Noname manuscript No.
(will be inserted by the editor)

Using Black-Box Performance Models to Detect Performance Regressions under Varying Workloads: An Empirical Study

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

Received: date / Accepted: date

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

Lizhi Liao, Jinfu Chen, Yi Zeng, Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
E-mail: {l_lizhi, fu_chen, ze_yi, shang}@encs.concordia.ca

Heng Li
Département de génie informatique et génie logiciel
Polytechnique Montréal
Montreal, Quebec, Canada
E-mail: heng.li@polymtl.ca

Jianmei Guo
Alibaba Group
Hangzhou, Zhejiang, China
E-mail: jianmei.gjm@alibaba-inc.com

Catalin Sporea, Andrei Toma,Sarah Sajedi
ERA Environmental Management Solutions
Montreal, Quebec, Canada
E-mail: {steve.sporea, andrei.toma, sarah.sajedi}@era-ehs.com

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

Keywords Performance regression · Black-box performance models · Field workloads · Performance engineering

1 Introduction

Many large-scale software systems (e.g., Amazon, Google, Facebook) provide services to millions or even billions of users every day. Performance regressions in such systems usually lead to both reputational and monetary losses. For example, a recent incident of performance regressions at Salesforce[1] affected eight data centers, resulting in a negative impact (i.e., slowdown and failures of the provided services) on the experience of clients across many organizations (Cannon, 2019). It is reported that 59% of Fortune 500 companies experience at least 1.6 hours of downtime per week (Syncsort, 2018), while most of the field failures are due to performance issues rather than feature issues (Weyuker and Vokolos, 2000). To avoid such performance-related problems, practitioners perform in-house performance testing to detect performance regressions of large-scale software systems (Jiang and Hassan, 2015).

For in-house performance testing, load generators (e.g., JMeter[2]) are used to mimic thousands or millions of users (i.e., the *workload*) accessing the system under test (SUT) at the same time. Performance data (e.g., CPU usage) is recorded during the performance testing for later performance analysis. To determine whether a performance regression exists, the performance data of the current software version is compared to the performance data from running an earlier version on the same workload, using various analysis techniques (Gao *et al.*, 2016) (e.g., control charts (Nguyen *et al.*, 2012)). However, such in-house performance testing suffers from two important challenges. First, the real workloads in the field are constantly changing and difficult to predict (Syer *et al.*, 2014), making it unrealistic to test such dynamic workloads in a predefined manner. Second, performance testing is usually very expensive as it requires

[1] https://www.salesforce.com
[2] https://jmeter.apache.org

extensive resources (e.g., computing power) and lasts for a long time (e.g., hours or even weeks). For software systems that follow fast release cycles (e.g., Facebook is released every few hours), conducting performance testing in a short period of time becomes particularly difficult.

There exist software testing techniques such as A/B testing (Xu *et al.*, 2015) and canary releasing (Sato, 2014) that depend on end user's data in the field for software quality assurance. Similarly, in this work, we propose to directly detecting performance regressions based on the end users' data of large-scale software systems when they are deployed in the field[3]. Such an automated detection can complement or even replace typical in-house performance testing when testing resources are limited (e.g., in an agile environment) and/or the field workloads are challenging to be reproduced in the testing environment. In particular, we leverage sparsely-sampled performance metrics and readily-available execution logs from the field to detect performance regressions, which only adds negligible performance overhead to SUT.

Inspired by prior research (Farshchi *et al.*, 2015; Yao *et al.*, 2018), we use black-box machine learning and deep learning techniques (i.e., CNN, RNN, LSTM, Linear Regression, Random Forest, and XGBoost) to capture the relationship between the runtime activities that are recorded in the readily-available logs of a software system (i.e., the independent variables) and its performance under such activities (i.e., the dependent variables). We use the black-box model that describes the current version of a software system and the black-box model that describes an earlier version of the same software system, to determine the existence of performance regressions between these two versions.

Prior research that builds black-box models to detect performance regressions (Farshchi *et al.*, 2015; Yao *et al.*, 2018) typically depends on the data from in-house performance testing, where the workload is consistent between two releases. However, when using the field data from the end users, one may not assume that the workloads from the two releases are the same. In order to study whether the use of black-box performance models under such inconsistent workloads may still successfully detect performance regressions, we performed empirical experiments on two open-source systems (i.e., OpenMRS and Apache James) and applied our approaches on a large-scale industrial system. In particular, our study aims to answer two research questions (RQs):

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

[3] Note that our approach serves different purposes and has different usage scenarios from A/B testing and canary releasing, as discussed in Section 9.

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

Our work makes the following key contributions:

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

The remainder of the paper is organized as follows. Section 2 introduces the background of black-box performance modeling. Section 3 outlines our approaches for detecting performance regressions in the field. Section 4 and Section 5 present the setup and the results of our case study on two open-source subject systems, respectively. Section 6 shares our experience of applying our approaches on a large-scale industrial system. The challenges and the lessons that we learned from the successful adoption of our approaches are discussed in Section 7. Section 8 and Section 9 discuss the threats to the validity of our findings and the related work, respectively. Finally, Section 10 concludes the paper.

2 Background: Black-box performance models

Performance models are built to model system operations in terms of resources consumed. Performance models can be used to predict the performance of systems in the purpose of workload design (Krishnamurthy *et al.*, 2006; Syer *et al.*, 2017; Yadwadkar *et al.*, 2010), resources control (Cortez *et al.*, 2017; Gong *et al.*, 2010), configuration selection (Guo *et al.*, 2013, 2018; Valov *et al.*, 2017), and anomaly detection (Farshchi *et al.*, 2015; Ghaith *et al.*, 2016; Ibidunmoye *et al.*, 2015). Performance models can be divided into two categories (Didona *et al.*, 2015). One is white box performance models, which is also called analytical models. White box performance models predict performance based on the assumption of system contexts, i.e., workload intensity and service demand.

Another type of performance model is black-box performance models. Black-box performance models employ statistical modeling or machine learning techniques to predict system performance by taking the system as a black box. Black-box performance models typically require no knowledge about the system's internal behavior (Didona *et al.*, 2015; Gao *et al.*, 2016). Such models apply various machine learning algorithms to model a system's performance behavior taking system execution logs or performance metrics as input. The main strength of using black-box modeling techniques is to model the system performance when the system is under field operations. While white-box performance models typically require knowledge about the system's internal behaviour, such knowledge is often not available when the system is deployed in the production environment. Thus, under the production environment, black-box approaches appear to be more effective than white-box modeling approaches.

Typical examples of black-box performance models are linear regression models (Shang *et al.*, 2015; Yao *et al.*, 2018). For example, prior studies use linear regression models to capture the relationship between a target performance metric (e.g., CPU usage) and the system operations (Yao *et al.*, 2018) or other performance metrics (e.g., memory utilization) (Shang *et al.*, 2015):

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n \tag{1}$$

where the response variable y is the target performance metric, while the explanatory variables $(x_1, x_2, ..., x_n)$ are the frequency of each operation of the system (Yao *et al.*, 2018) or each of the other performance metrics (Shang *et al.*, 2015). The coefficients $\beta_1, \beta_2, ..., \beta_n$ describe the relationship between the target performance metric and the corresponding explanatory variables.

Linear regression requires that the input data satisfies a normal distribution. With the benefit of being applicable to data from any distribution, regression tree (Xiong *et al.*, 2013) has been used to model system performance. Regression tree models a system's performance behavior by using a tree-like structure. Quantile regression (de Oliveira *et al.*, 2013) can model a system's different phases of performance behavior.

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

3 Approaches

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

3.1 Extracting metrics

We aim to identify whether there is performance regression in the new version of the system based on modeling the relationship between the system performance (e.g., CPU usage) and the corresponding logs that are generated during system execution. Specifically, we collect system access logs that are readily generated during the execution of web servers, such as the Jetty, Tomcat, and IIS (Internet Information Services). Those logs represent the workload of the system during a period of execution. The performance data is sparsely sampled at a fixed frequency (e.g., every 30 seconds).

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

3.1.1 Splitting logs and performance data into time periods

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

3.1.2 Extracting log metrics and performance metrics

We parse the collected logs into events and their corresponding time stamps. For example, a line of web log "*[2019-09-27 22:43:13] GET /openmrs/ws/rest/v1/person/HTTP/1.1 200*" will be parsed into the corresponding web request or URL "*GET /openmrs/ws/rest/v1/person/*" and time stamp "*2019-09-27 22:43:13*". Afterwards, each value of a log metric is the number of times that each log event executes during the period. For example, if the log event "*GET /openmrs/ws/rest/v1/person/*" is executed 10 times during a 30-second time period, the corresponding log metric for "*GET /openmrs/ws/rest/v1/person/*"'s value is 10 for that period. Then, we consider the aggregation (e.g., taking the average) of the records of performance data as the value of the performance metric of the time period, similar to prior research (Foo *et al.*, 2010). Table 1 shows an illustrative example of log metrics and performance metrics in five 30-second time periods.

These log metrics together with the performance metrics in the corresponding time periods are used to construct our performance models. Each data point of a log metric is based on a time period (e.g., 30 seconds) instead of a log entry. For example, if we run the subject system for eight hours and we take each 30 seconds as a time period, there will be 960 data instances in total: 8 (hours) * 60 (minutes per hour) * 2 (time periods per minute).

Table 1 An illustrative example of log metrics and performance metric data from OpenMRS

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

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

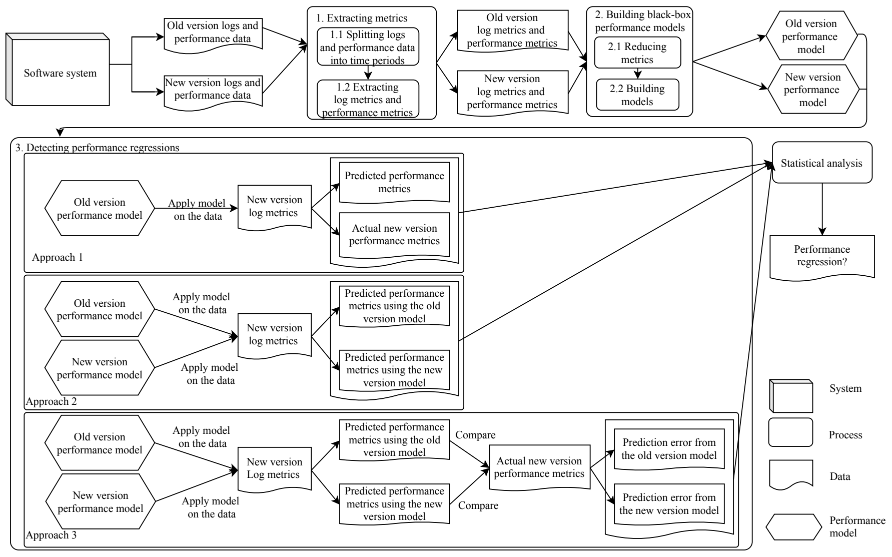

Software system
Old version logs and performance data
New version logs and performance data
1. Extracting metrics
1.1 Splitting logs and performance data into time periods
1.2 Extracting log metrics and performance metrics
Old version log metrics and performance metrics
New version log metrics and performance metrics
2. Building black-box performance models
2.1 Reducing metrics
2.2 Building models
Old version performance model
New version performance model
3. Detecting performance regressions
Old version performance model
Apply model on the data
New version log metrics
Predicted performance metrics
Actual new version performance metrics
Approach 1
Old version performance model
New version performance model
Apply model on the data
Apply model on the data
New version log metrics
Predicted performance metrics using the old version model
Predicted performance metrics using the new version model
Approach 2
Old version performance model
New version performance model
Apply model on the data
Apply model on the data
New version Log metrics
Predicted performance metrics using the old version model
Predicted performance metrics using the new version model
Compare
Compare
Actual new version performance metrics
Prediction error from the old version model
Prediction error from the new version model
Approach 3
Statistical analysis
Performance regression?
System
Process
Data
Performance model

Fig. 1 An overview of our studied approaches to detecting performance regressions.

3.2 Building black-box performance models

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

3.2.1 Reducing metrics

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

Different log events may always appear at the same time, e.g., user logging in and checking user's privilege, and provide repetitive information for the workloads. To avoid bias from such repetitive information, we then perform correlation analyses and redundancy analyses on the log metrics. We calculate Pearson's correlation coefficient (Benesty *et al.*, 2009) between each pair of log metrics (i.e., in total N*N/2 times of correlation calculations where N refers to the number of log metrics). If a pair of log metrics have a correlation higher than 0.7, we remove the one that has a higher average correlation with all other metrics. We repeat the process until there exists no correlation higher than 0.7. The redundancy analysis would consider a log metric redundant if it can be predicted from a combination of other log metrics. We use each log metric as a dependent variable and use the rest of the log metrics as independent variables to build a regression model. We calculate the R^2 of each model. If the R^2 is larger than a threshold (e.g., 0.9), the current dependent variable (i.e., the log metric) is considered redundant. We then remove the log metric with the highest R^2 and repeat the process until no log metrics can be predicted with an R^2 higher than the threshold.

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

3.2.2 Building models

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

3.3 Detecting performance regressions

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

Intuitively, one may use the model that is built from the old version of the system to predict the performance metrics from running the new version of the system. By measuring the prediction error, one may be able to determine whether there exists performance deviance (Cohen *et al.*, 2004; Nguyen *et al.*, 2012). However, such a naive approach may be biased by the choice of thresholds that are used to determine whether there is performance deviance. For example, a well-built performance model may only have less than 5% average prediction error; while another less fit performance model may have 15% average prediction error. In these cases, it is challenging to determine whether an average prediction error of 10% on the new version of the system should be considered as a performance regression. Therefore, statistical analyses are used to detect performance regressions in a systematic manner (Foo *et al.*, 2015; Gao *et al.*, 2016; Shang *et al.*, 2015).

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

Approach 1: *comparing the predicted performance metrics (using the model built form the old version) and the actual performance metrics of the new version.* The most intuitive way of detecting performance regression is to compare the predicted value and the actual value of performance metrics. Since the model is built from the old version of the system, if the actual value of the performance metrics from the new version of the system is higher than the predicted value and the difference between them is large, we may consider the existence of performance regressions. In particular, we use the data from the old version of the system, i.e., $Data_{old}$ to build a black-box performance model $Model_{old}$ (cf. Section 3.2). Afterwards, we apply the model $Model_{old}$ on the data from the new version of the system, i.e., $Data_{new}$. Then we compare the predicted and the actual values of the performance metrics.

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

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

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

Approach 3: *comparing the prediction errors on the new version using the model built from the old version and the model built from the new version.* The final approach of detecting performance regression is similar to the previous one (Approach 2), where instead of directly comparing the predicted performance metrics, we compare the distribution of the prediction errors by applying $Model_{old}$ on $Data_{new}$, and the predicted errors by applying each $Model^n_{new}$ on each of the n data points in $Data_{new}$. The intuition is that, when $Model_{old}$ has a larger prediction error than $Model_{new}$, it may be an indication of performance deviance between the two versions. We note that this approach would only be able to determine whether there exists performance deviance, which may actually be improvement instead of regression.

Statistical analysis. All three approaches generate two distributions of either actual/predicted performance metric values, or prediction errors. we compare the two distributions to determine if there are performance regressions between the two software versions. However, the differences between the data distributions may be due to experimental noises rather than the systematic difference between the two versions. Therefore, similar to previous studies (Chen *et al.*, 2016), we use statistical methods (i.e., statistical tests and effect sizes) to compare the data distributions of the two versions while taking into consideration the experimental noises.

In particular, we use the Mann-Whitney U test since it is non-parametric and it does not assume a normal distribution of the compared data. we run the test at the 5% level of significance, i.e., if the p-value of the test is not greater than 0.05, we would reject the *null hypothesis* in favor of the alternative hypothesis, i.e., there exists a statistically significant difference between the performance of old version and new version systems. In order to study the magnitude of the performance deviation without being biased by the size of the data, we further adopt the effect sizes as a complement of the statistical significance test. Considering the non-normality of our data, we utilize *Cliff's Delta* (Cliff, 1996) and thresholds provided in prior research (Romano *et al.*, 2006). For approach 1 and 2, we can use the direction (i.e., positive or negative) of the effect size to further tell whether the deviation indicates a performance regression or improvement.

4 Case Study Setup

To study the effectiveness of our approaches for detecting performance regressions under varying workloads, we perform case studies on two open-source systems[4]. In this section, we first present the subject systems. Then, we present the workloads applied to the systems, the experimental environment and performance issues. Finally, we present the choices of machine learning models that are used in our approaches.

4.1 Subject systems

We evaluate our approaches on two open-source systems, namely OpenMRS and Apache James. We also discuss the successful application of our approach in System X, a large-scale industrial system that provides government-regulation related reporting services (in Section 6). Apache James is a Java-based mail server. OpenMRS is an open-source health care system that supports customized medical records and it is wildly used in developing countries. The two open-source subject systems and the industrial system are all studied in prior research (Gao *et al.*, 2016; Yao *et al.*, 2018) and cover different domains, which ensures that our findings are not limited to a specific domain. The details of the two open-source subject systems and the industrial system are shown in Table 2.

Both two open-source subject systems and the industrial system are CPU-intensive. Besides, CPU is usually the main contributor to server costs (Greenberg *et al.*, 2008). Performance regression in terms of CPU would result in the need for more CPU resources to provide the same quality of service, thereby significantly increasing the cost of system operations. Therefore, in this study, we use CPU as the performance metric for detecting performance regressions.

[4] Our experimental setup, workloads, and results are shared online `https://github.com/senseconcordia/EMSE2020Data` as a replication package.

Table 2 Overview of the open-source subject systems and the industrial system.

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

* We share our experience of applying our approaches on System X in Section 6.

Table 3 Summary of the test actions with increased appearances (the ones with ●) in different load drivers.

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

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

4.2 Subject workload design

4.2.1 OpenMRS

OpenMRS provides a web-based interface and RESTFul services. We used the default OpenMRS demo database in our performance tests. The demo database contains data for over 5K patients and 476K observations. OpenMRS contains four typical scenarios: adding, deleting, searching, and editing operations. We designed different performance tests that are composed of eight various test actions, including 1) creation of patients, 2) deletion of patients, 3) searching for patients, 4) editing patients, 5) searching for concepts, 6) searching for encounters, 7) searching for observations, and 8) searching for types of encounters. Those performance tests are different in the ratio between the actions. In order to simulate a more realistic workload in the field, we added random controllers and random order controllers in JMeter to vary the workload. Moreover, we also simulated the variety of the number of users and activities in the field by setting random gaps between the repetitions of each user's activities, randomizing the order of the user activities, and setting a different number of maximum concurrent users for different workloads at different times.

Furthermore, we designed a total of five JMeter-based performance tests, each of which consists of a mixture of random workloads. In particular, to drive different workloads, for each of the load drivers, we pick several different test actions and put them in an extra JMeter loop controller that iterates a random number of times to increase their appearances in the workload. Table 3 shows the detailed choice of the looped actions of each load driver. To make sure that the two versions of the subject systems have different workloads, for one version of the system (e.g., the old version), we run four JMeter tests together to exercise the system. For the other version (e.g., the new version) of the system, we run one additional JMeter test, i.e, a total of five JMeter tests, to ensure that the two versions have undergone different workloads.

Despite the variation in the workloads, we keep our systems not saturated and running in normal conditions. The CPU saturation can be qualified as the average CPU load increases to a very large value (e.g., 90%) and remains for a long period of time (Krasic *et al.*, 2007). When the system is saturated, the services of the systems are not provided normally and the relationship between the performance of a software system (e.g., CPU usage) and the runtime activities may change. Besides, real software systems are usually not running under saturated conditions. The details of the workload, such as the CPU usage values, are shared in our replication package.

For the OpenMRS system, we do not have any evidence showing performance regressions of specific versions. Therefore, we manually inject several performance regressions in the system. The injected performance regressions are shown in Table 4. The original version, i.e., v0 of OpenMRS does not have any injected or known performance regressions. We separately injected four performance regressions (heavier DB request, additional I/O, constant delay and additional calculation) on the basis of version v0. These four versions are called v1, v2, v3, and v4, respectively. The details of the injected performance regressions of OpenMRS are explained below:

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

We deployed the OpenMRS on two machines, each with a configuration of Intel Core i5-2400 CPU (3.10GHz), 8 GB memory, and 512GB SATA hard drive. One machine is deployed as the application server and the other machine is deployed as the MySQL database server. We ran JMeter using the RESTFul API of OpenMRS on five extra machines with the same specification to simulate user workloads on the client side for five hours. Each machine hosts one JMeter instance with one type of workload. We used *Pidstat* (pid, 2019) to monitor the CPU usage that is used as the performance metrics of this study. To minimize the noise from the system warm-up and cool-down periods, we filtered out the data from the first and last half hour of running each workload. Thus, we only kept four hours of data from each performance test.

4.2.2 Apache James

Apache James is an open-source enterprise mail server. It contains two main actions: sending and retrieving mails. Those two actions can be further divided into many smaller actions, e.g., sending mails with or without attachments and retrieving an entire mail or only the header of the mail. In total, we built eight actions for the performance test, including 1) sending mails with short messages and without attachments, 2) sending mails with long messages and without attachments, 3) sending mails with short messages and with small attachments, 4) sending mails with short messages and with large attachments, 5) sending mails with long messages and with small attachments, 6) sending mails with long messages and with large attachments, 7) retrieving entire mails, and 8) retrieving only the header of mails. Similar to OpenMRS, we also created a total of five different workloads, where each workload consists of a mixture of eight test actions with different ratios. We conduct performance tests with different versions of Apache James with varying workloads (i.e., 4 tests for one version and 5 tests for the other version). We used JMeter to create performance tests that exercise Apache James. Different from OpenMRS, in which the performance regressions are manually injected, we performance-tested on three different versions of systems with known real-world performance issues for Apache James. Apart from the last stable version 2.3.2, there are multiple Milestone Releases for version 3.0. After checking the release notes on the Apache James website, we picked 3.0M1 and 3.0M2 to use in our study. These three selected versions have many bug fixes and performance improvements (Apa, 2019). The same subjects are studied in prior research on performance testing (Gao *et al.*, 2016). Table 4 summarizes the details of performance improvements of three selected versions.

We deployed Apache James on a server machine with an Intel Core i7-8700K CPU (3.70GHz), a 16 GB memory, and a 3TB SATA hard drive. We ran JMeter on five extra machines with a configuration of Intel Core i5-2400 CPU (3.10GHz), 8 GB memory and 320GB SATA hard drive. These JMeter instances generate multiple workloads for five hours. Similar to OpenMRS, we use *Pidstat* (pid, 2019) to monitor the CPU usage as the performance metrics of this study. We filtered out the data from the first and last half hour of the tests to minimize the system noises.

Table 4 Performance regressions in the studied open-source systems

System	Versions	Performance regressions
OpenMRS	v0	Original version
	v1	Injected heavier DB request
	v2	Added additional I/O access
	v3	Created constant delay
	v4	Injected additional calculation
Apache James	2.3.2	Stable release version
	3.0M1	Improved ActiveMQ spool efficiency (Apa, 2019)
	3.0M2	Improved large attachment handling efficiency (Apa, 2019)

4.2.3 Generated workloads

Our generated workloads are different across different software versions. By conducting chi-square tests of independence between the number of different actions and the versions, we find that in both OpenMRS and Apache James, the number of different actions and the versions are not statistically independent (p-value $\ll 0.05$), which means that the distribution of the actions are statistically different across versions. For example, for the OpenMRS system, the total number of all actions ranges from 10,673 to 22,392 across versions. In particular, the number of the *deletion of patients* action has a range from 293 to 1,861 across all versions. For another example from Apache James, the number of the *retrieving only the header of mails* action has a range from 411 to 25,532 across all versions. In addition, we provide the detailed numbers of each action that are sent to each version in our replication package.

Table 5 summarizes the average CPU utilization of each version of the systems under our generated dynamic workloads. We would like to note that when we observe a higher CPU usage, it may not correspond to a performance regression; instead it may be just due to the system having a higher workload. The influence of the dynamic workload on the CPU usage is one of the reasons that we leverage performance modeling to detect performance regressions in the field. The detailed CPU utilization information during the entire test time period is reported in the replication package.

4.3 Subject models

Our approach presented in Section 3 is not designed strictly for any particular type of machine learning models. In fact, practitioners may choose their preferred models. In this study, we study the use of six different types of models, namely linear regression, random forest, XGBoost, convolutional neural network (CNN), recurrent neural network (RNN) and long short-term memory

Table 5 Summary of average CPU utilization of each version in our case studies.

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

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

(LSTM). In particular, for linear regression, random forest, and XGBoost, the inputs of the models are vectors whose values are the number of appearances of each log event in a time period. For CNN, RNN, and LSTM, we sort the logs in each time period by their corresponding time stamp to create a sequence as the input of the neural networks. Our XGBoost model is fine-tuned using the GridSearchCV (gir, 2019) method. Our CNN, RNN, and LSTM have three, four and four layers, respectively, and they are trained with five, five and ten epochs, respectively. The number of layers and epochs are manually fine-tuned to avoid overfitting.

5 Case Study Results

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

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

Motivation

In order to use black-box machine learning models to detect performance regressions under varying workloads, we first need to understand whether such black-box models could accurately model the performance of a software system under varying workloads. Although prior research demonstrates promising results of using black-box models to capture the relationship between system performance and logs (Farshchi *et al.*, 2015; Yao *et al.*, 2018), these models are built on predefined in-house workloads instead of varying field workloads. If such black-box models are sensitive to the variance in the workloads, they may not be suitable for modeling the performance of software systems under the field workloads from real end users.

Approach

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

Performance modeling. For each open-source system (i.e., OpenMRS and Apache James), we select the version of the system that is without performance regressions. We first run the system with four different concurrent workloads (i.e., *4W*) and collect the logs and performance metrics. In order to ensure having new workloads to the system, we conduct another run by having an additional concurrent workload, i.e., having five different concurrent workloads (i.e., *5W*). We build the performance models using the data that is generated by running the *4W* workloads (*a.k.a.* the training set) and apply the model on the data that is generated by running the *5W* workloads (*a.k.a.* the testing set). The training and testing sets are derived from independent system runs. We evaluate the prediction performance of the black-box models on the 5W workloads by comparing the predicted performance and the measured performance.

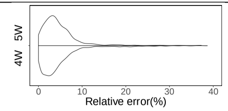
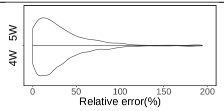
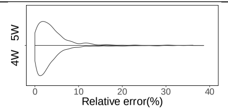
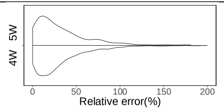
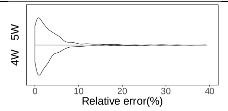
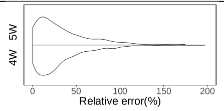
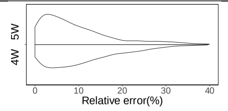
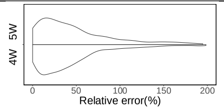
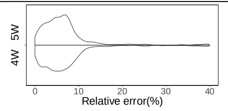
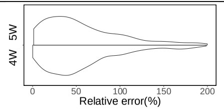
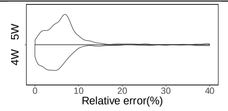
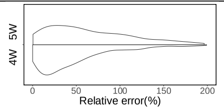
Analysis of modeling results. We calculate the prediction errors of the models on the new workloads (i.e., the *5W* workloads for the open-source systems). In order to understand the magnitude of the prediction errors, we use the prediction errors of the models on the old workloads (i.e., the 4*W* workloads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

– **Median relative error**. The difference between the predicted performance and the measured performance, normalized by the measured performance.
– **p-value** (Mann-Whitney U). In order to understand whether the models trained on the old workloads can equivalently capture the system performance under the new workloads, we use the Mann-Whitney U test (Nachar *et al.*, 2008) to determine whether there exists a statistically significant difference (i.e., $p\text{-value} < 0.05$) between the prediction errors on the new workloads and the prediction errors on the old workloads.
– **Effect size** (Cliff's delta). Reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, the p-value can be very small even if the difference is trivial) (Sullivan and Feinn, 2012). Therefore, we apply Cliff's Delta (Cliff, 1996) to quantify the effect size of the difference between the prediction errors on the old workloads and the prediction errors on the new workloads.

Results

Table 6 shows the detailed results of using six machine learning techniques to model the performance of the studied open-source systems (i.e., OpenMRS and Apache James) under varying workloads. The column "MRE" shows the median relative errors of applying the models (trained on the *4W* workloads) on the *4W* workloads and the *5W* workloads, respectively. The columns "p-value" and "effect size" show the statistical significance and the effect size of the difference between the prediction errors of the models on the *4W* workloads and the prediction errors of the models on the *5W* workloads, respectively. The "violin plot" column shows the distribution of the relative prediction errors under the *4W* and *5W* workloads.

Table 6 Prediction error details for OpenMRS and Apache James under different workloads.

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

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

Our black-box models can effectively model the performance of the studied systems using the dynamic runtime activities that are recorded in the logs. As shown in Table 6, the XGBoost model achieves the best results for modeling the performance of the OpenMRS system, with a median relative error of 2.11% on the *5W* workloads (i.e., the new workloads) and a median relative error of 2.16% on the *4W* workloads (i.e., the baseline workload). The random forest model achieves the best results for Apache James, reaching a median relative error of 22.99% and 23.08% for the *4W* workloads and *5W* workloads, respectively. All the machine learning models achieve better results for the OpenMRS system than the results for the Apache James system. The less-promising results for the Apache James system might be explained by the latency between the actual activities of the mail server system and the recorded logs. For example, the system can take an extended period of time to process an email with a large attachment, while a log about the successful processing of the email is only printed after the processing period.

The traditional models (e.g., linear regression and random forest) outperform the deep neural networks (e.g., CNN and RNN) for modeling the performance of the studied systems. As shown in Table 6, for both the OpenMRS and the Apache James systems, the three traditional models achieve better results than the three deep neural networks for modeling the system performance. These results indicate that the relationship between the system performance and the runtime activities recorded in the logs can be effectively captured by the simple traditional models. Such results also agree with a recent study (Dacrema *et al.*, 2019) that compares deep neural networks and traditional models for the application of automated recommendations.

Our black-box models can equivalently explain the performance of a system under new workloads that are unseen when constructing the models. Table 6 shows the statistical significance (i.e. the p-value) and the effect size of the difference between the prediction errors of applying the old models (i.e., trained from the *4W* workloads) on the new workloads (i.e., the *5W* workloads) and applying the old models on the old workloads (i.e., the *4W* workloads). Table 6 also compares the distributions of the prediction errors for the *4W* and *5W* workloads. The prediction errors of most of the models (except LSTM) have an either statistically insignificant or negligible difference between the *4W* and the *5W* workloads, indicating that the models trained from the old workloads can equivalently model the system performance under new workloads. The LSTM model results in a *small* difference of the prediction errors between the *4W* and the *5W* workloads. We suspect that the complex LSTM model is likely to over-fit towards the training workloads.

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

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

Motivation

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

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

Approach

The results from RQ1 show that random forest and XGBoost have the lowest prediction errors when modeling performance (cf. Table 6). Since XGBoost requires resource-heavy fine-tuning, we opt to use random forest in this research question. For the open-source systems, we first build the performance models from running the systems without performance regressions under the *4W* workloads (i.e., a combination of four different concurrent workloads). We then run the systems without performance regressions under the *5W* workloads (i.e., a combination of five different concurrent workloads). Ideally, our approach should not detect performance regressions from these runs. We use such results as a baseline to evaluate the effectiveness of our approach for detecting performance regressions under the new workloads (i.e., the *5W* workloads). Afterwards, we run the systems with the performance regressions (cf. Table 4) under the *5W* workloads. Our approach should be able to detect performance regressions from these runs.

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

Results

Our black-box-based approaches can effectively detect performance regressions under varying workloads. Table 7 shows the results of our three approaches of performance regression detection (cf. Section 3.3) on OpenMRS and Apache James. We find that with all three approaches, when there are known performance regressions between two versions, the statistical analysis always shows a significant difference between the two versions with medium or large effect sizes. In addition, the effects sizes from Approach 1 and 2 are negative, confirming the existence of performance regressions (negative values indicate performance regression and positive values indicate performance improvement).

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

Table 7 Performance regression detection results for OpenMRS and Apache James.

OpenMRS							
Versions		Approach 1		Approach 2		Approach 3	
Old version	New version	P-value	Effect size	P-value	Effect size	P-value	Effect size
v0	v0	$\ll$0.001	0.39 (medium)	$\ll$0.001	0.36 (medium)	$\ll$0.001	0.17(small)
v0	v1	$\ll$0.001	-0.59 (large)	$\ll$0.001	-0.69 (large)	$\ll$0.001	0.38 (medium)
v0	v2	$\ll$0.001	-0.44 (medium)	$\ll$0.001	-0.63 (large)	$\ll$0.001	0.37 (medium)
v0	v3	$\ll$0.001	-0.36 (medium)	$\ll$0.001	-0.42 (medium)	$\ll$0.001	0.53 (large)
v0	v4	$\ll$0.001	-0.69 (large)	$\ll$0.001	-0.76 (large)	$\ll$0.001	0.51 (large)

Apache James							
Versions		Approach 1		Approach 2		Approach 3	
Old version	New version	P-value	Effect size	P-value	Effect size	P-value	Effect size
3.0m2	3.0m2	0.008	0.09 (negligible)	$\ll$0.001	-0.12 (negligible)	$\ll$0.001	-0.03 (negligible)
3.0m2	3.0m1	$\ll$0.001	-0.65 (large)	$\ll$0.001	-0.76 (large)	$\ll$0.001	0.41 (medium)
3.0m2	2.3.2	$\ll$0.001	-0.90 (large)	$\ll$0.001	-0.93 (large)	$\ll$0.001	0.82 (large)

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

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

Our approaches can detect performance regressions as early as 15 minutes after running a new version. Table 8 shows the earliest time that our approach can detect performance regressions in the studied open-source systems. We find that all the performance regressions in the open-source systems can be detected by at least one approach with less than 20-minute data from the new version. In particular, the regressions from both versions of Apache James and three versions of OpenMRS can even be detected using only the first 15-minute data. The ability of early detection eases the adoption of our approaches in the practices of testing in the field, where performance regressions are detected directly based on the field data, instead of using dedicated performance testing. In our future work, we plan to investigate on further shortening the needed time for detecting performance regressions in the field to ease practitioners in adopting our approach.

Table 8 The earliest time for our approaches to detect regressions in the two open-source systems.

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

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

6 An Industrial Experience Report

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

6.1 Modeling system performance

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

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

6.2 Detecting performance regressions

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

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

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

7 Challenges and Lessons Learned

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

C1: Determining the sampling frequency of performance metrics

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

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

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

C2: Reducing the time cost of performance regression detection

Challenge. We choose random forest as our final black-box model to detect performance regressions, as random forest achieves the best results for modeling the performance of the studied systems (see RQ1). A random forest model contains a configurable multitude of decision trees and takes the average output of the individual decision trees as the final output. At first, we started with the default number of trees (i.e., 500 trees) (Breiman *et al.*, 2018). It took 5-6 hours to detect performance regressions between two releases of the industry system, including training and testing the random forest models and performing statistical tests. The industrial practitioners usually have an early need of checking if there is performance regression between the current version and multiple historical versions, which makes our approaches difficult to be adopted in a fast-paced development environment (e.g., an agile environment).
Solution. A larger number of trees usually result in a more accurate random forest model, at the cost of longer training and prediction time. In order to reduce the time cost of performance regression detection, we gradually decreased the number of trees in our random forest model while ensuring the model performance is not significantly impaired. In the end, we kept 100 decision trees in our random forest models. It took less than 2 hours to detect performance regressions between two releases of the industry system (i.e., training and testing the random forest models and performing statistical tests). Such a lighter model also enables us to detect performance regressions between the current version and multiple history versions (e.g., taking less than 10 hours when comparing the current version with five historical versions). We compared the prediction results of the 100-tree model with the original 500-tree models. We found that the 100-tree model only has a slightly higher median relative error (around 0.1%) than the previous 500-tree models, which is negligible in practice.
Lessons learned. In addition to the model performance, the time cost of training and applying a black-box model is also a major concern in the practice of performance regression detection in the field. Seeking an appropriate trade-off between the model performance and the time cost is essential for the successful adoption of performance regression detection approaches in the field.

C3: Early detection of performance regressions

Challenge. In an in-house performance testing process, performance engineers usually wait until all the tests finish before they analyze the testing results. However, in a field environment, any performance regressions can directly impact users' experience. We cannot usually wait for a long time to gather plenty of data before performing performance analysis. If our approach cannot detect performance regressions in a timely manner, the performance regressions

would already have brought non-negligible negative impact to users. This is a unique challenge facing the detection of performance regressions in the field.
Solution. In order to detect field performance regressions in a timely manner, we continuously apply our approach to detect performance regressions using the currently available data. For example, after a new version of the system has been up and running for one hour, we use only the one-hour data as our new workloads to determine the existence of performance regressions in the new versions. Using the data generated in a short time period also allows the analysis part of our approach (i.e., model training and predictions) to be processed faster. As discussed in RQ2, our approach can effectively detect performance regressions when the system has run for a very short time (e.g., down to 15 minutes for Apache James). In other words, our approach can detect early performance regressions in the field.
Lessons Learned. Different from performance regression detection in an in-house testing environment, performance regressions in the field need to be detected in a timely manner, in order to avoid notable performance impact to the users. Continuously detecting performance regressions using the currently available data can help detect early performance regressions in the field.

8 Threats to Validity

This section discusses the threats to the validity of our study.
External validity. Our study is performed on two open-source systems (i.e., OpenMRS and Apache James) and one industry system (i.e., System X) that are from different domains (e.g., health care system and mail server). All our studied systems are mature systems with years of history and have been studied in prior performance engineering research. Nevertheless, more case studies on other software systems in other domains can benefit the evaluation of our approach. All our studied systems are developed in either Java or .Net. Our approach may not be directly applicable to systems developed in other programming languages, especially dynamic languages (e.g., JavaScript and Python). Future work may investigate approaches to minimize the uncertainty in the performance characterization of systems developed in dynamic languages.
Internal validity. Our approach builds machine learning models to capture the relationship between the runtime activities that are recorded in logs and the measured performance of a system. However, there might be some runtime activities that are not recorded in logs and that also impact system performance. In our approach, we use logs that capture the high-level workloads of the system. Our experiments on our studied systems demonstrated that such logs can predict system performance with high accuracy. Nevertheless, the correlation between the runtime activities recorded in the logs and the measured system performance does not necessarily suggest a causal relationship between them.

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

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

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

We use the CPU usage as our performance metric to detect performance regressions. There exist other performance metrics, such as memory utilization and response time that can be considered to detect performance regressions. Considering other performance metrics would benefit our approach. However, monitoring more performance metrics would introduce more performance overhead to the monitored system. In the case of our industrial system, CPU usage is the main concern in performance regression detection. Besides, the three studied approaches are not limited to the performance metric of CPU usage. Practitioners can leverage our approach to consider other performance metrics that are appropriate in their context.

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

9 Related Work

We discuss related work along with four directions: analyzing performance test results, A/B testing and canary releasing, source code evolution and performance, and leveraging logs to detect performance-related anomalies.

9.1 Analyzing performance test results

Prior research has proposed many automated techniques to analyze the results of performance tests (Gao *et al.*, 2016; Jiang and Hassan, 2015). Initially, the Queuing Network Model (QNM) (Lazowska *et al.*, 1984) has been proposed to model the performance of a software system based on the queuing theory. Based on QNM, Barna *et al.* (2011) propose an autonomic performance testing framework to locate the software and hardware bottlenecks in the system. They use a two-layer QNM to automatically target hardware and software resources utilization limits (e.g., hardware utilization, web container threads number, and response time).

Due to the increasing complexity of software systems and their behaviors, the QNM-based performance model becomes insufficient. Therefore, prior work uses statistical methods to assist in performance analysis. Cohen *et al.* (2005) present an approach that captures the signatures of the states of a running system and then cluster such signatures to detect recurrent or similar performance problems. Nguyen *et al.* (2011, 2012) use control charts to analyze performance metrics across test runs to detect performance regressions. Malik *et al.* (2010, 2013) use principle component analysis (PCA) to reduce the large number of performance metrics to a smaller set, in order to reduce the efforts of performance analysts.

Prior work also uses machine learning techniques for performance analysis (Didona *et al.*, 2015; Foo *et al.*, 2015; Lim *et al.*, 2014; Shang *et al.*, 2015; Xiong *et al.*, 2013). Shang *et al.* (2015) propose an approach that automatically selects target performance metrics from a larger set, in order to model system performance. Xiong *et al.* (2013) propose an approach that automatically identifies the system metrics that are highly related to system performance and detects changes in the system metrics that lead to changes in the system performance. Foo *et al.* (2015) build ensembles of models and association rules to detect performance regressions in heterogeneous environments, in order to achieve high precision and recall in the detection results.

Different from existing approaches that analyze the results of in-house performance tests, our approach builds black-box performance models that leverage performance metrics and readily available logs to detect performance regressions in the field.

9.2 A/B testing and canary releasing

As the software system evolves, developers need to take actions to ensure that the new versions meet the expectations, in terms of both functional and non-functional requirements. In order to reduce the potential risk of introducing new releases, one common approach is to perform A/B testing to compare two versions of the system against each other to determine which one performs better. For example, Xu *et al.* (2015) build an A/B testing experimentation platform at LinkedIn to support data-driven decisions and improve user experiences. A/B tests is a standard way to evaluate user engagement or satisfaction with a new service, feature, or product and its main concern is not about the system performance. In comparison, our approach aims to detect performance regressions in the field.

Another prevalent industrial live experimentation technique is called canary releasing, which gradually delivers the new versions or features to an increasingly larger user group. Sato (2014) indicates that canary releasing reduces the risk associated with releasing a new version of the software by limiting its audience at the beginning. Different from canary releasing, our approach relies on the field data from the normal operations of a software system (i.e., without gradual releasing to gain confidence). Our approach has advantages over canary releasing when new features need to be delivered to all the users quickly, such as the scenario faced by our industrial partner. Nevertheless, our approach can be used in a canary releasing process to detect performance regressions based on the field data collected from a sample of the users, which can be a step in our future research.

In summary, although A/B testing, canary releasing and our approach all rely on the end user data from the field, our approach serves different purposes and has different usage scenarios from the other two techniques (as discussed above). Our approach is not a direct replacement or peer to A/B testing or canary releasing, and we do not aim to alter how the software is released to the end users. On the other hand, our approach can complement existing A/B testing and canary releasing practices.

9.3 Source code evolution and performance

Performance issues are essentially related to the source code. Thus, understanding how software performance evolves across code revisions is very important. Alcocer and Bergel (2015) conduct an empirical study on the performance evolution of 19 applications. Their findings show that every three code revisions introduce a performance variation and the performance variations can be classified into some patterns. Based on these patterns, Zhou *et al.* (2019) propose an approach called DeepTLE that can predict the performance of submitted source code before executing the system, based on the semantic and structural features of the submitted source code. While source-code level techniques aim to identify performance issues before running performance tests, we believe that the performance of the whole system is not just a simple combination of each code snippet's performance. Instead, the performance of the whole system needed to be measured by performance testing. In this work, we propose three approaches that identify performance issues when the system is running in the field, which can complement or even replace traditional performance testing.

9.4 Leveraging logs to detect performance-related anomalies

Prior research proposes various approaches that leverage execution logs to detect performance-related anomalies (He *et al.*, 2018; Jiang and Hassan, 2015; Tan *et al.*, 2010; Xu *et al.*, 2009). Jiang *et al.* (2009) propose a framework that automatically generates a report to detect and rank potential performance problems and the associated log sequences. Xu *et al.* (2009) extract event features from the execution logs and leverage PCA to detect performance-related anomalies. Tan *et al.* (2010) present a state-machine view from the execution logs of Hadoop to understand the system behavior and debug performance problems. Syer *et al.* (2017) propose to leverage execution logs to continuously validate whether the workloads in the performance tests are reflective of the field workloads. Syer *et al.* (2013) also propose to combine execution logs and performance metrics to diagnose memory-related performance issues. Similarly, He *et al.* (2018) correlate the clusters of log sequences with system performance metrics to identify impactful system problems (e.g., request latency and service availability).

In comparison, our approach leverages the relationship between the logs and performance metrics from the field to detect performance regressions between two versions of a software system.

10 Conclusions

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

Acknowledgement

We would like to thank ERA Environmental Management Solutions for providing access to the enterprise system used in our case 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

(2019). Apache james project-apache james server 3-release notes. `http://james.apache.org/server/release-notes.html`. Last accessed 10/09/2019.

(2019). Gridsearchcv. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`. Last accessed 10/11/2019.

(2019). pidstat: Report statistics for tasks - linux man page. `https://linux.die.net/man/1/pidstat`. Last accessed 10/11/2019.

Alcocer, J. P. S. and Bergel, A. (2015). Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, page 129–139, New York, NY, USA. Association for Computing Machinery.

Barna, C., Litoiu, M., and Ghanbari, H. (2011). Autonomic load-testing framework. In *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011*, pages 91–100.

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

Breiman, L., Cutler, A., Liaw, A., and Wiener, M. (2018). *Breiman and Cutler's Random Forests for Classification and Regression*. R package version 4.6-14.

Cannon, J. (2019). Performance degradation affecting salesforce clients. `https://marketingland.com/performance-degradation-affecting-salesforce-clients-267699`. Last accessed 10/11/2019.

Chen, T., Shang, W., Hassan, A. E., Nasser, M. N., and Flora, P. (2016). Cacheoptimizer: helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 666–677.

Cliff, N. (1996). Ordinal methods for behavioral data analysis.

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

Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. (2005). Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 105–118.

Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. (2017). Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 153–167.

Dacrema, M. F., Cremonesi, P., and Jannach, D. (2019). Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM Conference on Recommender Systems, RecSys 2019, Copenhagen, Denmark, September 16-20, 2019.*, pages 101–109.

de Oliveira, A. B., Fischmeister, S., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2013). Why you should care about quantile regression. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 207–218.

Didona, D., Quaglia, F., Romano, P., and Torre, E. (2015). Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, Jan 31 - Feb 4, 2015*, pages 145–156.

Farshchi, M., Schneider, J., Weber, I., and Grundy, J. C. (2015). Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 24–34.

Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., and Flora, P. (2010). Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 32–41.

Foo, K. C., Jiang, Z. M. J., Adams, B., Hassan, A. E., Zou, Y., and Flora, P. (2015). An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages

159–168.

Gao, R., Jiang, Z. M., Barna, C., and Litoiu, M. (2016). A framework to evaluate the effectiveness of different load testing analysis techniques. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, pages 22–32.

Ghaith, S., Wang, M., Perry, P., Jiang, Z. M., O'Sullivan, P., and Murphy, J. (2016). Anomaly detection in performance regression testing by transaction profile estimation. *Softw. Test., Verif. Reliab.*, **26**(1), 4–39.

Gong, Z., Gu, X., and Wilkes, J. (2010). PRESS: predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM 2010, Niagara Falls, Canada, October 25-29, 2010*, pages 9–16.

Greenberg, A., Hamilton, J., Maltz, D. A., and Patel, P. (2008). The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, **39**(1), 68–73.

Guo, J., Czarnecki, K., Apel, S., Siegmund, N., and Wasowski, A. (2013). Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 301–311.

Guo, J., Yang, D., Siegmund, N., Apel, S., Sarkar, A., Valov, P., Czarnecki, K., Wasowski, A., and Yu, H. (2018). Data-efficient performance learning for configurable systems. *Empirical Software Engineering*, **23**(3), 1826–1867.

He, S., Lin, Q., Lou, J., Zhang, H., Lyu, M. R., and Zhang, D. (2018). Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 60–70.

Ibidunmoye, O., Hernández-Rodriguez, F., and Elmroth, E. (2015). Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, **48**(1), 4:1–4:35.

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

Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. (2009). Automated performance analysis of load tests. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 125–134.

Krasic, C., Sinha, A., and Kirsh, L. (2007). Priority-progress CPU adaptation for elastic real-time applications. In R. Zimmermann and C. Griwodz, editors, *Multimedia Computing and Networking 2007*, volume 6504, pages 172 – 183. International Society for Optics and Photonics, SPIE.

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

Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. (1984). *Quantitative system performance - computer system analysis using queueing network models*. Prentice Hall.

Lim, M., Lou, J., Zhang, H., Fu, Q., Teoh, A. B. J., Lin, Q., Ding, R., and Zhang, D. (2014). Identifying recurrent and unknown performance issues. In *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 320–329.

Malik, H., Jiang, Z. M., Adams, B., Hassan, A. E., Flora, P., and Hamann, G. (2010). Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*, pages 222–231.

Malik, H., Hemmati, H., and Hassan, A. E. (2013). Automatic detection of performance deviations in the load testing of large scale systems. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1012–1021.

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

Nguyen, T. H. D., Adams, B., Jiang, Z. M., Hassan, A. E., Nasser, M. N., and Flora, P. (2011). Automated verification of load tests using control charts. In *18th Asia Pacific Software Engineering Conference, APSEC 2011, Ho Chi Minh, Vietnam, December 5-8, 2011*, pages 282–289.

Nguyen, T. H. D., Adams, B., Jiang, Z. M., Hassan, A. E., Nasser, M. N., and Flora, P. (2012). Automated detection of performance regressions using statistical process control techniques. In *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012*, pages 299–310.

Romano, J., Kromrey, J. D., Coraggio, J., and Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33.

Sato, D. (2014). Canary release. *MartinFowler. com*.

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

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

Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M. N., and Flora, P. (2013). Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 110–119.

Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M. N., and Flora, P. (2014). Continuous validation of load test suites. In *ACM/SPEC International Conference on Performance Engineering, ICPE'14, Dublin, Ireland, March 22-26, 2014*, pages 259–270.

Syer, M. D., Shang, W., Jiang, Z. M., and Hassan, A. E. (2017). Continuous validation of performance test workloads. *Autom. Softw. Eng.*, **24**(1), 189–231.

Syncsort (2018). White paper: Assessing the financial impact of downtime.

Tan, J., Kavulya, S., Gandhi, R., and Narasimhan, P. (2010). Visual, log-based causal tracing for performance debugging of mapreduce systems. In *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pages 795–806.

Valov, P., Petkovich, J., Guo, J., Fischmeister, S., and Czarnecki, K. (2017). Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 39–50.

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

Xiong, P., Pu, C., Zhu, X., and Griffith, R. (2013). vperfguard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic*, pages 271–282.

Xu, W., Huang, L., Fox, A., Patterson, D. A., and Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 117–132.

Xu, Y., Chen, N., Fernandez, A., Sinno, O., and Bhasin, A. (2015). 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*, KDD '15, page 2227–2236, New York, NY, USA. Association for Computing Machinery.

Yadwadkar, N. J., Bhattacharyya, C., Gopinath, K., Niranjan, T., and Susarla, S. (2010). Discovery of application workloads from network file traces. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 183–196.

Yao, K., B. de Pádua, G., Shang, W., Sporea, S., Toma, A., and Sajedi, S. (2018). Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 127–138.

Zhou, M., Chen, J., Hu, H., Yu, J., Li, Z., and Hu, H. (2019). Deeptle: Learning code-level features to predict code performance before it runs. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 252–259.