and always appear simultaneously in a specific order. One of the typical examples is that once users log

in to a mail system, they often check their inbox. According to prior studies [17, 241], when building regression models, the degree of correlation between independent variables should be low. Otherwise, the collinearity and redundancy among the independent variables may have a negative impact on fitting and interpreting the models.

Solution. To solve the above-mentioned challenge, we use correlation analysis and redundancy analysis to remove the collinearity and multicollinearity among the independent variables, respectively. As the system runtime activities associated with the removed independent variables can potentially cause performance regression, we maintain a mapping between the removed independent variables and the remaining independent variables that are highly corrected with the removed ones. When we detect a system runtime activity (for which the corresponding independent variable remains in the model) that may cause the performance regression (cf. Section 4.4), we also consider the system activities that have a high correlation with the detected activity as potential causes of the performance regression.

4.3.3 Challenge 3: Identifying problematic runtime activities related to performance regressions

Challenge. As *ES* adopts an agile development approach and the release cycle is within two weeks, our industrial partner does not have enough budget and time to conduct performance tests and examine the root causes of performance regressions. Therefore, we aim to help developers automatically locate the root causes of introducing the performance regressions without running performance tests. As the first step, we need to automatically identify the system activities related to the performance regressions. It is worth noting that although Challenge 1 and Challenge 3 are related, they are indeed different challenges. In particular, Challenge 1 describes whether one can model the system performance using system activities, especially when the system is deployed in the field and little knowledge about the system's internal behavior can be acquired. However, Challenge 3 is no longer to capture the relationship between the performance of a system and its runtime activities, instead, it is about whether one can locate the problematic system activities in a statistical way. In fact, from prior related work, much prior research aims at the detection of performance regression (mostly touching on Challenge 1) but not locating the particular factors that cause the regression (Challenge 3).

Solution. We propose a novel statistical solution that automatically identifies system activities related to performance deviance. Intuitively, if there is a performance regression, a performance model built on an old system version cannot equally explain the performance

of a new version. In such a performance model, the independent variables that contribute to the difference in model performance are related to the root causes of the performance regression. First, we build the black-box performance models (e.g., a random forest model) on the old version and on the new version, respectively. We then apply the two performance models to the new version data and measure the deviance between the two models' modeling errors. A larger deviance is more likely to suggest a performance regression.

In order to find out which system runtime activities are related to the performance regression (i.e., the deviance between the two performance models' modeling errors), we build a linear regression model taking the system activities (i.e., log appearances) as the independent variables and the deviance of modeling errors as the dependent variable. If an independent variable is statistically significant in the linear regression model, the independent variable is statistically significantly contributing to the difference in the modeling errors. Therefore, the associated system activities with the independent variable may be related to the performance regression, which are considered as candidates for performance regression causes.

There may be multiple candidates that are related to the performance regression. To prioritize *ES*'s resources on the system activities that are most likely to cause the performance regression, we rank the candidate system activities before providing them to developers. In particular, we use the effect of each independent variable on the model's output to rank the system activities that are associated with each independent variable.

4.3.4 Challenge 4: Linking problematic runtime activities to code changes

Challenge. Given the problematic system runtime activities that are related to a performance regression, developers still need to inspect the source code related to the problematic system activities and locate the code changes that cause the performance regression. Besides, during the regular meeting with our industrial partner, they mentioned that not all developers have a deep understanding of the system behaviors, and it is challenging for them to locate files, classes, or functions that are related to the system activities. Thus, motivated by the feedback from developers, we aim to further assist developers in finding the code changes that are associated with the problematic web activities related to the performance regressions.

Solution. In order to locate the code changes that lead to a performance regression, we focus on the web requests associated with the system activities that introduce the performance regressions. We aim to identify the code changes associated with the web

requests in the commit history between the two versions where regression is detected. We first search the entire source code to locate the methods that are associated with a web request, then use source code analysis to build a call graph of the web request. Finally, we identify the code changes in the commit history that affect the call graph, which are considered the potential root causes of the performance regression.

4.3.5 Challenge 5: Increasing the ease of adoption

Challenge. Although our approach can automatically locate the root cause of a performance regression, developers may be reluctant to adopt our approach in the production environment. Derived from the experience and feedback from the use of our industrial collaborators, we realize that, first, some developers are concerned that the additional performance monitoring may introduce system performance overhead, especially when the system is serving a large number of end users in field operations. Second, as many developers may lack knowledge about the statistical techniques, they may not completely understand and trust our approach to their system.

Solution. To address the challenge of adoption, we first study the system performance overhead after we integrate our performance monitoring into the existing system. The performance overhead depends on how often system performance data is sampled. While a higher sampling frequency can achieve a more accurate measurement, it can lead to a larger performance overhead. Therefore, we wish to find the optimal sampling frequency of system performance data. To achieve this, we load test the system with different scales of workloads while recording the impact of different sampling frequencies (e.g., every 10s, 30s, and 60s) on the system performance. At the same time, we worked closely with the senior operation support specialists to find acceptable performance overhead while ensuring a considerably high sampling frequency. Finally, we reached a consensus that sampling the performance data at a frequency of every 30 seconds is an acceptable balance between the accuracy and the overhead of the performance measurement.

On the other hand, to help developers understand our approach's mechanism and the output, we visualize the system performance and the root causes of performance regressions in a user-friendly manner. In particular, we integrated our approach into an Elastic Stack [14] platform such that developers can simply log into the platform to view the status of the system performance (e.g., CPU, memory, and disk I/O) and the workloads (e.g., total number of requests, slowest requests, and response status over time), as well as our newly added performance regression dashboard, which enables developers to observe the instantly measured system performance and the modeling errors of our model. With the

help of visualization, developers can have a better understanding and control of the overall system performance and the working of our approach, which can assist them in analyzing the root causes of performance regressions confidently and efficiently.

4.4 Approach

In this section, we briefly discuss the detailed process and the implementation of our approach for locating performance regression root causes. Figure 6.2 illustrates the overall process of our approach. Our intuition is that, if a system runtime activity (e.g., a web request) is associated with the deviation of the performance models built on two different software releases, then the system runtime activity is related to the performance regression between the two releases. It is also worth noting that, in order to deploy our approach for locating performance regression root causes, the target software system is supposed to be a web-based system deployed in the typical web server (e.g., Apache and IIS) and can generate logs that record the system runtime activities.

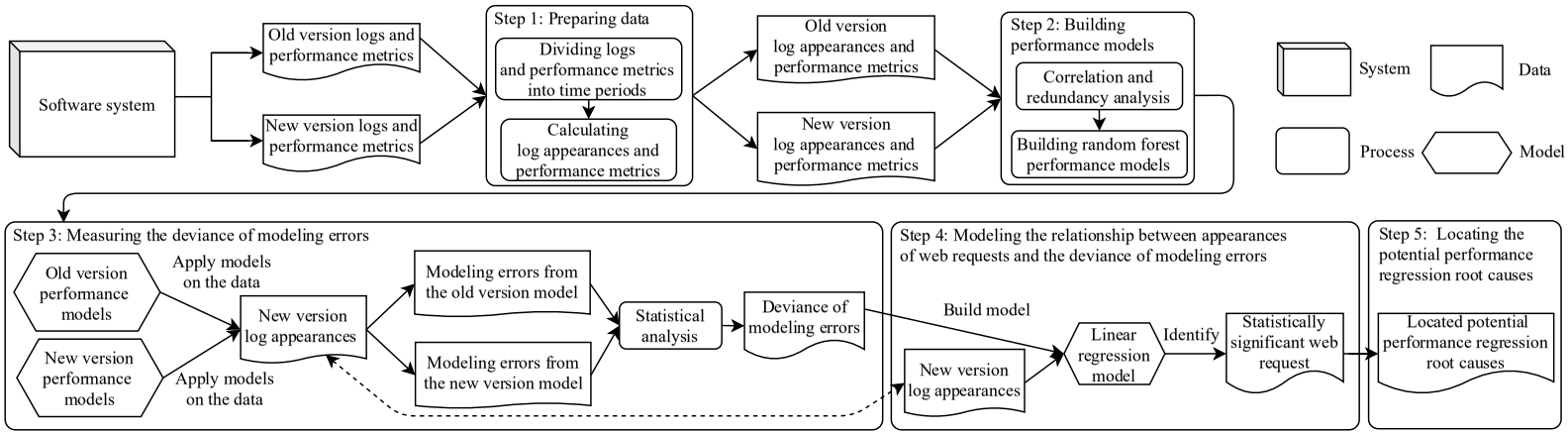


Figure 4.2: An overview of our approach to locating performance regression root causes

Step 1: Preparing data. In order to establish the relationship between the runtime activities of the system (i.e., the web requests) and the corresponding performance, we first need to align the web-access logs generated by the web servers (e.g., IIS) and the collected performance metrics (e.g., CPU usage). Specifically, we divide the data into small time periods (e.g., every 30 seconds) and allocate each line of web-access logs and each record of performance metrics based on their timestamps. Afterwards, we count the appearance of each type of web request in the web-access logs and we calculate the average value of the performance metrics during the time period.

Step 2: Building performance models. We build common black-box performance models [83], similar to prior research [36, 241, 233, 93], to capture the performance of a system and its runtime activities. We build one performance model for each version of the system. In particular, the independent variables of the model are the number of appearances of each type of web request in each time period, while the dependent variable is the corresponding performance metric, i.e., CPU usage, of each time period.

Similar to prior research using black-box performance models [36, 241], we also notice that different log events may always appear simultaneously in a specific order, e.g., a user logging in and then checking the user's inbox, and provide repetitive information for the workloads. Since such collinearity and redundancy among the independent variables may negatively impact the robustness of the performance models (i.e., the models built on old version data may not perform well on new data from the new version of the system), we remove the entangled independent variables that provide repetitive information about the workloads to avoid any bias. In particular, we conducted pair-wise correlation analysis to remove one variable from each pair of highly correlated independent variables whose Pearson correlation coefficient [46] is higher than 0.7. We also conducted a redundancy analysis [106] to remove any independent variables that can be modeled by the rest of the independent variables with a high model fit ($R^2 > 0.9$). Afterward, we opt to use the random forest regression model to build our performance models, due to its high accuracy shown in prior research [36, 241, 93].

Step 3: Measuring the deviance of modeling errors. Our intuition is that the root cause of a performance regression is related to the deviation of the performance model built on the new version from the performance model built on the old version. In particular, if the performance model built on the new version results in a worse modeling error than the performance model built on the old version, it is an indicator of a performance regression and can be used to locate the root causes of the performance regression. Intuitively, one may build the model on the old version of the system and then measure the modeling errors on the new version to determine the performance deviance between the old version and the new version. However, such a naive approach may be biased. For example, a

well-built performance model may only have less than 7% average prediction error; while another less fit performance model may have 10% average prediction error. In these cases, it is challenging to determine whether an average prediction error of 8% on the new version of the system should be considered as a performance regression. Therefore, we first build two performance models on the old version and the new version of the system separately and use the prediction error from the old version as a baseline to measure the deviance between the modeling errors of the new and old performance models. Specifically, we train two performance models using the old version data and the new version data, respectively, and apply the two models to the new version data to measure their respective modeling errors. However, we cannot directly calculate the modeling error of the new version's model with the new version's data, since applying a model to its training data can lead to over-optimistic results. To address this issue, we apply the leave-one-out approach [133, 53]. For each time period in the new version, we remove its data from the training data to rebuild the model and apply the rebuilt model to the time period. We repeat the process until all time periods are used as test data once.

Finally, having both the modeling errors from the old version and the new version for each time period, we calculate the deviance of modeling errors for each time period. To statistically measure the deviance of modeling errors, we analyze the random forest model and obtain the modeling errors from each decision tree inside the random forest. For a random forest with 100 decision trees, for each time period, we would have 100 modeling errors from the new version and the old version, respectively. Then, we calculate the Cohen's D effect size [72] between the modeling errors from the new and old versions (i.e., the deviance of modeling errors).

Step 4: Modeling the relationship between appearances of web requests and the deviance of modeling errors. Intuitively, if two versions of a software system have no performance deviance (no regression or improvement), the deviance of the modeling errors (from the last step) should be randomly distributed around zero. Otherwise, if there is a performance regression, there should be systematic deviance of modeling errors, and the web requests that contribute to the deviance are related to the root cause of the performance regression. Therefore, in this step, we use a linear regression model [92] to explain the relationship between the web requests and the modeling errors. The independent variables of the model are the number of appearances of each type of web request in each time period; the dependent variable is the corresponding deviance in modeling errors in that time period. We choose linear regression due to the fact that, unlike deep learning models that are usually considered as black boxes, linear regression is *explainable* as it has a good ability to explain the effect of each independent variable and we just need to explain such model in the next step (i.e., Step 5). Nevertheless, practitioners may also use other

explainable modeling techniques in this step. After building the linear regression model, we only keep the statistically significant independent variables (p-value $\leqslant$0.05), which are potential root causes of the performance regression.

On the other hand, the linear regression model may have no statistically significant independent variables or a poor model fit (very low R^2 cf. Section 4.5 and Section 4.6). Therefore, under this circumstance, the results mean that we cannot identify a relationship between the web requests and the deviance of the modeling errors, i.e., the new version of the system may not have any performance regression or improvement.

Step 5: Locating the potential performance regression root causes. In this step, we first rank the web requests that are potential performance regression root causes, such that practitioners can prioritize their efforts on the most likely root causes. Specifically, we calculate the **effect** of each statistically significant independent variable on the output of the linear regression model, i.e., the deviance of modeling errors. We keep all of the variables at their median value, except that we increase the median value of one variable by 10% and then re-predict the output. Then, we calculate the percentage of difference in the output caused by increasing the value of the variable (i.e., the **effect** of the variable). Such an approach has been widely used in prior software engineering research to understand the effect of independent variables [167, 198, 196]. These prior studies chose to increase the median value of one variable by either 10% or 100%, however, the exact increase of percentage does not impact the ranking of the results. We would like to note that, such an **effect** may be either positive or negative. Similarly, the appearance of a certain web request may also contribute to either better or worse performance of the systems. Therefore, we only rank the potential performance regression root causes if a higher appearance of the web request is associated with both worse performance (from the model built in step 2) and higher deviance in the modeling error (from the model built in step 4).

The ranked list of web requests that are associated with the performance regression can help developers find the root cause. However, in our approach, we step further and link the web requests to the specific code changes that lead to the performance regression to provide more detailed insights. First, we automatically search the entire source code of the software to locate the methods that are associated with each of the web requests that are related to the performance regression. Then, we use source code analysis frameworks, such as .NET Compiler Platform SDK called "Roslyn" [18] and Eclipse Java development tools called JDT [13], to parse the source code and build a call graph of the web request. We seek the code that is called by the request and identify all the places in the source code where the web request can be triggered, e.g., dynamically called by another web request. Finally, we identify the code changes that change any methods along with the call graph related to the web requests. In addition, we also provide the metadata of the commits that

contain these code changes, including:

- Commit hashes
- Timestamp
- Comment
- Committer
- Code churn
- Code difference

Such changed methods along with the commits (and the corresponding metadata) are considered the potential causes of the performance regression and they are provided to developers. We would like to note that since we built the call graph of the web requests that are related to the performance regression from the new version of the system and match all the new version code changes that change any methods along with such call graph, so even if the call graph of the new version of the system is different from the old version, our approach still can locate the root cause of the performance regressions.

4.5 Evaluation

We evaluate our approach on three open-source systems and one industrial system. In this section, we present our evaluation that is conducted on three open-source systems. We present our evaluation of the industry system in Section 4.6.

4.5.1 Open-source systems and their workloads

We use three open-source subject systems including *TeaStore*, *OpenMRS*, and *CloudStore* to evaluate our approach. Our three open-source subject systems are all web-based systems from different domains, which ensures that our findings are effective to a variety of web-based systems while not limited to a specific domain, and they are also studied as performance benchmarking systems in prior research [241, 86]. An overview of the subject systems is shown in Table 4.1.

Table 4.1: An overview of the open-source subject systems

Subject	Version	Domain	SLOC (K)
TeaStore	1.3.4	Microservice e-commerce	29.7
OpenMRS	2.1.4	Medical record system	67.3
CloudStore	v2	E-commerce	7.7

Note: SLOC of the subject systems is measured with cloc [11].

TeaStore is an open-source reference application that is designed to be used for benchmarking performance testing and modeling. Its main function is emulating a basic web store for tea and tea supplies [219]. *TeaStore* is developed in a distributed micro-service architecture and it consists of five distinct services (i.e., WebUI, Image Provider, Authentication, Recommender, and Persistence). In addition to five primary functional services, there is also a registry service responsible for the necessary service discovery and load balancing. We choose *TeaStore* since the result of previous work [86] shows that due to the sufficient complexity and performance properties of the system itself, *TeaStore* can serve as an appropriate candidate case study for performance modeling. We deployed version 1.3.4 of *TeaStore* in our case study. *TeaStore* has a few quintessential use cases, including logging in the system, browsing the store, browsing the user's profile, browsing products, shopping products, and logging out of the system.

OpenMRS is an open-source healthcare system that supports customizable electronic medical records and it is commonly used in developing countries. *OpenMRS* is built by a global open community that aims to improve health care delivery in resource-constrained environments by creating a robust, scalable, user-driven, and open-source medical record system platform [19]. We deployed the *OpenMRS* version 2.1.4 and the REST web services module version 2.24. The database data we used are from the MySQL backup files provided by *OpenMRS* developers. The demo database file contains data for over 5K patients and 476K observations. The typical usage scenario of *OpenMRS* consists of four operations: adding, deleting, searching, and editing resources. In total, we created eight different simulated system activities in our case study, comprising 1) creation of patients, 2) deletion of patients, 3) searching for patients by ID, 4) searching for patients by name, 5) searching for concepts, 6) searching for encounters, 7) searching for observations, and 8) searching for types of encounters.

CloudStore is an open-source e-commerce web application designed to be used in the scenarios of analyzing the cloud characteristics of systems, such as capacity, scalability, elasticity, and efficiency [12]. It was developed as a showcase application to validate the European Union-funded project which is called CloudScale [12]. It follows the functional

requirements defined by the TPC-W standard which is a web e-Commerce benchmark for transaction processing [8]. We deployed the *CloudStore* version v2 and the database data we used was generated using the scripts provided by *CloudStore* developers. The data in the database contains about 864K customers, 777K orders, and 300 items. We constructed the simulated system activities to cover searching, browsing, adding items to shopping carts, checking out, and paying for commodities.

Simulating dynamic field workloads. The goal of our approach is to locate the root causes of performance regressions in the field, i.e., without pre-assumption of consistent workloads between versions. To simulate such dynamic field workloads, we follow three steps to design our workloads: 1) Each subject system is driven by a mixture of multiple (four or five) JMeter-based load drivers, where each load driver has different profiles of mixed workloads. In particular, the original versions (i.e., the versions without injected regressions) are driven by four load drivers, and the new versions (i.e., the versions with injected regressions) have five load drivers, in order to ensure that there are some workload profiles unseen from the original version; 2) To simulate inconsistent workloads, for each load driver, the runtime activities of each system are driven in a random order, with a random length of gaps between two activities. In addition, we set different numbers of maximum concurrent users for different workloads at different times. Hence, each load driver itself produces inconsistent workloads in different time periods; 3) Furthermore, for each of the load drivers, we pick several different test actions and put them in an extra JMeter loop controller that iterates a random number of times to increase their appearances in the workload, to make sure that each of the five load drivers has different characteristics in workload actions. Each run of the system lasts a total of eight hours, where only the seven hours in the middle are used in our analysis (i.e., to exclude the warm-up and cool-down periods).

4.5.2 Injected performance regressions

For the open-source subject systems (i.e., *TeaStore*, *OpenMRS*, and *CloudStore*), we cannot find any historical performance regressions of specific versions via checking the system version history. Therefore, we had to manually inject four types of performance regressions in multiple arbitrarily selected positions of each subject system. Inspired by the previous work [102, 195, 93], we considered four types of performance regressions that are commonly encountered in practice and cover various software system performance aspects. The injected performance regressions are explained below:

A: Injected additional calculation. We added additional calculations to the source code that is frequently executed under our simulated workloads.

B: Generated excessive garbage collection. Creating large numbers of temporary objects will lead to excessive garbage collection and consequently high CPU utilization. Therefore, due to the fact that *Strings* are immutable objects in Java, we injected string concatenation operations (i.e., "+=") into the source code.

C: Added excessive I/O access. Since accessing I/O storage devices (e.g., hard drives) is usually slower than accessing memory, we added redundant logging statements to the source code that is frequently executed. The execution of the logging statements may cause excessive I/O operations and introduce performance regressions.

D: Created superfluous use of multi-threading. When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer, etc. of the current thread, and load the local data, program pointer, etc. of the next thread to execute. We introduced large numbers of threads which may cause the CPU to be busy switching from the context of one thread to the context of another.

In order to reduce the bias that may be introduced by injecting the synthetic performance regressions in a particular location in the source code, we manually examine the source code and arbitrarily identify the candidate locations in the source code without the preference for any specific locations, to inject each type of the synthetic performance regressions. Since the arbitrarily determined injected locations in the source code are selected through an unsystematic manual process, they may still be influenced by subjective factors. In order to mitigate this effect, in our experiments, we opt to separately inject each synthetic performance regression in four different arbitrarily selected places (i.e., p1 to p4) in the source code, which makes up a total of 16 different versions with performance regressions per open-source system.

4.5.3 Experiment setup

The experiments on the open-source subject systems are conducted in the Google Cloud Platform [15], where three separate virtual machines are set up for each subject system. These virtual machines have the same hardware configuration, which includes an Intel Haswell 4 cores CPU, 8GB memory, and a 300GB SATA hard drive. These virtual machines are connected to the same internal network. All virtual machines run the Linux Ubuntu 16.04 LTS (Xenial Xerus) operating system. We deploy the subject web application in Apache Tomcat [10] on the first machine (i.e., the web server). The second machine is deployed as a database server. Finally, we run the JMeter [9] load driver with varying workloads on the third machine to simulate real-world users using the system under test (SUT).

For the open-source subject systems, we use *pidstat* [20] to monitor the resource utilization (e.g., CPU usage) of the systems. To minimize the noise of other background processes, we only monitor the system resource usage for the process of the subject system. A higher sampling frequency of collecting performance metrics can capture the system performance more accurately. However, a higher sampling frequency would also introduce more performance overhead. For the purpose of achieving an optimal balance between the monitoring accuracy and the performance overhead, we monitor the CPU usage of the open-source systems every 10 seconds.

In this study, we use CPU as the performance metric for locating performance regression root causes, since our closed-source industrial system and three open-source subject systems are all CPU-intensive. Besides, CPU is usually the main contributor to server costs [100]. Performance regression in terms of CPU would result in the need for more CPU resources to provide the same quality of service, thereby significantly increasing the cost of system operations. However, our approach is not limited to the performance metric of CPU usage. Practitioners can leverage our approach to consider other performance metrics (e.g., memory and disk I/O) that are appropriate in their context.

4.5.4 Evaluation results of open-source systems

For each of the studied open-source systems, we first run the system without performance regressions (i.e., v0) under the combination of four different concurrent workloads. Then, we run the system with an injected performance regression under the combination of five different concurrent workloads. Our approach should be able to detect and locate the root cause of the performance regression. For comparison, we also run the system without performance regressions (i.e., v0) under the combination of five different concurrent workloads. Ideally, our approach should not detect and locate the root causes of performance regressions from this version. Table 4.2 and Table 4.3 show the detailed results of applying our approach to locate the root causes of performance regressions for the studied open-source systems (i.e. *TeaStore*, *OpenMRS*, and *CloudStore*). We use three evaluation metrics to evaluate our approach.

1) $\underline{R^2}$ is the model fit of the linear regression models that are built to model the relationship between the appearances of web requests and the errors of performance modeling. A higher R^2 indicates that certain web requests are associated with the performance modeling errors, i.e., being related to performance regressions. Therefore, the R^2 values from the models trained from the versions with injected performance regressions are expected to be higher than the ones without injected regressions (i.e., the "No regressions" column).

2) **Effect** with regressions. **Effect** quantifies the relationship between the appearances of one type of web request and the deviance of performance modeling error (cf. Step 5 in Section 4.4). The higher the **effect**, the more likely that the web request is the root cause of the performance regression. **Effect** with regressions is the **effect** of the web request where the performance regression is injected.

3) Highest **effect** without regressions refers to the highest **effect** of the web requests without injected regressions. Therefore, the values of the highest **effect** without regressions are expected to be lower than the **effect** with regressions. The bigger the difference between the two values, the better our approach can differentiate web requests with and without performance regressions.

In addition, we also measure the precision of using the existing baseline approach (cf. Section 4.2) to locate the injected performance regressions and present it in Table 4.5. We do not measure recall since false-positive results are the main limitation of the baseline approach. All the web requests with the injected performance regressions can be covered by the baseline approach (i.e., recall of 100%).

Our approach can effectively distinguish between system versions with and without performance regressions. Table 4.3 shows that, for all the subject systems, we obtain much higher R^2 values for the versions with injected performance regressions than for the versions without injected regressions. For example, for *OpenMRS*, the R^2 of the version without injected regressions is only 0.14, i.e., a poor model fit; while for all other versions with injected regression, the lowest R^2 is 0.50. Such a large difference in R^2 values shows the effectiveness of using the R^2 to detect performance regressions. The results also demonstrate the high quality of our models for capturing the relationship between the appearances of web requests and the errors of performance modeling when there are performance regressions; while when there is no performance regression, the low R^2 values indicate that the errors of performance modeling are not likely to be associated with the appearances of any particular web request.

Our approach can successfully locate the root causes of the injected performance regressions, always ranking the web requests with injected regressions in the first place. We find that for all the subject systems (i.e., *TeaStore*, *OpenMRS*, and *CloudStore*), after applying our approach, the web requests with the injected performance regressions always rank at the top 1st, i.e., with the highest **effect**. In addition, from Table 4.2, we observe that there is usually a large difference (e.g., twice the effect) between the **effect** of the web request with the performance regression (i.e., "Effect with regressions" in Table 4.2) and the highest **effect** from the web requests without injected performance regression (i.e., "Highest effect without regressions" in Table 4.2).

By a closer look at the gap between the **effect** with regressions and the highest **effect** without regressions from the evaluation results of the three open-source systems, we noticed that our proposed approach may perform differently between the software systems and between different injected regressions. In order to study such differences in a statistically rigorous manner, we use a non-parametric statistical hypothesis test called the Wilcoxon rank-sum test [224] to determine whether there exists a statistically significant difference (i.e., p-value $<$ threshold) between the gap between the **effect** with regressions and the Highest **effect** without regressions from different systems and between different injected performance regressions. To counteract the effect of multiple comparisons, in our study, the Bonferroni correction [50] is used together with the Wilcoxon rank-sum test [224] in the statistical analysis. For example, we have three different systems *TeaStore*, *OpenMRS*, and *CloudStore* to compare them pairwise, and in total there are three trials of testing needed (i.e., *TeaStore* vs. *OpenMRS*, *TeaStore* vs. *CloudStore*, and *OpenMRS* vs. *CloudStore*), so the Bonferroni correction would test each individual hypothesis at the threshold of 0.05 (original p-value threshold) / 3 (number of testing trials) $\approx$ 0.017. From the statistical test results between different open-source software systems presented in Table 4.4, we observe that none of our three subject systems (i.e., *TeaStore*, *OpenMRS*, and *CloudStore*) have statistically significant difference (i.e., p-value $>$ threshold) from the other two subjects at the same time, which implies that when applying our approach to the different subject systems selected in this work, there exists no obvious difference in the performance of locating the performance regression root causes. We also perform such statistical analysis between different injected performance regressions, and from the statistical test results between different performance regressions summarized in Table 4.4, we find that all the statistical test results (i.e., p-value) are above the significance threshold, which indicates that there exist no significant differences between the four injected performance regressions (i.e., performance regression A to D) and the root causes for these performance regressions are statistically equivalently located by our approach.

Table 4.2: Results of locating performance regression root causes for TeaStore, OpenMRS, and CloudStore

System	Metrics	Regression-A				Regression-B				Regression-C				Regression-D			
		p1	p2	p3	p4	p1	p2	p3	p4	p1	p2	p3	p4	p1	p2	p3	p4
TeaStore	Effect with regressions	0.07	0.06	0.07	0.08	0.08	0.06	0.06	0.12	0.09	0.08	0.06	0.13	0.10	0.06	0.07	0.09
	Highest effect without regressions	0.01	n/a	0.01	0.03	0.01	0.02	0.02	n/a	0.03	0.04	0.04	0.03	n/a	0.01	0.01	0.05
OpenMRS	Effect with regressions	0.06	0.10	0.08	0.10	0.06	0.08	0.07	0.06	0.07	0.15	0.14	0.11	0.09	0.06	0.08	0.13
	Highest effect without regressions	0.03	n/a	n/a	0.01	0.02	0.01	0.01	0.02	0.03	0.01	0.01	n/a	0.04	0.01	0.01	0.01
CloudStore	Effect with regressions	0.15	0.12	0.20	0.08	0.09	0.07	0.09	0.08	0.14	0.12	0.11	0.21	0.14	0.14	0.05	0.11
	Highest effect without regressions	0.01	0.07	0.04	0.05	0.01	0.01	0.02	n/a	0.02	0.01	0.03	0.02	0.02	0.04	n/a	0.06

Note 1: The “Regression-A” to “Regression-D” columns refer to the versions of the system where the performance regression is injected. For each performance regression, we separately inject that bug into four different positions, i.e., “p1” to “p4”.

Note 2: The value “n/a” of the “Highest effect without regressions” metric in the “Regression-A” to “Regressions-D” columns indicates that, in these versions, our approach generates only one candidate web request, i.e., the web request with an injected regression.

Table 4.3: Model fit (i.e., R^2) of the linear regression models built to model the relationship between the appearances of web requests and the performance modeling errors for TeaStore, OpenMRS, and CloudStore

System	No regressions	Regression-A				Regression-B				Regression-C				Regression-D			
	v0	p1	p2	p3	p4	p1	p2	p3	p4	p1	p2	p3	p4	p1	p2	p3	p4
TeaStore	0.30	0.54	0.36	0.56	0.40	0.70	0.63	0.72	0.45	0.40	0.65	0.66	0.59	0.62	0.56	0.75	0.50
OpenMRS	0.14	0.51	0.58	0.50	0.63	0.50	0.63	0.51	0.56	0.50	0.58	0.66	0.54	0.55	0.53	0.58	0.54
CloudStore	0.06	0.38	0.41	0.43	0.38	0.58	0.57	0.72	0.48	0.53	0.55	0.53	0.59	0.50	0.50	0.25	0.50

Note 1: "No regressions" column represents the version of the system without injected performance regressions and we call it "v0". The "Regression-A" to "Regression-D" columns refer to the versions of the system where the performance regression is injected. For each performance regression, we separately inject that bug into four different positions, i.e., "p1" to "p4".

Table 4.4: Statistical test results (i.e., p-value) of comparing the gap between the **effect** with regressions and the highest **effect** without regressions from different open-source systems and different performance regressions

Between different open-source systems			
	TeaStrore	OpenMRS	CloudStore
TeaStrore	-	0.243	0.024
OpenMRS		-	0.274
CloudStore			-

Note 1: p-value threshold is 0.017 (corrected by Bonferroni correction).

Between different performance regressions				
	Reg-A	Reg-B	Reg-C	Reg-D
Reg-A	-	0.686	0.326	0.840
Reg-B		-	0.112	0.665
Reg-C			-	0.260
Reg-D				-

Note 1: "Reg" is short for "Regression".
Note 2: p-value threshold is 0.008 (corrected by Bonferroni correction).

On the other hand, from Table 4.5 which presents the precision of the baseline approach of locating performance regression root causes for *TeaStore*, *OpenMRS*, and *CloudStore*, we observe that the baseline approach only achieves an average precision of 0.11, 0.45, and 0.52 in *TeaStore*, *OpenMRS*, and *CloudStore*, respectively, whereas our approach can achieve the top-1 precision as high as 1.0. In particular, when using the baseline approach to locate performance regression root causes on *TeaStore*, all the precision values are low ($\leqslant 0.2$). Such a low precision in the baseline approach would indicate that many root causes are located, but only one of them is true positive. The false positives in locating performance regression root causes in open-source systems confirm our experience that practitioners from *ES* have wasted much effort due to a large amount of false positive results produced by the baseline approach in *ES*.

After identifying the web requests that are associated with the performance regressions, our approach successfully matches the source code on the call graph of the web requests

Table 4.5: Precision of the baseline approach of locating performance regression root causes for TeaStore, OpenMRS, and CloudStore

Type	Position	TeaStore	OpenMRS	CloudStore
Regression-A	p1	0.10	0.33	0.50
	p2	0.11	0.33	0.14
	p3	0.13	0.50	1.00
	p4	0.07	0.07	1.00
Regression-B	p1	0.17	0.33	0.13
	p2	0.09	0.25	1.00
	p3	0.10	1.00	1.00
	p4	0.06	0.25	0.13
Regression-C	p1	0.08	1.00	0.50
	p2	0.20	0.33	0.25
	p3	0.14	0.25	0.14
	p4	0.13	0.25	0.25
Regression-D	p1	0.10	0.25	0.13
	p2	0.10	0.50	1.00
	p3	0.10	1.00	1.00
	p4	0.07	0.50	0.13

where the arbitrarily injected synthetic performance regressions are located in the source code. We would like to note that there is one code change (i.e., changeset in the context of *ES*) per regression and each has around 10 to 30 lines of changed source code.

4.6 A Success Story from an Industrial Deployment

Our approach has been deployed to locate root causes of performance regressions for an industrial software system (i.e., *ES*), on a daily basis. In particular, the first author of this work was embedded on-site with the development team of *ES* for over a year to enable a faster feedback loop from practitioners to guarantee the smooth adoption of our approach in the large-scale complex industrial setting. In this section, we present the results that are obtained from the industrial deployment. The workload for system *ES* used in our evaluation is not predetermined since *ES* is deployed in a production environment and used by real end users.

Our approach can successfully locate more root causes of real-world performance regressions with much higher precision than the baseline approach

We have deployed our approach for all releases of *ES* during the year 2020, which is a total of 22 releases **without** any pre-known (for both developers and authors) performance regressions. For each of the releases, we apply our approach to the field data and hold a 30-minute to one-hour meeting with the developers of *ES* to discuss the results in order to know whether each located root cause is a true positive result or a false positive result. The process of the meetings is as follows: Before attending the meeting, developers have no prior knowledge about our identified performance regressions between the old version and the new version of the system. During the meeting, we present the results of the performance regression root causes determined by our approach and discuss them with the developers to confirm whether each located performance regression root cause indeed results in the performance regressions or not. After confirming the performance regression root causes, developers would also inform us of the performance regressions that are not identified by our approach if there exist any. In particular, we first discuss the ranked (by the **effect** on the performance regression) list of web requests that are associated with the performance regression with developers. After that, we present the corresponding code changes that change any methods along the call graph related to the web requests. In addition, we also provide the metadata of the commits that include these code changes, e.g., commit hashes, timestamp, comment, committer, code churn, and the code difference to assist developers in locating the performance regression root causes (cf. Step 5 in Section 4.4).

From the meetings, we identified that three releases have performance regressions and developers were not aware of the existence of these performance regressions before. After the discussions with developers, we confirmed that these three releases indeed have performance regressions, and the located root causes of these performance regressions together with the corresponding code changes are also confirmed by developers. Furthermore, we note that there are no performance regressions that are known to developers but not detected by our approach. Specifically, there are three commits associated with one regression and one commit associated with each of the other two regressions. By further inspection of these commits, we observe that these commits vary greatly in size, i.e., with a wide range of changed source lines of code (SLOC) from 20 to 170. Based on the multifaceted information we have provided, developers can save a lot of time and effort while requiring relatively less expert knowledge to locate and fix the root causes of performance regressions compared to manually going through all the code changes between the old and new releases. For the other releases, the developers of *ES* have not yet known or received any reported performance issues from the end users as of the time of writing this work. However, since the evaluation is conducted in a real-life industrial setting, without any pre-known performance regressions, we cannot make an assertion that these releases are free of performance regressions.

Table 4.6: Results of locating performance regression root causes for the industrial system (i.e., *ES*).

	Baseline approach	Our approach
Total # releases	22	
# releases with detected performance regressions	18	4
# releases with confirmed performance regressions	3	3
# located root causes	65	6
# confirmed root causes	2	3
Precision	3.08%	50%

Note: The 3 releases with confirmed performance regressions from the baseline approach and that from our approach are the same.

Table 4.6 summarizes the results of applying both the baseline approach and our ap-

proach to locate the performance regression root causes for the industrial system (i.e., *ES*). In particular, for the 22 releases of *ES*, our approach has identified 4 releases containing performance regressions and there are 6 performance regression root causes located in these releases. After meeting with industry partners, we confirm that 3 of them are true positives, which shows that our approach achieves precision at 50%. Our approach can also successfully locate the root causes of these performance regressions and provide corresponding code changes. It is worth pointing out that we choose the threshold of **effect** at 0.04 in the industrial system (i.e., *ES*) to determine the performance regression root causes from a list of potential candidates (i.e., significant variables in the regression model (cf. Section 4.4)), we make this choice since in the results from the open-source subjects, the average highest **effect** without regression are around 0.05 and the average **effect** with regression are above 0.05, so based on this finding, we choose a relatively conservative threshold at 0.04 to not miss anything in the industrial environment. Whereas for the baseline approach, 18 out of 22 releases are deemed to have performance regressions and a total of 65 web requests have significantly slower response time (i.e., p-value < 0.05 and effect size is large) in the new version than in the old version, thus are located as the root causes of the performance regressions. Among these located root causes, only two of them are true positive, which shows the baseline has a low precision at only 3.08%. The results show that although we chose a rather strict threshold, the baseline still suffers from very low precision (a lot of false positives). On the other hand, although we don't know the true recall of our approach and the baseline approaches, we do know that one true positive case detected by our approach is missed by the baseline approach. In other words, even though with a low precision of the baseline approach, its recall is still lower than our approach.

For the releases with the confirmed performance regressions, we conduct further investigation to reveal how the performance regressions are introduced. The first performance regression our approach located is a code change in which developers added nested code loops whose computation has repetitive and partially similar patterns across loop iterations. The outer loop iterates over all the items in a data set and each item calls a method x, which in turn calls another method y. The inner loop in method y makes computations on all the items in that data set. The repeated computation is redundant and can be performed only once, as the values of the items do not change between the calls. The outer loop amplifies the performance penalty of the inner loop, which makes the performance regression even more severe. This regression was not identified by code review as it involves the interaction between multiple methods.

From the second located performance regression root cause, we observed that in the new version, developers added a complex SQL query that joins multiple temporary tables in order to load some employee-related information. However, such tables lack indexes

and the query is frequently executed, both of the factors make the corresponding query suffer from the performance regression. After we discussed with the developers who are responsible for this module, we confirmed this performance regression to the software.

The third performance regression located by our approach is caused by an inefficient SQL query that is business logic-related within a SQL stored procedure. After discussion and confirmation with developers, they optimized this query in the next release to provide the service in a much faster manner and consume fewer hardware resources (e.g., CPU).

Finally, by a close examination of the detection results of the 22 releases, we find that model fit (R^2) of the model that captures the relationship between the appearances of web requests and the deviance of modeling errors (step 4 in our approach) can be an effective indicator for differentiating releases with and without confirmed performance regressions. Specifically, the R^2 of the versions with the confirmed performance regression is up to 0.25, with an average value of 0.21; while for the other releases, the maximum R^2 is only 0.15, with an average value of 0.05. In addition, the **effect** values that are calculated in step 5 of our approach can also be used as an effective indicator. The confirmed root cause has an average **effect** of 0.05, while the **effect** from the other releases have a maximum of only 0.038 with an average value of 0.02. Such results also confirm with the evaluation of our approach on the open-source subject systems (cf. Section 4.5) that we can adopt the model fit (R^2) and the **effect** values as indicators to effectively identify the performance regression root causes.

4.7 Lessons Learned

In this section, we summarize the lessons that we learned from addressing the challenges while designing our approach and adopting our approach in the industry setting, in order to provide insights for researchers and practitioners who are interested in locating the root causes of performance regressions using the field-operation data. The corresponding faced challenges and solutions are described in Section 4.3.

Lessons learned from challenge 1

Black-box performance models can capture the relationship between the performance of a system and its dynamic activities in the field-operation environment. Software systems usually produce logs (e.g., web-access logs) at runtime, in order to track, monitor, and debug system runtime activities. Our study confirms that such readily

available logs can be used to understand the system performance. In order to understand how effective the black-box models are in capturing the relationship between the performance of a software system and its runtime activities, we build the black-box performance models on the data from the version of the software system without performance regressions. We then calculate the mean absolute percentage error (MAPE) of the models as the metrics to evaluate the effectiveness of the model. In particular, we measure the prediction errors (i.e., MAPE) using 10-fold cross-validation to avoid the bias of having the same training and testing data. From the results, we find that using a simple black-box model (i.e., a random forest regression model) can already achieve a high modeling performance with low MAPE at the value of 5.91%, 5.15%, and 8.28% for *TeaStore*, *OpenMRS*, and *CloudStore*, respectively, which demonstrates high effectiveness of using black-box models to capture the relationship between the system runtime activities and its performance.

Without the need for knowledge of the system's internal behaviors, black-box models rely on logs to model system performance. Black-box models are particularly helpful when the system is under field operation where the workloads and the system itself are constantly evolving.

> Without the knowledge of system internal behaviors, black-box performance models can help understand the relationship between the performance of a system and its runtime activities in the field-operation environment.

Lessons learned from challenge 2

Removing the collinearity and redundancy among system runtime activities can improve the performance of the black-box models. Some of the system runtime activities may be highly correlated and provide redundant information, which can adversely impact the robustness of the black-box performance models. We also explore how the degrees of correlation between independent variables affect the performance of our approach, i.e., locate the performance regression root causes using just the field-operation data of software systems. For example, table 4.7 shows the measured results when applying our approach to the *TeaStore* subject system with and without reducing the highly correlated and redundant system activities. From the table, we can see that, by applying the correlation analysis and redundancy analysis one can significantly reduce the number of the variables (i.e., from 32 to 8) and achieve a higher value of R^2 (i.e., 0.541) than not doing so (i.e., 0.402). Our results also confirm with prior work [17, 241] that the

collinearity and redundancy among the independent variables may have a negative impact on fitting and interpreting the models. More importantly, since the model without the variable reduction has lower performance (i.e., lower R^2 value), the relationship between the problematic runtime activities and performance regressions may not be well captured (i.e., cannot identify system activities associated with regressions). In contrast, the model with reduced variables can successfully locate the root cause of the performance regressions. In this case, performing correlation analysis and redundancy analysis can boost the performance of the black-box performance models.

Correlation and redundancy analysis may risk ignoring the system runtime activities that actually cause the performance regression. During the regular meetings with our industrial collaborator, we find that the removed highly correlated or redundant independent variables can be associated with the system runtime activities that lead to performance regressions. Therefore, we simply and practically maintain a mapping between the removed independent variables and the remaining ones to help avoid missing possible root causes of the performance regression.

> Removing the collinearity and redundancy among the independent variables can improve black-box performance models. However, one needs to pay attention to the removed independent variables as they may be associated with the system runtime activities that lead to the performance regression.

Table 4.7: An example of the influence of collinearity and redundancy among system runtime activities from TeaStore

Metrics	Without reducing variables	With reducing variables
# Variables	34	8
R^2	0.402	0.541
The rank of the root cause of the regression	n/a	1

Lessons learned from challenge 3

Statistical techniques can be used to identify system activities related to performance regressions. Many of the existing approaches in finding performance root causes need to rely on performance testing which is conducted in an in-house testing environment with predetermined workloads or they can only locate the performance regressions root cause at a relatively high level (e.g., service level). On the other hand, based on our results, we observe that, when the system runs in a production environment, statistical techniques (e.g., linear regression) can be effectively leveraged to identify the system activities that are related to performance regressions.

Ranking the system activities that are related to a performance regression can help developers prioritize their efforts. There are still considerable resources and efforts needed to manually check all the system activities related to a performance regression. Therefore, we use statistical techniques to prioritize the candidate system activities that may cause the performance regression to help developers optimize their resources and efforts.

> Statistical techniques (e.g., linear regression) can be used to identify and prioritize system runtime activities related to a performance regression.

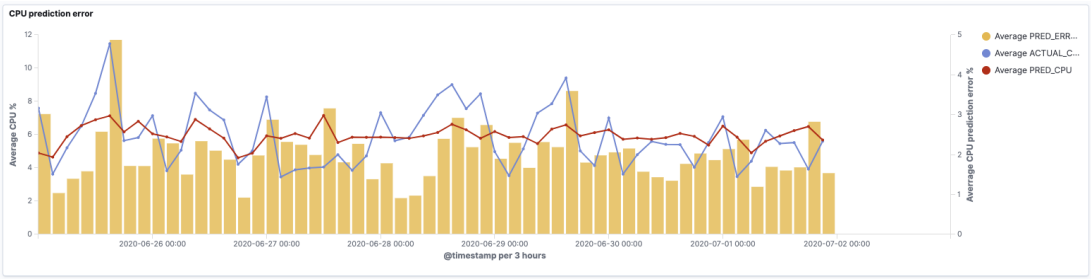
Lessons learned from challenge 4

A combination of code-level recommendations and high-level guidelines for developers can improve their analysis of the causes of performance regressions. Using static program analysis to recommend code changes that are related to performance regressions can help developers narrow down the causes of performance regressions. In addition, as system performance is usually complicated and most of the developers from our industrial partner are not experts in system performance, we also provide developers with high-level performance guidelines from previous research, e.g., introducing locks and synchronization may introduce performance regressions. Both the code-level recommendations and the high-level guidelines can assist developers in diagnosing the causes of performance regressions.

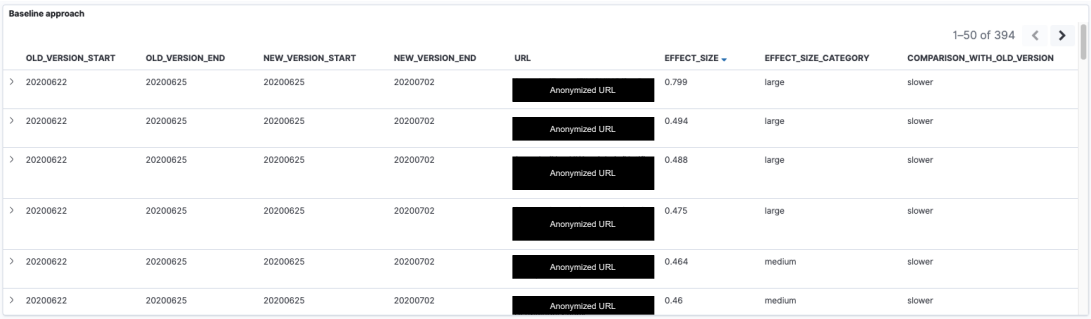
The granularity of locating the code changes leading to performance regressions is a trade-off. Locating the code changes at a high granularity, like the directory or file level, can be much easier to achieve. However, developers still need much effort to

gain insight into the provided scope. On the other hand, locating the code changes at a low granularity, like instruction or line level, can provide much detailed information, while it is more difficult to achieve this granularity since performance regressions are often introduced not with merely a single line of code [58]. After discussing with the developers of our industrial partner, we chose to locate code changes at the method level as it is sufficient for them to understand and fix the performance issue.

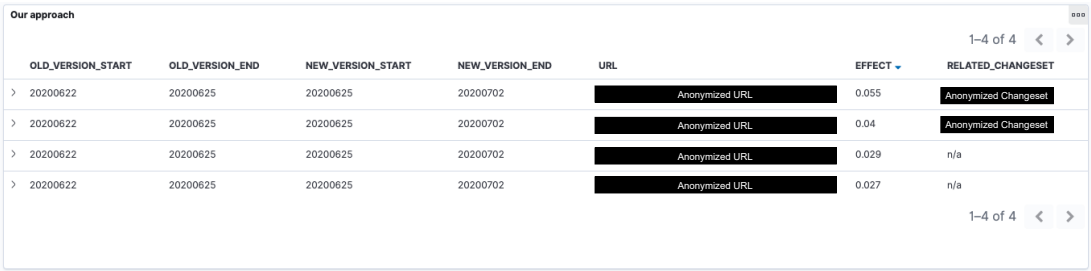
> Providing code-level recommendations together with high-level performance guidelines can assist developers in understanding and fixing performance regressions.



(a) Visualization of CPU prediction error



(b) Visualization of the baseline approach's result



(c) Visualization of our approach's result

Figure 4.3: Screenshot of our Elastic Stack unified management platform

Note 1: Our approach locates 4 instead of 394 performance regression root causes.
Note 2: The URL and related changeset are known to the authors, but are anonymized due to the NDA.

Lessons learned from challenge 5

Finding a balance between the monitoring accuracy and the monitoring overhead is crucial. System monitoring infrastructures usually come with performance over-

head. When applying research to practice, we would suggest that it is of great importance to minimize the performance overhead of our approach while keeping sufficient monitoring accuracy.

Visualization can assist in the adoption of our approach in our industry partner's environment. By visualizing the working mechanism of our approach, developers use our approach more frequently. In addition, visualization also improves the efficiency of analyzing the root causes of performance regressions. In particular, we present the screenshots of the major parts of the Elastic Stack unified management platform in our industrial environment in Figure 4.3. We would like to note that the actual detected URLs and the changeset IDs are anonymized with black color since they contain confidential information from our industrial collaborator. Specifically, there are three main components in our Elastic Stack platform: 1) Visualization of CPU prediction error. As shown in Figure 4.3a, this component presents the actual target performance metrics (i.e., CPU) in the blue line, the predicted performance metrics by the performance model in the red line, and the prediction errors (i.e., calculated by the absolute difference between actual and predicted performance metrics) in yellow bars; 2) Visualization of the baseline approach's result. As shown in Figure 4.3b, this component visualizes the performance regression root cause located results from the baseline approach including effect size, effect size category, whether it is faster or slower in the new version compared to the old version for each URL; 3) Visualization of our approach's result. As shown in Figure 4.3c, this component presents the results of locating performance regression root cause which include a list of URLs, and for each URL we also show the corresponding effect and the ID of associated code changes (i.e. "RELATED_CHANGESET") during the release period. If we cannot find code changes that are associated with the URL, we then mark it as "n/a". Components 2) and component 3) also indicate the release date information of the old and new versions in columns "OLD_VERSION_START", "OLD_VERSION_END", "NEW_VERSION_START" and "NEW_VERSION_END". In summary, we have a suggestion for developers that an interactive visualization is crucial for the successful transfer of research into practice.

Finding an optimal balance between the monitoring accuracy and the monitoring overhead is crucial to the adoption of our approach in practice. In addition, visualizing the working mechanism of our approach improves the understandability and usability of our approach.

4.8 Threats to Validity

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

4.8.1 External validity.

In our approach, we target locating performance regression root causes for just the web-based systems for the following considerations: first, the state of practice is an important starting point that motivates our study and the studied industrial system (i.e., *ES*) is a commercial web-based software system that is developed with .NET framework and deployed in the Microsoft IIS (Internet Information Services) web server; second, the system access logs are the minimum requirement in our proposed approach as such access logs represent the workload of the web-based system during a period of execution; last but not least, unlike other systems (e.g., mail server applications), the web servers in which the web-based systems are deployed, such as Jetty, Tomcat, and IIS web servers, are able to automatically generate the system access logs during system runtime. More investigation and studies on locating performance regression root causes in other types of systems are in our ongoing future work.

Our study is only conducted on one industrial system (i.e., *ES*) and three open-source projects (e.g., *TeaStore*, *OpenMRS*, and *CloudStore*). The three open-source projects are all benchmark systems and they do not aim to demonstrate the quality of the development process, thus, there may be performance issues in the system, but these issues are not indicated in the development history. Since we do not have evidence of historical performance regressions of specific versions via checking the system version history, we manually injected four types of synthetic performance regressions that are described in Section 4.5. Although the practice and results from the industrial deployment of our approach can compensate for the open-source experiments for demonstrating the effectiveness of the proposed approach, more case studies on other software systems with other types of performance regressions can benefit the evaluation of our approach.

4.8.2 Construct validity.

In our work, we use traditional system monitoring tools (e.g., pidstat) to collect the system runtime performance data (e.g., CPU usage). The quality of the recorded system performance data may be a threat to the construct validity of this study. Besides, we use CPU usage as our performance metric to locate performance regression root causes since

in the case of our industrial system and the studied open-source systems, CPU usage is the main concern in performance regressions. However, our approach is not limited to the performance metric of CPU usage. Evaluation with more performance metrics may lead to a better understanding of the applicability of our approach. Similarly, practitioners can consider other performance metrics that are appropriate in their own context while applying our approach to solve their problems.

4.8.3 Internal validity.

Our approach captures the relationship between the system runtime activities and the measured performance of a system. In order to achieve that, we utilize machine learning techniques to model such a relationship. Although our models achieve a good fit, we admit that the relationship between the runtime activities recorded in the logs and the measured system performance does not necessarily suggest a causal relationship between them. Another threat to the validity of our work is that in the experiments of open-source subjects, the locations in the system source code where we inject the synthetic performance regression are arbitrarily chosen without the preference for any specific locations (i.e., to avoid the bias that may be introduced by injecting performance regressions in a particular location). However, such arbitrarily determined injected locations in the source code are selected through an unsystematic manual process, therefore, they may still be influenced by subjective factors. To mitigate this threat, we opt to separately inject each synthetic performance regression in four different arbitrarily selected places in the source code, which makes up a total of 16 different versions with performance regressions per open-source system.

Our approach aims to automatically locate the root causes of performance regressions without the need for performance testing (i.e., requiring just the field-operation data of software systems). However, since the software systems are already running in the production environment, if the version of the deployed system indeed has performance regressions and runs in the field for hours or days before discovering these performance regressions, the delay in locating the performance regressions root causes may potentially pose an adverse impact (e.g., higher resource utilization, slow user response, and even financial losses) on the end users and the company. In addition, we are aware that if a new version has fundamentally different features (i.e., different types of runtime activities) that do not exist in the old version, the model constructed on the old version would not have the knowledge about the performance impact of the new features, making it impractical to compare performance models between these two versions. Hence, our approach is not applicable to

locating the root causes of performance regressions introduced by the features that only exist in the new version.

4.9 Conclusion

In this chapter, we provide an experience report on the challenges and lessons learned from designing and adopting our automated approach for locating performance regression root causes in practice. Our approach uses the readily available web-access logs (by default generated by the web servers) and performance metrics that are directly generated when the system is running in the field (i.e., interacting with end users) without the need for time and resource-consuming in-house performance testing. In particular, our approach relies on machine learning models and statistical techniques to identify the factors that contribute to the difference between the models built on two software releases. By evaluating our approach on three popular open-source projects (i.e., *TeaStore*, *OpenMRS*, and *CloudStore*) and applying our approach on a large-scale industrial system (i.e., *ES*), we find that our approach can successfully locate the root causes of both the arbitrarily injected synthetic performance regressions in the open-source systems and the real-world performance regressions in a large-scale industrial system. We believe that our approach and documented experience can benefit both practitioners and researchers in the use of field-operation data as a main source to conduct performance assurance activities during their fast-paced software development and release cycles.

Chapter 5

Adapting Existing Performance Analytic Techniques in a Real-Life Database-Centric System

Database-centric architectures have been widely adopted in large-scale software systems in various domains to deal with the ever-increasing amount and complexity of data. Prior studies have proposed a wide range of performance analytic techniques aimed at assisting developers in pinpointing software performance inefficiencies and diagnosing performance issues. However, directly applying these existing techniques to large-scale database-centric systems can be challenging and may not perform well due to the unique nature of such systems. In particular, compared to typical database-based systems like online shopping systems, in database-centric systems, a majority of the business logic and calculations reside in the database instead of the application. As the calculations in the database typically use domain-specific languages such as SQL, the performance issues of such systems and their diagnosis may be significantly different from the systems dominated by traditional programming languages such as Java.

In this chapter, we share our experience of adapting performance analytic techniques in a large-scale database-centric system from our industrial collaborator. Our adapted performance analysis pays special attention to the database and the interactions between the database and the application with minimal reliance on expert knowledge and manual effort. Moreover, we document our encountered challenges and how they are addressed during the development and adoption of our solution in the industrial setting as well as the corresponding lessons learned. We also discuss the real-world performance issues detected

by applying our analysis to the target database-centric system. We anticipate that our solution and the reported experience can be helpful for practitioners and researchers who would like to ensure and improve the performance of database-centric systems.

5.1 Introduction

Performance is critical to the success of a modern software system since it directly impacts the system's speed, responsiveness, and reliability, which are crucial to user satisfaction and productivity [49]. Unsatisfactory performance of the system may potentially lead to significant adverse consequences to the stakeholders of the system, e.g., additional operation costs, damaged customer relations, or loss of profit [227, 200, 204]. This is also the case for our industrial collaborator. In particular, the system from our industrial collaborator is an enterprise application (*EA*) that provides important government regulation-related reporting services to clients across the globe, and due to the importance of the services, the system performance is considered a vital matter. *EA* is described in detail in Section 5.2.

Due to the practical advantages in the ease of development, scalability, maintainability, and reliability [134], our target industrial system (i.e., *EA*) adopts a database-centric architecture based on two tiers (i.e., application and database) to cope with the continuously increasing variety, quantity, and complexity of user data. Unlike typical database-based systems, like online shopping systems, where only simple CRUD (create, read, update, and delete) operations are applied to the database, *EA* extensively extends the database with pre-defined stored procedures to perform in-database operations to process business logic and calculations. By placing the database at the central position, the application written in regular source code (e.g., C#) is designed around the database and only serves as a connector between user requests and complex business logic and calculations in database stored procedures.

In recent years, many studies have proposed analytics-driven approaches to assisting in analyzing software system performance, such as monitoring and managing tremendous performance data [67, 179, 108, 216], identifying software performance issues [175, 90, 195, 157, 233, 161], and locating the root causes of performance issues [32, 111, 155, 58, 183]. Inspired by prior studies, we have deployed traditional performance monitoring and analytics in *EA* to assist developers in ensuring the system performance. However, since the existing analytic techniques are not originally designed for database-centric systems, we face major challenges in the target industrial context. In particular, the results of traditional performance analytics consider only the application part of the system, while in our target system, the application source code is simply responsible for connecting

user requests to the complex business logic and calculations residing in the database. Focusing solely on the application and ignoring the crucial impact of the database and the interactions between the database and the application often leads to failures in identifying real performance issues and their underlying root causes.

In this chapter, we present our industrial experience of adapting performance analytic techniques in the large-scale database-centric system (i.e., *EA*) from our industrial collaborator. During the development and adoption of our solution, we encountered many engineering challenges. In particular, we first need to expose the runtime measurement of the database for performance analytics. We then need to expose the source code from the database for performance root cause analysis. For these challenges, we document our solution to address them and what we have learned during the process. We also share the implemented analysis in the target industrial setting and discuss the real-world cases detected by our analysis. We believe that our experience and learned lessons can provide software practitioners and researchers with helpful insights into ensuring and improving the performance of database-centric systems.

The main contributions of this study include:

- We provide an industrial experience report that discusses the encountered challenges, proposed solutions, and the learned lessons when adapting performance analytic techniques in a real-world database-centric system.
- We present the implemented analysis that can effectively assist developers in ensuring and improving the database-centric system performance from multiple aspects, including identifying performance anomalies within one release, detecting performance regressions between two releases, and locating the corresponding root causes.
- Our engineering process and experience can be helpful for software practitioners and researchers who are interested in ensuring and improving the performance of database-centric software systems.

The remainder of the chapter is organized as follows. Section 5.2 introduces our target system and explains the motivational background. Section 5.2 discusses the encountered challenges, our proposed solutions, and the learned lessons. Section 5.4 describes our implementation of the adapted performance analytic techniques. Section 5.5 discusses the detected real-world performance issues and improvements. Section 5.6 discusses the threats to the validity of our study. Finally, Section 5.7 concludes the chapter.

5.2 Background

In this section, we briefly discuss the industrial system that we study and the backstory for adapting performance analytic techniques for the target system.

The industrial system under study. The industrial system (i.e., *EA*) studied in this work is a large-scale database-centric enterprise application from our industrial collaborator. *EA* is a commercial software system that provides reporting services related to government regulations (e.g., environmental regulations) to enterprises across the globe and it is also the market leader in its field. Due to the non-disclosure agreement (NDA), we cannot give the exact details about the system. *EA* has more than 20 years of history and is currently under active development for new features and maintenance activities. *EA* has more than two million lines of code based on the Microsoft .Net framework and adopts a database-centric architecture. The database plays a more vital role in the system and is extensively used by other software components (e.g., the web application) designed around it. *EA* utilizes the Microsoft SQL server as the database server to store all the user data and extensively extends the database with pre-defined stored procedures to perform in-database operations to process the business logic and calculations. The database is large in scale and high in complexity, consisting of more than 1,000 stored procedures and 5,000 tables.

The current practice in *EA* and its limitations. Due to the large client base and the importance of the service, the performance of the *EA* is a primary concern of our industrial collaborator. Like many other industrial systems, *EA* follows fast-paced agile software development and release practices. Therefore, instead of conducting resource- and time-consuming in-house performance testing before each new release, the existing performance assurance activities are conducted based on the system runtime data (e.g., web access logs or CPU usage) that is directly collected under real-world client operations in a canary releasing [2] manner. In particular, the new version of the system runs simultaneously with the old version, while the new version only serves a small portion of clients. After making sure there are no performance issues in the new version, it is then expanded to reach more clients until its full release.

The existing practice of performance analysis consists of the following two main parts. First, we put all the collected system performance metrics (e.g., CPU or memory usage) under the management of the ELK stack for data storing, processing, and visualization. Through the visualizations of the collected performance metrics, developers can have an overall comprehension of the system performance and then utilize the expertise of the system to further diagnose the performance issues. Furthermore, we also extract the response time for each type of web request from web application execution logs and adopt

a pair-wise analysis to compare the response time of each web request type across versions of the system. We utilize statistical hypothesis testing (i.e., Wilcoxon rank-sum test (WRS) [224]) for the comparison and the null hypothesis assumes that the response times of a web request type are similar between the two versions while the alternate hypothesis is that there is a statistical difference (can be regression or improvement). The null hypothesis is rejected when p-values are smaller than 0.05. If there are web requests detected slow in the new release, we further adopt static code analysis techniques to identify the associated code changes in the commit history. Such slow web requests and code changes are provided to senior developers for further investigation.

However, there exist major limitations in the current practice of performance analysis. One limitation is that our existing practices focus solely on the application and do not have enough information on the crucial impact of the database (e.g., database performance or runtime behaviors) and the interactions between the database and the application (e.g., how application code relates to database operations). Such a limitation often leads to failures in identifying the performance issues and their underlying root causes. Additionally, in our experience, we have observed that the pair-wise analysis often reports a large number of web requests as performance issues even though only a few or none of them were confirmed by senior developers (i.e., false positives), which leads to developers wasting a significant amount of time and effort in examining the results. Furthermore, unlike in-house performance testing where the workload is often fixed or constant, the real workload in the production environment is constantly changing, which makes it challenging for developers to determine whether the performance impact is caused by performance issues or workload variations solely by visualizing the collected performance metrics.

Current practices of addressing these challenges largely rely on developers' manual analysis in an ad-hoc manner, however, due to the increasing complexity and scale of the system, such a manual process can be extremely difficult and tedious, and even senior developers with expert knowledge may still need to spend a significant amount of manual effort and extensive time to diagnose the performance issues and fix them. Worse, the delay in the detection and fixing of performance issues may potentially cause a significant adverse impact in terms of both the reputational and financial aspects of the company.

Our proposed solution. In order to overcome these limitations, we propose to adapt performance analytic techniques in the context of the target database-centric system (i.e., *EA*). In particular, our adapted solution pays special consideration to the performance of the database and the interactions between the database and the application. Further, we aim to provide multi-aspect analysis, including detecting performance anomalies within one release, identifying performance regressions between two releases, and locating the root causes. Our approach has been adopted in the internal platform of the industrial

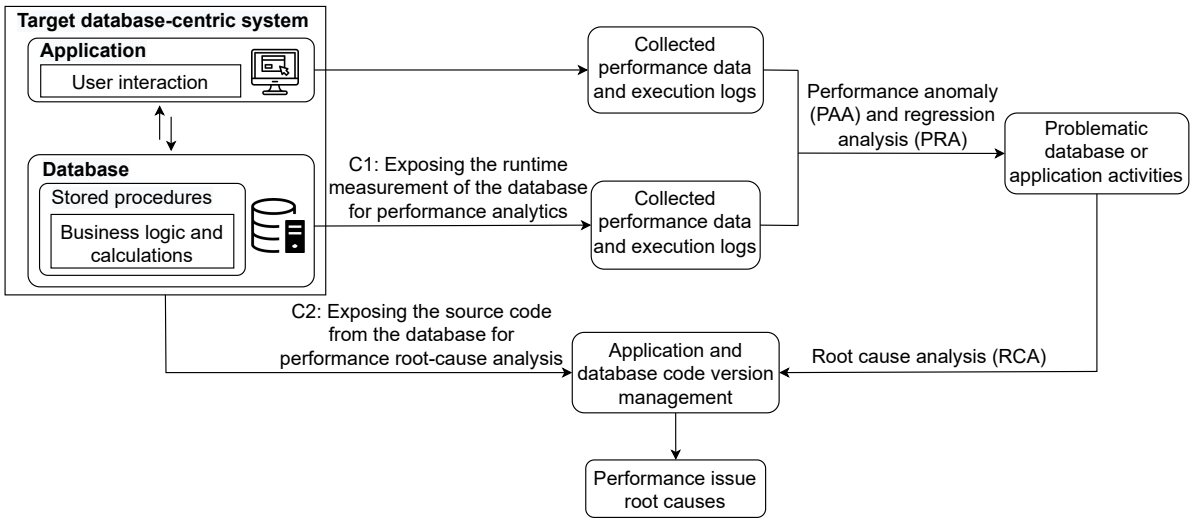


Figure 5.1: An overall engineering process of adapting performance analytic techniques in *EA*

collaborator to assist developers in ensuring and improving the system performance on a daily basis.

5.3 Challenges and Lessons Learned

In this section, we discuss the challenges encountered during the process of adapting performance analytic techniques to our target real-world system (i.e., *EA*) that is database-centric and large in scale. For each challenge, we describe the corresponding solutions and the lessons that we learned from addressing the challenge.

Figure 5.1 outlines our engineering process. Overall, we first expose the runtime measurement of the database for performance analytics, then we expose the source code from the database for performance root-cause analysis. The details of our implemented analysis are described in Section 5.4 and the detected real-world performance issues are discussed in Section 5.5.

5.3.1 C1: Exposing the runtime measurement of the database for performance analytics

Challenge. Analyzing system performance is a critical task for a large-scale software system since it helps to guarantee and improve the system performance. In order to adapt the existing performance analytic techniques in *EA*, we first need to understand how the system behaves under real-world workloads, especially when the system misbehaves or exceeds our expectations. However, unlike traditional applications, whose system performance or runtime behaviors are well captured by many available techniques, for example, application performance monitoring (APM) tools monitor and manage the performance of the application, and web servers (e.g., Apache Tomcat [10]) typically generate web access logs by default in order to support performance analysis, the performance and runtime behaviors of the database are often unexposed or not well understood. In database-centric systems without such essential information, when severe performance issues arise, what developers can only do is make decisions based on uninformed guesses or personal experiences. On the other hand, database-centric systems tend to have frequent database access operations that need to be captured. We could not add too much monitoring overhead to the system since the performance of the database is critical. Therefore, our first challenge is to comprehensively understand how the database in *EA* behaves under dynamic field workloads for performance analytics, without introducing significant monitoring overhead to the system.

Solution. To address the challenge, we propose to leverage two types of monitoring solutions to gain insight into how the database in *EA* performs under the dynamic field workloads and its corresponding runtime behavior from different perspectives and granularities. Resource utilization (e.g., CPU or memory usage) is an important aspect to consider when understanding the performance of a system, especially when some resources are overused or their utilization becomes unreasonable. Therefore, we first add the recording of system-level performance metrics for all the database servers (based on Windows platforms) by utilizing TypePerf [22]. TypePerf is a lightweight tool in Microsoft Windows operating systems used for performance monitoring. It provides a wide range of performance counters to collect the system performance, such as CPU usage, memory utilization, and disk activity, and it can output the data in various formats, like CSV or binary log files. These collected performance counters provide us with the window to observe how busy various resources of the system are when running under a specific workload.

Furthermore, to support a more comprehensive performance analysis of *EA*, we also leverage sp_WhoIsActive [3] to record a more fine-grained (i.e., at the query level) performance and runtime behavior of the database. sp_WhoIsActive is a powerful and lightweight

stored procedure that can help gain important insights into the internal workings of the SQL Server database. sp_WhoIsActive provides real-time information about running processes, such as query text, execution time, I/O activity, session information, and the outer batch or stored procedure information. By combining the abundant information from these sources, developers can obtain a clear and comprehensive understanding of the database performance in *EA*. This provides developers with a solid foundation to analyze which component may be malfunctioning and the corresponding contextual data, such as what else is running on the system, which helps identify and troubleshoot performance issues.

To avoid significant performance overhead to the system while keeping an adequate amount of the system runtime information, we have attempted to performance test the system with various data collecting intervals (e.g., 10 to 60 seconds) under different workloads. Through working closely with the senior developers, we determined that recording system-level performance metrics every 30 seconds and query-level database runtime behaviors every 15 seconds achieves an acceptable balance between the accuracy and the overhead of the measurement. It is worth noting that we also put these massive and valuable data that record system performance and runtime behaviors in our ELK Stack-based performance data platform (cf. Section 5.2) for the unified management and preliminary visualizations. Our solution enables further performance anomaly analysis (PAA) and performance regression analysis (PRA) described in Section 5.4.

Lessons learned. Leveraging monitoring techniques to understand how the database in the database-centric system behaves under dynamic field workloads with minimal performance overhead. Due to the high complexity and large scale of modern software systems, if there is a lack of adequate information about the system performance and runtime behavior, it can be extremely challenging to analyze and diagnose the system performance when the system misbehaves. This is even harder for large-scale database-centric systems since the database plays a vital role in the performance of such systems, and the performance and runtime behaviors of the database are usually unexposed or not well understood. Therefore, it is crucial to have a comprehensive monitoring solution from various perspectives to record the database performance and runtime behavior to gain more insights into how a database-centric system behaves under dynamic field workloads.

On the other hand, collecting more system runtime information (e.g., in a high recording frequency) often means more usable data and a more accurate representation of the system performance. However, it also brings more performance overhead, which is a major concern of developers, especially in the production environment. Therefore, it is critical to work closely with senior developers to run various experimental trials with different recording frequencies at various workloads to understand the introduced performance impact. While maintaining enough data and accuracy for the analysis, it is important to

minimize additional performance costs to increase developers' confidence in adopting our solution in practice.

Providing centralized management and visualization support to developers. Based on our experience, we observe that large-scale database-centric systems tend to have a large number of performance metrics and runtime behavior logs since the system often involves frequent database access operations that need to be monitored and recorded. Such data are valuable and contain rich information about the system's workload and running status for further performance analysis of the system. Therefore, putting these runtime monitoring data under the management of centralized processing and storage mechanisms (e.g., in the ELK stack-based platform) is important. In addition, putting these system runtime data into visualization also helps developers obtain a preliminary grasp and analysis of the performance of the large-scale database-centric system in a more intuitive manner.

> We monitor both the system-level performance and query-level runtime behavior to comprehensively understand how the database in a database-centric system performs under dynamic field workloads. Minimizing monitoring overhead and providing centralized management and visualization of the collected data is helpful to increase developers' confidence in adopting our solution.

5.3.2 C2: Exposing the source code from the database for performance root-cause analysis

Challenge. In *EA*, the database is highly complex and large in scale, comprising over 5,000 tables and 1,000 stored procedures, with an average of over 300 lines of code (LOC) per stored procedure. Unlike application source code that is well managed by various modern version control systems, the code changes in the database are often treated as second-class citizens and are left out of the development process. This is because database code may be strongly integrated with the database (e.g., SQL code in stored procedures), which makes it more challenging and requires more specialized knowledge and skill sets to manage the changes. On the other hand, in a database-centric system, the application and the database are often tightly connected, for example, the SQL queries are often highly dynamically constructed through a series of string operations (e.g., concatenation and substitution) in the application code and the application code usually invokes stored

procedures to perform in-database operations to process the business logic and calculations. Since the database plays a critical part in the performance of a large-scale database-centric system, if there is no proper management of database code changes and the link between the application code and the database code, when severe performance issues arise in the system, it becomes even more challenging to diagnose their root causes.

Solution. Exposing database code histories. In order to expose database code histories, we implemented a lightweight solution to automate the process of managing the database changes in version control. In particular, we first export a version of the database schema, including table structures, views, functions, triggers, and stored procedures as text files. After that, we link these files to a version control system (e.g., Git or TFVC). Whenever there are new database schema changes made in the new version of the system, it will trigger our automated tool to export the database schema again and developers can commit it to version control. By following such a process, it becomes straightforward for developers to track what the change to the database was and visualize the difference over time. More importantly, it also makes it possible to revert to previous versions of the schema if needed and trace back to the source changes in the database when analyzing and diagnosing the performance issues in the system.

Linking database activity to application code. We developed an automated solution that helps developers link database activities to concrete application code. Our solution first extracts the accessed database tables in the application code that generates database queries and the accessed database tables in the database query. By matching the accessed database tables from the application code and database query, we can establish a coarse-grained mapping between the database activity and the application code. Based on this mapping, we then leverage static code and string analysis techniques to further capture a more fine-grained relationship between database activity and application code. In particular, starting from each location in the application code that executes a database query, we extract a list of possible query strings by analyzing intra-procedural and inter-procedural control flow with the help of the abstract syntax tree (AST). We then calculate their similarities to the database query to link the database activity to the application code. We find that by conducting the analysis, the required time and effort for developers to link database activities to the application code can be significantly reduced.

Linking application activity to database code. In an effort to link application activity to database code, we first utilize static code analysis techniques to inspect the entire application source code and construct a call graph for the target application activities (e.g., web requests). Through traversing the methods in this call graph, we search for locations where the *SqlCommand* object is created with *CommandType.StoredProcedure* as its command type. We then extract the name of the stored procedure being executed

in the method, which is passed as the command text parameter. By doing so, we can successfully link the application activity to a list of database stored procedures that the activity may invoke.

Our solution of managing database code changes and building the link between the application code and the database code enables further root cause analysis (RCA) presented in Section 5.4 when we identify any performance issues in the system.

Lessons learned. Managing the changes of the database is critical for the performance analysis of the database-centric system. Database version control brings many benefits in terms of improving management efficiency and performance in database-centric systems. In particular, by checking the database version control, it is convenient for developers to understand what changes have happened to the database, what has been done currently, and who is doing the changes. Having the database code changes under version control also helps developers and database administrators to synchronize applications and databases better, which makes it easy to map the application changes to the database changes. Furthermore, in the event that any performance issues related to the database are identified, developers can promptly reference the database source changes and efficiently resolve them.

In large-scale database-centric systems, the performance issues are often due to the activities in the database rather than in the application. Analyzing and diagnosing the performance issues in database-centric systems are quite different from that in other systems. All too often, the developer cannot locate the root cause of performance issues in the application code, since most of the business logic and calculations are processed by stored procedures that run in the database, rather than relying on the logic running in the application. With the help of static code analysis techniques, we can locate not only the application code but also the database code that is related to the problematic application activity, which effectively helps developers diagnose the real root cause of performance issues.

Leveraging static code and string analysis can help link the application code and dynamically generated SQL queries. Database activities, such as SQL queries, are often constructed by application code in a highly dynamic manner. Even the same code may generate significantly different SQL query commands depending on the corresponding parameter or conditions, which has remarkably increased the difficulty for developers to link a query to the corresponding application code. Through the utilization of the accessed table analysis combined with the static code and string analysis, we are able to assist developers in locating the application code that is associated with a problematic database activity.

Exposing database code history simplifies tracking database code modifications and enables tracing performance issues back to database code changes. By leveraging static code and string analysis, the database code and application code can be effectively linked to each other while requiring less expertise and effort from developers for performance analysis.

5.4 Implemented Analysis

In this section, we present the detailed implementation of our adapted performance analytic techniques in the context of the target large-scale database-centric system (i.e., *EA*). In total, our implemented performance analysis consists of three main parts: 1) performance anomaly analysis within one release, 2) performance regression analysis between two releases, and 3) root cause analysis. Part 1) and Part 2) of the analysis are supported by our solution to challenge C1 described in the previous section, while the analysis in Part 3) is enabled by our solution to challenge C2.

5.4.1 Performance anomaly analysis within one release (PAA)

In the first type of implemented analysis, our objective is to automatically identify the existence of performance issues that arise during a particular release. We identify the problematic application activity (i.e., web request) and database activity (i.e., query or stored procedure) through outlier analysis (PAA-1), and for the problematic database stored procedures, we also locate the corresponding performance bottleneck through proportion analysis (PAA-2).

PAA-1: Performance outlier analysis. Performance outliers refer to the performance data that deviates significantly from the expected or normal behavior of the system such as an extremely slow response of a SQL query or web request. It is a common yet important performance issue in large-scale database-centric systems dealing with massive amounts of data. To identify performance outliers, we first parse the recorded database and application runtime activity logs to extract the execution duration of each database activity (e.g., SQL query or stored procedures) and each application activity (e.g., web request) within the current version of the system. Then, we utilize two statistical techniques that have been commonly utilized in prior work [192, 33, 218], namely Z-sore and

inter-quartile range (IQR), to effectively detect performance outliers in *EA*. In particular, the Z-score measures how far an observation deviates from the mean in the unit of standard deviation. If the Z-score of an observation is larger than three, it is an indicator that this observation is quite different from other ones and can be classified as an outlier [33]. By contrast, IQR is a measure of the dispersion of data, calculated by the difference between the third quartile and the first quartile. Typically, observations that fall below the value of the first quartile minus 1.5 times the IQR or exceed the value of the third quartile plus 1.5 times the IQR are identified as performance outliers. In the event of any performance outliers detected by both techniques, we leverage ElastAlert to provide a timely alert to the developers with emails. When an outlier is only detected by one technique, we do not produce an alert to avoid false alarms. Furthermore, we store and visualize the extracted execution duration and analysis results in our ELK stack, enabling developers to utilize our analysis in a more transparent and intuitive manner. Without this analysis, developers will not know in certain cases, a database query can take significantly longer than usual.

PAA-2: Performance proportion analysis. Database activities typically do not occur in isolation, but rather in the form of a sequence, for example, in the execution of database stored procedures, there may be a series of queries that are executed in a specific order. Identifying performance bottlenecks in the sequence of these activities is critical to help developers pinpoint the real cause of the identified performance issues. To this end, after we have identified the problematic database stored procedures from PAA-1, we further analyze the recorded database execution logs and recover the activities sequence by linking the related log lines into sequences. For instance, we group the log lines that have the same session ID and user ID as the problematic stored procedures, and these log lines can be used to represent the activity sequence of execution of the stored procedure. After that, we calculate the proportion of execution time of each individual activity (i.e., a query) compared to the performance of the entire sequence of activities (i.e., the stored procedure). Our intuition is that if a database stored procedures have performance issues, the activity that takes up the majority portion (e.g., more than two-thirds) of the execution time of that entire activity sequence is more likely to be the real cause of the problem. Without this analysis, developers would not identify the time-costly database query. We further prioritize the potentially problematic activities based on their contribution to overall performance issues. This allows developers to allocate their time and effort more effectively when conducting further investigations.

5.4.2 Performance regression analysis between two releases (PRA)

Different from the first type of analysis, in this subsection, we employ two complementary approaches, including statistical process control chart-based analysis and machine learning model-based analysis to point out any performance regressions introduced in the development of a new release compared to the previous release.

PRA-1: Statistical process control-based performance regression analysis. In an effort to identify the problematic system activities that may cause the performance regression, we leverage a statistical process control technique called control charts (similar to prior research [175, 176]) to analyze the execution time of every type of system activity (including application activities and database activities) across the old and the new releases of the system. In particular, we extract the execution time for each system activity from the database and web application execution logs collected during the execution of both the old and new releases. The extracted performance from the old release serves as a baseline and is utilized to create control limits, including the center line (CL), upper control limit (UCL), and lower control limit (LCL). The LCL, CL, and UCL are commonly defined as the 10^{th}, 50^{th}, and 90^{th} percentiles of the old release performance, respectively [175]. We create a separate control chart for each type of activity and once the control limits are established, we then compute the percentage of performance extracted from the new release that falls outside these limits, which is known as the violation ratio. This metric represents the likelihood of performance regression for a particular system activity type in the new release. If the violation ratio of an activity type surpasses a threshold set by experts (e.g., 20%) and its average execution time in the new version is higher than in the old version, then that system activity type is identified as a performance regression. Before our adaption, our analysis did not know the performance regressions occurring in the database activity.

PRA-2: Machine learning model-based performance regression analysis. To alleviate the impact of continuously varying workloads in the production environment, in this analysis, we rely on a machine learning model-based approach from prior work [148] to identify problematic system activities. In particular, when the system has a new release, we construct two random forest models to capture the relationship between the performance of the system and the runtime activities (i.e., system workload) by using the data from the old and new releases, respectively. We then apply these models to the new release data to evaluate their respective modeling errors. Afterward, we compare these two models by evaluating the deviation of their modeling errors through statistical analysis (i.e., performing statistical tests and measuring effect sizes). The idea behind this is that if a system runtime activity contributes to the deviation of the performance models built on two dif-

ferent software releases, then the system activity has a high chance of being the cause of performance regressions between the two releases. Finally, we utilize a linear regression model [92] to explain the relationship between the system activities and the deviation in modeling errors. If a system activity contributes significantly (with a p-value smaller than 0.05) to the deviation, then it is considered problematic. Prior to our adaption, our analysis was unaware of performance regressions related to the problematic database activity.

5.4.3 Root cause analysis (RCA)

Once performance issues have been identified through the above-mentioned analysis, we intend to locate their root causes from two perspectives, including problematic application activity and problematic database activity.

RCA-1: Root cause identification from problematic application activity. To locate the root causes (i.e., source code or code changes) of the identified problematic application activity (i.e., web request), we first search through the entire application source code by keywords, function names, or URLs to find out one or more methods associated with the target application activity. Then, we utilize the static code analysis framework (i.e., .NET Compiler Platform SDK, also known as Roslyn) to parse the source code and construct a call graph for the method(s) associated with the web request. Moreover, for the methods in the call graph that invoke database stored procedures, we further extract the name of the stored procedures being executed. To achieve this, we traverse all the invoked stored procedures using static analysis and save their names in a list. Afterward, we enhance the previously constructed call graph by annotating the extracted stored procedure names onto the methods that invoke them. The source code, including both application code and database code (i.e., stored procedures), in these call graphs is identified as the potential root cause of the problematic application activity. In particular, when performance issues are detected between two releases of the system, we conduct a more in-depth investigation by analyzing the system version history. Specifically, we look for any changes between the two releases made to the application or database code that may have affected the methods or stored procedures presented in the enhanced call graph. We then consider these code changes, along with additional information such as code churn and committer data, as potential causes of the performance issues. Finally, we provide this information to the developers responsible for the affected code for further investigation and resolution.

RCA-2: Root cause identification from problematic database activity. The identified problematic database activities can be generally divided into two categories:

problematic queries and problematic stored procedures. In cases where problematic stored procedures are identified from our performance anomaly analysis (PAA) or performance regression analysis (PRA), we can directly examine the corresponding database code and its version history, and consider this code or any associated code changes as the root cause of the issue (i.e., similar to RCA-1). However, for problematic queries, pinpointing the corresponding root cause becomes more challenging since these queries are often dynamically constructed through a series of string operations in the application code such as concatenation and substitution. Due to the complexity of the system, the same source code may generate remarkably different queries based on the different parameters or code paths. From a straightforward example shown in Listing 5.1, this application code constructs queries dynamically based on the user input, and the query may vary for different inputs.

```
public static string GenerateQuery(string userInput)
{
    string sqlQuery = "SELECT * FROM Products";
    if (!string.IsNullOrEmpty(userInput))
    {
        sqlQuery += " WHERE ProductName LIKE '%" + userInput + "%'";
    }
    return sqlQuery;
}
```

Listing 5.1: Exmaple of dynamically constructing a SQL query

To tackle this challenge, we first perform a coarse-grained matching between the identified problematic database activities and the application code, based on the accessed database table. Firstly, we parse the application code to identify all the methods that involve database operations. Then, for each of these methods, we extract the database tables accessed by it (e.g., using regular expressions to extract the "FROM" clause). We also extract the accessed database tables of the SQL query under investigation. By matching these results, we can generate a list of methods in the application code that are potentially responsible for generating the target SQL query.

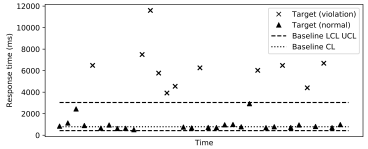
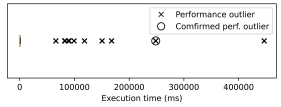
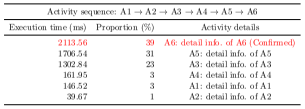
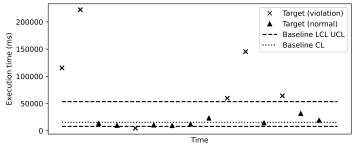
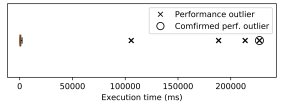
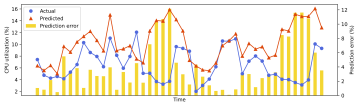
Once we have narrowed down the scope, we conduct a more fine-grained static code and string analysis (similar to prior research [166]) to further locate the root causes of the problematic database activities. In particular, for each of the candidate methods in the application code, we extract the abstract syntax tree (AST) from the corresponding source code and leverage the visitor pattern provided by the .NET compiler platform SDK (Roslyn) to identify the locations where SQL queries are sent to the database (e.g., *statement.executeQuery(query)*). Then we take each of these locations as the starting point

and extract all the potential SQL string values passed to it. To achieve this, we extract the control flow that leads to the starting point with the help of Roslyn. Afterward, we trace back all the related expressions of the SQL query following the control flow previously extracted and try to determine the values of the query expression by resolving each variable involved in it. However, sometimes constructing a SQL query may involve variables passed as parameters, for example, "select" + *column* + "from table1", where *column* is passed to the method as a parameter. In order to determine the different possible values of the required parameters, we recursively resolve the query expression by following the call graph of the application in a backward direction. At the end of our analysis, a set of different possible SQL string values that can be produced at a specific location in the source code is generated. After that, we compare these SQL strings to the one that was identified as problematic. We leverage the Gestalt pattern matching algorithm [182] to calculate the similarities between the two SQL query strings and provide the ones with relatively high similarity (e.g., the top 5) as well as the corresponding locations (e.g., the names of the application methods) that could potentially generate these queries to developers for further investigation. In particular, if performance issues are detected between two releases of the system, we automatically scan the system version history to identify any changes made to these candidate methods and provide them to developers (i.e., similar to RCA-1).

5.5 Detected Real-World Cases

Our solutions and analysis have been adopted and used by the industrial collaborator to ensure the performance of *EA* on a daily basis. Since the performance analysis is based on the data collected in the production environment, the workload for *EA* is not predetermined (cf. Section 5.2). Table 5.1 summarizes a set of real-world issues that were detected by applying our adapted analysis in the target industrial setting. In particular, we have detected and confirmed a total of six cases, including five performance issues and one performance improvement that occurred either within one release or across two releases, and we also identified the corresponding root causes. In the following, we present more details about each detected case.

Table 5.1: The summary of real-world cases detected in the target industrial setting (i.e., *EA*)

ID	Type	Scope	Involved analysis	Description	Visualization of detection results
DI-1	Issue	BTR	PRA-1, RCA-1	This is a performance regression caused by newly added complex SQL queries in database stored procedures.	
DI-2	Issue	WOR	PAA-1, RCA-2	This is a performance anomaly due to an inefficient SQL query for a new feature.	
DI-3	Issue	WOR	PAA-1, PAA-2, RCA-2	This is a performance anomaly caused by a problematic SQL stored procedure related to business logic.	 Activity sequence: A1 → A2 → A3 → A4 → A5 → A6 Execution time (ms) Proportion (%) Activity details 2113.56 39 A6: detail info. of A6 (Confirmed) 1706.54 31 A5: detail info. of A5 1302.84 23 A3: detail info. of A3 161.95 3 A4: detail info. of A4 146.52 3 A1: detail info. of A1 39.67 1 A2: detail info. of A2
DI-4	Issue	BTR	PRA-1, PRA-2, RCA-1	This is a performance regression caused by an inefficient SQL stored procedure when handling a large amount of data.	
DI-5	Issue	BTR	PRA-2, PAA-1, RCA-1	This is a performance regression caused by an inefficient SQL query generated by the application code.	
DI-6	Improve-ment	BTR	PRA-2, RCA-1	This is a performance improvement due to the optimization of a SQL query in the new version.	

Note: In the "Scope" column, "WOR" stands for within one release and "BTR" stands for between two releases.

5.5.1 Detected Issue 1 (DI-1)

The first detected case (i.e., DI-1) is a performance regression between two consecutive releases of *EA* and it is caused by newly added complex SQL queries in the database stored procedures. In particular, the issue was detected by the control charts that track the response times of each web request type and examine the violation ratio of the same type of web request for a new release (i.e., PRA-1). From the results, we observed that there is one type of web request in the new release showing a significantly slower response time than that in the previous release with a violation ratio of 31.43% (the violation threshold is set at 20% by senior developers). The constructed control chart for the target web request is also shown in Table 5.1. Prior to exposing the source code from the database (as discussed in Section 5.3.2), we find that there seem to be no application code changes that are related to the performance regressions. After successfully addressing the challenge and performing the root cause analysis for problematic application activity (i.e., RCA-1), we managed to identify two suspicious updates of database stored procedures in the new release and then confirmed them with the developers from our collaborator. These updates introduced multiple complex SQL queries to support new requirements, resulting in the target web request that frequently invokes these stored procedures taking longer to respond to clients.

5.5.2 Detected Issue 2 (DI-2)

DI-2 is an identified performance anomaly due to an inefficient SQL query for a new feature. This issue was missed by our prior pair-wise analysis (cf. Section 5.2) and it caused a direct impact on end users. After investigation, we found that this issue is caused by a new feature (i.e., a new type of web request) released in the current version, however, our prior performance analytics focused only on the performance regressions of the same system feature during consecutive versions, which has limited support for this case. By utilizing our adapted analysis on the recorded database execution logs (i.e., PAA-1), we have successfully identified performance outliers caused by an inefficient SQL query. The detection results are also shown in Table 5.1. In particular, our root cause analysis (i.e., RCA-2) identified a specific section of the application source code that generates a complex query. Within this query, the subqueries implicitly create multiple temporary tables in the database. However, it was observed that these tables lack indexes, resulting in poor performance for both the query and the new web request that triggers this query. This adverse performance impact becomes more prominent when users interact with larger-than-usual amounts of data. After discussing the analysis results with developers, they confirmed

the identified root cause and implemented optimizations to address the inefficiency. After applying the fix, we observed a significant improvement in performance, with the execution time of the SQL query being approximately 10 times faster.

5.5.3 Detected Issue 3 (DI-3)

DI-3 is a performance anomaly caused by a problematic stored procedure related to business logic. Similar to the previous performance issue (i.e., DI-2), this case went undetected in the prior analysis, and the clients reported experiencing a noticeable slowdown when performing a specific operation. Our performance outlier analysis (i.e., PAA-1) has successfully identified that the issue stemmed from a problematic stored procedure associated with business logic. However, due to confidentiality concerns, we are unable to provide further details regarding its specifics. Our further performance proportion analysis (i.e., PAA-2) extracted the sequence of database queries related to the target stored procedure from execution logs and from the results (also shown in Table 5.1), we identified a particular query that consumed a significant portion of the overall processing time, approximately 39%. After presenting the corresponding root cause through our root cause analysis (i.e., RCA-2) to developers, they delightedly accepted our findings and promptly addressed the issue in the subsequent release.

5.5.4 Detected Issue 4 (DI-4)

DI-4 is a performance regression caused by an inefficient stored procedure when handling a larger amount of data than before. Our machine learning model-based performance regression analysis (i.e., PRA-2) has located a problematic web request. With the support of our solution of exposing the source code from the database, we successfully located the corresponding root cause through our root cause analysis (i.e., RCA-1). However, we also noticed that the target web request potentially calls a wide range of stored procedures (>10), making the manual inspection of these results time-consuming and requiring significant expertise. By combing the results from our statistical process control-based analysis (i.e., PRA-1) on the recorded database execution logs, we identified a stored procedure responsible for the performance regression. As shown in Table 5.1, the stored procedure in the new release shows a remarkably longer execution time than that in the previous release with a violation ratio as high as around 40%. Furthermore, we also found that this issue remained unnoticed in previous releases primarily due to the fact that clients are currently handling a larger volume of data than before.

5.5.5 Detected Issue 5 (DI-5)

DI-5 is a performance regression between two consecutive releases caused by a newly added inefficient SQL query in the application code. Our machine learning model-based performance regression analysis (i.e., PRA-2) has detected a problematic application activity, but from the results of the root cause analysis (i.e., RCA-1), we found that the identified problematic application activity seems to be responsible for a lot of work and is associated with many changed source code files (around 30 files), making it difficult to pinpoint the real root cause. By combining these findings with the results from our performance outlier analysis (i.e., PAA-1), we confirmed that one of the changed source code files added an inefficient SQL query that executes for a couple of minutes (the results are shown in Table 5.1). We also provided the corresponding application code changes related to this query to developers for further optimization.

5.5.6 Detected Improvement 6 (DI-6)

The last detected case (i.e., DI-6) is a performance improvement due to the optimization of a SQL query in the new version. Our machine learning model-based performance regression analysis (i.e., PRA-2) successfully detected this case. The results shown in Table 5.1 reveal that the new version is more likely to perform better (i.e., consumes less CPU) than the old version when handling similar workloads since the system performance predicted by the model constructed from the old version (i.e., the red line) is often higher than the actual performance (i.e., the blue line). The results of our root cause analysis (i.e., RCA-1) indicate that this performance improvement was accomplished by several updates to an existing complex SQL query. After discussing the results with senior developers, we confirmed that the developers who made these updates opted to use a subquery with the EXISTS function in replace of the DISTINCT function in an SQL query to retrieve data from multiple tables. Such an update helps to avoid the additional performance overhead of scanning the entire table and sorting the retrieved rows before suppressing the duplicates. Furthermore, the updates also enable the query execution to stop early when specific conditions are met, thereby further improving performance, particularly when processing extensive amounts of user data. This detected case also indicates that our solution can derive good practices from detected performance improvement cases. We can utilize these cases as examples for developers to help them learn good coding practices and avoid suboptimal practices in future development tasks.

5.6 Threats to Validity

This section discusses potential threats to the construct, external, and internal validity of our study.

5.6.1 Construct validity.

In our study, we opt to leverage a built-in performance profiling tool in Windows (i.e., TypePerf) to monitor the system-level runtime performance and a community-maintained SQL Server stored procedure (i.e., sp_whoisactive) to capture the query-level database runtime behavior. Therefore, the quality of the recorded runtime performance metrics and behaviors may be a threat to the construct validity of our study. In addition, for the performance outlier analysis (i.e., PAA-1), we do not distinguish each type of database or application activity. Future work may separate each activity to examine the anomalies within each type of activity.

5.6.2 External validity.

Our work is conducted on a large-scale enterprise system from our industrial collaborator, which is developed in the .NET framework with the Microsoft SQL server. Although our subject is a mature database-centric system with many years of development and maintenance history, which has a certain representativeness, it may still limit the generalization of our study results and findings to other systems.

5.6.3 Internal validity.

Our solution depends on multiple statistical analysis techniques to detect performance anomalies or regressions within or across system releases. However, our approach may not perform well on small systems with an insufficient amount of data due to the nature of the statistical analysis. Besides, statistical analysis techniques often rely on the threshold to determine the statistical significance. However, in practice, determining the significance usually depends on the subjective judgment of the performance analysts.

5.7 Conclusion

In this chapter, we report our experience in adapting performance analytic techniques to effectively assist developers in detecting performance issues and identifying the corresponding root causes for a large-scale database-centric system from our industrial collaborator. We share our challenges and solutions for exposing the runtime measurement of the database for performance analytics and exposing the source code from the database for performance root-cause analysis. We also share the implemented database-aware performance analysis, including detecting performance anomalies within one release, identifying performance regressions between two releases, and root cause analysis, as well as the discussions of the real-world performance issues detected by our analysis. We anticipate that our engineering solution and documented experience can provide helpful insights into ensuring and improving the performance of large-scale database-centric systems for both research and practice.

Part II

Early Detection of Performance Regressions during the Development Phase

Chapter 6

Early Detection of Performance Regressions by Bridging Local Performance Data and Architectural Models

During software development, developers often make numerous modifications to the software to address existing issues or implement new features. However, certain changes may inadvertently have a detrimental impact on the overall system performance. To ensure that the performance of new software releases does not degrade (i.e., absence of performance regressions), existing practices rely on system-level performance testing, such as load testing, or component-level performance testing, such as microbenchmarking, to detect performance regressions. However, performance testing for the entire system is often expensive and time-consuming, posing challenges to adapting to the rapid release cycles common in modern DevOps practices. In addition, system-level performance testing cannot be conducted until the system is fully built and deployed. On the other hand, component-level testing focuses on isolated components, neglecting overall system performance and the impact of system workloads.

In this chapter, we propose a novel approach to early detection of performance regressions by bridging the local performance data generated by component-level testing and the system-level architectural models. Our approach uses local performance data to identify deviations at the component level, and then propagate these deviations to the architectural model. We then use the architectural model to predict regressions in the performance of

the overall system. In an evaluation of our approach on two representative open-source benchmark systems, we show that it can effectively detect end-to-end system performance regressions from local performance deviations with different intensities and under various system workloads. More importantly, our approach can detect regressions as early as in the development phase, in contrast to existing approaches that require the system to be fully built and deployed. Our approach is lightweight and can complement traditional system performance testing when testing resources are scarce.

6.1 Introduction

Performance is a critical aspect of the quality of service (QoS) of software systems. It is important to ensure that software consistently delivers optimal performance after each new release (i.e., absence of performance regressions) [152, 223, 121, 49, 202]. However, throughout the development cycle, developers may implement various modifications to the software to implement new features or address existing issues, some of which could potentially impact the performance of the system adversely [246, 60, 68]. Such performance regressions can result in higher resource consumption (e.g., excessive memory or CPU usage), increased response time, or even field failures, thereby causing significant financial and reputation losses [227, 23]. For instance, according to a recent report [23], even a mere two-second difference in the website response time can drastically decrease user satisfaction, causing the bounce rate to surge from 9% to 38%. Ultimately, this could result in a remarkable loss of market share and revenue. Therefore, it is crucial to detect and resolve any performance regressions before deploying the system.

In practice, traditional system-level performance testing is widely adopted to prevent potential performance regressions from sneaking into production [93, 175, 157]. Existing practices involve running field-like workloads for an extended period of time (from hours to days) to exercise both the old and the new versions of the system in an in-house performance testing environment. During testing, a large number of performance metrics and execution logs are generated, and performance analysts collect and analyze this information from both old and new versions to determine the existence of performance regressions [93]. However, performance testing can be expensive, especially for large-scale systems, requiring significant resources and time to set up the environment and execute the performance tests [59]. Furthermore, such a process is often conducted late in the software development and release cycle, i.e., after the system is built or even integrated, making it challenging and laborious to diagnose and address the performance regressions at such a late stage.

To tackle this challenge, there has been a growing interest in leveraging component-

level (e.g., function or class) performance information to detect performance regressions in software systems. These approaches [58, 115, 84, 130, 183, 140, 119] propose to leverage unit tests (e.g., JUnit tests) or microbenchmarks (e.g., Java Microbenchmark Harness) to collect the corresponding performance metrics, such as execution time or throughput. Through conducting statistical analysis on the performance metrics collected from the test runs before and after code changes, developers can identify any remarkable performance improvement or regressions in individual components of the system. Such approaches are relatively lightweight to execute and avoid the expensive resources and time required for running traditional performance tests on the entire system. However, due to the unique nature of component-level tests, they typically do not explain the system-level performance well [59]. In particular, the performance changes of components may be propagated to the system very differently. For instance, a function executed once may have a negligible performance impact whereas a function executed in a long loop may have a much more significant impact. In addition, it is also challenging to consider realistic workloads in the component-level testing.

In this chapter, we propose a novel approach to early detection of performance regressions by bridging the local performance data generated by component-level testing and the architectural models of the system. After developers make code changes during development, we first collect local performance information by running component-level tests and then identify performance deviations at the component level. Afterward, we propagate these component-level performance deviations to the architectural model. Finally, we leverage the architectural model to predict performance regressions of the entire system. To evaluate the effectiveness of our approach, we conduct experiments on two representative open-source benchmark systems, i.e., TeaStore [220] and TrainTicket [247]. In the experimental results, we find that by bridging local performance data and architectural models, our approach can effectively detect end-to-end system performance regressions with different intensities of local performance deviations and can maintain its effectiveness when the system experiences various workloads. The experimental results also demonstrate that our approach can effectively and efficiently assist developers in identifying and addressing performance regressions as early as the software development phase and can be adapted into the rapid software development and release practices (e.g., DevOps) to complement traditional system performance testing in resource-constrained scenarios.

This work makes the following main contributions:

- We develop a novel approach to the early detection of performance regressions by bridging the local performance testing data and the system architectural models.
- Our approach is lightweight and can be integrated into fast-paced software develop-

ment and release practices (e.g., DevOps) to complement traditional system performance testing in situations where testing resources are scarce.

The remainder of the chapter is organized as follows. Section 6.2 discusses the background of our work. Section 6.3 outlines our approach to detecting system performance regressions. Section 6.4 introduces the evaluation setup. Section 6.5 presents the evaluation design, data analysis, and results. The threats to the validity of our work are discussed in Section 6.6. Finally, Section 6.7 concludes our work.

6.2 Background

In this section, we introduce the background of our study, including software performance testing and model-based performance analysis.

6.2.1 Software performance testing

Software performance testing evaluates a software system's performance under various conditions to ensure the system meets the specified performance requirements and operates efficiently. Software performance testing can be conducted at both the system and component levels.

At the system level, performance testing evaluates the overall performance of the entire software system under a workload similar to real-world scenarios. It simulates user interactions and workload scenarios to assess performance metrics such as response times, throughput, and resource utilization of the system. Typically, there are four phases in system performance testing: 1) defining a system workload (e.g., Apache JMeter HTTP(S) test script [9]), 2) preparing an in-house testing environment, 3) executing the performance tests, and 4) analyzing the testing results. System performance testing has demonstrated its effectiveness in assisting developers in determining compliance with performance goals, identifying system bottlenecks, and detecting performance regressions [93]. However, since it involves the entire software system, and modern software systems are often large in scale and highly complex, the testing process can be extremely time- and resource-consuming.

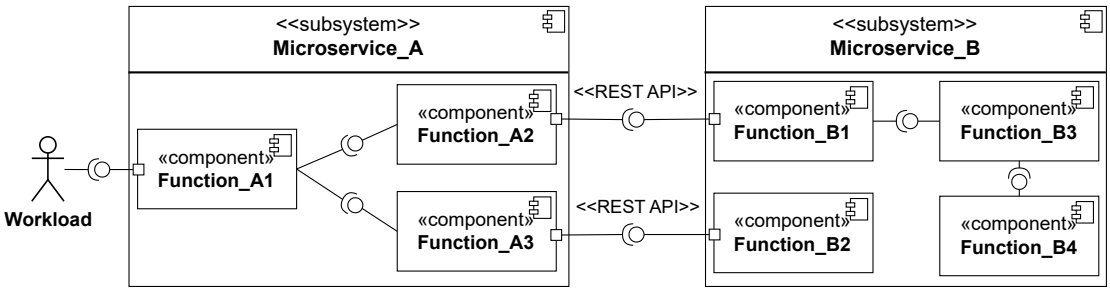
On the other hand, at the component level, performance testing focuses on evaluating the performance of individual software components or modules in isolation. These components can be different functions or classes that can be independently tested. Component-level performance testing involves developers writing tests (microbenchmarks) targeting

specific components of concern with performance unit testing frameworks, such as JMH, JunitPerf, and ContiPerf, or directly using JUnit to test the performance combined with functional tests [75, 141]. This type of performance testing is lightweight to execute and effective to uncover performance issues at the component level and ensure components meet performance requirements individually [84]. However, it often does not explain the system-level performance well [59], since performance changes of a component may be propagated to the system very differently (e.g., the function executed once versus executed within a long loop). Furthermore, it is also challenging to consider realistic workloads in the component-level testing.

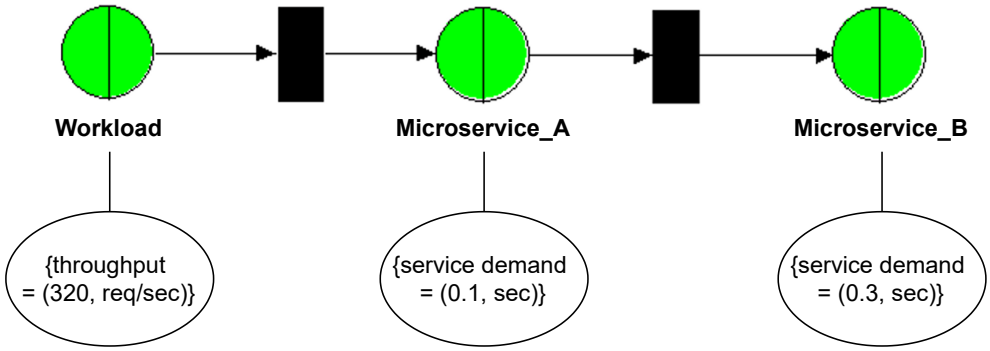
6.2.2 Model-based performance analysis

Model-based performance analysis is a methodology that utilizes analytical models to predict and assess the performance of software systems [40]. Model-based performance analysis involves constructing abstract models of the system, describing its components, interactions, and resource utilization, and then employing mathematical and computational techniques for analysis and simulation. Models can take various forms, such as queueing network (QN) [131, 127, 228, 187], layered queueing network (LQN) [91, 199, 214], or queueing Petri nets (QPN) [38, 43, 42, 44], to capture different aspects and characteristics of the system. Figure 6.1 shows an illustrative example of a software system with two subsystems and its corresponding architectural model. Model-based performance analysis provides an efficient and effective means for developers and system designers to better understand and optimize the performance characteristics of software systems. Through modeling, developers can identify potential performance bottlenecks, predict system performance under various workload conditions, and make design optimizations to enhance system performance [135]. However, when dealing with modern systems that are typically large-scale and complex, accurately modeling the performance of the entire system is challenging, leading to the adoption of high-level modeling representations, such as modeling at the subsystem or service levels [136] as illustrated in Figure 6.1. While these high-level models offer a macroscopic understanding of the overall system performance, they also constrain the model's ability to capture nuanced performance changes at the system's finer levels (e.g., functions).

In this chapter, we propose a novel cost-effective approach to detecting system performance regressions during the development phase by bridging local performance data generated from component-level performance testing and architectural models. Our approach allows developers to understand earlier that there is a performance regression, instead of



(a) An example software system with two subsystems



(b) The corresponding architectural (QPN) model (the green dots indicate service queueing places, the underneath circles indicate corresponding performance parameters, and the black boxes indicate transitions between queueing places)

Figure 6.1: An example software system with two subsystems and its corresponding architectural model

running expensive system performance testing after the system is built or even integrated. We present our approach in detail in the next section.

6.3 Approach

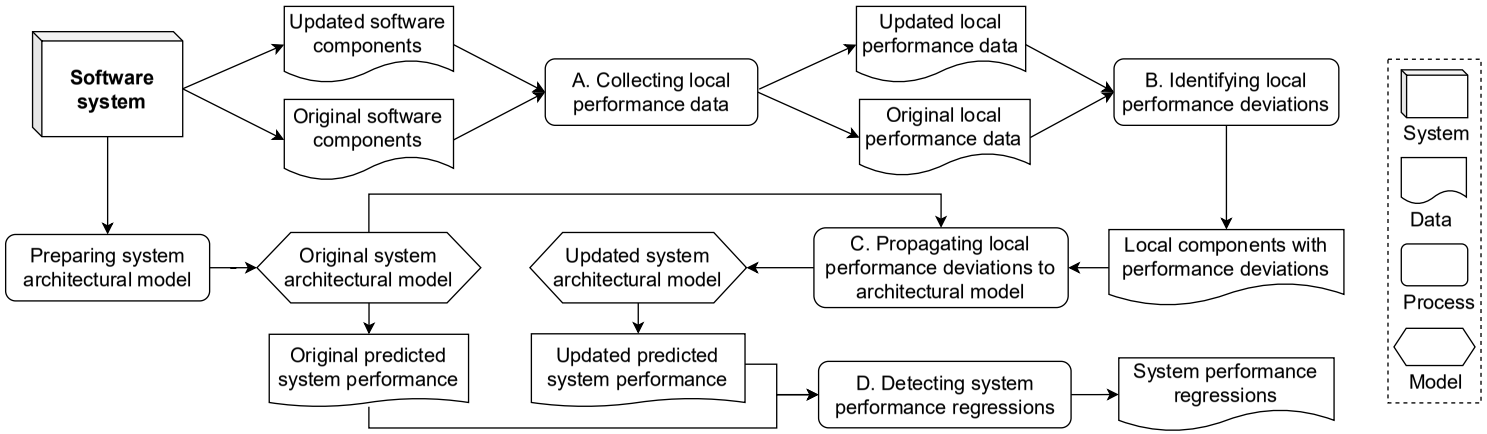


Figure 6.2: An overview of our approach for system performance regression detection by bridging local performance data and architectural models

In this section, we present our approach for detecting system performance regressions by bridging local performance data and architectural models. Figure 6.2 gives an overview of our approach, and each subsection corresponds to a step in the figure. To ease the illustration of our approach, we show a running example of a software system with two subsystems, each with three and four functions, as shown in Figure 6.1a. However, the number of subsystems and functions is much larger in our evaluation and real-life scenarios. The goal of our approach is to detect whether there are system performance regressions after developers make changes to the software (e.g., after a commit) without running expensive system-level performance testing.

Preparing system architectural model. Our approach relies on the architectural model to detect performance regressions. The model is typically available as part of the software artifact [113] and can be directly utilized for our approach. As an alternative when there is no available model, we can also attain it through relatively low-frequency system-level performance testing [199, 135]. In particular, compared to component-level performance testing that typically occurs after each commit (alongside unit tests in the CI pipeline), system-level performance testing is often conducted only once at the beginning of a major release [93]. It is also worth noting that our approach does not necessarily depend on system performance testing and specific types of architectural models. Instead, our approach primarily focuses on bridging the local performance data and architectural models to detect performance regressions, assuming the availability of the system architectural model.

6.3.1 Collecting local performance data

In the first step, we collect the performance data of local components (i.e., functions) before and after developers make changes to the software during development. In particular, we run the component-level tests of the software, e.g., unit tests or JMH microbenchmarks, for the performance testing of the local components. We execute these test cases on both the original and the updated version of the code. Each test is executed with a total of 30 iterations to minimize environmental noise and system randomness, similar to what was done in prior research [81]. Furthermore, during testing, we actively utilize application performance monitoring tools to monitor and collect the software system's runtime behavior (i.e., the interactions of local components – functions), and the corresponding performance data (i.e., the execution time of each component).

6.3.2 Identifying local performance deviations

We analyze the local performance data collected in the previous step to estimate the performance deviation (in execution time) of local software components. We first conduct Wilcoxon rank-sum statistical tests [224] to compare the performance data distributions before and after the update. We run the statistical test at the 5% level of significance, and the null hypothesis is that there exists no statistically significant difference between the performance of the original and updated versions of the component. If the p-value of the test is not greater than 0.05, we would reject the null hypothesis and favor the alternative hypothesis, i.e., there is a statistically significant difference. However, recognizing that statistical tests solely reveal the existence of differences without quantifying their magnitude, we complement this statistical analysis by calculating Cliff's Delta effect size [69] to determine the magnitude of the observed differences. We employ the widely adopted thresholds for Cliff's Delta effect size [189], as shown in the following:

$$\text{Effect size} = \begin{cases} negligible, & \text{if } |\delta| \leq 0.147 \\ small, & \text{if } 0.147 < |\delta| \leq 0.33 \\ medium, & \text{if } 0.33 < |\delta| \leq 0.474 \\ large, & \text{if } 0.474 < |\delta| \leq 1 \end{cases}$$

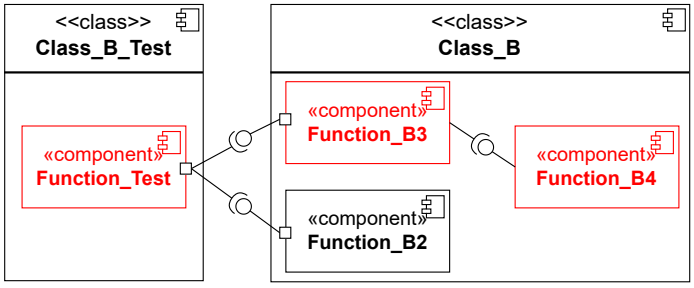
It is worth noting that the choices of the Wilcoxon rank-sum test and Cliff's Delta effect size are deliberate since they do not assume a specific distribution for the performance data.

In the cases where statistical analysis indicates significant differences with an effect size larger than negligible, we further propagate the local performance deviation to the system architectural model. To do so, we calculate the mean difference (MD) of local performance before and after updates. A positive value indicates decreased performance (i.e., longer execution time) in the updated version, while a negative value suggests performance improvement (i.e., shorter execution time).

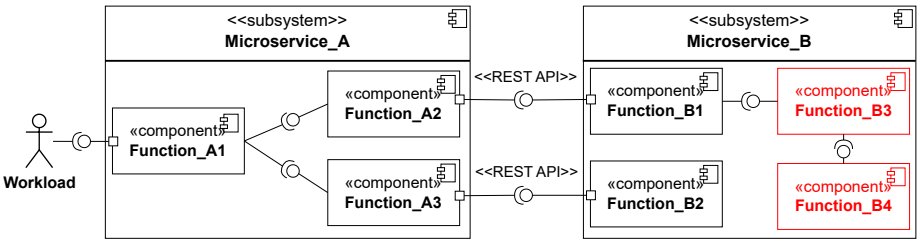
6.3.3 Propagating local performance deviations to architectural model

With the identified local software components that suffer from deviated performance, intuitively, one may utilize the local performance deviations and the number of times the components are called as a proxy to determine system performance regressions. However, such a naïve approach may overlook the crucial impact of the system's complexity and

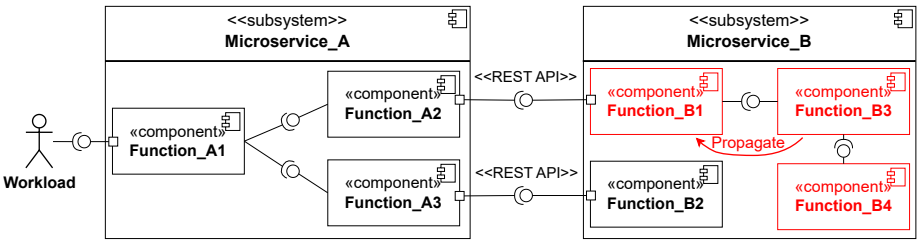
dynamics, such as resource contention, queue wait times, and various workloads. Therefore, the architectural model is used to detect performance regressions, especially when the system is under various workloads. In this step, we first analyze the local performance data collected from the prior step (cf. Section 6.3.1) to establish a dependency graph to represent the structure and performance of the local components. Such a graph is a directed acyclic graph, with nodes representing the local components, and edges representing their interaction dependencies (e.g., function calls). Performance information is represented as node attributes. We then extract a subgraph with local components that have deviated performance (identified from the previous step in Section 6.3.2) within the local dependency graph and mark them accordingly, as depicted in Figure 6.3a.



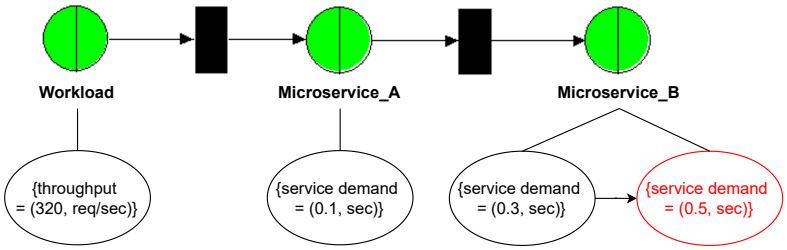
(a) Extracting local performance deviations



(b) Mapping local performance deviations to the system level via maximum common subgraph analysis



(c) Propagating performance deviations to the top level of the subsystem



(d) Updating architectural model

Figure 6.3: An illustrative example of propagating local performance deviations to architectural model (for subfigures (a), (b), and (c), the red boxes indicate components with performance deviations, while for subfigure (d), the red box indicates updated performance parameter in the architectural model)

Afterward, we map the local performance deviations to the corresponding system level through graph mapping. In particular, we first identify common substructures shared between the extracted local graph and the existing system graph by leveraging the ISMAGS algorithm [116], which finds the maximum common subgraph of two graphs. We then align the performance deviation (i.e., MD) from the local level to the corresponding system level. Figure 6.3b shows an example of mapping the local performance deviation marked in Figure 6.3a to the corresponding system level.

Since system architectural models typically operate at a relatively higher level, such as the subsystem level, merely understanding the performance deviations at the lower (component) level is insufficient. Therefore, we further employ a bottom-up strategy to propagate performance deviations upwards through the hierarchical dependency layers of the system. We begin this process from the identified components of concern and extend it until it reaches the top-level components of that subsystem, since these components encompass the performance of their invoked components and constitute the performance of the subsystem. As illustrated in the example in Figure 6.3c, after we successfully map the local performance deviation to the system level for Function_B3 and Function_B4, we further propagate to the top-level component Function_B1.

Finally, we calculate the overall performance of the subsystem by summing up the top-level component performance (e.g., Function_B1 and Function_B2 in Figure 6.3c) and calculate the deviation (i.e., relative difference) between the overall performance of the subsystem before and after the changes. With this deviation in the overall performance of the subsystem, we then update the corresponding performance parameters within the architectural model. For example, in Figure 6.3d, we propagate the local performance deviation to the architectural model by updating the service demand parameter for Microservice_B from 0.3 sec to 0.5 sec.

6.3.4 Detecting system performance regressions

The last step of our approach is to identify system performance regressions by comparing the original architectural model and the model updated in the last step. We first use both models to get the original and updated predicted performance, including the resource utilization and response time. We then utilize a statistical analysis approach similar to the one used for identifying local performance deviation in Section 6.3.2. In particular, we utilize the Wilcoxon rank-sum test and Cliff's Delta effect size, to determine whether there is a significant difference between the two performance distributions and the magnitude of the difference. If the p-value is smaller than 0.05 and the effect size is larger than negligi-

ble, we would consider that the changes to the software may lead to system performance regressions.

6.4 Evaluation Setup

To evaluate the effectiveness of our approach, we conduct case studies on two prevalent open-source benchmark systems (i.e., TeaStore and TrainTicket) with injected performance regressions. Our selected subject systems have been widely used in prior work [73, 148, 205]. Table 6.1 gives an overview of our subject systems. In this section, we present the details about the subject systems, evaluation environment, performance regressions, component-level and system-level performance test design, and architectural models.

6.4.1 Subject systems

TeaStore [220] is a microservice reference and benchmark application that serves as a simulated web store. As a distributed microservice application, TeaStore comprises five distinct services plus a separate service registry. We choose TeaStore as a subject since it is positioned as a benchmark application and widely utilized for evaluating various performance aspects [220], including performance modeling, auto-scaling, and energy-efficiency analysis.

TrainTicket [247] is a web train ticket booking system that provides various typical train ticket booking functionalities, such as ticket inquiry, reservation, payment, and rebooking. This system is a benchmark system and also adopts a microservice architecture, containing up to 41 microservices. The system is large in scale, comprising approximately 289k source lines of code and 2k files.

Table 6.1: An overview of our subject systems

Subject	Version	Source lines of code (K)	# files	# tests
TeaStore	v1.4.2	32.9	406	163
TrainTicket	v0.2.1	289.3	2,072	628

6.4.2 Evaluation environment

The evaluation of our subject systems is conducted on two separate machines interconnected within the same internal network. Each machine has a configuration of Intel Core i7-6700 CPU, 32GB of RAM, and operates on Ubuntu 22.04.3 LTS. The first machine functions as the application server to host the subject system, while the second machine serves to generate workloads (via JMeter load driver [9]) to simulate real-world users interacting with the system. We leverage Podman container (v4.6.2) and Podman-Compose orchestration (v1.0.6) for the deployment and management of our subject systems, where each container runs a particular microservice. It is worth noting that we deliberately restrict the CPU usage for each service container to avoid resource contention among containers. We utilize the logs generated by the load generator to collect the response time of each request and leverage *podman stats* to gather the resource (CPU) usage statistics for each of the containers. To ensure sufficient precision of measurement while not introducing significant performance overhead, we sample the performance data at a 5-second interval. Furthermore, we have instrumented the subject systems with Kieker [217] to collect the execution time of each function and their interactions. Kieker offers comprehensive dynamic analysis capabilities at runtime with low overhead and does not require modifying the source code.

6.4.3 Performance regressions

By checking the development history (i.e., commit logs, issue tracking, and comments) of our subject systems, we did not find any historical performance regressions of specific commits or versions. Therefore, we opt to manually inject performance regressions in the source code of the subject systems for the evaluation of our approach. Inspired by the previous studies [140, 119, 81], we consider injecting performance regression by adding a busy waiting to simulate additional heavy-weight operations in the source code to slow down the software execution. We inject this performance regression at the beginning of a function to avoid the effect of data-flow or control-flow complexities within the function's code, thus ensuring the unconditional execution of the injected regressions. To mitigate the bias introduced by injecting performance regressions at one specific location in the software, we systematically examine the source code of our subject systems and arbitrarily identify three distinct locations from different subsystems and different components for regression injection, denoted as L1, L2, and L3.

Furthermore, we also investigate the performance regressions with various intensities, i.e., different waiting time lengths. According to prior work on evaluating software microbenchmark suites, Laaber and Leitner [140] observe that slowdowns exceeding 50%

(i.e., 1.5x slower) in most Java projects can often be reliably detected. Inspired by this finding, at each of the selected locations in the source code, we respectively inject busy waiting with three distinct time lengths, including 10%, 50%, and 250% of the component's original execution time, and denote them as Low, Medium, and High. Due to the system's complexity, there can be inherent randomness and variation of the injected performance regressions. It is often challenging to precisely control the percentage of the slowdown, therefore, we inject regressions roughly equal to the specified percentage. This process would yield a total of nine distinct versions per subject system, each of which incorporates the performance regression with a specific intensity and at a specific location.

6.4.4 Component-level and system-level performance test design

Our evaluation relies on component-level performance tests to understand the impact of the injected performance regressions at the component level and further propagate such impact to the architectural model. For both of our subject systems, although they lack existing performance microbenchmarks, there exists a large number of unit tests, i.e., TeaStore has 163 test cases and TrainTicket has 628 test cases. In addition, prior studies [119, 84, 141], have pointed out that unit tests can be a good proxy for performance tests (if there are no real performance tests). Therefore, we leverage these existing unit tests to facilitate the performance testing of the local components of our subject systems. We run each test a total of 30 times to minimize environmental noise and system randomness, similar to what was done in prior work [81].

While our approach does not inherently require system-level performance tests (cf. Section 6.3), our study still relies on these tests to establish an oracle regarding the impact of performance regressions on the system-level performance, which is crucial for evaluating the effectiveness of our approach. We examine the design documentation of our subject systems and identify typically triggered usage scenarios to exercise various aspects of the system. We choose the workload intensity that exercises the system performance without saturation since our approach is intended to work before extreme performance degradation. In particular, for the TeaStore application, we have designed a diverse workload scenario in which the clients check the store, log into the system, and browse various categories and products, with the intensity of 50 requests/sec, and for the TrainTicket subject, we have realized a workload scenario in which the admin logs into the system, updates user information, and clients check their order and change a ticket reservation, with the intensity of 133 requests/sec. To ensure steady-state performance, we conduct 15-minute system performance testing for each run, in which we exclude the initial 5-minute warm-up period

data and retain the subsequent 10-minute data as steady-state performance data. Furthermore, we repeat each system performance testing run three times to ensure robust results and mitigate the impact of the environmental noise and randomness during performance measurements.

6.4.5 Architectural models

Our study leverages architectural models of the system to detect system performance regressions, however, after manually checking both of our subject systems, we find that their architectural performance models are not explicitly provided. Therefore, we need to construct the architectural models of our subject systems for evaluation purposes. To achieve it, we rely on the collected system runtime behavior and performance information during system performance testing (cf. Section 6.4.4) to construct the system's structure and calculate the model service demands, which is similar to the process in prior studies [199, 135]. In our study, we utilize the QPN [43] modeling formalism to construct the architectural model due to its superior expressiveness and effectiveness in modeling the performance of complex software systems [214]. It is also worth noting that our study does not rely on specific types of architectural models, and the construction of architectural models is also not the primary focus of our study. Practitioners are encouraged to refer to the existing literature (cf. Section 6.2) for insights into various modeling techniques and methodologies.

6.5 Evaluation Design and Results

In this section, we first introduce the detailed evaluation design of our approach. Then, we present the data analysis for the evaluation. Finally, we discuss the evaluation results.

6.5.1 Evaluation design

We consider two evaluation scenarios to assess the effectiveness of our approach. In the traditional in-house performance testing scenario, developers often fix the workloads when they test and compare the performance of different versions of systems [93]. Therefore, we consider **fixed-workload** as our first evaluation scenario. In this scenario, for each of our subject systems, we first perform system-level performance testing on both the original version (with no injected regression) and the updated versions (with injected regressions of varying intensities and located at different locations in the source code) with our designed

workload (cf. Section 6.4.4). We then compare the performance testing results from the original and updated versions to understand the impact of performance regressions on the system performance in terms of response time and CPU percentage. It is noteworthy that our approach does not inherently require system-level performance testing (cf. Section 6.3), and the purpose of doing so is to establish an oracle regarding the impact of performance regressions on the system-level performance, which is crucial for evaluating the effectiveness of our approach. Afterward, we apply our approach to each of the updated versions with the same workload and then examine whether our approach can detect performance regressions and keep consistent with the performance testing results.

In the second evaluation scenario, we assess whether our approach works when system workloads change between system versions. Changes in the system workload are common, as they are influenced by the dynamics of the operating environment (e.g., an increase in the number of users over time). In such cases, the same local performance deviation may impact the system-level performance differently. Therefore, we also consider a **various-workload** evaluation scenario to explore whether our approach can capture these diverse impacts and remain effective under various workloads. In this scenario, we carefully devise multiple workload variants of the original workload. In particular, the first variant increases the workload intensities, the second variant changes the execution ratio of various request types, and the third variant combines both the first and second variants simultaneously. Similar to the fixed-workload scenario, we first conduct system-level performance testing to understand the impact of the same regressions under various workloads on the system-level performance and then apply our approach to these various workloads. We pay special attention to the minimum detectable regressions at each location in the fixed-workload scenario (if absent, then the highest intensity is used) to examine whether our approach can still successfully detect these performance regressions and remain consistent with the performance testing results under various workloads.

6.5.2 Data analysis

Table 6.2 and Table 6.3 show the detailed results of applying our approach to detecting end-to-end system performance regressions for our subject systems in the **fixed-workload** and **various-workload** scenarios, respectively. We use the following three metrics to evaluate our approach.

MPD refers to the mean performance degradation between two versions (with and without performance regressions). It explains how the regression impacts the mean of system performance. A positive value indicates a longer response time or higher CPU percentage in the updated version.

Effect size is a statistical metric used to quantify the practical significance of the observed effects of performance regressions in a standardized way (independent of the measurement scales or units). It compares the performance distributions before and after injecting regressions. This metric can help determine whether the observed effects are meaningful or merely coincidental, and its magnitude reflects effect strength. We opt to calculate Cliff's Delta effect size since it does not assume a specific distribution for the performance data.

Outcome examines whether the impact of performance regressions measured during performance testing (i.e., the oracle) and the impact predicted by our approach are consistent. In particular, for response time, we consider the effect size larger than negligible as the presence of regressions, and the "Outcome" indicates the classification (including TP for true positive, TN for true negative, FP for false positive, and FN for false negative) of model prediction compared to performance testing based on the effect size, while for CPU percentage, since only mean CPU percentage is produced, the "Outcome" measures the absolute difference ($|\Delta|$) between the MPD of performance testing and model prediction.

6.5.3 Evaluation results

Evaluation in the fixed-workload scenario:

When there exist significant local performance deviations, their impact on end-to-end system performance may not always be significant. For example, as shown in Table 6.2, in the TeaStore subject, for the local performance deviation at location L2 with Low and Medium intensities and L3 with all three intensities (i.e., Low, Medium, and High), there exist significant local performance deviations with up to 250% slowdown (cf. Section 6.4.3), however, their impact on end-to-end system performance is all insignificant, i.e., with a p-value > 0.05 or only negligible effect size in response time and very low MPD (≤ 1.01) in CPU percentage. After further investigation, we observe that these components exhibiting performance deviations have an inherently low impact in terms of system performance. For instance, even when subjected to the most severe performance regression (i.e., injecting an extra 250% waiting time), the execution time of these components mostly does not exceed 1 millisecond. Furthermore, these components are often executed infrequently during system runtime (e.g., only once), further diminishing their impact on system performance. Such a finding indicates that significant local performance deviations may not always cause a notable impact at the system level, potentially leading to false positives. Therefore, if developers consistently choose to conduct system performance testing whenever there are notable local performance deviations, it could result in

Table 6.2: Overall results of detecting end-to-end system performance regressions in the **fixed-workload** scenario

Local performance deviation		**System performance regression**							
		Response time					**CPU percentage**		
Location	**Intensity**	**Performance testing**		**Model prediction**		**Outcome**	**Performance testing**	**Model prediction**	**Outcome ($\lvert\Delta\rvert$)**
		MPD (ms)	**Effect size**	**MPD (ms)**	**Effect size**		**MPD**	**MPD**	
L1	Low	7.02	medium	7.60	small	TP	6.53	6.09	0.44
	Medium	41.68	large	37.67	large	TP	33.03	28.82	4.21
	High	250.37	large	749.01	large	TP	110.60	149.55	38.95
L2	Low	5.31	$p > 0.05$	0.39	negligible	TN	0.13	0.27	0.14
	Medium	4.52	negligible	1.36	negligible	TN	1.01	-0.08	1.09
	High	8.65	small	6.42	small	TP	3.91	5.21	1.30
L3	Low	-1.08	negligible	0.11	negligible	TN	-0.12	< 0.01	0.12
	Medium	0.52	$p > 0.05$	0.51	negligible	TN	-0.19	0.31	0.50
	High	-0.88	negligible	2.11	negligible	TN	-0.11	1.68	1.79

(a) TeaStore

Local performance deviation		**System performance regression**							
		Response time					**CPU percentage**		
Location	**Intensity**	**Performance testing**		**Model prediction**		**Outcome**	**Performance testing**	**Model prediction**	**Outcome ($\lvert\Delta\rvert$)**
		MPD (ms)	**Effect size**	**MPD (ms)**	**Effect size**		**MPD**	**MPD**	
L1	Low	3.63	$p > 0.05$	-0.25	$p > 0.05$	TN	0.41	-0.03	0.44
	Medium	-1.61	$p > 0.05$	1.01	negligible	TN	< 0.01	0.28	0.28
	High	12.51	small	6.37	negligible	FN	1.97	1.69	0.28
L2	Low	0.48	small	0.33	negligible	FN	1.16	0.89	0.27
	Medium	1.47	medium	1.64	small	TP	3.77	4.25	0.48
	High	5.34	large	8.00	large	TP	13.81	20.12	6.31
L3	Low	0.04	negligible	0.46	negligible	TN	0.08	1.18	1.10
	Medium	0.40	negligible	1.72	negligible	TN	1.05	4.43	3.38
	High	1.86	large	9.12	large	TP	4.92	22.76	17.84

(b) TrainTicket

Note 1: "MPD" refers to the mean performance degradation between two versions (with and without performance regressions).
Note 2: For response time, the "Outcome" indicates the classification of model prediction compared to performance testing based on effect size, while for CPU percentage, since only mean CPU percentage is produced, the "Outcome" measures the absolute difference ($|\Delta|$) between the MPD of performance testing and model prediction.
Note 3: "TP" (true positive) and "TN" (true negative) indicate the successfully detected presence and absence of performance regressions by our approach, respectively, while "FP" (false positive) and "FN" (false negative) indicate falsely detected presence and absence of performance regressions, respectively.

a considerable waste of resources and time. This finding signifies the importance of our approach.

Our approach does not generate false alarms when there is local performance deviation but no system performance regression. As shown in Table 6.2, when there is local performance deviation but no performance regression in system performance testing, our approach does not generate false alarms (i.e., FP). For instance, in TeaStore, for the local performance deviation at the location of L2 with Low and Medium intensities and L3 with all three intensities (i.e., Low, Medium, and High), when there exists insignificant system performance impact observed from performance testing, the results predicted from our approach always indicate a negligible difference between the system response times before and after injecting the performance regressions. In addition, regarding CPU percentage, for both of our subject systems (i.e., TeaStore and TrainTicket), the MPD between our model predictions and performance testing results are also similar. The largest difference between the model predicted and the actual MPD for CPU percentage is only 3.38 when there exist no system performance regressions.

Our approach can effectively detect end-to-end system performance regressions caused by local performance deviations in the components. As presented in Table 6.2, for the cases where the local performance deviation leads to notable system performance regression (i.e., with a small to large effect size in performance testing), our approach can detect seven out of nine cases in our subject systems with comparable results (in MPD and effect size) to the performance testing results in terms of response time. The two (FN) cases missed by our approach are all from TrainTicket, i.e., L1-High and L2-Low. It is worth noting that despite missing these two cases, their actual impact on the system performance is relatively low (i.e., both have small effect sizes in performance testing). Furthermore, we suspect that one potential reason for these cases could be the high complexity and variability of the system (cf. Section 6.4.1), leading to challenges in accurately building architectural models and thus impacting the effectiveness of detecting performance regressions. Concerning CPU percentage, we observe that the differences between the model predicted and the actual MPD when there are system performance regressions are relatively larger than those when there are no regressions. However, such differences are consistently in the right direction, i.e., model prediction shows a larger MPD than performance testing, therefore, they do not fundamentally impact the determination of performance regressions.

The presence of local performance deviations may not always indicate a notable deviation of the end-to-end system performance, thus conducting system performance testing in such cases would lead to wastage in terms of both time and resources. Our approach can successfully detect end-to-end system performance regressions without the execution of time- and resource-consuming system-level performance testing.

Evaluation in the various-workload scenario:

Our approach can maintain its effectiveness in detecting end-to-end system performance regressions even when the system experiences various workloads. As shown in Table 6.3, we find that our approach can consistently demonstrate effectiveness when the system experiences various workloads. In particular, for TrainTicket, except for the local performance deviation of L1, which is undetectable even under the original workload, the model prediction results consistently align with the performance testing results. As for TeaStore, our approach maintains consistency with the performance testing results in seven out of nine cases. We notice a FN case in TeaStore for local performance deviation of L1-Low under variant 3, i.e., small regression in performance testing while negligible regression in model predictions. Upon a closer examination, we find that the MPD in response time from performance testing and model prediction is quite close, with values of 7.22 ms compared to 7.90 ms. In addition, the effect size for the model prediction results is 0.141, which is just slightly different from the threshold of small effect size at 0.147. In addition, we identify a FP case in TeaStore at L2-High under variant 3, i.e., negligible regression in performance testing but small regression in model predictions. After further investigation, we find that from the performance testing results, the mean response time after injecting the performance regression is even slightly lower (i.e., with a negative MPD), which indicates a faster execution compared to the pre-injection state. Such a phenomenon would be explained by the nature of the system's intricacies, where the system may cache certain hotspot codes or adopt other optimization strategies to improve overall performance. Nevertheless, capturing such optimizations in models can prove challenging and requires more consideration of the underlying structure and mechanisms of the system, which we leave for further research.

Component-level performance testing can hardly capture the variety in the system workloads, thus it cannot consistently reflect the true effect of local performance deviation on the end-to-end system performance under various

Table 6.3: Overall results of detecting end-to-end system performance regressions in the **various-workload** scenario

Local performance deviation		System performance regression								
			Response time					CPU percentage		
Location	Intensity	Workload	Performance testing MPD (ms)	Effect size	Model prediction MPD (ms)	Effect size	Outcome	Performance testing MPD	Model prediction MPD	Outcome ($\lvert\Delta\rvert$)
L1	Low	Original	7.02	medium	7.60	small	TP	6.53	6.09	0.44
		Variant 1	7.25	small	8.51	small	TP	12.24	12.29	0.05
		Variant 2	7.27	medium	7.58	small	TP	4.17	4.15	0.02
		Variant 3	7.22	small	7.90	negligible	FN	8.48	8.22	0.26
L2	High	Original	8.65	small	6.42	small	TP	3.91	5.21	1.30
		Variant 1	2.03	small	6.65	small	TP	6.24	10.59	4.35
		Variant 2	4.14	small	6.46	small	TP	4.28	7.10	2.82
		Variant 3	-1.82	negligible	7.04	small	FP	7.99	14.09	6.10
L3	High	Original	-0.88	negligible	2.11	negligible	TN	-0.11	1.68	1.79
		Variant 1	-3.16	negligible	2.08	negligible	TN	0.66	3.52	2.86
		Variant 2	-2.73	negligible	2.06	negligible	TN	0.31	3.54	3.23
		Variant 3	-1.81	negligible	2.16	negligible	TN	1.61	7.06	5.45

(a) TeaStore

Local performance deviation		System performance regression								
			Response time					CPU percentage		
Location	Intensity	Workload	Performance testing MPD (ms)	Effect size	Model prediction MPD (ms)	Effect size	Outcome	Performance testing MPD	Model prediction MPD	Outcome ($\lvert\Delta\rvert$)
L1	High	Original	12.51	small	6.37	negligible	FN	1.97	1.69	0.28
		Variant 1	32.79	small	7.47	negligible	FN	5.31	2.80	2.51
		Variant 2	3.73	small	6.80	negligible	FN	0.73	1.73	1.00
		Variant 3	11.64	small	7.50	negligible	FN	3.10	2.89	0.21
L2	Medium	Original	1.47	medium	1.64	small	TP	3.77	4.25	0.48
		Variant 1	1.36	medium	1.67	small	TP	5.26	6.33	1.07
		Variant 2	1.31	medium	1.66	small	TP	5.05	6.03	0.98
		Variant 3	2.49	medium	1.71	small	TP	7.92	9.07	1.15
L3	High	Original	1.86	large	9.12	large	TP	4.92	22.76	17.84
		Variant 1	1.77	medium	9.71	large	TP	6.76	34.17	27.41
		Variant 2	1.74	medium	8.91	large	TP	3.20	16.17	12.97
		Variant 3	1.90	medium	9.18	large	TP	5.29	24.22	18.93

(b) TrainTicket

Note 1: "MPD" refers to the mean performance degradation between two versions (with and without performance regressions).
Note 2: For response time, the "Outcome" indicates the classification of model prediction compared to performance testing based on effect size, while for CPU percentage, since only mean CPU percentage is produced, the "Outcome" measures the absolute difference ($\lvert\Delta\rvert$) between the MPD of performance testing and model prediction.
Note 3: "TP" (true positive) and "TN" (true negative) indicate the successfully detected presence and absence of performance regressions by our approach, respectively, while "FP" (false positive) and "FN" (false negative) indicate falsely detected presence and absence of performance regressions, respectively.

workloads. From Table 6.3, we also observe that even the local performance deviation with the same intensity and at the same location in the source code can lead to different impacts (different MPD and effect size) on the end-to-end system performance when the system is under various workloads. For instance, in the performance testing results of TeaStore, for the local performance deviation L1-Low, when the system experiences the workload variant 2, the MPD of CPU percentage is only 4.17. However, when the system experiences workload variant 1 (cf. Section 6.5.1), the impact of local performance deviation on CPU percentage drastically escalates to 12.24 in MPD, approximately tripling the original value. Upon deeper investigation, it has come to our attention that due to the increased workload intensity in variant 1, the component experiencing performance deviations is being executed much more frequently. As a result, the workload variant (i.e., variant 1) has resulted in a more significant performance impact at the system level, even with the same local performance deviation in the source code. Such a finding implies that only focusing on the local performance deviation without considering the various system workloads is often not enough to accurately reflect the true effect of local performance deviation on the end-to-end system performance.

> Without considering the system workloads, it is often challenging to reflect the true effect of local performance deviations on the system performance. Our approach remains effective in detecting end-to-end system performance regressions even when the systems experience various workloads.

6.6 Threats to Validity

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

6.6.1 External validity.

Our study is performed on two representative and prevalent open-source systems. Although both subject systems adopt the microservice architecture, our approach is not limited to the unique characteristics of microservices, but rather applicable to all systems with subsystems. Nevertheless, more case studies on other software systems adopting other types of architecture and in other domains can benefit the evaluation of our approach.

6.6.2 Internal validity.

Our study relies on the performance data collected during the component-level and system-level performance testing of the system. The unsatisfactory quality of the performance data, e.g., non-stable or high-noisy data, can adversely impact the internal validity of our study. To mitigate this threat, we conduct our experiments with rigorous measures. We repeat the experiment 30 times for local performance testing, and for system performance testing, we repeat the experiment 3 times, each lasting 15 minutes. Additionally, we exclude data from the initial 5-minute warm-up period. Our approach also depends on the availability and the accuracy of an architectural model of the software system. Therefore, our approach may not be feasible when there is no architectural model and may not perform well when the model provides poor accuracy or the model is outdated. In addition, our study only considers one type of architectural model (i.e., QPN model) developed from performance testing data (cf. Section 6.4.5). However, in practice, there exist other types of architectural performance models (e.g., the LQN model), and the models can also be developed from other sources, like actual field performance data. Considering more types of architectural models developed from various sources of performance data is in our future plan.

6.6.3 Construct validity.

Our study is conducted under constant load during system performance testing, and the system behaviors may differ when conducted under continuously varying loads. Future work may extend our approach under continuously varying loads. Furthermore, our evaluation results depend on thresholds to determine the statistical significance, while in practice, developers usually rely on their subjective judgment to determine the significance for their systems.

6.7 Conclusion

In this chapter, we propose a novel approach to detecting end-to-end system performance regressions as early as in the software development phase, i.e., before the expensive system performance testing. By bridging the local performance data collected during the performance testing of individual components and the architectural models of the entire system, our approach can predict the impact of component-level performance deviations

on the end-to-end system performance, enabling lightweight performance regression detection. Our evaluation results on two representative benchmark systems demonstrate that our approach can effectively detect system performance regressions with different intensities of local performance deviations and under various system workloads. Furthermore, our proposed approach also offers a promising solution to complement traditional system performance testing to ensure the software performance in the fast-paced software development and release practice (e.g., DevOps) where testing resources are limited.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Performance plays a vital role in large-scale software systems since these systems often need to provide high-quality services to millions or even billions of users on a daily basis. Performance regressions are common and vital in large-scale software systems. Prior research often relies on system performance testing results to detect performance regressions and locate the corresponding root causes. However, existing practices are often expensive in both time and resources and therefore do not fit well in the fast-paced process of DevOps in addressing software performance regressions. We hypothesize that there is abundant information during software development and field operations that can be leveraged to assist developers and operators in effectively and timely addressing software performance regressions. To evaluate our hypothesis, in the first part of this thesis, we propose a series of automated approaches to helping developers and operators first detect performance regressions and then locate performance regression root causes directly based on analyzing field operational data without the need for system performance testing. Finally, we adapt the existing performance analytic techniques in a real-life database-centric system. In the second part of this thesis, we propose a novel approach to provide developers with an early impression of how the software changes in specific components are related to the presence of performance regressions of the entire software system. Our results and approaches to addressing software performance regressions would provide insights and support to benefit practitioners and researchers in conducting performance assurance activities during DevOps without expensive system performance testing.

7.2 Future Research

We believe that this thesis makes major contributions toward the goal of assisting developers and operators in addressing software performance regressions. However, there are still many open challenges and research opportunities to improve software performance in practice. We now discuss some directions for future research.

7.2.1 Proactive Software Performance Assurance

Software performance issues are often addressed late in the software development and release cycle, for instance, after the system is built, integrated, or even released. Not only does this make it laborious for developers to detect, locate, and fix performance issues, but it can also lead to potential adverse effects on users and companies. In the future, we plan to carry out empirical studies on the perceptions of performance anti-patterns from the perspective of both research and industry. In addition, we also plan to design an approach to detect performance anti-patterns in the source code and estimate their impact on the end users.

7.2.2 Reduce the Length of Performance Testing with Production Data

Some companies may still consider system performance testing prior to each new release as necessary to ensure the software quality. In response to this, we plan to investigate production data for the purposes of more effective and more efficient system performance testing. In particular, based on the production data, we plan to guide developers in creating more realistic and representative system performance test scenarios while maintaining the testing minimum. Therefore, we can increase the effectiveness of performance testing in finding real performance regressions and at the same time, reduce the required time and resources of performance testing.

7.2.3 Performance Assurance of AI-Centric Software

Due to the remarkable capabilities of artificial intelligence (AI) in tackling many complex problems across diverse application domains, it is undoubtedly that AI will be at the

core of future software systems. The performance (e.g., speed and energy cost) of AI-centric software is crucial, especially since many AI-centric software systems often run on low-power devices (e.g., IoT devices) with limited computing capabilities. However, we are not sure whether the existing practices for traditional software systems still work for these AI-centric software systems. In addition, when designing and developing AI-centric software systems, even though AI components are at their core, people often use these AI components off-the-shelf and do not actively optimize the performance of the AI components. Therefore, what I plan to do in my future research is treat the performance of the AI component as a first-class citizen, considering it alongside the overall system performance in a proactive way, to ensure the performance of AI-centric software in a holistic manner.

References

[1] 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/, 2012.

[2] Canary release. https://martinfowler.com/bliki/CanaryRelease.html, 2014.

[3] sp_whoisactive sql server monitoring stored procedure by adam machanic. http://whoisactive.com, 2017.

[4] Apache james project-apache james server 3-release notes. http://james.apache.org/server/release-notes.html, 2019.

[5] The cost of performance issues. https://focusaps.com/2019/09/26/the-costs-of-performance-issues/, 2019.

[6] Gridsearchcv. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html, 2019.

[7] psutil documentation. https://psutil.readthedocs.io, 2019.

[8] Tpc benchmark w (tpc-w). http://www.tpc.org/tpcw/, 2019.

[9] Apache jmeter - apache jmeter™. https://jmeter.apache.org/, 2020.

[10] Apache tomcat® - welcome! http://tomcat.apache.org/, 2020.

[11] cloc - count lines of code. https://github.com/AlDanial/cloc/, 2020.

[12] Cloudscale project. https://www.cloudscale-project.eu/, 2020.

[13] Eclipse java development tools (jdt). https://www.eclipse.org/jdt/, 2020.

[14] Elastic stack: Elasticsearch, kibana, beats & logstash — elastic. https://www.elastic.co/elastic-stack/, 2020.

[15] Google cloud: Cloud computing services. https://cloud.google.com/, 2020.

[16] Mozilla performance regressions policy. https://www.mozilla.org/en-US/about/governance/policies/regressions/, 2020.

[17] Multicollinearity in regression analysis: Problems, detection, and solutions. https://statisticsbyjim.com/regression/multicollinearity-in-regression-analysis/, 2020.

[18] The .net compiler platform sdk. https://docs.microsoft.com/en-ca/dotnet/csharp/roslyn-sdk/, 2020.

[19] Openmrs. https://openmrs.org/, 2020.

[20] pidstat(1): Report statistics for tasks - linux man page. https://linux.die.net/man/1/pidstat, 2020.

[21] Apache spark - unified engine for large-scale data analytics. https://spark.apache.org/, 2022.

[22] typeperf—microsoft. https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/typeperf, 2022.

[23] Website load time statistics: Why speed matters in 2023, 2023.

[24] Rui Abreu and Arjan J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009*. AAAI, 2009.

[25] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*, pages 39–46. IEEE Computer Society, 2006.

[26] Stephen Adler. The slashdot effect: an analysis of three internet publications. *Linux Gazette*, 38(2):623, 1999.

[27] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exp.*, 23(6):589–616, 1993.

[28] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 246–256. ACM, 1990.

[29] Zuhoor A. Al-Khanjari, Martin R. Woodward, Haider Ali Ramadhan, and Narayana Swamy Kutti. The efficiency of critical slicing in fault localization. *Softw. Qual. J.*, 13(2):129–153, 2005.

[30] Juan Pablo Sandoval Alcocer and Alexandre Bergel. Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, page 129–139, New York, NY, USA, 2015. Association for Computing Machinery.

[31] Juan Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, and Marcus Denker. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*, pages 1–9. IEEE Computer Society, 2013.

[32] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*, pages 37–48. ACM, 2016.

[33] Hamzeh Alimohammadi and Shengnan Nancy Chen. Performance evaluation of outlier detection techniques in production timeseries: A systematic review and meta-analysis. *Expert Syst. Appl.*, 191:116371, 2022.

[34] Erik R. Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 739–753. ACM, 2010.

[35] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *26th*

IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011, pages 520–523. IEEE Computer Society, 2011.

[36] Moiz Arif, Weiyi Shang, and Emad Shihab. An empirical study on the discrepancy between performance testing results from virtual and physical environments. *Empirical Software Engineering*, page To Appear, 2017.

[37] George K. Baah, Andy Podgurski, and Mary Jean Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Trans. Software Eng.*, 36(4):528–545, 2010.

[38] F Baccelli, G Balbo, RJ Boucherie, J Campos, and G Chiola. Annotated bibliography on stochastic petri nets. *Performance Evaluation of Parallel and Distributed Systems—Solution Methods*, 105:1–24, 1994.

[39] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[40] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.

[41] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. Autonomic load-testing framework. In *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011*, pages 91–100, 2011.

[42] Falko Bause. Qn+ pn= qpn. *Combining Queueing Networks and Petri Nets. Technical report*, (461), 1993.

[43] Falko Bause. Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models, PNPM 1993, Toulouse, France, October 19-22, 1993*, pages 14–23. IEEE Computer Society, 1993.

[44] Falko Bause, Peter Buchholz, and Peter Kemper. Integrating software and hardware performance models using hierarchical queeing petri nets. In *MMB*, pages 87–101, 1997.

[45] Lee A Becker. Effect size (es). *Accessed on October*, 12(2006):155–159, 2000.

[46] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[47] Alexandre Bergel, Felipe Bañados, Romain Robbes, and Walter Binder. Execution profiling blueprints. *Softw. Pract. Exp.*, 42(9):1165–1192, 2012.

[48] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Mónica Villavicencio, Jürgen Walter, and Felix Willnecker. How is performance addressed in devops? In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*, pages 45–50. ACM, 2019.

[49] André B Bondi. *Foundations of software and system performance engineering: process, performance modeling, requirements, testing, scalability, and practice*. Pearson Education, 2015.

[50] Carlo Bonferroni. Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze*, 8:3–62, 1936.

[51] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[52] Leo Breiman, Adele Cutler, Andy Liaw, and Matthew Wiener. *Breiman and Cutler's Random Forests for Classification and Regression*, 2018. R package version 4.6-14.

[53] Leo Breiman and Philip Spector. Submodel selection and evaluation in regression. the x-random case. *International statistical review/revue internationale de Statistique*, pages 291–319, 1992.

[54] Jennifer Cannon. Performance degradation affecting salesforce clients. https://marketingland.com/performance-degradation-affecting-salesforce-clients-267699, 2019.

[55] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. In *Formal Concept Analysis, 6th International Conference, ICFCA 2008, Montreal, Canada, February 25-28, 2008, Proceedings*, volume 4933 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2008.

[56] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *Proceedings of the 23rd International Conference*

on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011, pages 238–243. Knowledge Systems Institute Graduate School, 2011.

[57] Boyuan Chen, Zhen Ming Jiang, Paul Matos, and Michael Lacaria. An industrial experience report on performance-aware refactoring on a database-centric web application. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 653–664. IEEE, 2019.

[58] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 341–352. IEEE Computer Society, 2017.

[59] Jinfu Chen, Weiyi Shang, Ahmed E. Hassan, Yong Wang, and Jiangbin Lin. 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*, pages 669–681. IEEE, 2019.

[60] Jinfu Chen, Weiyi Shang, and Emad Shihab. Perfjit: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Trans. Software Eng.*, 48(5):1529–1544, 2022.

[61] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, and Eric Brewer. Failure diagnosis using decision trees. *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 36–43, 2004.

[62] T. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 243–252, 2017.

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

[64] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. Detecting problems in the database access code of large scale systems: an industrial experience report. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 71–80. ACM, 2016.

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

[66] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Trans. Software Eng.*, 42(12):1148–1161, 2016.

[67] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 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*, pages 243–252. IEEE Computer Society, 2017.

[68] Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xiao Ling, and Michael R. Lyu. Adaptive performance anomaly detection for online service systems via pattern sketching. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 61–72. ACM, 2022.

[69] Norman Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[70] Ira Cohen, Jeffrey S. Chase, Moisés Goldszmidt, Terence Kelly, and Julie Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 231–244, 2004.

[71] Ira Cohen, Steve Zhang, Moisés Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 105–118, 2005.

[72] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 1988.

[73] Vittorio Cortellessa, Daniele Di Pompeo, Romina Eramo, and Michele Tucci. A model-driven approach for continuous performance engineering in microservice-based systems. *J. Syst. Softw.*, 183:111084, 2022.

[74] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 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*, pages 153–167, 2017.

[75] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. What's wrong with my benchmark results? studying bad practices in jmh benchmarks. *IEEE Transactions on Software Engineering*, 47(7):1452–1467, 2021.

[76] Marc Courtois and C. Murray Woodside. Using regression splines for software performance analysis. In *Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000.*, pages 105–114, 2000.

[77] Deborah S Coutant, Sue Meloy, and Michelle Ruscetta. Doc: A practical approach to source-level debugging of globally optimized code. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*, pages 125–134, 1988.

[78] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM Conference on Recommender Systems, RecSys 2019, Copenhagen, Denmark, September 16-20, 2019.*, pages 101–109, 2019.

[79] Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Why you should care about quantile regression. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 207–218, 2013.

[80] Vidroha Debroy, W. Eric Wong, Xiaofeng Xu, and Byoungju Choi. A grouping-based strategy to improve the effectiveness of fault localization techniques. In *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, pages 13–22. IEEE Computer Society, 2010.

[81] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, and Inmaculada Medina-Bulo. Performance mutation testing. *Software Testing, Verification and Reliability*, 31(5):e1728, 2021.

[82] Alin Deutsch, Richard Hull, Yuliang Li, and Victor Vianu. Automatic verification of database-centric systems. *ACM SIGLOG News*, 5(2):37–56, 2018.

[83] Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, Jan 31 - Feb 4, 2015*, pages 145–156, 2015.

[84] 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 *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1435–1446. ACM, 2020.

[85] Jermaine Charles Edwards. Method, system, and program for logging statements to monitor execution of a program, March 2003.

[86] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dusan Okanovic, and André van Hoorn. Microservices: A performance tester's dream or nightmare? In *ICPE '20: ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada, April 20-24, 2020*, pages 138–149, 2020.

[87] Simon Eismann, Diego Elias Costa, Lizhi Liao, Cor-Paul Bezemer, Weiyi Shang, André van Hoorn, and Samuel Kounev. A case study on the stability of performance tests for serverless applications. *J. Syst. Softw.*, 189:111294, 2022.

[88] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber, and John C. Grundy. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 24–34, 2015.

[89] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, pages 32–41. IEEE Computer Society, 2010.

[90] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 159–168, 2015.

[91] Greg Franks, Tariq Omari, C. Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Software Eng.*, 35(2):148–161, 2009.

[92] David A Freedman. *Statistical models: theory and practice*. cambridge university press, 2009.

[93] Ruoyu Gao, Zhen Ming Jiang, Cornel Barna, and Marin Litoiu. 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*, pages 22–32, 2016.

[94] Shadi Ghaith, Miao Wang, Philip Perry, Zhen Ming Jiang, Patrick O'Sullivan, and John Murphy. Anomaly detection in performance regression testing by transaction profile estimation. *Softw. Test., Verif. Reliab.*, 26(1):4–39, 2016.

[95] Shadi Ghaith, Miao Wang, Philip Perry, and John Murphy. Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 379–383, 2013.

[96] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM 2010, Niagara Falls, Canada, October 25-29, 2010*, pages 9–16, 2010.

[97] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[98] Mark Grechanik, B. M. Mainul Hossain, and Ugo A. Buy. Testing database-centric applications for causes of database deadlocks. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 174–183. IEEE Computer Society, 2013.

[99] Mark Grechanik, B. M. Mainul Hossain, Ugo A. Buy, and Haisheng Wang. Preventing database deadlocks in applications. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 356–366. ACM, 2013.

[100] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[101] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[102] Harvey W. Gunther. WebSphere Application Server Performance and Scalability. *IBM WebSphere Application Server Standard and Advanced Editions - White paper*, 2000.

[103] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 301–311, 2013.

[104] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. Data-efficient performance learning for configurable systems. *Empirical Software Engineering*, 23(3):1826–1867, 2018.

[105] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 145–155, 2012.

[106] Frank E Harrell Jr. *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer, 2015.

[107] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Softw. Test. Verification Reliab.*, 10(3):171–194, 2000.

[108] Wilhelm Hasselbring and André van Hoorn. Kieker: A monitoring framework for software engineering research. *Softw. Impacts*, 5:100019, 2020.

[109] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 156–164. ACM, 2004.

[110] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 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*, pages 60–70, 2018.

[111] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013*, pages 27–38. ACM, 2013.

[112] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, André van Hoorn, Antonio Filieri, and David Lo. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Softw. Pract. Exp.*, 49(8):1197–1224, 2019.

[113] Robert Heinrich. Architectural runtime models for integrating runtime observations and component-based models. *J. Syst. Softw.*, 169:110722, 2020.

[114] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):323–344, 1982.

[115] Vojtech Horký, Peter Libic, Lukás Marek, Antonín Steinhauser, and Petr Tuma. Utilizing performance unit tests to increase performance awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*, pages 289–300. ACM, 2015.

[116] Maarten Houbraken, Sofie Demeyer, Tom Michoel, Pieter Audenaert, Didier Colle, and Mario Pickavet. The index-based subgraph matching algorithm with general symmetries (ismags): exploiting symmetry for faster subgraph enumeration. *PloS one*, 9(5):e97896, 2014.

[117] Jez Humble and Gene Kim. *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations*. IT Revolution, 2018.

[118] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, 48(1):4:1–4:35, 2015.

[119] Mostafa Jangali, Yiming Tang, Niclas Alexandersson, Philipp Leitner, Jinqiu Yang, and Weiyi Shang. Automated generation and evaluation of jmh microbenchmark suites from unit tests. *IEEE Transactions on Software Engineering*, 2022.

[120] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Performance monitoring and root cause analysis for cloud-hosted web applications. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 469–478. ACM, 2017.

[121] Zhen Ming Jiang. Automated analysis of load testing results. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 143–146. ACM, 2010.

[122] Zhen Ming Jiang and Ahmed E. Hassan. A survey on load testing of large-scale software systems. *IEEE Trans. Software Eng.*, 41(11):1091–1118, 2015.

[123] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, pages 307–316. IEEE Computer Society, 2008.

[124] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 125–134. IEEE Computer Society, 2009.

[125] Bjarne Johansson, Alessandro Vittorio Papadopoulos, and Thomas Nolte. Concurrency defect localization in embedded systems using static code analysis: An evaluation. In *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*, pages 7–12. IEEE, 2019.

[126] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. *CoRR*, abs/1302.4964, 2013.

[127] Krishna Kant. *Introduction to computer system performance evaluation*. International Edition, 1992.

[128] S. Khanduja, V. Nair, S. Sundararajan, A. Raul, A. B. Shaj, and S. Keerthi. Near real-time service monitoring using high-dimensional time series. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 1624–1627, 2015.

[129] Gene Kim, Jez Humble, Patrick Debois, and John Willis. *The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations*. IT Revolution, 2016.

[130] Heejin Kim, Byoungju Choi, and W. Eric Wong. Performance testing of mobile applications at the unit test level. In *Third IEEE International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2009, Shanghai, China, July 8-10, 2009*, pages 171–180. IEEE Computer Society, 2009.

[131] Leonard Kleinrock. Queuing systems, volume i: Theory, 1975.

[132] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[133] Ekrem Kocaguneli and Tim Menzies. Software effort models should be assessed via leave-one-out validation. *J. Syst. Softw.*, 86(7):1879–1890, 2013.

[134] Toon Koppelaars. A database-centric approach to j2ee application development. *Oracle Development Tools Users Group (ODTUG)*, 2004.

[135] Samuel Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Trans. Software Eng.*, 32(7):486–502, 2006.

[136] Samuel Kounev and Alejandro P. Buchmann. Performance modeling and evaluation of large-scale J2EE applications. In *29th International Computer Measurement Group Conference, December 7-12, 2003, Dallas, Texas, USA, Proceedings*, pages 273–283. Computer Measurement Group, 2003.

[137] Charles Krasic, Anirban Sinha, and Lowell Kirsh. Priority-progress CPU adaptation for elastic real-time applications. In Roger Zimmermann and Carsten Griwodz, editors, *Multimedia Computing and Networking 2007*, volume 6504, pages 172 – 183. International Society for Optics and Photonics, SPIE, 2007.

[138] Diwakar Krishnamurthy, Jerome A. Rolia, and Shikharesh Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. Software Eng.*, 32(11):868–882, 2006.

[139] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empir. Softw. Eng.*, 7(1):49–76, 2002.

[140] Christoph Laaber and Philipp Leitner. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 119–130. ACM, 2018.

[141] Christoph Laaber, Joel Scheuner, and Philipp Leitner. Software microbenchmarking in the cloud. how bad is it really? *Empir. Softw. Eng.*, 24(4):2469–2508, 2019.

[142] Leslie Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[143] Giuseppe Lancia and Ramamoorthi Ravi. Gestalt: Genomic steiner alignments. In *Annual Symposium on Combinatorial Pattern Matching*, pages 101–114. Springer, 1999.

[144] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance - computer system analysis using queueing network models*. Prentice Hall, 1984.

[145] Meir M. Lehman and Juan F. Ramil. Software evolution and software evolution processes. *Ann. Softw. Eng.*, 14(1-4):275–309, 2002.

[146] Lizhi Liao. Addressing performance regressions in devops: Can we escape from system performance testing? In *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*, pages 203–207. IEEE, 2023.

[147] 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. *Empir. Softw. Eng.*, 25(5):4130–4160, 2020.

[148] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Locating performance regression root causes in the field operations of web-based systems: An experience report. *IEEE Trans. Software Eng.*, 48(12):4986–5006, 2022.

[149] Lizhi Liao, Heng Li, Weiyi Shang, and Lei Ma. An empirical study of the impact of hyperparameter tuning and model optimization on the performance properties of deep neural networks. *ACM Trans. Softw. Eng. Methodol.*, 31(3):53:1–53:40, 2022.

[150] Lizhi Liao, Heng Li, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Adapting performance analytic techniques in a real-world database-centric system: An industrial experience report. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1855–1866. ACM, 2023.

[151] Meng-Hui Lim, Jian-Guang Lou, Hongyu Zhang, Qiang Fu, Andrew Beng Jin Teoh, Qingwei Lin, Rui Ding, and Dongmei Zhang. Identifying recurrent and unknown performance issues. In *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 320–329, 2014.

[152] Henry H. Liu. *Software Performance and Scalability - A Quantitative Approach*. Wiley series on quantitative software engineering. Wiley, 2009.

[153] Siyang Lu, BingBing Rao, Xiang Wei, Byung-Chul Tak, Long Wang, and Liqiang Wang. Log-based abnormal task detection and root cause analysis for spark. In *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, pages 389–396. IEEE, 2017.

[154] Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *J. Softw. Evol. Process.*, 26(2):172–219, 2014.

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

[156] Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. SAND: a static analysis approach for detecting SQL antipatterns. In *ISSTA '21: 30th ACM SIG-*

SOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021, pages 270–282. ACM, 2021.

[157] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1012–1021. IEEE Computer Society, 2013.

[158] Haroon Malik, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*, pages 222–231, 2010.

[159] David Maplesden, Karl von Randow, Ewan D. Tempero, John G. Hosking, and John C. Grundy. Performance analysis using subsuming methods: An industrial case study. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 149–158. IEEE Computer Society, 2015.

[160] L. Mariani, C. Monni, M. Pezzé, O. Riganelli, and R. Xin. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 262–273, 2018.

[161] Leonardo Mariani, Cristina Monni, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. Localizing faults in cloud systems. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 262–273. IEEE Computer Society, 2018.

[162] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling java programs for diagnosis. In *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000*, pages 171–175. IOS Press, 2000.

[163] Wolfgang Mayer and Markus Stumptner. Modeling programs with unstructured control flow for debugging. In *AI 2002: Advances in Artificial Intelligence, 15th Australian Joint Conference on Artificial Intelligence, Canberra, Australia, December 2-6, 2002, Proceedings*, volume 2557 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2002.

[164] Daniel A. Menascé. Load testing, benchmarking, and application performance management for the web. In *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings*, pages 271–282, 2002.

[165] Daniel A. Menascé and Mohamed N. Bennani. On the use of performance models to design self-managing computer systems. In *29th International Computer Measurement Group Conference, December 7-12, 2003, Dallas, Texas, USA, Proceedings*, pages 1–9, 2003.

[166] Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of dynamic database usage in java systems. In *Advanced Information Systems Engineering: 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings 28*, pages 491–506. Springer, 2016.

[167] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 117–126, 2010.

[168] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. Accurate modeling of performance histories for evolving software systems. 2019.

[169] Nadim Nachar. 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, 2008.

[170] Vinod Nair, Ameya Raul, Shwetabh Khanduja, Vikas Bahirwani, Sundararajan Sellamanickam, S. Sathiya Keerthi, Steve Herbert, and Sudheer Dhulipalla. Learning a hierarchical monitoring system for detecting and diagnosing service issues. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 2029–2038. ACM, 2015.

[171] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[172] Syeda Nessa, Muhammad Abedin, W. Eric Wong, Latifur Khan, and Yu Qi. Software fault localization using n-gram analysis. In *Wireless Algorithms, Systems, and Applications, Third International Conference, WASA 2008, Dallas, TX, USA, October 26-28, 2008. Proceedings*, volume 5258 of *Lecture Notes in Computer Science*, pages 548–559. Springer, 2008.

[173] T. H. D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated verification of load tests using control charts. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 282–289, 2011.

[174] Thanh H. D. Nguyen. Using control charts for detecting and understanding performance regressions in large software. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 491–494. IEEE Computer Society, 2012.

[175] Thanh H. D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. Automated verification of load tests using control charts. In *18th Asia Pacific Software Engineering Conference, APSEC 2011, Ho Chi Minh, Vietnam, December 5-8, 2011*, pages 282–289. IEEE Computer Society, 2011.

[176] Thanh H. D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. Automated detection of performance regressions using statistical process control techniques. In *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012*, pages 299–310. ACM, 2012.

[177] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. An industrial case study of automatically identifying performance regression-causes. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 232–241. ACM, 2014.

[178] Seikoh Nishita. More precise analysis of dynamically generated string expressions in web applications with input validation. *IEICE Trans. Inf. Syst.*, 96-D(6):1278–1285, 2013.

[179] George Papadimitriou, Cong Wang, Karan Vahi, Rafael Ferreira da Silva, Anirban Mandal, Zhengchun Liu, Rajiv Mayani, Mats Rynge, Mariam Kiran, Vickie E. Lynch, Rajkumar Kettimuthu, Ewa Deelman, Jeffrey S. Vetter, and Ian T. Foster. End-to-end online performance data capture and analysis for scientific workflows. *Future Gener. Comput. Syst.*, 117:387–400, 2021.

[180] John C. Platt. *Fast Training of Support Vector Machines Using Sequential Minimal Optimization*, page 185–208. MIT Press, Cambridge, MA, USA, 1999.

[181] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[182] John W Ratcliff, David Metzener, et al. Pattern matching: The gestalt approach. *Dr. Dobb's Journal*, 13(7):46, 1988.

[183] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. Peass: A tool for identifying performance changes at code level. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1146–1149. IEEE, 2019.

[184] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1140–1152. ACM, 2020.

[185] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA):211:1–211:30, 2020.

[186] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 667–682. USENIX Association, 2020.

[187] Jerome A. Rolia and Kenneth C. Sevcik. The method of layers. *IEEE transactions on software engineering*, 21(8):689–700, 1995.

[188] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, volume 177, page 34, 2006.

[189] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, volume 177, page 34, 2006.

[190] David S Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th international conference on Software engineering*, pages 92–104, 1992.

[191] David S. Rosenblum. A practical approach to programming with assertions. *IEEE transactions on Software Engineering*, 21(1):19–31, 1995.

[192] Peter J Rousseeuw and Mia Hubert. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery*, 1(1):73–79, 2011.

[193] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *J. Funct. Program.*, 3(2):217–245, 1993.

[194] Danilo Sato. Canary release. *MartinFowler. com*, 2014.

[195] Weiyi Shang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*, pages 15–26. ACM, 2015.

[196] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empir. Softw. Eng.*, 20(1):1–27, 2015.

[197] Walter Andrew Shewhart. *Economic control of quality of manufactured product*. Macmillan And Co Ltd, London, 1931.

[198] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: A study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, page 300–310, 2011.

[199] Yasir Shoaib and Olivia Das. Web application performance modeling using layered queueing networks. In *Fifth International Workshop on the Practical Application of Stochastic Modelling, PASM@ICPE 2011, Karlsruhe, Germany, March 2011*, volume 275 of *Electronic Notes in Theoretical Computer Science*, pages 123–142. Elsevier, 2011.

[200] Connie U. Smith. Introduction to software performance engineering: Origins and outstanding problems. In *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, volume 4486 of *Lecture Notes in Computer Science*, pages 395–428. Springer, 2007.

[201] Connie U. Smith and Lloyd G. Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *28th International Computer Measurement*

Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings, pages 667–674. Computer Measurement Group, 2002.

[202] Connie U Smith and Lloyd G Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley Reading, 2002.

[203] Connie U. Smith and Lloyd G. Williams. More new software antipatterns: Even more ways to shoot yourself in the foot. In *29th International Computer Measurement Group Conference, December 7-12, 2003, Dallas, Texas, USA, Proceedings*, pages 717–725. Computer Measurement Group, 2003.

[204] Connie U. Smith and Lloyd G. Williams. Software performance engineering. In *UML for Real - Design of Embedded Real-Time Systems*, pages 343–365. Kluwer, 2003.

[205] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. Automating serverless deployments for devops organizations. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 57–69. ACM, 2021.

[206] Petr Stefan, Vojtech Horký, Lubomír Bulej, and Petr Tuma. Unit testing performance in java projects: Are we there yet? In Walter Binder, Vittorio Cortellessa, Anne Koziolek, Evgenia Smirni, and Meikel Poess, editors, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 401–412. ACM, 2017.

[207] Gail M Sullivan and Richard Feinn. Using effect size—or why the p value is not enough. *Journal of graduate medical education*, 4(3):279–282, 2012.

[208] Mark D. Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 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*, pages 110–119. IEEE Computer Society, 2013.

[209] Mark D. Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. Continuous validation of load test suites. In *ACM/SPEC International Conference on Performance Engineering, ICPE'14, Dublin, Ireland, March 22-26, 2014*, pages 259–270, 2014.

[210] Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, and Ahmed E. Hassan. Continuous validation of performance test workloads. *Autom. Softw. Eng.*, 24(1):189–231, 2017.

[211] Syncsort. White paper: Assessing the financial impact of downtime, 2018.

[212] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. 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*, pages 795–806, 2010.

[213] Hannes Thaller, Lukas Linsbauer, Alexander Egyed, and Stefan Fischer. Towards fault localization via probabilistic software modeling. In *IEEE Workshop on Validation, Analysis and Evolution of Software Tests, VST@SANER 2020, London, ON, Canada, February 18, 2020*, pages 24–27. IEEE, 2020.

[214] Nidhi Tiwari and Prabhakar Mynampati. Experiances of using LQN and QPN tools for performance modelling of a J2EE application. In *32nd International Computer Measurement Group Conference, December 3-6, 2006, Reno, Nevada, USA, Proceedings*, pages 537–548. Computer Measurement Group, 2006.

[215] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 39–50, 2017.

[216] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012*, pages 247–248. ACM, 2012.

[217] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC international conference on performance engineering*, pages 247–248, 2012.

[218] HP Vinutha, B Poornima, and BM Sagar. Detection of outliers using interquartile range technique from intrusion dataset. In *Information and decision sciences*, pages 511–518. Springer, 2018.

[219] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of*

the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, September 2018.

[220] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

[221] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar T. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4):14, 2007.

[222] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.

[223] Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Software Eng.*, 26(12):1147–1156, 2000.

[224] Frank Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

[225] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani M. Thuraisingham. Effective software fault localization using an RBF neural network. *IEEE Trans. Reliab.*, 61(1):149–169, 2012.

[226] W. Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *Int. J. Softw. Eng. Knowl. Eng.*, 19(4):573–597, 2009.

[227] C. Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 171–187. IEEE Computer Society, 2007.

[228] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Trans. Computers*, 44(1):20–34, 1995.

[229] C. Murray Woodside, Tao Zheng, and Marin Litoiu. The use of optimal filters to track parameters of performance models. In *Second International Conference on the Quantitative Evaluaiton of Systems (QEST 2005), 19-22 September 2005, Torino, Italy*, pages 74–84, 2005.

[230] Murray Woodside, Greg Franks, and Dorina C Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE'07)*, pages 171–187. IEEE, 2007.

[231] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 746–757. Springer, 2002.

[232] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *International Symposium on Search Based Software Engineering*, pages 224–238. Springer, 2013.

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

[234] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. 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*, pages 117–132, 2009.

[235] Ya Xu, Nanyu Chen, Addrian Fernandez, Omar Sinno, and Anmol Bhasin. From infrastructure to culture: A/b testing challenges in large scale social networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 2227–2236, 2015.

[236] Neeraja J. Yadwadkar, Chiranjib Bhattacharyya, Kanchi Gopinath, Thirumale Niranjan, and Sai Susarla. 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*, pages 183–196, 2010.

[237] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 1299–1308. ACM, 2017.

[238] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 800–810. ACM, 2018.

[239] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. Powerstation: automatically detecting and fixing inefficiencies of database-backed web applications in IDE. 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*, pages 884–887. ACM, 2018.

[240] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 127–138, 2018.

[241] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: suggesting and updating logging locations for web-based systems' performance monitoring. *Empirical Software Engineering*, 25(1):488–531, 2020.

[242] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In *International Symposium on Search Based Software Engineering*, pages 244–258. Springer, 2012.

[243] Kai Yu, Mengxiang Lin, Qing Gao, Hui Zhang, and Xiangyu Zhang. Locating faults using multiple spectra-specific models. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, pages 1404–1410. ACM, 2011.

[244] Zhongxing Yu, Hai Hu, Chenggang Bai, Kai-Yuan Cai, and W. Eric Wong. GUI software fault localization using n-gram analysis. In *13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011*, pages 325–332. IEEE Computer Society, 2011.

[245] Tao Zheng, C. Murray Woodside, and Marin Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Trans. Software Eng.*, 34(3):391–406, 2008.

[246] M. Zhou, J. Chen, H. Hu, J. Yu, Z. Li, and H. Hu. Deeptle: Learning code-level features to predict code performance before it runs. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 252–259, 2019.

[247] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans. Software Eng.*, 47(2):243–260, 2021.

[248] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Softw.*, 33(3):32–34, 2016.