Locating Performance Regression Root Causes in the Field Operations of Web-based Systems: An Experience Report

Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Catalin Sporea, Andrei Toma, Sarah Sajedi

Abstract—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 response time of the system may not be able to help locate the root causes of performance regressions. In this paper, 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 address 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.

Index Terms—performance regression, performance regression root causes, field testing, experience report, industry case study.

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 for providing 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 [3, 32, 52, 89]. 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 [13].

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 [62, 97]. 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 [58]. However, performance testing is usually very costly, requiring expensive resources, complex environment configuration, and excessive execution time [39, 62]. More and more software systems are delivered along fast-paced release cycles (e.g., a new version is released every few hours) [60, 66]. 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 [23, 72, 81, 91]. In addition to load tests, unit tests are also leveraged in prior studies [37, 58, 70] to locate performance regression root causes. However, load testing usually requires extensive resources and a long time

- *Lizhi Liao, Yi Zeng, and Weiyi Shang are with the Department of Computer Science and Software Engineering, Concordia University, Canada. E-mail: {l_lizhi, ze_yi, shang}@encs.concordia.ca*
- *Jinfu Chen is with Centre for Software Excellence, Huawei Technologies, Canada. E-mail: jinfu.chen1@huawei.com*
- *Heng Li is with the Département de génie informatique et génie logiciel, Polytechnique Montreal, Canada. E-mail: heng.li@polymtl.ca*
- *Catalin Sporea, Andrei Toma, and Sarah Sajedi are with ERA Environmental, Canada. E-mail: {steve.sporea, andrei.toma, sarah.sajedi}@era-ehs.com*

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) [24, 61, 78, 99], or only target specific performance regression patterns [24].

In this paper, 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 be reproduced 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 [100] 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 as 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 on 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 paper 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 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 paper is organized as follows. Section 2 introduces the background of locating performance regression root causes for an industrial system. Section 3 surveys prior research related to this paper. The challenges and corresponding solutions are discussed in Section 4. Section 5 outlines our approach for locating performance regression root causes in the field. Section 6 and Section 7 respectively present our results of applying our approach on three open-source subject systems and a large-scale industrial system. Section 8 summarizes the lessons that we learned from the addressing the faced challenges and successful adoption of our approach. Section 9 discusses the threats to the validity of our findings. Finally, Section 10 concludes the paper.

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 of 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 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, ESs 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 ESs performance is evaluated in the field with end users, one cannot put extensive instrumentation to monitor the resource cost for each

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

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 [86]) 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 Mann-Whitney U test [77] to determine whether there exists a statistically signi cant 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 [40], 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 [86], 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 eld 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 users' data from the eld (cf. Section 4). To address these challenges, we designed and developed an approach that automatically detects and locates performance regression root causes for ES (cf. Section 5). 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.

3 Related Work

In this section, we discuss prior work related to this paper.

3.1 Performance regression detection and localization

We summarize the related work of performance regression detection and localization in Table 1, including the category of the approach proposed in prior work, the analyzed data type, a brief description, and their major limitations.

Performance regression detection. Prior research designs various types of approaches to detecting performance regressions. The most straightforward approaches of detecting performance regressions are based on directly comparing performance metric values in two versions [37, 71, 80, 81, 82]. This type of research adopts statistical techniques, such as control charts [80, 82] and statistical tests [37, 71] to compare and analyze performance metrics to detect performance regressions. Prior research also investigates the relationship among performance metrics and/or system logs and uses the deviation of such relationship as indicators of performance regressions [48, 57, 91, 101]. For example, Foo et al. [48] build association rules between performance metrics to detect performance regressions. In addition, data-mining-based and statistical techniques like clustering [57, 91] and linear regression models [101] are also leveraged to mine such a relationship between execution logs and performance metrics. Performance models (e.g., queue models [29], rule-based models [80, 82], and machine learning models [44, 86, 99]) are also widely leveraged as vehicles to capture performance regressions. For example, Shang et al. [86] 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 [37, 45, 83, 90]. For example, Reichelt et al. [83] 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 performance regression root causes.

However, prior research on the detection of performance regressions are designed to be conducted based on data generated from time and resource-consuming performance testing with prede ned or xed workloads; while our approach considers the problem of the real workloads in the eld that are continuously varying and only depends on the readily available data that is generated in the eld from end users without the need for performance testing.

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

Various prior research for locating the root causes of performance regressions is based on load testing the target software system [23, 72, 81, 91]. Such approaches compare the load testing results (e.g., performance metrics) from different versions of the software system to locate performance regression root causes. For example, Nguyen et al. [81] propose to mine the regression root cause repository that stores the past load testing results and performance regressions. They use classi cation techniques (e.g., Bayes

TABLE 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	Pro ling data	Injected trace data		Cannot locate root cause	Coarse-grained root cause	Target speci c performance issue patterns	Expensive testing	Not aware of system-level performance	Not consider varying workloads	Expect system knowledge
Based on load testing	[80]	X						Using control charts to compare and analyze performance metrics to identify performance regressions	X	n/a		X		X	
	[29]	X						Modeling the software system using a two-layer queuing model to nd the performance bottlenecks in the system	X	n/a		X		X	X
	[82]	X						Comparing performance counters across test runs using control charts to detect performance regressions	X	n/a		X		X	
	[71]	X	X					Using machine learning techniques (e.g., K-Means, PCA) to nd the best subset of performance counters and build models for performance deviation prediction	X	n/a		X		X	
	[91]	X	X					Adopting hierarchical clustering on time-slices data consists of events and performance counters and analyzing the in uence of outliers in clusters to locate anomalous events		X	X	X		X	
	[81]	X						Mining regression root cause repository and using classi cation techniques to match the behavior of a new version to prior tests to locate performance regression root causes		X		X		X	
	[48]	X						Adopting association rule to capture performance patterns and ensemble learning to detect performance regressions in heterogeneous environments	X	n/a		X		X	
	[86]		X					Clustering various performance counters into groups and build regression models on the clusters to detect performance regressions	X	n/a		X		X	
	[72]	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	
	[23]	X				X		Proposing a sampling technique called horizontal pro ling that assigns a cost to each source code change based on the run-time load testing history				X		X	
Based on unit testing	[58]			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	
	[70]			X	X	X		Using a combination of search-based input pro ling and change impact analysis techniques across software versions to identify code changes that are potentially responsible for performance regressions					X	X	
	[37]			X	X			Adopting unit tests and performance micro-benchmarks to locate performance regression introducing code changes					X	X	
	[83]			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	
	[45]			X	X			Utilizing the tests from the release pipeline to identify performance issue root causes during the rapid development process					X	X	
Based on eld data	[24]		X					Abstracting the states of idleness observed in a running program based on a set of rules de ned declaratively by an expert to diagnosis the root cause of idle time in applications			X				X
	[99]	X	X					Constructing performance models capturing the relationship between application performance and system resources to locate possible performance bottleneck locations		X					
	[78]	X	X					Proposing an machine learning based approach that utilize unsupervised learning algorithms, i.e., af nity propagation clustering, on the monitored performance counters and system execution logs to locate anomalies on cloud-hosted web applications		X					
	[61]	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					
	[69]		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					
	[57]	X	X					Proposing a cascading clustering approach to cluster log sequences and correlating clusters with performance metrics to identify performance problems		X					
	[73]	X						Combining machine learning with graph centrality algorithms to locate the nodes spreading the performance anomalies		X					
	Our approach	X	X	X				Propose an approach combining both historical repository data and runtime information while just requires minimum knowledge about the internal behaviors of the system to effectively assist developers in identifying the root causes of performance regressions							

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

Classi er [64]) 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 load testing results from different versions, a similar or even the same testing workload is needed. However, it may be dif cult to design load test cases that represent the eld workloads. In particular, there may exist special real-world cases where the eld workload is completely different from other workloads (e.g., when a popular site (e.g., Reddit and Twitter) links to a small site, the throughput of that small site would be much greater than the average daily workload [3], a phenomenon also known as the Slashdot effect [20]). The impact of different workloads between the old and new versions of the system are not considered in these approaches; 2) in practice, load 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).

In order to avoid the need for expensive load 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 [37, 45, 58, 70, 83]. For example, Heger et al. [58] 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. However, due to the nature that these approaches are based on small-scale testing, such testing can not capture the performance of the system and be only aware of the performance of separate components. Similar to load testing-based approaches, they do not take the impact of the continuously varying workloads into consideration either.

Prior research also proposes to use system runtime information (e.g., execution logs and performance metrics) collected directly from the eld to locate performance regression root causes [24, 38, 57, 61, 65, 69, 73, 78, 99]. For example, Nair et al. [78] propose a machine learning-based approach that utilizes unsupervised learning algorithms, i.e., af nity propagation clustering, constructed on the monitored runtime performance metrics and system execution logs to locate anomalies in cloud-hosted web applications. Lu et al. [69] 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 speci c cause related factors to determine the probability of the root causes. Such eld data-based approaches bene t from saving the effort and resources for load testing, and it also considers the system level performance under the impact of varying workloads. However, some approaches may suffer from the limitations that the located root causes are too coarse-grained (e.g., only at service level) [24, 57, 61, 69, 78, 99], which can only assist IT operators (instead of developers) to locate the high-level service or modules with performance problems but may not be suf cient to support developers to locate the speci c root causes (e.g., at class or method level), or only target speci c performance regression patterns [24].

Different from prior approaches, in this paper, we propose an approach that combines both historical repository data (e.g., code change history) and eld runtime information (e.g., web-access logs and performance metrics) while just requiring minimum knowledge about the internal behaviors of the system to effectively assist developers in identifying and xing the root causes of performance regressions. Our approach makes the use of statistical and machine learning models to effectively capture the relationship between the workloads of the system and the system performance, thereby having the capability to handle the continuously varying workloads in the eld scenario.

3.2 Software fault localization

A great amount of prior research has been proposed to locate faults in software systems. The traditional practice of fault localization often adopts the most intuitive system analysis techniques, e.g., logging [46], pro ling [28, 56, 85], and debugging (e.g., assertions and breakpoints) [42, 59, 84] to identify the exact locations of program faults. However, such traditional techniques require developers to have suf cient experience and expert knowledge about the system to locate the faults and they are often time and effort-consuming.

To improve the effectiveness and ef ciency, various advanced fault localization techniques have been proposed. Prior research [21, 22, 25] employ program slicing (both static and dynamic) techniques to extract the relevant parts of the source code that in uence or are in uenced by the variables at a given point, in particular, an incorrect variable value that causes the test case to fail, allowing developers to focus on a reduced search space rather than the entire program to locate faults. Program spectrum, which records the execution information (e.g., code coverage) of program entities (e.g., statements or methods), is also employed in prior studies [19, 43, 55, 102, 103] for fault localization. Such spectrum-based approaches rank program entities according to a suspiciousness score which indicates their risk of being faulty. This score is calculated by various ranking formulae based on the program spectrum information collected from passing and failing test cases. In addition, prior works [18, 27, 74, 75, 92, 98] also propose to leverage model-based diagnosis techniques to locate program faults. Particularly, they utilize statistical analysis or source code analysis to build various types of models (e.g., dependency models [27, 98]) to represent the program structures and behaviors. If the test case fails, i.e., con icting the expected output, the model will help to determine the statements whose incorrectness can explain incorrect outcomes. Prior research [35, 36, 79, 95, 96, 104] also advocates the use of data mining and machine learning techniques by learning a model or deriving patterns from a huge volume of software data to locate program faults. For example, Wong et al. [95, 96] propose approaches based on neural networks to capture the relationship between the coverage data of each test case and the corresponding execution result, then generate the suspiciousness of each statement containing the

3. https://web.archive.org/web/20141101224936/http://blogs.abc.net.au/newseditors/2012/08/the-reddit-effect.html

Fig. 1. An overall process of developing our approach in an industrial setting

bug.

Compared to our work, the preceding approaches lack the consideration of non-functional faults, e.g., performance regressions, which are crucial in industrial applications, especially for large-scale systems with a large user base. Our approach (especially step 5) employs static code analysis, a commonly used technique for fault localization [27, 63, 92, 98] to locate the code changes related to a web request that causes the performance regression. However, in the adoption of our approach, practitioners can also opt to use other preferred techniques to associate the web request provided by our approach to the code changes that potentially result in performance regressions.

4 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 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 5 and the lessons that we learned from addressing these challenges are summarized in Section 8.

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 [31] finds that few practitioners construct performance models (e.g., queuing networks [68]) 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 [44, 50]. Moreover, the workloads of a large-scale system are impacted by thousands or millions of users interacting with the system [39]. 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 the black-box models [44]. 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 [44, 50]. 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 logs 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.

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 the specific order. One of the typical examples is that once users login to a mail system, they often check their inbox. According to prior studies [14, 101], 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 the 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 5), we also consider the system activities that have a high correlation with the detected activity as potential causes of the performance regression.

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 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 (cf. Section 3), 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 the 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 on 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 of 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 ESs 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.

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 a 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.

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 the field operations. Second, as many developers may lack knowledge about our used statistical techniques, they may not completely understand and trust our approach for 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 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 the acceptable performance overhead while ensuring a considerable 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 [11] platform such

Fig. 2. An overview of our approach of locating performance regression root causes

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 con dently and ef ciently.

5 Approach

In this section, we brie y discuss the detailed process and the implementation of our approach for locating performance regression root causes. Figure 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.

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 rst need to align the web-access logs generated by the web servers (e.g., IIS) and the collected performance metrics (e.g., CPU usage). Speci cally, 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 [44], similar to prior research [26, 50, 99, 101], 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 [26, 101], we also notice that different log events may always appear simultaneously in a speci c order, e.g., user logging in and then checking 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 coef cient [30] is higher than 0:7. We also conducted a redundancy analysis [54] to remove any independent variables that can be modeled by the rest of the independent variables with a high model t ($R^2 > 0{:}9$). Afterwards, we opt to use the random forest regression model to build our performance models, due to its high accuracy shown in prior research [26, 50, 101].

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 t 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 rst

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. Speci cally, we train two performance models using the old version data and the new version data, respectively, and apply the two models on 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 [34, 67]. For each time period in the new version, we remove its data from the training data to re-build the model and apply the re-built model on 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 [41] 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 [49] 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 explainableas 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 signi cant independent variables (p-value 6 0.05), which are potential root causes of the performance regression.

On the other hand, the linear regression model may have no statistically signi cant independent variables or a poor model t (very low R^2 cf. Section 6 and Section 7). 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 rst rank the web requests that are potential performance regression root causes, such that practitioners can prioritize their effort on the most likely root causes. Speci cally, we calculate the effect of each statistically signi cant 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 [76, 87, 88]. 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 nd the root cause. However, in our approach, we step further and link the web requests to the speci c 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” [15] and Eclipse Java development tools called JDT [10], to parse the source code and build a call graph of the web request. We seek for 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 as the potential causes of the performance regression and they are provided to developers. We would like to note that since we build the call graph of the web requests that are related to the performance regression from the new version of the system

TABLE 2
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 [7].

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.

6 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[4]. We present our evaluation on the industry system in Section 7.

6.1 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 [47, 101]. An overview of the subject systems is shown in Table 2.

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 [93]. 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 [47] 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 the version 1.3.4 of TeaStore in our case study. TeaStore has a few quintessential use cases, including logging in system, browsing the store, browsing user's profile, browsing products, shopping products, and logging out the system.

OpenMRS is an open-source health care 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 [16]. 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 [9]. It was developed as a showcase application to validate the European Union funded project which is called CloudScale [8]. It follows the functional requirements defined by the TPC-W standard which is a web e-Commerce benchmark for transaction processing [4]. 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 with 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).

6.2 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

4. Our experiment setup, workloads, and results are shared online https://doi.org/10.5281/zenodo.5659008 as a replication package.

positions of each subject system. Inspired by the previous work [50, 53, 86], 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 calculation 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) are 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 super uous 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 speci c 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 in uenced 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.

6.3 Experiment setup

The experiments on the open-source subject systems are conducted in the Google Cloud Platform [12], where three separate virtual machines are set up for each subject system. These virtual machines have the same hardware con guration, which includes an Intel Haswell 4 cores CPU, an 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 [6] on the rst machine (i.e., the web server). The second machine is deployed as a database server. Finally, we run the JMeter [5] 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 [17] 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 regressions 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 [51]. Performance regression in terms of CPU would result in the need for more CPU resources to provide the same quality of service, thereby signi cantly 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.

6.4 Evaluation results of open-source systems

For each of the studied open-source systems, we rst 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 ve 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 ve different concurrent workloads. Ideally, our approach should not detect and locate root causes of performance regressions from this version. Table 3 and Table 4 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) R^2 is the model t 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 quanti es the relationship between the appearances of one type of web request and the deviance of performance modeling error (cf. Step 5 in Section 5). 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 2) to locate the injected performance regressions and present it in Table 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 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 3, 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 3) and the highest effect from the web requests without injected performance regression (i.e., "Highest effect without regressions" in Table 3).

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 [94] 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 [33] is used together with the Wilcoxon rank-sum test [94] 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) 0.017. From the statistical test results between different open-source software systems presented in Table 6, 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 6, 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.

On the other hand, from Table 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 (6 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 where the arbitrarily injected synthetic performance regressions locate 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.

7 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 paper 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 our 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.

TABLE 3
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
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 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

TABLE 6
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).

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 eld 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 identi ed 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 con rm whether each located performance regression root cause indeed results in the performance regressions or not. After con rming the performance regression root causes, developers would also inform us of the performance regressions that are not identi ed by our approach, if there exist any. In particular, we rst 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 5).

From the meetings, we identi ed 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 con rmed 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 con rmed by developers. Furthermore, we note that there are no performance regressions that are known to developers but not detected by our approach. Speci cally, 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 x 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 paper. 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 7 summarizes the results of applying both baseline approach and our approach 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 identi- ed 4 releases containing performance regressions and there are 6 performance regression root causes located in these releases. After meeting with industry partners, we con rm that 3 of them are true positives, which shows that our approach achieves the precision at 50%. Our approach can also successfully locate the root causes of these performance regressions and provide corresponding code changes. It is worth point 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., signi cant variables in the regression model (cf. Section 5)), 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 nding, 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 signi cantly 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.

TABLE 7
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 con rmed performance regressions	3	3
# located root causes	65	6
# con rmed root causes	2	3
Precision	3.08%	50%

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

For the releases with the con rmed performance regressions, we conduct further investigation to reveal how the performance regressions are introduced. The rst 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 turns 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 ampli es the performance penalty of the inner loop, which makes the performance regression even more severe. This regression was not identi ed 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 con rmed this performance regression to the software.

The third performance regression located by our approach is caused by an inef cient SQL query that is business logic-related within a SQL stored procedure. After discussion and con rmation 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 nd that model t (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 con rmed performance regressions. Speci cally, the R^2 of the versions with the con rmed 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 con rmed root cause has an averageeffect of 0.05, while the effect from the other releases have the maximum of only 0.038 with an average value of 0.02. Such results also con rm with the evaluation of our approach on the open-source subject systems (cf. Section 6) that we can adopt the model t (R^2) and the effect values as indicators to effectively identify the performance regression root causes.

8 Lessons Learned

In this section, we summarize the lessons that we learned from addressing the challenges during 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 eld-operation data. The corresponding faced challenges and solutions are described in Section 4.

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 eld-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 con rms 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 nd 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% forTeaStore, 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 the knowledge of the system internal behaviors, black-box models rely on logs to model system performance. Black-box models are particularly helpful when the system is under eld-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 eld-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 regressions root causes using just the eld-operation data of software systems. For example, table 8 shows the measured results when applying our approach to the TeaStoresubject 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 signi cantly 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 con rm with prior work [14, 101] that the collinearity and redundancy among the independent variables may have a negative impact on tting 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 nd 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 8
An example of the in uence 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
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 nding 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 effort. 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 the 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 le 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 dif cult to achieve this granularity since performance regressions are often introduced not with merely a single line of code [37]. After discussing with the developers of our industrial partner, we choose to locate code changes at the method level as it is suf cient for them to understand and x the performance issue.

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

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 overhead. When applying research to practice, we would suggest that it is of great importance to minimize the performance overhead of our approach while keeping suf cient 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 ef ciency of analyzing the root causes of performance regressions. In particular, we present the screenshots of the major parts of the Elastic Stack uni ed management platform in our industrial environment in Figure 3. We would like to note that the actual detected URLs and the changeset ids are anonymized with black color since they contain con dential information from our industrial collaborator. Speci cally, there are three main components in our Elastic Stack platform: 1) Visualization of CPU prediction error. As shown in Figure 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 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 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 nd 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.

9 Threats to Validity

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

External validity. In our approach, we target locating performance regression root causes for just the web-based systems for the following considerations: rst, the state of practice is an important starting point that motivates our study and the studied industrial system (i.e., ES) is a commercial

(a) Visualization of CPU prediction error

(b) Visualization of the baseline approach's result

(c) Visualization of our approach's result

Fig. 3. Screenshot of our Elastic Stack uni ed 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.

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 on other types of systems is 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 the evidence of historical performance regressions of speci c versions via checking the system version history, we manually injected four types of synthetic performance regressions that are described in Section 6. 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 bene t the evaluation of our approach.

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 the 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.

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 while 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 deployed system indeed has performance regressions and run 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 locate root causes of performance regressions introduced by the features that only exist in the new version.

10 Conclusions

In this paper, we provide an experience report on the challenges and lessons learned from designing and adopting our automated approach for locating performance regressions 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 of 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 on the use of field-operation data as a main source to conduct performance assurance activities during their fast-paced software development and releasing cycles.

Acknowledgment

We would like to thank ERA Environmental Management Solutions for providing access to the enterprise system used in our study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of ERA Environmental Management Solutions and/or its subsidiaries and affiliates. Moreover, our results do not reflect the quality of ERA Environmental Management Solutions' products.

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, Last accessed 05/6/2020.

[2] "Canary release," https://martinfowler.com/bliki/CanaryRelease.html, 2014, Last accessed 05/8/2020.

[3] "The cost of performance issues," https://focusaps.com/2019/09/26/the-costs-of-performance-issues/, 2019, Last accessed 05/8/2020.

[4] "Tpc benchmark w (tpc-w)," http://www.tpc.org/tpcw/, 2019, Last accessed 05/7/2020.

[5] "Apache jmeter - apache jmeter™," https://jmeter.apache.org/, 2020, Last accessed 03/8/2020.

[6] "Apache tomcat® - welcome!" http://tomcat.apache.org/, 2020, Last accessed 05/8/2020.

[7] "cloc - count lines of code," https://github.com/AlDanial/cloc/, 2020, Last accessed 06/7/2020.

[8] "Cloudscale project," https://www.cloudscale-project.eu/, 2020, Last accessed 05/9/2020.

[9] "Cloudscale-project/cloudstore," https://github.com/CloudScale-Project/CloudStore, 2020, Last accessed 05/9/2020.

[10] "Eclipse java development tools (jdt)," https://www.eclipse.org/jdt/, 2020, Last accessed 03/6/2020.

[11] "Elastic stack: Elasticsearch, kibana, beats & logstash — elastic," https://www.elastic.co/elastic-stack/, 2020, Last accessed 04/8/2020.

[12] "Google cloud: Cloud computing services," https://cloud.google.com/, 2020, Last accessed 05/8/2020.

[13] "Mozilla performance regressions policy," https://www.mozilla.org/en-US/about/governance/policies/regressions/, 2020, Last accessed 05/8/2020.

[14] "Multicollinearity in regression analysis: Problems, detection, and solutions," https://statisticsbyjim.com/regression/multicollinearity-in-regression-analysis/, 2020, Last accessed 03/8/2020.

[15] "The .net compiler platform sdk," https://docs.microsoft.com/en-ca/dotnet/csharp/roslyn-sdk/, 2020, Last accessed 03/8/2020.

[16] "Openmrs," https://openmrs.org/, 2020, Last accessed 05/10/2020.

[17] "pidstat(1): Report statistics for tasks - linux man page," https://linux.die.net/man/1/pidstat, 2020, Last accessed 05/6/2020.

[18] R. Abreu and A. 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.

[19] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coef cients for software fault localization," in 12th IEEE Paci c Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA IEEE Computer Society, 2006, pp. 39–46.

[20] S. Adler, "The slashdot effect: an analysis of three internet publications," Linux Gazette, vol. 38, no. 2, p. 623, 1999.

[21] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," Softw. Pract. Exp., vol. 23, no. 6, pp. 589–616, 1993.

[22] Z. A. Al-Khanjari, M. R. Woodward, H. A. Ramadhan, and N. S. Kutti, "The ef ciency of critical slicing in fault localization," Softw. Qual. J., vol. 13, no. 2, pp. 129–153, 2005.

[23] J. P. S. Alcocer, A. Bergel, and M. T. 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, A. Avritzer, A. Iosup, X. Zhu, and S. Becker, Eds. ACM, 2016, pp. 37–48.

[24] E. R. Altman, M. Arnold, S. Fink, and N. 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, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, 2010, pp. 739–753.

[25] E. Alves, M. Gligoric, V. Jagannath, and M. 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 IEEE Computer Society, 2011, pp. 520–523.

[26] M. Arif, W. Shang, and E. Shihab, "An empirical study on the discrepancy between performance testing results from virtual and physical environments," Empirical Software Engineering, p. To Appear, 2017.

[27] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," IEEE Trans. Software Eng., vol. 36, no. 4, pp. 528–545, 2010.

[28] T. Ball and J. R. Larus, "Optimally pro ling and tracing programs," ACM Trans. Program. Lang. Syst., vol. 16, no. 4, pp. 1319–1360, 1994.

[29] C. Barna, M. Litoiu, and H. Ghanbari, "Autonomic load-testing framework," in Proceedings of the 8th ACM International Conference on Autonomic Computing, ICAC 2011, p. 91–100.

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

[31] C. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. 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 ACM, 2019, pp. 45–50.

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

[33] C. Bonferroni, "Teoria statistica delle classi e calcolo delle probabilita," Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze, vol. 8, pp. 3–62, 1936.

[34] L. Breiman and P. Spector, "Submodel selection and evaluation in regression. the x-random case," International statistical review/revue internationale de Statistique, pp. 291–319, 1992.

[35] P. Cellier, M. Ducassé, S. Ferré, and O. 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, ser. Lecture Notes in Computer Science, vol. 4933. Springer, 2008, pp. 273–288.

[36] ——, "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 Knowledge Systems Institute Graduate School, 2011, pp. 238–243.

[37] J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017 IEEE Computer Society, 2017, pp. 341–352.

[38] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," International Conference on Autonomic Computing, 2004. Proceedings., pp. 36–43, 2004.

[39] 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), 2017, pp. 243–252.

[40] N. Cliff, Ordinal methods for behavioral data analysis Psychology Press, 2014.

[41] J. Cohen, Statistical Power Analysis for the Behavioral Sciences Routledge, 1988.

[42] D. S. Coutant, S. Meloy, and M. 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, 1988, pp. 125–134.

[43] V. Debroy, W. E. Wong, X. Xu, and B. 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 IEEE Computer Society, 2010, pp. 13–22.

[44] D. Didona, F. Quaglia, P. Romano, and E. 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, ICPE 2015, pp. 145–156.

[45] Z. Ding, J. Chen, and W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet?" in ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1435–1446.

[46] J. C. Edwards, "Method, system, and program for logging statements to monitor execution of a program," Mar. 25 2003, uS Patent 6,539,501.

[47] S. Eismann, C. Bezemer, W. Shang, D. Okanovic, and A. 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, 2020, pp. 138–149.

[48] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. 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, 2015, pp. 159–168.

[49] D. Freedman, Statistical Models : Theory and Practice. Cambridge University Press, August 2005.

[50] R. Gao, Z. M. Jiang, C. Barna, and M. 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, pp. 22–32.

[51] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," SIGCOMM Comput. Commun. Rev., vol. 39, no. 1, p. 68–73, 2009.

[52] B. Gregg, Systems performance: enterprise and the cloud. Pearson Education, 2014.

[53] H. W. Gunther, "WebSphere Application Server Performance and Scalability," IBM WebSphere Application Server Standard and Advanced Editions - White paper, 2000.

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

[55] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," Softw. Test. Verification Reliab., vol. 10, no. 3, pp. 171–194, 2000.

[56] M. Hauswirth and T. 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. ACM, 2004, pp. 156–164.

[57] S. He, Q. Lin, J. Lou, H. Zhang, M. R. Lyu, and D. 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, 2018, pp. 60–70.

[58] C. Heger, J. Happe, and R. 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, S. Seelam, P. Tuma, G. Casale, T. Field, and J. N. Amaral, Eds. ACM, 2013, pp. 27–38.

[59] J. Hennessy, "Symbolic debugging of optimized code," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 4, no. 3, pp. 323–344, 1982.

[60] J. Humble and G. Kim, Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations. IT Revolution, 2018.

[61] H. Jayathilaka, C. Krintz, and R. 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, R. Barrett, R. Cummings, E. Agichtein, and E. Gabrilovich, Eds. ACM, 2017, pp. 469–478.

[62] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," IEEE Trans. Software Eng., vol. 41, no. 11, pp. 1091–1118, 2015.

[63] B. Johansson, A. V. Papadopoulos, and T. 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. IEEE, 2019, pp. 7–12.

[64] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," CoRR, vol. abs/1302.4964, 2013.

[65] 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), 2015, pp. 1624–1627.

[66] G. Kim, J. Humble, P. Debois, and J. Willis, The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations. IT Revolution, 2016.

[67] E. Kocaguneli and T. Menzies, "Software effort models should be assessed via leave-one-out validation," J. Syst. Softw., vol. 86, no. 7, pp. 1879–1890, 2013.

[68] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc., 1984.

[69] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. 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. IEEE, 2017, pp. 389–396.

[70] Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software," in Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 25–36.

[71] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in Proceedings of the 2013 International Conference on Software Engineering, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1012–1021.

[72] D. Maplesden, K. von Randow, E. D. Tempero, J. G. Hosking, and J. 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, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 149–158.

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

[74] C. Mateis, M. Stumptner, and F. Wotawa, "Modeling java programs for diagnosis," in ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000. IOS Press, 2000, pp. 171–175.

[75] W. Mayer and M. 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, ser. Lecture Notes in Computer Science, vol. 2557. Springer, 2002, pp. 107–118.

[76] A. Mockus, "Organizational volatility and its effects on software defects," in Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering 2010, p. 117–126.

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

[78] V. Nair, A. Raul, S. Khanduja, V. Bahirwani, S. Sellamanickam, S. S. Keerthi, S. Herbert, and S. 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. ACM, 2015, pp. 2029–2038.

[79] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. 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*, ser. Lecture Notes in Computer Science, vol. 5258. Springer, 2008, pp. 548–559.

[80] 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*, 2011, pp. 282–289.

[81] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "An industrial case study of automatically identifying performance regression-causes," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, p. 232–241.

[82] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, 2012, pp. 299–310.

[83] D. G. Reichelt, S. Kühne, and W. 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*. IEEE, 2019, pp. 1146–1149.

[84] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.

[85] C. Runciman and D. Wakeling, "Heap profiling of lazy functional programs," *J. Funct. Program.*, vol. 3, no. 2, pp. 217–245, 1993.

[86] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, p. 15–26.

[87] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empir. Softw. Eng.*, vol. 20, no. 1, pp. 1–27, 2015.

[88] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. 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*, 2011, p. 300–310.

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

[90] P. Stefan, V. Horký, L. Bulej, and P. Tuma, "Unit testing performance in java projects: Are we there yet?" in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, W. Binder, V. Cortellessa, A. Koziolek, E. Smirni, and M. Poess, Eds. ACM, 2017, pp. 401–412.

[91] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. N. Nasser, and P. 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*. IEEE Computer Society, 2013, pp. 110–119.

[92] H. Thaller, L. Linsbauer, A. Egyed, and S. 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*. IEEE, 2020, pp. 24–27.

[93] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. 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.

[94] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[95] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. M. Thuraisingham, "Effective software fault localization using an RBF neural network," *IEEE Trans. Reliab.*, vol. 61, no. 1, pp. 149–169, 2012.

[96] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 4, pp. 573–597, 2009.

[97] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 171–187.

[98] F. Wotawa, M. Stumptner, and W. Mayer, "Model-based debugging or how to diagnose programs automatically," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, 2002, pp. 746–757.

[99] P. Xiong, C. Pu, X. Zhu, and R. Griffith, "Vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, 2013, p. 271–282.

[100] Y. Xu, N. Chen, A. Fernandez, O. Sinno, and A. 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*, 2015, p. 2227–2236.

[101] K. Yao, G. B. de Pádua, W. Shang, C. Sporea, A. Toma, and S. Sajedi, "Log4perf: suggesting and updating logging locations for web-based systems' performance monitoring," *Empirical Software Engineering*, vol. 25, no. 1, pp. 488–531, 2020.

[102] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 244–258.

[103] K. Yu, M. Lin, Q. Gao, H. Zhang, and X. 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*. ACM, 2011, pp. 1404–1410.

[104] Z. Yu, H. Hu, C. Bai, K. Cai, and W. E. 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*. IEEE Computer Society, 2011, pp. 325–332.

Lizhi Liao is a Ph.D. student in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada, supervised by Weiyi Shang. He has received his M.Eng. degree from Concordia University and he obtained B.Eng. from Chongqing University of Posts and Telecommunications. His research interests contain software performance engineering, software log mining and mining software repositories. Contact him at l_lizhi@encs.concordia.ca.