AN EMPIRICAL ASSESSMENT ON THE TECHNIQUES USED IN LOAD TESTING

RUOYU GAO

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
NOVEMBER 2016

Abstract

There are two main problems associated with load testing research: (1) the testing environment might not be realistic and (2) lack of empirical research. To address the first problem, we systematically assess the performance behavior of the system with various realistic environment changes. Results show that environment changes can have a clear performance impact on the system. Different scenarios react differently to the changes in the computing resources. When predicting the performance of the system under new environment changes, our ensemble-based models significantly out-perform the baseline models.

To address the second problem, we have empirically evaluated 23 test analysis techniques. We have found all the evaluated techniques can effectively build performance models using data from both buggy or non-buggy tests and flag the performance deviations. It is more cost-effective to train models using two recent previous tests collected under longer sampling intervals.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Zhen Ming Jiang for giving me the opportunity to do research on load testing and support my master study. Without him, this thesis would not have been completed.

I would like to thank my committee members: Dr. Marin Litoiu and Dr. Zijiang Yang for taking time out of their busy schedule to read my thesis and to provide insightful comments.

I also want to thank other members in the SCALE lab for working and having fun together. Last but not the least, I want to thank my family for the continuous support and encouragement.

Table of Contents

List of Tables

List of Figures

1 Introduction

Performance is a important non-functional requirement for large-scale systems like Google and Amazon. These systems handle concurrent requests from thousands or even millions of users everyday. Failure to satisfy performance and scalability requirements can have extensive impact on the success of the system and may result in negative news coverage (e.g., the crash of online train ticket booking system in China [1] and the botched launch of US Government's Health Care website [2]). Hence, in order to make sure that the system in the field is able to provide stable service, load testing is necessary apart from functional testing.

Load testing usually uses one or more load generators mimicking the desired number of users sending concurrent requests to the system under test (SUT) during a period of time. When performing load testing, the runtime behavior of the SUT is logged in order to do analysis. Generally speaking, there are two types of system behaviour data: system logs and resource counters. System logs are logs produced by the SUT (e.g., print("customer login")), while resource counters are counters reflecting resource usage of software resources as well as hardware resources (e.g., CPU queue length, disk I/O, etc.). Moreover, in order to capture the performance of the SUT, testing environment needs to be closely mimicking the field environment to have an accurate assessment of the expected field behaviour. In the field there may have other processes running along with the SUT, resulting in interference with the SUT. For example, if there is an antivirus software running on the node in which the database is running, it might slow down the I/O of the database, hence, increase the response time of requests which are related to the database requests.

The general process of a load test can be divided into the following phases: test design, test setup and execution and test analysis [3]. There are many test design techniques proposed and many test execution tools available in industry and open source software systems. In this thesis, we focus on the test setup and test analysis, since these two areas have not been thoroughly studied. In particular, there are a few challenges associated with setting a test environment and analyzing a load test:

1. **Additional Tools**: In the field environment, system may need to run along with additional tools such as auto backup or antivirus software without informing the performance analysts.
2. **Shared Environment**: Many of the current system are run in a shared environment such as cloud environment. In this case, different systems on

the same physical box may interfere with each other.

3. **Rapid Changes**: In order to make fast reactions to the needs of users, Continuous Integration has been adopt to constantly push the changes to the field. Hence, it's very hard to all ways make sure that the test environment is up-to-date.
4. **Large Volume of Data:** Usually a load test will last for hours or even days, generating hundreds megabyte or gigabytes of logs, which is hard for manual analysis.
5. **Limited Time:** Load testing is usually done after all functional testing has been finished to make sure there are few functional failures confounding the test results, so the time remains for load testing is usually very limited.
6. **Lack of Test Oracle:** Functional testing has clear pass and fail criteria, such as whether a user has successfully registered or not. But the objectives of a load test are not clear [4] and are often defined later on a case-by-case basis [5].

These challenges can be further divided in two categories: the first three challenges are for re-creating a field-like test environment; and the remaining three challenges are for analyzing load tests. To tackle with the challenge of test vs. field environment, we take an different approach. We first conduct environment-variation-based load testing to obtain different performance behaviour of the target system under realistic changes. Then we use performance modeling techniques to predict the performance behaviour of the system under the new environment changes. To address the second challenge, researchers have proposed several automatic performance analysis techniques to tackle some of these problems (e.g., [6, 7, 8, 9]). The idea of these techniques are using different models to evaluate whether the performance of the current tests is different from the previous tests. But none these techniques have been compared against each other. Hence, we have proposed an evaluation framework to empirically evaluate 23 different performance analysis techniques.

The contributions of this thesis are:

1. This is the first study, which systematically evaluates the effectiveness of different test analysis techniques.
2. We proposed a flexible framework that can evaluate different test analysis techniques on a range of threshold values. In addition, it can easily be used to evaluate other systems or techniques.
3. This is the first work which assesses the impact of the test results under the variations in the system deployment environment. Compared to the existing testing techniques like Chaos Monkeys [10], our environment-variation-based load testing is more fine-grained.
4. We have proposed an ensemble-based performance model to leverage data

from previous runs to predict the system performance under new SDE changes.

5. We put our test data used in the thesis online [11] [12] to encourage further research on this topic.

Thesis Organization

The rest of the thesis is organized as follows: chapter 2 describes our approach of assessing system performance under realistic environment changes . Chapter 3 introduces our framework to evaluate the effectiveness of load testing analysis techniques. Chapter 4 explains the related work and chapter 5 concludes the thesis and discusses some future work.

2 Systematically Assessing the Impact of Environment Variations on the Results of Load Tests

Most of the in-house performance tests are done in a clean lab environment, whereas most of the field environments have additional tool(s) running along with the system. In this chapter, we will investigate whether this kind of environment variations would impact the system and how to properly model the performance behaviour of the SUT under these variations.

2.1 Introduction

Load testing is an important type of non-functional testing technique to ensure the performance and the scalability of large-scale software systems like AT&T's telecommunication systems [13] and Microsoft's web services [14]. These systems are used by millions of users concurrently everyday. The failure to process high volumes of user traffic could result in catastrophic consequences and unfavorable media coverage (e.g., the crashes of the Consensus websites for Statistics Canada [15] and Bureau of Statistics in Australia [16]). Load testing in general refers to the process of assessing system behavior under load in a field-like environment [13]. The load drivers are used to generate hundreds or thousands of concurrent requests, which mimic the realistic user behavior, to the system under test (SUT). During the course of the load test, execution logs and performance counters are recorded for further analysis.

Although some load tests are conducted in the field environment, it is often very difficult and costly to access the actual production environment [17, 18]. Most of the load testing is done in a lab environment. The computing hardware can be purchased in-house [17] or by renting virtual machines from the cloud computing service providers [19, 20]. However, extra care should be taken of to ensure the lab environment closely resembles various characteristics in the deployment field. For example, if the database and the web sever are far apart (e.g., in two different continents), network delays should be added when deploying the SUT in the lab to simulate cross-the-continent network latency using Network Virtualization tools like Shunra [21].

However, re-creating the field-like environment in the lab is very challenging due to the following three reasons: (1) **Additional Tools**: System operators may install additional tools in the field for security [22] and monitoring purposes [23] without informing the performance analysts; and (2) **Shared Environment**: Many of the systems are installed in a shared environment (e.g., public cloud or multi-tenant), in which one or multiple resources are shared by different users [24, 20]. Both cases can cause the instability of the performance behavior of the SUT [19, 25] and degrading user experience [26]. (3) **Rapid Changes**: Nowadays, many companies adopt the Continuous Integration/Continous Delivery process, in which software changes and environment configuration changes are constantly pushed to the field. Companies like Netflix are constantly updating their various components in the field [10]. It would not be feasible to re-create an up-to-date test environment.

In this chapter, we tackle the problem of deviations in the lab/field environments with a different approach: rather than trying to create a "realistic" lab environment, which captures and simulates all the characteristics in the field, we want to conduct environment-variation-based load testing to systematically assess the behavior of the SUT under different changes in the system deployment environment (SDE). For example, we will conduct a load test in an environment with limited computing resources (e.g., CPU) to simulate the case of the SUT deployed in an environment with a resource-intensive process. We also will conduct a load test in an environment where other processes (e.g., anti-virus) are active. Our objective is to learn the performance behavior of the SUT under these tests and to pin-point resource-sensitive scenarios. In addition, we also want to build performance models based on the results of these environment-variation-based tests. These models can help system operators to anticipate potential performance deviations in other changes to the SDE.

The contributions of this chapter are:

1. This is the first work to systematically assess the impact of the test results with respect to the variations in the SDE. Compared to the existing testing techniques like Chaos Monkeys [10], which assess the SUT behavior by randomly turning off some processes or machines, our environment-variation-based load testing can assess the behavior of the SUT in a more fine-grained manner. The results of our testing can provide useful suggestions to the system operators (e.g., process A cannot be co-deployed with the SUT due to high contention of the CPU resources).

2. We have proposed an ensemble-based performance model to predict the system performance for changes to the SUT, which have not been previously tested. Case studies show that our ensemble models out-performs the baseline models.

3. In this chapter, we have executed over 80 hours of environment-variation-based load tests executed on three open source systems. We have made this

data available online [11] to support replications and to encourage further research on this topic.

Chapter Organization

The rest of the chapter is organized as follows: Section 2.2 presents a motivation story. Section 3.4 provides an overview of our case study setup by explaining the target systems, the motivation and the descriptions of the proposed three research questions. Sections 2.4, 2.5 and 2.6 study the three research questions. Section 3.8 discusses the threats to validity. Section 3.9 concludes the chapter and discusses some future work.

2.2 Motivation

Bryan was a performance analyst working for an e-commerce startup, which adopted the Continuous Integration/Continous Delivery development process. The new version of the system had already been successfully load tested. However, right before the system upgrade, the system operators informed Bryan that all the field machines had anti-virus software ABC and data backup program DEF installed due to recent security breaches. After the installation of ABC and DEF, the system operators had noticed some response time fluctuations on the current version of the system. Hence, they were not sure whether these two pieces of software would have a large performance impact on the new version of the system. Unfortunately, it was not feasible for Bryan to install these two additional software applications in the lab machines due to high costs associated with the software licenses. In addition, Bryan had also been inquired by the management team, as customers had been expecting some of the features and bug fixes in the new version of the system for a while.

Pressured by cost and time, Bryan built a performance model using the results of his environment-variation-based load tests. Using this model and the performance characteristics of software ABC and DEF, Bryan was able to predict the performance behavior of the new version of the system in the field. Bryan analyzed the prediction results and only found small performance deviations compared to the existing lab testing results. After obtaining the analysis results from Bryan, the system operators carried out the upgrade process without any issues.

2.3 Case Study Setup

In this section, we will introduce the setup of our case studies. In particular, we will discuss about the case study systems, the motivation and the descriptions of three research questions and the load test setup.

Table 2.1: Case Study Systems

System	Domain	Category	# of Test Scenarios	SLOC
DS2	Benchmark application	JSP	4	~ 100
PET	E-commerce	Hibernate	15	> 2000
JMS	Mail Server	Mailet Container	6	$> 4{,}000$

2.3.1 Target Systems

In order to assess the behaviour of different types of the system under SDE changes, we use these three open source systems to conduct our case study: Dell DVD Store (DS2) [27], Petclinic (PET) [28] and James Mail Server (JMS) [29]. Each of them has different types of components, configurations and uses different development technologies. Table 3.2 shows the summary of these three systems.

- **DS2**: DS2 is an open source e-commerce website made by Dell to do performance testing on their hardware. It has 2 component: web server and database. It is deployed on the Tomcat as web server, and uses MySQL as database. The system has 4 different pages: login, browsing DVD, purchasing items and registrations for new customers.

- **PET**: Petclinic is a web application based on Java Spring framework and Hibernate. Similar to DS2, it is also deployed on Tomcat as web server and uses MySQL as database. While DS2 contains only 4 scenarios, Petclinic has 15 different test scenarios related to pet owners (e.g., view pet owners, edit owners information), pet (e.g., view pet, add visit log to pet), and veterinarians (e.g., view list of veterinarians).

- **JMS**: The Apache James Mail Server (JMS) is an open source mail server based on Java. Different from DS2 and Petclinic, James Mail Server is a stand-alone system, so it does not need any other components. As a mail server, James Mail Server has two generic scenarios: send mail and receive mail. Based on [30, 31], these two scenarios can be further divided into more smaller scenarios. In our chapter we divided them into six scenarios: sending or receiving emails with or without attachments, reading only mail header or reading the whole mail.

2.3.2 Research Questions

We have proposed the following three research questions (RQs) to assess the system's performance behavior under different variations to the SDE.

- **(RQ1) Impact Analysis**: *How different can the results of a load test be* *under realistic changes to the SDE?*

Although many efforts have been made to mimic the lab environment close to the actual field, in many cases it is not possible to completely capture all the environment characteristics in the field. For example, the system operators might decide to install additional probes or anti-virus software in the field for purposes like monitoring and security without informing the performance analysts. Hence, in this research question, we want to examine the degree of performance deviation for SUT when running under an ideal environment (no additional software systems) and under an environment with some realistic changes.

- **(RQ2) Sensitivity Analysis**: *Which test scenarios are sensitive to the variations in the SDE?*

  The system consumes different amount of computing resources, while processing different scenarios. For example, the scenario of compressing a file would probably consume more CPU and disk resources, instead of memory; whereas uploading an image would probably consume more disk and network resources, rather than CPU. In this research question, we want to perform a sensitivity analysis on the performance impact of different scenarios with respect to the amount of computing resources allocated.

- **(RQ3) Model-based Analysis**: *Can we effectively model the performance impact of the SUT due to the variations in the SDE?*

  In practice, each software application consumes various computing resources. Furthermore, the resource consumption patterns would not be uniform throughout the time. For example, the amount of the computing resources consumed would be much higher when an anti-virus program is actively scanning, rather than idle. Finally, due to the complexity of the field environment, in many cases, it would not be feasible to exercise all the variations in the SDE. Some enterprise systems (e.g., SAP's ERP systems) could be deployed in thousands of customers' site, each of which has different field configurations set by the system operators. Hence, it would be very useful to build a performance model which can predict the anticipated performance when the SUT is deployed in the field. In this research question, we want to research into modeling techniques which can effectively predict the SUT's performance behavior for changes to the SDE that are not previously assessed.

2.3.3 Test Setup

We use two machines to execute our load tests. One machine is used to deploy the SUT. Its configuration is Intel® Core™ 2 CPU 2.40 GHz, 2 GB memory and a 160 GB 7200 RPM hard drive. The other machine is used as a load generator to mimic the hundreds or thousands of users using the system concurrently. The load generator machine has Intel® Core™ i7-4790 CPU 3.60 GHz, 16 GB memory and a 2TB 7200 RPM hard drive. The SUT and the load generator are deployed on separate machines, as this is the standard practice to eliminate the performance interference caused by the load generator [3].

We use an open source tool, called Apache JMeter [32], as our load generator and pidstat [33] to monitor and record the performance behavior during the course of a load test. In addition, we have also recorded the response time logs from JMeter and the access logs from Tomcat for each load test. These logs are used to extract the workload and the response time information. After we finish each run, we parse the data as each row representing the system state in the past one minute. To simulate the variations of the SDE, we have used the following three tools:

- **rsync**: Many of the field machines perform period data backup to avoid potential data losses due to machine failures or security breaches. In this chapter, we use a data backup utility, called rsync [34], which is used to perform automated remote data backup during a load test.

- **ClamAV**: We use an open source anti-virus program, ClamAV [35] to evaluate the performance impact of SUT under virus scan.

- **stress-ng**: The above two applications are used to simulate realistic changes to the SDE. However, there can be more variations to the SDE in reality. Hence, we need to have a way to assess the SUT behavior under more variations to the SDE (e.g., a different computing resource usage pattern compared to rsync and ClamAV). To address this problem, we use an open source application, stress-ng [36] to control the amount of available computing resources in a more fine-grained manner. Stress-ng can be configured to consume different amount of computing resources like CPU, memory and disk, and hence, constraining the SDE that SUT is running under.

For the details on the experiments, please refer to the next three sections. In total, we run more than 80 hours of load tests and collected about 150 performance counters and approximately 8 GB logs from three target systems.

2.4 RQ1 - Impact Analysis

In many cases, a load test is running in an ideal environment, in which only the SUT is running. The reasons are two-fold. First, it is much easier to monitor and analyze the performance of the SUT under load in a cleaner environment. Second, the performance analysts are not clear about the actual characteristics of the field environment. Hence, in this research question, we want to assess the degree of performance deviations between load tests running in an ideal environment and in an realistic environment (e.g., an environment in which the anti-virus is running together with the SUT).

2.4.1 Experiment

The same load tests are executed in various environment setup. All three subject systems are run under the following four environment settings:

- *ideal test*: First, two load tests, ideal_1 and ideal_2, are run in an ideal environment, where no additional applications are activated. The reasons for executing the *ideal test* twice is because we want to check whether the performance behavior in the ideal setting is similar to each other.

  Then, we run three different load tests in the following three more realistic settings. Collectively, we call these three types of environment-variation-based load tests (*backup* test, *sscan* test, and *dscan* test) as the "realistic runs".

- *backup test*: We run one load test in an environment, in which rsync is running along with the SUT. This setup is to simulate the field environment in which remote data back-up is enabled.

- *sscan test*: We run another load test in an environment, in which the anti-virus application ClamAV is running along with the SUT. The ClamAV is configured to perform anti-virus scan using only one thread. This setup is to simulate the field environment in which the anti-virus application is installed.

- *dscan test*: This load test setup is the same as the *sscan test*, except ClamAV is configured to perform anti-virus scan using two threads. The goal here is to simulate the field environment in which the operators want to perform a faster anti-virus scan.

In total, five load tests are run for each of the subject systems.

2.4.2 Result Analysis

In order to investigate the impact of variations in the SDE on model accuracy, we perform the following two types of analyses: statistical comparison and performance model assessment.

Statistical Comparison

We statistically compare the response time between the ideal runs and the three realistic runs using the Wilcoxon Rank Sum test and Cliff's Delta (WC). Wilcoxon Rank Sum (WRS) test is a non-parametric test which compares the distributions of the two datasets. For example, if two sets of response time are about one minute (1 minute vs. 1.0006 minute) and the WRS test shows statistical significance between these two datasets. However, the differences between the two sets are so small (36 milliseconds), humans would hardly notice the differences. Hence, to quantify the strength of the differences between the two distributions, we use Cliff's Delta (CD). CD is a non-parametric technique to calculate the effect size between two distributions [37]. CD quantifies the level of the differences as follows:

$$\text{effect size} = \begin{cases} \text{trivial} & \text{if } CD < 0.147 \\ \text{small} & \text{if } 0.147 \leq CD < 0.33 \\ \text{medium} & \text{if } 0.33 \leq CD < 0.474 \\ \text{large} & \text{if } 0.474 \leq CD \end{cases}$$

The results are shown in Table 2.2. Table 2.2 shows the number of scenarios which are impacted by realistic changes to the SDE: each cell indicates how many scenarios have different performance behaviour when comparing one ideal run and one other run. We consider the differences are significant when the WRS test shows statistical significance and CD shows medium to large effect sizes. For example, in DS2, when comparing the ideal$_1$ and the backup runs, there is one out of the total four tested scenarios which have different performance behaviour. The cell shows an "-" when it is compared against itself (e.g., ideal$_1$ comparing against ideal$_1$). In general, the performance behavior is the same for all the scenarios when comparing two ideal runs. However, there is at least one scenario from all three subjects whose

Table 2.2: Percentage of test scenarios which have different performance behaviour between the ideal and the realistic runs.

Systems		**Test Runs**				
		Ideal_1	Ideal_2	Backup	Dscan	Sscan
DS2	Ideal_1	-	0.00%	25.00%	50.00%	75.00%
	Ideal_2	0.00%	-	25.00%	100.00%	100.00%
PET	Ideal_1	-	0.00%	73.33%	93.33%	100.00%
	Ideal_2	0.00%	-	73.33%	100.00%	100.00%
JMS	Ideal_1	-	0.00%	100.00%	100.00%	100.00%
	Ideal_2	0.00%	-	100.00%	100.00%	100.00%

performance behaviour different when comparing against the ideal and the realistic runs.

Performance Model Assessment

As performance requirements are seldom available and up-to-date [38, 5], performance analysts usually adopt the no-worse-than previous principle. In other words, performance analysts build performance models based on the load testing data from previous runs and predict on the current runs. There can be performance problems, if there is a large deviation between the actual measurements and the predicted results. Hence, we also build performance models using the load testing results from the ideal runs and compare the prediction results on the ideal runs as well as the realistic runs. We use the following three statistical techniques to build the performance models:

- **Generalized Linear Model (GLM)**: Generalized Linear Model models the system behavior using a global function of workload mixes and resource usage utilization [39]. Extra care is taken to ensure the assumptions associated with the GLM model are met.

- **Multivariate Adaptive Regression Splines (MARS)**: MARS [40] is a model that can split data with different kinds of behaviour into different sub models using one or more "hinge function". Each hinge function is a function indicating two different behaviours based on certain cut-offs. All the hinge functions can be automatically produced by the model, modeling system in different period rather than as a whole like GLM. We want to use MARS to model the different phases in a load test (e.g., when anti-virus program is active vs. idle).

- **Regression Tree (RegTree)**: Compared to GLM, RegTree does not have many assumptions associated with the model [41]. Similar to MARS, RegTree can also model the behavior of the system in different phases, but in a tree-like manner. RegTree builds a decision tree model by splitting the data points into certain leaves.

We build the performance models using the results from the ideal runs. Then we either predict the performance of another ideal run or the performance of one realistic run. Then we perform the WC analysis between the predicted values and the actual measured results. If they are statistically significantly different by the WRS analysis and have medium to large effect sizes, we consider this scenario have large performance deviations between the predicted values and the actual values. Table 2.3 shows the percentage of scenarios from each system where there large deviations between the predicted and the measured values. For example, in DS2, when using RegTree as the prediction model and trained on the data from the ideal run, there are 12.50% of the test scenarios which are different when predicting on another ideal run. However, when predicting on the realistic runs, 50% of the scenarios have large deviations between the actual and the predicted values.

In general, the prediction results on the ideal runs are better than the realistic runs when using MARS or RegTree as the modeling techniques. This is the opposite when using the GLM as the modeling technique. We believe this is mainly due to the performance of the modeling techniques, as the model performance of the GLM is significantly worse than the other two techniques. For example, when training

Table 2.3: Percentage of test scenarios in which there are different between the predicted and the actual values.

Models	**DS2**		**PET**		**JMS**	
	Ideal	Realistic	Ideal	Realistic	Ideal	Realistic
GLM	37.50%	37.50%	43.33%	38.89%	50.00%	66.67%
MARS	12.50%	37.50%	40.00%	45.56%	16.67%	72.22%
RegTree	12.50%	50.00%	3.33%	28.89%	8.33%	94.44%

on the ideal run and predicting on another ideal run in DS2, the performance of GLM is more than three times worse than the performance of MARS and RegTree. For RegTree, although it can predict the performance from another ideal run with high performance, the prediction results on the realistic runs are much worse.

2.4.3 Summary

Findings: There is a clear performance impact when incorporating some realistic changes to the SDE. Such changes can cause the system performance to be noticeably different. Consequently, the performance models built using the ideal runs cannot be used to accurately predict the behavior of the realistic runs.

Implications: As it can be unavoidable to have variations to the SDE, there could be large performance deviations between the load tests running in an ideal environment and the field. It is worthwhile to conduct environment-variation-based load testing to assess the performance impact due to changes to the SDE.

2.5 RQ2 - Sensitivity Analysis

In the previous RQ, we have demonstrated the need to conduct environment-variation-based load testing. When the system contains abundant computing resources, the SUT and the other applications can run without interfering each other. However, when the resources are limited, the performance of the SUT can be impacted if some of its scenarios are competing for the same computing resources against another application. In this RQ, by leveraging the environment-variation-based load tests, we want to conduct a sensitivity analysis on the performance impact of different scenarios with respect to the amount of available computing resources.

2.5.1 Experiment

If one scenario consumes much CPU, limiting the CPU resource would result in a large increase in its response time. Otherwise, its response time would not be impacted. Hence, in order to find out which scenario is sensitive to what kinds of computing resources, we run the load tests with limited computing resources. We run the load test along with stress-ng multiple times. Each time, stress-ng is configured to consume a different amount of computing resources, which forces the SUT to be run under different constrained conditions. For each system, we run six additional environment-variation-based load tests apart from the two ideal tests. The details are as follows:

- *ideal test*: We ran the load test under the ideal environment twice. We call these two runs as ideal_1 and ideal_2. These two runs are used as our baseline in this RQ.

- *CPUStr test*: We ran the load tests with stress-ng configured to consume different amount of CPU resources. We run two CPUStr tests in this RQ: stress-ng configured to consume 30% and 70% of the single core CPU resources. We call these two runs as CPUStr_1 and CPUStr_2, respectively.

- *DISKStr test*: Similar to the CPUStr tests, we ran the load tests with stress-ng configured to perform writes to the disk with one or two worker threads.

We call these two runs as DISKStr_1 and DISKStr_2, respectively.

- *MEMStr test*: Similar to the CPUStr and the DISKStr tests, we ran the load tests with stress-ng configured to consume 512 MB or 1 GB of memory. We call these two runs as MEMStr_1 and MEMStr_2, respectively.

Collectively, we call the the CPUStr test, the DISKStr test and the MEMStr test as "lab stress runs".

2.5.2 Result Analysis

We use the WC analysis to compare the response time between the ideal runs and a particular type of lab stress runs. For example, if WC result of the login scenario shows a large difference when comparing the ideal run and the CPUStr runs, the performance of the login scenario is sensitive to the amount of available CPU resource. The overall WC results are shown in Table 2.4. Each row shows the percentage of scenarios from that subject system which has different response time from the ideal runs. For example, the third row in the table shows that 75% of the scenarios in DS2 are impacted when running with lower level CPU stress (CPUStr_1).

As expected, the two ideal runs are similar to each other. Hence, no scenarios are flagged. Generally speaking, lower level of stress will have less effect to the

Table 2.4: Percentage of test scenarios which are different between the ideal and the lab runs.

Systems		**Ideal Runs**		**CPU Stress Runs**		**Disk Stress Runs**		**Memory Stress Runs**	
		Ideal$_1$	Ideal$_2$	CPUStr$_1$	CPUStr$_2$	DISKStr$_1$	DISKStr$_2$	MEMStr$_1$	MEMStr$_2$
DS2	Ideal$_1$	-	0.00%	75.00%	100.00%	100.00%	100.00%	0.00%	0.00%
	Ideal$_2$	0.00%	-	75.00%	100.00%	100.00%	100.00%	0.00%	0.00%
PET	Ideal$_1$	-	0.00%	73.33%	93.33%	100.00%	100.00%	0.00%	93.33%
	Ideal$_2$	0.00%	-	66.67%	80.00%	100.00%	100.00%	0.00%	93.33%
JMS	Ideal$_1$	-	0.00%	16.67%	50.00%	100.00%	100.00%	100.00%	100.00%
	Ideal$_2$	0.00%	-	33.33%	66.67%	100.00%	100.00%	100.00%	100.00%

scenario performance. But if the scenario heavily relies on certain type of resource, limiting this resource will have significant a impact on this scenario. DS2 and PET both use database for information storage, and JMS needs to access the disk to read and write emails. Hence, both levels of disk stresses show significant impact for all three systems. JMS consumes large amount of memory to store the users' states and PET uses Hibernate, which caches the database information to optimize database performance. Hence, the scenarios from these two systems are impacted by memory stresses. DS2 is a simple benchmarking system using the JSP technology. The scenarios from DS2, which do not consume much memory, are not impacted by the changes in the memory resource. Compared to DS2 and PET, the impact of the CPU stress is smaller in JMS.

2.5.3 Summary

Findings: Varying the amount of computing resources can impact the performance behavior of the SUT. However, there is no unified rules in terms of the amount of available computing resources and the performance of individual scenarios. Different scenarios may or may not be impacted by the underlying changes to the SDE.

Implications: There is a need to create better performance models to assess the performance implications of different scenarios with respect to the changes in the SDE.

2.6 RQ3 - Model-based Analysis

RQ2 shows that there is no one-size-fits-all answer to the relationship between the performance of individual scenarios and the amount of available computing resources. In this RQ, we will propose an ensemble-based modeling approach to systematically assess the performance implications of different scenarios with respect to the changes in SDE.

2.6.1 Ensemble Modeling

Our proposed ensemble models consist of many small performance models, built using the data from segments of environment-variation-based load tests. These

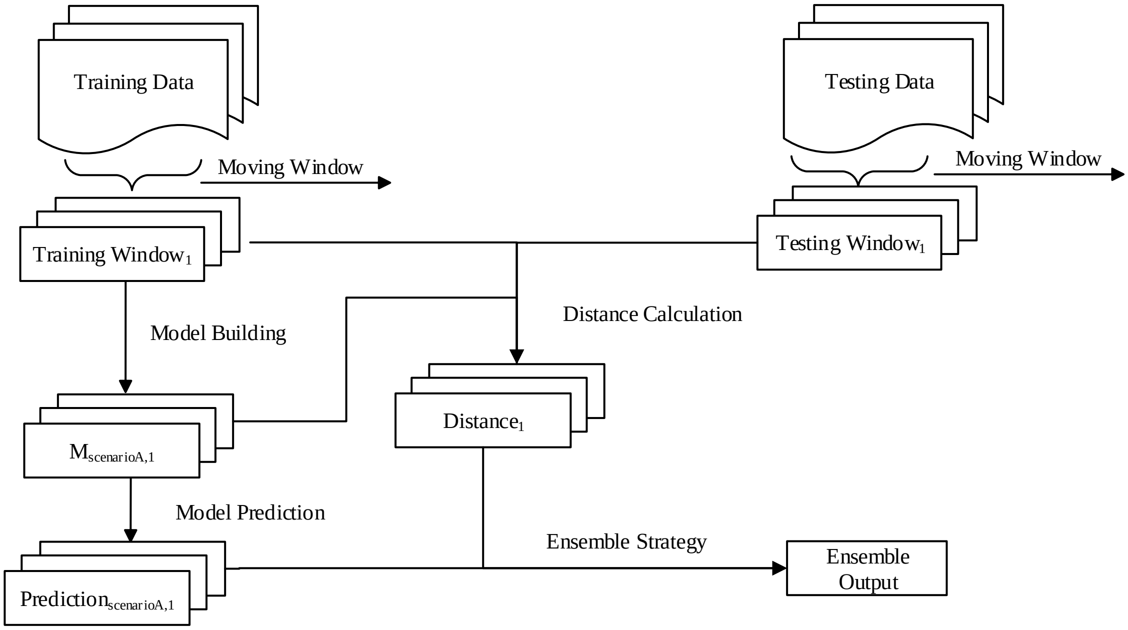

Figure 2.1: Our ensemble-based modeling approach.

small performance models are called sub-models. During runtime, each of these sub-models performs the prediction. The overall ensemble model output is based on how close the state of the current test environment and the environment enclosed in the sub-models. The more similar the state of the two environments are, the higher weight it should be given to the prediction results of this sub-model. Figure 2.1 illustrates the overall process of our ensemble modeling. It consists of three steps: model building, distance calculation and model prediction.

Step 1 - Model Building

As the available computing resources can fluctuate during the course of a load test due the state of different application processes (e.g., anti-virus actively scanning vs. anti-virus being idle). Hence, in order to accurately capture SUT's behaviour under different environment conditions, we split the load testing data with a moving window. The window contains data within a fixed time window size i. Hence, the first window (Training window$_1$) contains the data from time 0 to time $i - 1$; the second window (Training window$_2$) contains the data from time 1 to time i, and so on. For the data within that window, we built performance models using the data from that window as the training data. One performance model is built for the response time of each scenario. For example, the model $M_{scenarioA,i}$ corresponds to the performance model built to predict the response time of scenarioA using the training data from Training window$_i$.

This process is applied on all our executed environment-variation-based load tests (a.k.a., the ideal runs, the realistic runs, and the lab stress runs). For the data from each window, we build three kinds of performance models: GLM-based models, MARS-based models and RegTree-based models.

Step 2 - Distance Calculation

In the previous step, we have built many small performance models, which captures the performance behavior of SUT under different environment variations. The next step is to defined how to determine the final output from the ensemble model.

When the ensemble model is used (a.k.a, during the testing stage), each of these small performance models would give a prediction value. We need an approach, which automatically selects the most suitable model(s) and use their prediction result(s) as the final ensemble output. Our distance function is defined to measure the similarity between the training environment and the testing environment. If the SUT is running in a training environment, which is similar to the current target environment, the performance behavior should be similar with each other. Hence, the prediction results from these performance models should be given more trust than the prediction results from the other models in the final ensemble output. We will explain the calculation of our distance function with an example. Figure 2.2 shows an example of the resulting performance model built with the regression tree technique. In this case, the model contains two independent variables: CPU and disk. Hence, our distance function measures the degree of similarity of these two counters between the training window and the current target environment.

Similar to the *Training window*, we define the *Test window* with a fixed time

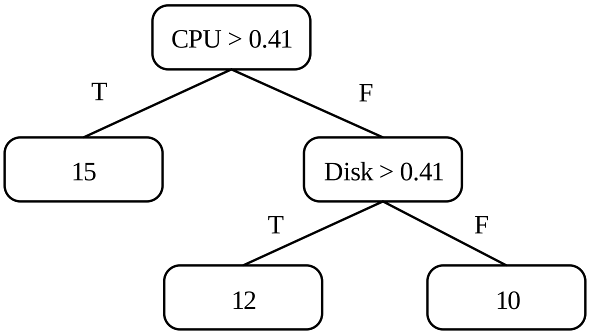

Figure 2.2: An example of a regression tree-based performance model.

window with a fixed time window size j. For example, at current time t, the window (Test Window$_t$) consists of the data from $t - j - 1$ to t (a.k.a., the data from the past j minutes). Note that the size of the training and the testing windows are configurable and they are not necessarily the same. For each prediction, we conduct the WC analysis on the CPU data between the current ($Test\ Window_t$) and all the ($Training\ Windows$). For the details of the WC analysis, please refer to Section 2.4. Similarly, the WC analysis is also applied on the disk data. Hence, for each of the performance models, we have two WC values: one for the CPU counter and the other one for the disk counter. The value of the distance function for this model is just the average of the two WC values. For example, if the WC values for the CPU and the disk counters are 0.4 and 0.6. The resulting distance function for this performance model is 0.5 ($\frac{0.4+0.6}{2}$). Different performance models would have different types of performance counters. For example, if a MARS-based performance model contains only one independent variable: the disk. The distance function for this model is just the WC values for the disk counter.

Step 3 - Model Prediction

Once we have calculated the distance functions of all the models, we need to have a way to combine the prediction results from all these models into one final output for the ensemble model. In this chapter, we have experimented with the following three ensemble strategies:

- **Mean** calculates the average of the prediction results from all the performance models. Note that in this case, the values calculated from the distance function are not used.
- **Winner-takes-all** just uses the prediction results with the smallest distance values. If there is a tie, we just use the median of the prediction results from all the best models.
- **Weighted Average** provides a weight for each of the prediction results and aggregates them. For example, there are three performance models, which have WC values 0.2, 0.4, 0.6, respectively. The prediction results from these three models are: 0.6, 0.3, and 0.4. The final ensemble output would be $\frac{\frac{1}{0.2+1}}{\frac{1}{0.2+1}+\frac{1}{0.4+1}+\frac{1}{0.6+1}}\times 0.6+\frac{\frac{1}{0.4+1}}{\frac{1}{0.2+1}+\frac{1}{0.4+1}+\frac{1}{0.6+1}}\times 0.3+\frac{\frac{1}{0.6+1}}{\frac{1}{0.2+1}+\frac{1}{0.4+1}+\frac{1}{0.6+1}}\times 0.4 = 0.44$. We

take the inverse of WC values, as bigger WC values indicate larger differences. We also added one to each of the WC values to avoid division by zero errors.

For each test window, the ensemble model is only going to predict the last point (a.k.a., the prediction value for the current time).

2.6.2 Evaluation

We evaluate our ensemble modeling techniques by training on the results of environment-variation-based load tests and predicting on the results of three realistic runs (backup test, sscan test, and dscan test). In order to effectively evaluate our ensemble modeling technique, we vary the configurations of our ensemble models in the following three dimensions:

- **The size of the training window**: we use 10, 15, or 20 minutes as three different window sizes. We fix the testing window size as 10 minutes in order to compare the effect of the training window sizes.

- **The training dataset**: We evaluated our ensemble models with these three options: (1) ideal runs only, (2) ideal runs and lab stress runs, and (3) all runs. There are in total six different kinds of lab stress runs as described in Section 2.5: two CPU stress runs (CPUStr$_1$ and CPUStr$_2$), two memory stress runs (MEMStr$_1$ and MEMStr$_2$) and two disk stress runs (DISKStr$_1$ and

DISKStr$_2$). For option (3), all runs refer to all the ideal runs, all the lab stress runs and two out of the three realistic runs. For example, if we are predicting the results of the backup test, the two realistic runs in option (3) would be the dscan test and the sscan test.

- **Ensemble strategies**: We used using mean, winner-takes-all and weighted average as three options to calculate the final ensemble results.

To demonstrate the effectiveness of our ensemble-based modeling technique, we compare our results against the baseline technique, which is the single-model based approach. The single-model based approach only uses the data from the ideal run and builds just one performance model. Similar as the other two RQs, we calculate the percentage of scenarios whose predicted values and the measured values are different. In other words, the two sets of data are statistically different by the WRS test and the CD results shows medium to large effect sizes. In this case, we consider the performance model produces "bad prediction results".

Table 2.5 shows the results of the RegTree models. The bottom row with row name "Baseline" shows the prediction of baseline model. For example, half of the scenarios in DS2 suffer from bad prediction results when using window size as 20, the weighted average as the ensemble strategy and only the ideal runs as the training data.

Overall, we can see the results from our ensemble models are the same or better than the baseline. Among the three ensemble strategies, winner-takes-all always outperforms the other two under the same window size and training dataset. If we only compare within the winner-takes-all strategy, the performance is similar to the baseline model when simply using the ideal runs as the training dataset. Except for two cases in DS2, adding more runs into the training dataset would generally get better the prediction results. This makes sense, as more runs will provide more variations of the SUT under variations to the SDE. In consequence, the ensemble model has a better chance to find similar time windows. Comparing among different window sizes, Win_{15} and Win_{20} generally perform the same or better than Win_{10}.

We have also experimented the ensemble-based modeling techniques using the GLM and the MARS as our modeling techniques. We analyze RegTree results for the purpose of illustrating the effectiveness of ensemble modeling approach compared with baseline approach. Please refer to [11], if you want the evaluation results for the GLM and the MARS-based ensemble models.

Table 2.5: Percentage of scenarios which are different between the predicted results and the actual measurements. To conserve space, the three model selection techniques, mean, winner-takes-all, and weighted average, are shown as “Mean”, “Winner” and “Weighted” in the table. Win_{10}, Win_{15} and Win_{20} refer to the three different training window sizes.

Window size	**Ensemble strategy**	**DS2**			**PET**			**JMS**		
		Ideal	Lab	All	Ideal	Lab	All	Ideal	Lab	All
Win_{10}	Mean	58.33%	41.67%	41.67%	51.11%	33.33%	84.44%	100.00%	66.67%	66.67%
	Winner	66.67%	16.67%	33.33%	55.56%	6.67%	4.44%	100.00%	33.33%	22.22%
	Weighted	58.33%	41.67%	41.67%	51.11%	15.56%	66.67%	100.00%	66.67%	66.67%
Win_{15}	Mean	58.33%	41.67%	41.67%	51.11%	33.33%	84.44%	100.00%	66.67%	66.67%
	Winner	66.67%	33.33%	16.67%	48.89%	6.67%	6.67%	100.00%	5.56%	0.00%
	Weighted	58.33%	41.67%	41.67%	51.11%	22.22%	75.56%	100.00%	66.67%	66.67%
Win_{20}	Mean	50.00%	41.67%	41.67%	44.44%	13.33%	51.11%	100.00%	66.67%	66.67%
	Winner	66.67%	16.67%	25.00%	64.44%	4.44%	6.67%	100.00%	72.22%	27.78%
	Weighted	50.00%	41.67%	41.67%	44.44%	11.11%	44.44%	100.00%	61.11%	66.67%
Baseline		(	50.00%	)	(	28.89%	)	(	94.44%	)

2.6.3 Summary

Findings: When predicting on realistic runs, ensemble-based performance models perform the same or better than the single models trained only using the ideal runs. Adding more results from the environment-variation-based tests would generally improve the model performance. The model built using the larger window sizes (e.g., 15 or 20 minutes window sizes) are usually better than the models using smaller window sizes.

Implications: We can effectively model the performance behavior under realistic changes to the SDE using the results from the environment-variation-based tests. Adding more variety of environment-variation-based load tests would improve the model performance. Hence, this motivates the need of further research into other kinds of environment-variation-based load tests (e.g., stressing the database component at various levels).

2.7 Threats To Validity

In this section, we will discuss the threats to validity.

2.7.1 Construct Validity

Performance Monitoring

We monitor the SUT's resource usage using an open source performance monitoring tool, called pidstat. This tool is very low overhead ($< 1\%$ CPU, $< 0.1\%$ memory, and very few disk writes). In addition, we also collect the logs from JMeter and Tomcat for workload and response time information. JMeter and the SUT are run on different machines to avoid resource contention. The access logs for Tomcat are enabled by default.

Evaluation Method

In this chapter, we use the WC method to check whether the predicted results match with the actual measurements. Mean Percentage Absolute Error (MAPE) is also a popular method to evaluate the quality of the predicted results [42]. Equation 3.1 shows the formula of how MAPE can be calculated. The higher the MAPE values, the worse the prediction models are.

$$MAPE = \frac{1}{n}\sum\nolimits_{i=1}^{n}\left|\frac{predicted_i - actual_i}{actual_i}\right| \tag{2.1}$$

In this chapter, we use the WC method instead. We flag a prediction model to be bad if the WRS test shows statistically significant and CD test shows medium

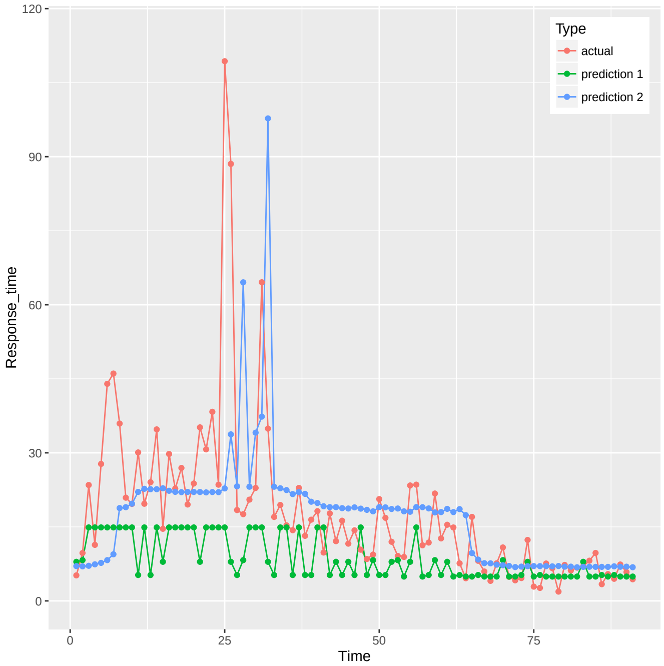

Figure 2.3: Comparison of the prediction results from two performance models.

to large effect sizes between the predicted and the actual measured values. We have originally tried both the WC and MAPE as our evaluation method and have found in certain scenarios MAPE values cannot highlight the strength and weaknesses of different models. Figure 2.3 shows one such example.

Prediction 1 (the model trained with the ideal runs) and prediction 2 (the ensemble-based model trained with more tests) have MAPE values of 0.452 and 0.507, respectively. This indicates that the baseline model is better than the ensemble-based model. However by looking at the graph, the baseline model did not accurately capture the trend of the data. The reason of the baseline model has a smaller MAPE is because the baseline model always gives a small prediction value than the ensemble model. When we use the WC analysis on these data, the CD values for the baseline and the ensemble models are 0.435 (medium effect sizes) and 0. This clearly shows that the prediction results from the baseline model is worse than the ensemble model's results.

2.7.2 Internal Validity

In order to make sure different runs of load tests do not affect each other, we cleaned up the system environment every time after each single test was finished. There are various configurable parameters in our ensemble-based models: the ensemble strategy, the training window size, test window size, and the type of training data. In RQ3 we have kept the test window size fixed (10 minutes) and systematically assess the impact of all three other configuration parameters.

2.7.3 External Validity

In this chapter, we assess the performance impact of environment changes using three open source systems. We picked these three open source systems because they are from different application domains, implemented with different technology, and have different deployment topologies. However, our findings may not be generalizable to other systems. For example, for large-scale industrial systems which have hundreds of scenarios and thousands of customer deployment sites, our existing environment-variation-based load tests may not be able to accurately capture and model the SUT behavior. In addition, although we have proposed various kinds of environment-variation-based load tests (e.g., realistic runs and lab stress runs), there can be more. For example, if the SUT is deployed in a multi-tenant environment, another type of realistic load test can be running the SUT while the database is doing additional unrelated processing tasks.

2.8 Conclusion and Future Work

In this chapter, we have evaluated the impact of the SDE variations on the results of load tests. We ran over 80 hours test on three different target systems with different variations of SDE. Our analysis shows that there is a clear performance impact on the performance of the SUT due to the changes in the SDE. However, not all scenarios react the same to the changes of the computing resources (e.g., CPU, memory and disk). In order to accurately predict the behaviour of SUT under realistic SDE changes, we have proposed an ensemble-based modeling technique, which leverages the results of existing environment-variation-based load tests. Case studies show that our ensemble-based models out-perform the baseline models when predicting the performance of the realistic runs.

In the future, we plan to extend this study by proposing and experimenting more types of environment-variation-based load tests, both in lab stress runs as well as realistic runs. In addition, we also want to research into techniques which dynamically adjust the test environment (e.g., adding or removing the computing resources) based on the performance of the system.

Once the load tests are executed, performance analysts need to go through the generated logs and counters data to find potential load-related problems. The next chapter conducts an empirical analysis to evaluate among the various load test analysis techniques.

3 A Framework to Evaluate the Effectiveness of Different Load Testing Analysis Techniques

One of the goal for performance testing is to detect performance deviations between current version and previous version. In this chapter, we will introduce how a load testing is done, why we need performance analysis techniques and how to compare them.

3.1 Introduction

Many large-scale software systems (e.g., BlackBerry's telecommunication systems [6] and Microsoft's web services [14]) are used by thousands and millions of users everyday. Study finds that most failures in the field are due to systems not able to scale rather than feature bugs [43]. The failure to scale could result in catastrophic consequences and unfavorable media coverage (e.g., the botched launch of Apple's MobileMe [44] and US Government's Health Care Website [2]). Hence, to ensure the quality of the system in the field, load testing is a required testing procedure in addition to the conventional functional testing (e.g., unit and system integration testing).

Load testing is usually conducted after all the functional testing is completed [13]. It uses one or more load generators, which mimic hundreds or thousands of real users accessing the system at the same time (called the *load*). During the course of the load testing, there are two types of system behavior data generated: execution logs and counters. Execution logs are generated by debug statements (e.g., printf and System.out.println) that developers insert into the code. Counters record the workload and the resource usage (e.g., CPU, memory and disk utilization) of system during runtime. Since the goal of a load test is to assess the system behavior under load, the duration of a load test can be a few hours (e.g., 8 hours for a working day) or even weeks. Hence, the volume of the generated system behavior data can be huge. Unfortunately, analyzing the results of a load test is challenging due to the following three reasons:

1. **Lack of Test Oracle:** Unlike functional testing, which has clear pass and fail criteria, the objectives of a load test are not clear [4] and are often defined later on a case-by-case basis [5].
2. **Large Volume of Data:** A load test can generate hundred megabytes or even terabytes of system behavior data. It would be impossible for load testing practitioners to analyze them manually.

3. **Limited Time:** Load testing is usually the last step in the already delayed software development cycle. The time allocated for test analysis can be very limited.

To cope with the above challenges, there have been a few techniques proposed to automatically analyze the results of a load test (e.g., [6, 7, 8, 9]). These techniques automatically derive the performance models from the previous test(s) using various approaches (e.g., queuing theory or data mining) and check whether the behavior of the current test matches with the previous test(s). However, none of the aforementioned works compared the effectiveness of their techniques against others nor provided replication packages. We suspect this is mainly due to two reasons: (1) most of the studies are conducted on commercial systems, whose test results cannot be made available to the public due to confidentiality concerns; and (2) certain details are missing. For example, many tools described in these works are not publicity available to download. In addition, many techniques impose threshold values. For example, the CPU utilizations in the current test maybe anomalous, as they are 30% above the predicted values. However, the exact threshold values are either not disclosed or should be tunable based on different systems. In this chapter, we have re-implemented and systematically evaluated the effectiveness of 23 different test analysis techniques. The contributions of this chapter are:

1. This is the first study, which systematically evaluates the effectiveness of

different test analysis techniques used in a load test.

2. Our evaluation framework is very flexible. It evaluates different test analysis techniques on a range of threshold values. In addition, it can easily integrate new test analysis techniques or new test data.
3. The test data used in this chapter is made available online [12] to encourage further research on this topic. Our data is a collection of system behavior data generated by load tests executed on three open source systems. For each system, load testing is conducted both on "buggy" versions, which contain manually injected or real-world performance bugs, and on "healthy" versions. Such data takes a long time to prepare and is rarely available.

Chapter Organization

The rest of the chapter is organized as follows: section 3.2 provides some background knowledge about different performance modeling and anomaly detection techniques. Section 3.3 describes our evaluation framework. Section 3.4 explains the case study setup and proposes three research questions. Sections 3.5, 3.6, 3.7 investigate the three research questions and present the results. Section 3.8 discusses the threats to validity. Section 3.9 concludes the chapter and discusses some future work.

3.2 Background

Different from functional testing, which has clear pass and fail criteria, there is usually no test oracle available for a load test [5]. The pass/fail criteria of a load test are usually derived based on the "no-worse-than-before" principle. This principle states that the performance of the current version should be at least as good as the previous version(s). In this section, we will briefly explain various test analysis techniques used in a load test. This chapter focuses on the automated test analysis techniques using counters. For a survey on automated test techniques using execution logs, please refer to [3].

Each test analysis technique can be further broken down in two parts: (1) *the performance modeling* part (section 3.2.1, in which the system behavior from the past test(s) is summarized into models; and (2) *the anomaly detection* part (section 3.2.2), in which the anomalous behavior in the current test is flagged.

3.2.1 Performance Modeling

Table 3.1 provides an overview of eight different performance modeling methods. We have divided them into three categories: queuing theory-based methods, data mining-based methods and rule-based methods. For each modeling method, we also list their assumptions and limitations, as well as the data used in the input

Table 3.1: Performance Modeling Techniques

Category	Techniques	Assumptions	Input	Output	Limitations
Queuing models	LQN Model [7]	System is stable	Workload mix, topology information	Utilization for hardware and software, response time and throughput	Needs topology information
Data mining models	Multiple Linear Regression [8]	Normality, Independence, Linearity, Homoscedasticity	Any counters	Any counters	Assumptions
	MARS [40]	N/A	Any counters	Any counters	N/A
	Quantile Regression [45]	N/A	Any counters	Any counters	Needs to specify quantile
	Regression Tree [41]	N/A	Any counters	Any counters	N/A
Rule-based models	Control Chart [6]	Normality, Linearity, Independence	Any counters	Any counters	N/A
	Descriptive Statistics [9]	N/A	Any counters	Any counters	N/A
	Load Profile [46]	System is stable	Workload mix, topology information	Service demand for each scenario	Needs topology information

and output.

Queuing Theory-based Methods

Queuing models apply queuing theory [47] to model the performance behavior of a system. Queuing models mimic system resources (e.g., CPU and memory) as queues, predict the time that different scenarios spend on different queues and estimates the resource utilizations and throughput information. Each scenario corresponds to one type of workload request (e.g., browsing a page or purchasing an item). For each request on each resource, there are two types of timing information that queuing theory tracks and predicts: waiting time and processing time. The waiting time, also called the queuing time, is the time that each request waits in this resource queue to be serviced. The service time, also called the *service demand (SD)*, is the processing time that each request spends on this resource. The overall response time for one request is the combination of the service time and queuing time from all the resources. The resource utilizations and throughput can also be estimated based on the SDs. Since queuing models require the knowledge of the system internals (e.g., the deployment topology, the queues and locks involved in a scenario, and whether the call is synchronous or asynchronous) to properly model the performance behavior, they are also called *white-box models* [48].

In addition to hardware resources, there are also various software resources (e.g., database or middleware) in a system. The single layer queuing models can only be used to model the performance of hardware resources. To model the performance of both hardware and software resources, Layered Queuing Models (LQN) are required [7].

Data Mining-based Methods

Compared to LQN, data mining models treat the system under test as a black box and do not require the knowledge of the system internals. Hence, they are also called *black-box models* [48]. Data mining models rely on various statistical-based and AI-based techniques to model the system's performance behavior. Most of the data mining models used in a load test are regression-based models:

- **Multiple Linear Regression (MLR)**: Linear regression models the system behavior as a linear function of workload mixes and resource usage utilization [8]. As there are certain assumptions associated with linear regression models, extra care is needed to verify these assumptions are met before applying the model. For example, linear regression assumes the input data is normally distributed. If the normality assumption is not met, the input data needs to be transformed (e.g., log transformation) before building the model. In addition, MLR also suffers from confounding problem, which may impact the accuracy of the model. Hence, a set of input variables, which are highly correlated with each other, are removed. This process is called *attribute selection*. In this chapter, we evaluate the following two attribute selection approaches: (1) removing the independent variables which are highly correlated with each other [8]; and (2) removing the independent variables which have low correlations with dependent variables [41]. For MLR applied with the first approach, we abbreviate this method as MLR_D; and MLR_V for MLR applied with the second approach.

- **Regression Tree (RegTree)**: Compared to MLR, in which extra care is required to ensure the input data satisfies the assumptions of the model, RegTree does not have such restrictions [41]. RegTree predicts the system's performance behavior using a tree-liked structure.

- **Multivariate Adaptive Regression Splines (MARS)**: In some cases, a system can undergo different phases during a load test. For example, the performance behavior of a system can be different when the cache is filled up. MARS [40] is proposed to model such behavior (a.k.a., multiple phases), as the simple MLR model cannot perform well in this case. MARS automatically splits the data (a.k.a., phases) and builds different sub-models.

- **Quantile Regression (QUANT)**: Similar to MARS, QUANT [45] can be used to model different phases. Compared to MLR, which imposes equal penalty on overestimation and underestimation, QUANT sets different penalty on overestimation and underestimation based on the given quantiles. These quantiles need to be specified manually in the model. For example, quantile regression built on 25% quantile, denoted as QUANT.25, estimates the relation between input variables and an output variable, on 25% quantile of the output data. In this chapter, we have studied the following QUANT models: QUANT.25, QUANT.50, QUANT.75, QUANT.90, QUANT.100. The reason for studying a range of QUANT models is to assess whether different quantiles would impact the effectiveness of detecting performance problems.

Rule-based Methods

Compared to the queuing theory and data mining-based methods, which predict the performance behavior of the current test using the derived models, there are no predictions involved in rule-based methods. Rather they analyze the system behavior using a set of rules.

- **Descriptive Statistics (DescStats)**: Descriptive statistics like median and mean [9] provide a high level summary of the system performance data. They are the easiest to implement and used widely in practice. In this chapter, we compare the absolute relative difference between the mean of the current and the previous test(s).

- **Control Chart (CtrlChart)**: A control chart has three components: a control line (CL), a lower control limit (LCL) and an upper control limit (UCL). If a point lies outside the controlled regions (above the UCL or below the LCL), the point is counted as a violation [6]. The three components can be defined in various ways. In this chapter, we use the median of the previous test(s) as our CL, the median ± one standard deviation of the previous test(s) as our UCL and LCL. Similar to MLR, certain assumptions need to be satisfied before applying the CtrlChart.

- **Load Profile (LP)**: The main idea behind LP is that among different versions of the system, the SDs of different requests should be the same. Otherwise, there could be performance problems [46]. There are different approaches to estimate the SDs. In this chapter, we use the Kalman filter to estimate the SDs, as it is very flexible (i.e., can work with data from any distributions) and produces high accurate results [49].

3.2.2 Anomaly Detection

In general, the data from the current test is considered anomalous, if it "significantly" deviates from the past test(s). The anomaly detection methods are different based on the three categories of performance modeling methods. The anomaly detection methods for the queuing and data mining-based methods are the same, as these methods predict the outcomes based on the input values from the current test. The predicted outcomes are compared against the actual measured values in the current test. The anomaly detection methods for the rule-based methods are different, as the actual values from the current test are evaluated against some rules.

Anomaly Detection Methods for Queuing and Data Mining-based Methods

In this chapter, the following two statistical methods are studied. Both methods aim to quantify the differences between the predicted and the measured values:

- **Mean Percentage Absolute Error (MAPE)**: MAPE is one of the evaluation methods of model prediction [42]. The formula to calculate MAPE is shown in Formula 3.1 below. For each predicted outcome, the absolute relative errors between each predicted and measured values are calculated. Then all these absolute relative errors are combined to provide an average value.

$$MAPE = \frac{1}{n}\sum\nolimits_{i=1}^{n}\left|\frac{measured_i - actual_i}{actual_i}\right| \tag{3.1}$$

- **Wilcoxon Rank Sum test and Cliff's Delta (WC)**: Wilcoxon Rank Sum (WRS) test is a non-parametric test which compares two distributions. However, in some cases, even if the two distributions are different, the differences between them are small. Hence, we also need to quantify the strength of the differences between the two distributions, called the *effect size*. For example, if the response time is more than 5 minutes and the differences of response time between the two tests are very small (e.g., 1 millisecond), the performance deviation from the new test should not be a big concern. Cliff's

Delta (CD) is a non-parametric technique to calculate the effect size between two distributions [37]. The strength of the differences and the corresponding range of CD values are shown below:

$$\text{effect size} = \begin{cases} \text{trivial} & \text{if } CD < 0.147 \\ \text{small} & \text{if } 0.147 \leq CD < 0.33 \\ \text{medium} & \text{if } 0.33 \leq CD < 0.474 \\ \text{large} & \text{if } 0.474 \leq CD \end{cases}$$

Anomaly Detection Methods for Rule-based Methods

The anomaly detection methods for rule-based methods are mainly based on threshold values. For example, if more than 30% of the CPU utilizations data from the current test are violations (above UCL or below LCL), the CPU metric in the current test is considered anomalous [6].

3.3 An Overview of Our Framework

In this section, we will introduce our framework to evaluate the effectiveness of different test analysis techniques as illustrated in Figure 3.1. Our framework consists of the following five steps: (1) load testing; (2) model building; (3) test oracle derivation; (4) anomaly detection & result evaluation, and (5) ranking. Our frame-

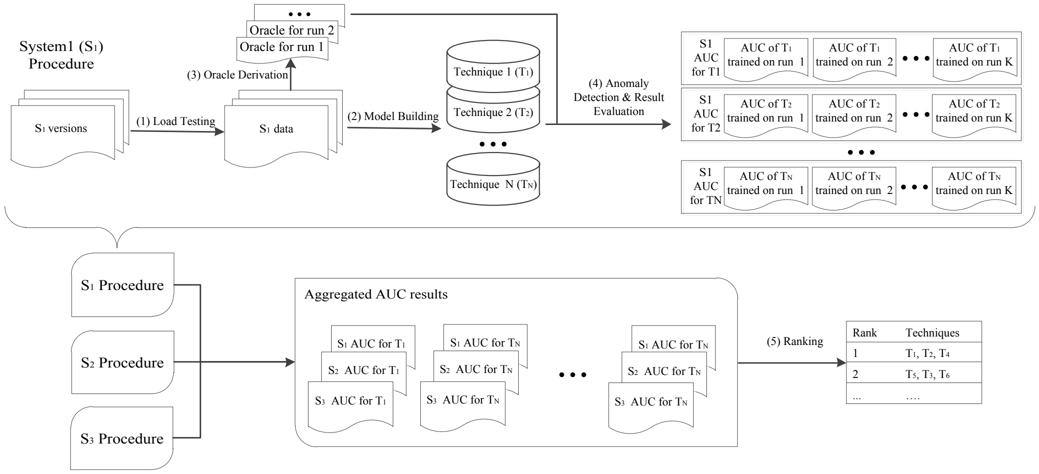

Figure 3.1: An Overview of Our Framework

work is very flexible, as it can easily accommodate new test analysis techniques, new load testing data, and new evaluation systems.

Step 1 - Load Testing

Multiple versions of the same system are load tested. The selected versions to be tested should be a mixture of "good" and "bad" versions. Bad versions contain known performance problems. Good versions refer to the versions of the system whose performance is satisfactory under load. All the selected versions of the same system are load tested using the same load profile. A load profile is characterized along two dimensions: workload mixes and workload intensities. The *workload mix* refers to the ratios among different types of requests (e.g., 30 percent browsing, 10 percent purchasing and 60 percent searching). The *workload intensity* refers to the rate of the incoming requests (e.g., browsing, purchasing and searching), or the number of concurrent users. During the course of each load test, while the system is processing requests generated by the load profile, the system behavior data (a.k.a., logs and counters) is collected. In this chapter, we use the load profile, which generates stable request rate, to test the target systems.

This same load profile is applied on multiple versions of the system to counter potential bias. The test plan will be updated if the system evolves. For example, if the URL of the browsing scenario changed from "brs.jsp" to "browse.jsp" in the next version, the test plan will be updated accordingly.

Step 2 - Model Building

Using the generated system behavior data from the previous step, different performance modeling methods are applied to summarize the performance behavior of the previous test(s). For descriptions of different performance modeling methods, please refer to Section 3.2.1. Additional data processing is carried out to ensure the data is cleaned up (e.g., removing the warm-up and cool-down period) and transformed into the corresponding input file formats (e.g., csv or xml) required by different modeling techniques.

In addition, extra care is taken to ensure the assumptions are met for each model. For example, MLR and CtrlChart require input data to be normally distributed. Hence, we first use Shapiro-Wilk test to verify the normality assumption. If the normality assumption is not met, log transformation is performed on the input data. For QUANT, we choose a range of different quantizes: 25%, 50%, 75%, 90% and 100%, to investigate if models built on different quantiles performed the same . In this chapter, we use the readily available OPERA tool [50] as our model implementation for LQN and LP methods. OPERA automatically estimates the SD using Kalman filter based on the workload mixes, hardware resource utilizations and response time for different scenarios. OPERA models and predicts the system performance using the Mean-Value-Analysis (MVA) algorithm. All the other methods are re-implemented using R.

Step 3 - Test Oracle Derivation

In order to evaluate the effectiveness of different test analysis techniques, a "test oracle", which flags the problematic counters in each run, is needed. We manually derive the test oracle for each run based on bug reports, release notes and close examinations on the counters. For example, if a version of a system injected with a CPU intensive bug, the CPU utilization of this version should be flagged as "buggy". As another example, if the response time for the browsing scenario is much longer in the current version than the previous version(s) and there is a response time issue reported in the bug report for this version, the response time for the browsing scenario is also flagged as “buggy”. Our test oracle has been derived and verified by the first two authors of this chapter.

Step 4 - Anomaly Detection & Model Evaluation

Using the derived performance models in the previous step, different anomaly detection techniques are applied to flag suspicious behavior from the current test. Please refer to Section 3.2.2 for descriptions on different anomaly detection methods. All the anomaly detection methods are re-implemented in R. Different test analysis techniques may use different threshold values to detect anomalous counters:

- **MAPE**: The higher the MAPE values are, the bigger the deviations between the predicted and the measured values in the current test. Ideally, higher MAPE values for one counter indicate higher probability that this counter is anomalous. However, if the MAPE cut-off value is set to be too high, we might miss some problematic counters.

- **WC**: For the WC method, the threshold values are only used when there is a statistical difference between the current and the past test(s). The strength of the differences are quantified by the CD values. The counters with higher CD values, indicating bigger differences, and are more likely to be problematic.

- **CtrlChart**: For CtrlChart, the threshold value corresponds to the violation ratio, which is the percentage of data points in the current test lying outside the control limit (a.k.a., above the UCL or below the LCL). Higher violation ratios indicate bigger differences between the current and past test(s).

- **DescStats**: For DescStats, the threshold value corresponds to the absolute percentage errors between the current test and the previous test(s).

A test analysis technique corresponds to one combination of a modeling technique and an anomaly detection techniques. The predicted values from each performance modeling method are evaluated using all the applicable anomaly detection methods to ensure all valid test analysis techniques are studied. For example, the predicted values from the RegTree are applied using both MAPE and WC. There are ten different data mining and queuing methods (LQN, MLR_D, MLR_R, MARS, RegTree, and 5 different QUANT methods), each of which has two different anomaly detection methods (WC and MAPE). In addition, we have three different rule-based performance modeling methods, each of which has one anomaly detection method. Hence, we have evaluated 23 different testing analysis techniques in total.

We use the Receiver Operating Characteristic (ROC) curve [51] to assess the effectiveness of different test analysis techniques under a range of threshold values.

Once the ROC curve for each test analysis technique is built, the AUC (Area Under the Curve) values are calculated. The AUC values are used in the ranking process below.

Step 5 - Ranking

In this chapter, we use the Scott-Knot test [52] to rank the effectiveness of different test analysis techniques. Scott-Knott test uses hierarchical clustering to group the test analysis techniques. Each group contains techniques with similar performance. The smaller the group number that one technique is assigned to, the more effective this test analysis technique is. For example, if CtrlChart is assigned to Group 1, it means that CtrlChart is more effective than techniques assigned in Groups 2 and 3. Scott-Knot test has been used successfully in other software engineering tasks (e.g., comparing the performance of different bug prediction techniques [53]). The input for the Scott-Knott test is the AUC values from different test analysis techniques exercised on different tests.

There are two reasons for using Scott-Knot test and the AUC values to rank the test analysis techniques. First of all, compared to other techniques (e.g., Nemenyi's test or hierarchical clustering), Scott-Knot test automatically clusters different test analysis techniques into distinctive groups, which are non-overlapping. Second, most test analysis techniques require threshold values, which are either not specified

Table 3.2: Case Study Systems

System	**Domain**	**Category**	**SLOC**
DS2	Benchmark application	JSP	100
Petclinic	E-commerce	Hibernate	> 2000
JMS	Mail Server	Mailet Container	> 4,000

in the chapter, or are tunable based on different projects. AUC values enable us to evaluate the effectiveness of different techniques under a range of threshold values.

3.4 Case Study Setup

We intentionally picked three different open source systems with different deployment setup and application domains: Dell DVD Store (DS2) [27], Petclinic [28] and James Mail Server (JMS) [29] to demonstrate the usefulness of our framework. Each system is developed under different technologies and has different deployment configurations. Table 3.2 shows the details.

In total, about 60 hours of load testing were run on various versions of the three systems. We collected over 150 counters from three target systems, containing resource utilizations (e.g., CPU, memory and disk I/O) for various components of the systems, request rates, throughput and response time for each scenario.

The rest of this section is organized as follows: section 3.4.1 explains the test scenarios as well as the rationales for selecting different versions. Section 3.4.2 describes our test execution and test monitoring process. Section 3.4.3 introduces a few research questions that we want to answer using our framework.

3.4.1 Test Design

DS2

DS2 is an open source e-commerce website, which Dell uses to benchmark the performance of their hardware. The system contains four different JSP pages: login, browsing catalogs, purchasing items and registrations for new customers. It is deployed on the top of a Tomcat web server and is connected to a MySQL database.

The test details are shown in Table 3.3. The original version (v0) does not have any injected or known performance bugs, so we call it "good version". V0 was load tested six times. We injected three performance bugs (a busy CPU bug, a database performance bug, and a constant delay bug) into three versions of DS2 (v1, v2, and v3), respectively. The CPU and the database bugs are from [6]. We have added the delay bug in this study, as it represents another performance issue (a.k.a., blocking v.s. busy waiting). These three versions are called "bad versions". Following the

Table 3.3: Load Testing Details for Three Systems

System	# of Test Scenarios	Versions	# of Test Runs	Injected/Known Bugs or Improvements
DS2	4	v0	6	Original version
		v1	1	Injected a CPU intensive bug in purchasing
		v2	1	Removed indices in two database tables
		v3	1	Created a constant delay in purchasing
PetClinic	15	v0	1	Original version
		v1	1	Changed to stateless and non-blocking I/O to improve request throughput
		v2	6	Removed JVM locks to further improve request throughput
JMS	6	2.3.2	1	Stable release version
		3.0M1	1	Improved the efficiency of queueing emails
		3.0M4	6	Improved the efficiency of handling MIME messages

practice of [8], in which a good version is load tested multiple times and each bad version is tested once. If a load test is executed on a good version, we call it a "good run", and a "bad run" means a load test is executed on a bad version.

PetClinic

Petclinic is a web application written in Java Spring and Hibernate. There are 15 different test scenarios related to pets, pet owners, and veterinarians. Example scenarios include scheduling visits to veterinarians and locating pet owners. Same as DS2, Petclinic deploys on top of a Tomcat web server and connects to a MySQL database.

Different from DS2, in which performance bugs are manually injected, we load tested on versions of systems with known real-world performance issues for PetClinic. In the blog post by Dubois [54], he detailed a series of changes that he has made in order to improve the performance and scalability of PetClinic. All his changes are archived in the GitHub repository [55]. In this chapter, we picked three versions from that GitHub repository. We called the last version as the "good version", as it contains all the fixes that he has performed. The other two older versions as "bad versions", as they contain various performance issues. The descriptions for the performance issues associated with the bad versions are shown in Table 3.3. We executed the load testing on good version 6 times and on each bad version once.

JMS

The Apache James Mail Server (JMS) is an open source, mail server written in Java. It contains two generic scenarios: sending emails and receiving emails. The two generic scenarios can be further divided into many smaller scenarios, like sending and receiving emails with or without attachments, reading only the mail header or loading the whole mail, etc. In total, 6 scenarios are tested for JMS. Different from DS2 and PetClinic, JMS is a stand-alone system, which does not have any additional supporting components.

Unlike Petclinic, we could not find two or more continuous revisions, which contain performance fixes. Hence, we picked three different release versions in our study. Apart from the latest stable version 2.3.2, there were three MR releases for version 3.0. Since 3.0M3 was not available to download in the official website, the other two MRs, 3.0M1 and 3.0M4, are used in this study. The selected three versions have many bug fixes and performance improvements. Between these versions, numerous modules have been upgraded, such as updating to more efficient JavaMail [56] and ActiveMQ versions [57]. Table 3.3 summarizes the various performance bug fixes and improvements from the two released versions. This information is extracted from JMS's bug reports and release notes. Similar to DS2 and PetClinic, since 3.0M4 (the “good version”) is the last version containing most performance fixes, it was load tested 6 times. The other two versions (the “bad versions”) were tested once.

3.4.2 Test Execution and Monitoring

All three systems are deployed on the same machine, which has the following hardware configurations: Intel® Core™ 2 CPU 2.40 GHz, 2 GB memory and a 160 GB 7200 RPM hard drive. We use JMeter [32] as our load generator. JMeter is deployed on a separate machine with the following hardware specifications: Intel® Core™ i7-4790 CPU 3.60 GHz, 16 GB memory and a 2TB 7200 RPM hard drive. The reason for the separate deployment of JMeter and the case study systems is to ensure no overhead caused by the load generator to the case study systems. We executed the load for each system and collected the data. For DS2, the overall request rate is around 35 request per second; for PET, it's around 230 requests per second; for JMS, it's around 10 requests per second. Notice that the request rate may vary among good versions and bad versions, as the performance is impacted in bad versions.

The resource utilizations for the systems during load testing are monitored and recorded using pidstat [33]. In addition, we also archived the logs from JMeter, MySQL and Tomcat at the end of each load test. These logs can be used to extract the workload and response time information.

3.4.3 Research Questions

We propose the following three research questions to evaluate the effectiveness of different test analysis techniques:

- **RQ1 (Build Dependency)**: *Can we use data from a bad run to analyze the results of other tests?*

  In some cases, there could be no "good versions" available for a system, as the system is constantly being fixed and improved until release. Hence, most of the load testing runs could be "bad runs". In this research question, we seek to answer whether we can still leverage the data from the bad runs to effectively analyze the results of a load test. RQ1 is discussed in Section 3.5.

- **RQ2 (History Dependency)**: *Would more historical data help improve the performance of different test analysis techniques?*

  Since all the test analysis techniques compare the current test data against the historical data (a.k.a., past runs). This research question aims to answer whether more historical data would help improve the effectiveness of different test analysis techniques. RQ2 is discussed in Section 3.6.

- **RQ3 (Sampling Interval Dependency)**: *What's the impact of different sampling intervals on detecting performance problems?*

  During the course of a load test, the behavior of the system is monitored. There can be different monitoring granularities. For example, the average CPU utilizations can be sampled at every 5 seconds interval or every 60 seconds interval. Different sampling intervals generate different levels of details and different sizes of data to be analyzed. This research question aims to answer whether more fine-grained sampling intervals would improve the effectiveness of different test analysis techniques. RQ3 is discussed in Section 3.7.

3.5 RQ1 - Build Dependency

In practice, it can be hard to obtain one version of a system which is "bug free" due to the following two main reasons: (1) constant evolution of the system due to bug fixes and feature improvements; and (2) undetected performance bugs due to the challenging nature of analyzing the results of a load test [3]. Hence, in this RQ, we want to investigate whether we can still leverage the data from "bad runs" to effectively analyze the results of current load test.

Approach

We built a set of performance models, which use the data from one good run or one bad run. These performance models are used to evaluate the test results from other test runs. For example, eight LQN techniques are built using the test runs from PetClinic. Among them, four techniques are built using the data from bad versions (v0 and v1 of PetClinic) and the rest four techniques are built using the data from good versions (v2 of PetClinic). Each model is used to predict the results of all the other test runs from PetClinic. Similar process is applied on other techniques and on other systems. The prediction results from different test runs of the same test analysis techniques are grouped. The overall results across all systems are ranked using the Scott-Knot test.

Analysis

Table 3.4 shows the ranking results for different test analysis techniques. The second column shows the list of performance modeling methods. Each cell corresponds to the ranking of one particular test analysis technique, as technique corresponds to one unique combination of a modeling method and an anomaly detection method. The columns marked with "B" and "G" correspond to the techniques built with training data from the good and bad runs, respectively. All the queu-

Table 3.4: Aggregated Scott-Knot Ranking for RQ1

Category	**Techniques**	**Rankings**			
		G		B	
Rule-based Models	CtlChart	1		1	
	DescStats	1		1	
	LP	2		2	
		M_B	M_G	W_B	W_G
Data Mining Models	MLR_D	2	2	2	2
	MLR_V	2	2	2	2
	MARS	2	2	2	2
	QUANT.25	2	2	2	2
	QUANT.50	2	2	2	2
	QUANT.75	2	2	2	2
	QUANT.90	2	2	2	2
	QUANT.100	2	2	2	2
	RegTree	1	1	1	1
Queuing Models	LQN	2	2	2	2

ing and data mining methods are evaluated under two anomaly detection methods: MAPE (marked as “M” in the table) and WC (marked as “W”). Hence, the column “M_B” refers to the modeling techniques trained with data from the bad run and use MAPE as its anomaly detection method.

There are two groups generated by the Scott-Knott test. Among all the test analysis techniques, DescStats, CtrlChart, both RegTree techniques perform better than the other techniques. Among all the test analysis techniques, their rankings are the same regardless of using data from the good or the bad runs. We want to note that some performance models do not summarize the training well. For example, when we apply the LQN model built from the bad runs (e.g., v3 from DS2) to predict the values from the same run (v3), the accuracy is low. However, when the LQN model built from the good runs (e.g., v0 from DS2) to predict the values from the same run (v0), the accuracy is high. However, in general, the accuracy of the modeling methods does not seem to impact the effectiveness of different test analysis techniques. We believe this is because each technique can effectively detect the performance deviations between the good and the bad runs.

Findings: In general, the effectiveness of different test analysis techniques are not impacted by its training data.

Implications: Load testing practitioners can use the test analysis techniques as an effective "diffing" tool to check if there are any performance deviations between the current and the past test(s).

3.6 RQ2 - History Dependency

All the studied load testing analysis techniques compare the data from the current test against the historical data (a.k.a., past runs) to ensure the performance of the current test is not worse-than the past test(s). As there can be many past test(s), the time required for load testing analysis can be increased due to the number of past test(s) that need to be compared. If adding more past tests could provide a significant boost for the effectiveness of detecting performance bugs, we need to know the number of past test(s) required. However, if we can achieve the same effectiveness by analyzing fewer past test(s), we can save the limited test analysis time. The goal of this RQ is to investigate whether more historical data could help improve the effectiveness of different test analysis techniques.

Approach

The previous RQ shows that all the test analysis techniques are effective regardless of building performance models using good or bad runs. In this RQ, we built a set of performance models with varying number of good runs as their training data for each test analysis technique. For example, we combined two different good runs as training data. This process was repeated six times to ensure each good run can appear at least once in one of the six training datasets. The same approach is repeated for combining 3–5 good runs. In the end, there are 30 different training sets for each modeling technique. These models are used to detect performance problems for the other test runs. For example, if the model is built using data from three good runs in JMS, the anomaly detection methods would be applied on the other 3 good runs and 2 bad runs. The same process is applied on all three systems.

Analysis

Table 3.5 shows the ranking results for different test analysis techniques based on varying number of previous tests as their training data. Similar to the previous RQ, we compared 23 different test analysis techniques based on their performance modeling and anomaly detection techniques. The second column corresponds to different performance modeling techniques. The capital letter "R" refers to the number of good runs used in the training data. For example, “1R” means 1 good run is used to build the model, and “2R” means 2 good runs are used to build the model, and so on. Each cell stores the ranking for each test analysis technique. “M” and “W” refer to the MAPE and WC anomaly techniques, respectively. “M_1R” refers to using 1 good run as the training data and MAPE as its anomaly detection method.

As we can see in Table 3.5, there are total five groups. Among all the test analysis techniques, CtrlChart, DescStats and (RegTree + WC) are ranked the top. All the test analysis techniques share similar trends based on the amount of training data: (1) each technique performs at least the same or better giving more previous test(s) as their training data; and (2) the effectiveness of the test analysis techniques seems to be stabilizing after 2R.

Findings: The test analysis techniques perform better by providing more previous tests as their training data. Based on our load testing data, the effectiveness of the test analysis techniques seems to stabilize after using two previous test run(s) as their training data.

Implications: Comparing the current test against multiple previous test runs is time-consuming and might not improve the effectiveness of detecting performance bugs. More research is required to investigate the optimal amount of data required for training the performance models.

Table 3.5: Aggregated Scott-Knot Ranking Results for RQ2

Category	**Techniques**	**Rankings**									
		1R		2R		3R		4R		5R	
Rule-based Models	CtlChart	1		1		1		1		1	
	DescStats	1		1		1		1		1	
	LP	5		5		5		5		5	
		M_1R	M_2R	M_3R	M_4R	M_5R	W_1R	W_2R	W_3R	W_4R	W_5R
Data Mining Models	MLR_D	3	3	3	3	3	3	3	2	2	2
	MLR_V	3	3	3	3	3	3	3	3	3	3
	MARS	3	3	3	3	3	3	3	2	2	2
	QUANT.25	3	3	3	3	3	4	3	3	3	3
	QUANT.50	4	3	3	3	3	4	3	3	3	3
	QUANT.75	4	3	3	3	3	4	3	3	3	3
	QUANT.90	4	3	3	3	3	4	3	3	3	3
	QUANT.100	3	3	3	3	3	3	3	3	3	3
	RegTree	2	2	2	2	2	1	1	1	1	1
Queuing Models	LQN	3	3	3	3	3	3	3	3	3	3

3.7 RQ3 - Sampling Interval Dependency

During the course of the load test, the system behavior is monitored and recorded periodically. For example, the average CPU and memory utilization can be sampled at every 5 seconds, 30 seconds, 60 seconds or more. The sampling interval affects the resulting test data in terms of size and granularity. As the test duration is fixed for one load test, the amount of recorded data decreases when the sampling interval is large. However, it might lose some details. For example, a sudden spike in the CPU could be visible using the 5 seconds sampling interval but might not be visible using the 180 seconds sampling interval. However, using a smaller sampling interval generates larger volumes of data to be analyzed and could slow down the system execution. For example, it is usually not recommended to set the sampling interval for pidstat to be less than 5 seconds [58], as the monitoring overhead is too high and significantly slows down the test execution. Such slow-down could impact the validity of the collected testing data, as the system's performance no longer reflects the actual behavior in the field. In this section, we want to examine the impact of different sampling intervals on the effectiveness of test analysis techniques.

Approach

The load tests conducted for the three systems were monitored using 5 seconds as its sampling interval. For all the systems, we aggregate the testing data into four additional granularities: 30, 60, 180 and 300 seconds to represent the data collected at 30, 60, 180 and 300 seconds sampling intervals. For each sampling interval, multiple performance models, which use the data from one good run, are built. The resulting models are used to analyze the rest of test runs.

Analysis

Table 3.6 shows the ranking results for different test analysis techniques based on the test runs monitored under different sampling intervals. Similar to the previous two RQs, the second column shows different performance modeling techniques. The numbers after the letter "i" indicate the sampling intervals. For example, "i5"

Table 3.6: Aggregated Scott-Knot Ranking Results for RQ3

Category	Technique	Rankings									
		i5		i30		i60		i180		i300	
Rule-based Models	CtlChart	2		1		1		1		1	
	DescStats	1		1		1		1		1	
	LP	4		4		4		4		4	
		M_i5	M_i30	M_i60	M_i180	M_i300	W_i5	W_i30	W_i60	W_i180	W_i300
Data Mining Models	MLR_D	3	2	2	2	2	2	2	2	2	2
	MLR_V	3	2	2	2	2	2	2	2	2	2
	MARS	3	2	2	2	2	2	2	2	2	2
	QUANT.25	3	2	3	2	2	3	3	3	2	2
	QUANT.50	3	2	3	2	2	2	3	3	2	2
	QUANT.75	3	3	3	2	2	3	3	3	2	2
	QUANT.90	3	3	3	2	2	3	3	3	2	2
	QUANT.100	3	3	2	2	2	3	3	2	2	2
	RegTree	2	2	1	1	1	3	1	1	1	1
Queuing Model	LQN	3	2	2	2	2	3	2	2	2	2

means data collected using 5 seconds as its sampling interval, "i30" means data collected using 30 seconds as its sampling interval, and so on. "M_i5" refers to using the data collected at 5 seconds sampling interval as the training data and MAPE as its anomaly detection method.

As we can see in Table 3.6, there are four groups outputted by the Scott-Knott tests. Similar to the previous two RQs, CtrlChart, DescStat and two RegTree techniques rank the best. For each technique, surprisingly, using data from smaller sampling intervals usually performs the same or worse than using data from larger sampling intervals. In general, the test analysis techniques perform the best when using data sampled at every 180 seconds or more. We suspect this is because the data trend is smoother under larger sampling intervals.

Findings: For most of the analysis techniques, using the data collected at smaller sampling intervals yields worse performance than larger sampling intervals. The techniques perform the best when using data sampled at $\geq$ 180 seconds.

Implications: Fine-grained monitoring could bring overhead to the system execution and does not necessarily improve the effectiveness of different test analysis techniques. Larger sampling interval may miss some of the runtime behavior. More research is required to investigate the optimal sampling interval for load testing analysis.

3.8 Threats To Validity

In this section, we will discuss the threats to validity.

3.8.1 Construct Validity

Sampling interval

In this chapter, we evaluated different test analysis techniques under different sampling intervals. All the test runs are monitored using pidstat under the 5 seconds sampling interval. We aggregated the pidstat values in RQ3 to simulate the pidstat data generated under the 30, 60, 180, and 300 seconds interval. We also ran one load test using the 30 seconds sampling interval and compared the resulting data against our aggregated data. The differences are negligible.

The accuracy of the performance data

Each load test was run for a few hours to avoid measurement bias and errors [59].

The resource utilizations are collected using pidstat. The throughput and the response time data from each scenario are extracted either from the JMeter or Tomcat access logs. We cross-checked the data extracted from JMeter and Tomcat and found they would agree most of the time. Hence, we used the data extracted from access logs for DS2 and PetClinic, and used JMeter logs for JMS as JMS doesn't use Tomcat.

3.8.2 Internal Validity

In order to avoid interfere among different load tests, we cleaned up the system environment every time. For MLR and CtrlChart, we performed log transformation to ensure the model assumptions are met before building the model. In addition, to avoid the confounding problem, we also applied two variable section techniques [8, 41] for MLR to reduce the number of input variables. We used the Kalman filter to estimate the SD for different scenarios. The values of SD are used in the LQN and LP models. Compared to other SD estimation techniques, studies show that Kalman filter is the least restrictive (a.k.a., works on data from any distributions) and produces high accurate results [49].

For each RQ, we keep all two of the three factors (good or bad runs, number of past tests, sampling intervals) the same while investigating the impact of one factor in order to avoid the risk of confounding. For example, in RQ1, we keep the number of past tests and the sampling intervals the same, while varying the types of training data (good run or bad run).

3.8.3 External Validity

Most of the previous load testing research is usually conducted on industry systems. Hence, the testing data from such systems are rarely available to download due to confidentiality concerns. In this chapter, we have evaluated different test analysis techniques on multiple versions of three open source systems. These three systems are from different domains (e.g., benchmarking, e-commerce and mail servers) and are implemented using different technologies. In addition, when selecting the versions of the systems to be load tested on, we have used two approaches: (1) for DS2, we manually injected a few performance bugs, which are also used in previous studies (e.g., [6, 8]); (2) for PetClinic and JMS, we have selected the versions which contain real-world performance bugs. Our findings are basically focused on the e-commerce systems and mailing systems under stable load, but we still need to verify if it also applied on the other types of system or other load types. For example, we have applied the stable load profiles for one system. Other systems may be load tested using bursty load profiles. In this case, some test analysis techniques may not perform well. However, our proposed evaluation framework is very flexible, new test data (from the same or different load profiles or systems) can be easily incorporated and compared in the framework.

3.9 Conclusion and Future Work

In this chapter, we have proposed an evaluation framework which systematically compares the effectiveness of different load testing analysis techniques. To demonstrate the usefulness of our framework, we have applied it on the load testing data exercised on 25 runs of three open source systems (DS2, PetClinic and JMS). We have found that all the techniques perform well regardless of training on good or bad test runs. Using more than two runs of previous tests as the training data takes longer time to analyze and may or may not yield better performance. The test analysis techniques are more accurate in terms of detecting performance deviations when building and analyzing data from higher sampling intervals (≥ 180 seconds). Among all the techniques, CtrlChart, DescStats, and RegTree are the most effective ones.

In the future, we plan to extend our evaluation study using load testing data from additional systems as well as incorporating new test analysis techniques. In addition, we also plan to evaluate the test analysis techniques along other dimensions (e.g., usefulness to practitioners).

4 Related Work

Here we discuss the prior research related to the thesis.

4.1 Chaos Engineering

Recently, large companies like Amazon [60], Netflix [10] and Microsoft [61] adopt a type of field testing technique called "Chaos Engineering". In the actual testing, there are system operators or programs (Chaos Monkeys) which randomly turns off some processes or even machines in the production environment. The goal of Chaos Engineering is to assess the system behavior under extreme conditions. As these systems are used by millions of people worldwide, they have very high requirements in terms of reliability and robustness. The system is expected to be still available for service even when some of its processes got terminated or even some machines are crashed.

Our environment-variation-based load testing is different from chaos engineering as they have different goals. Our new load testing technique in chapter 2 tries to assess the system's performance behavior under various realistic changes to the SDE. The goal of chaos engineering is not performance assessment, but rather assessing the reliability or robustness of the SUT.

4.2 Load Testing

In general, a load test can be broken down into three phases: test design, test execution and test analysis. For a survey on various techniques used in a load test, please refer to [3].

In general, there are two types of load profiles that can be used in a load test: *realistic testing loads* and *fault-inducing testing loads*. Realistic testing loads aim to generate load profiles which mimic the actual request rates in the field [9]. This type of load profiles are usually derived from past usage data (e.g., Tomcat access logs). Compared to realistic testing loads, which require repeatedly executing the same scenarios for a long duration of time (hours or days), the fault-inducing load profiles aim to generate load profiles, which likely lead to performance problems. For example, Grechanik et al. [62] propose an iterative technique which automatically learns the system behavior using randomly selected inputs. Then machine learning techniques are applied to generate inputs which lead to performance problems. Zhang et al. [63] use Java PathFinder to automatically generate load profiles, which cause performance problems. They assign a time value for each step along the code path. By summing up the costs for each code path, they can identify the paths that lead to the longest response time. The values that satisfy the path constraints form the load profiles. In chapter 3, we focus on evaluating different test analysis techniques, which analyze the data generated from realistic testing loads.

4.3 Evaluation

Although chapter 3 is the first work which systematically evaluates different load testing analysis techniques, there have been a few prior works which compare the techniques used to solve other software engineering problems. For example, Lutellier et al. [64] compare the accuracy of different architecture recovery techniques. Ghotra et al. [53, 65] compare the performance of different bug prediction techniques. Bellon et al. [66, 67] compare the accuracy of different clone detection techniques. Parnin and Orso [68] empirically evaluate different automated debugging techniques. Lo et al. [69] compare the effectiveness of different fault-localization techniques.

5 Conclusion and Future Work

In this thesis, we have explore the following two areas associated with load testing research: the impact of environment changes in the test environment; and the effectiveness of different test analysis techniques.

In order to evaluate the impact of the SDE changes to the results of load tests, we did over 80 hours tests on three SUT with different variations of SDE. Our analysis shows the difference between SUT with SDE changes and SUT without SDE changes are significant, but different scenarios react differently to the changes of certain computing resources (e.g., CPU, disk and memory). In order to leverage the previous tests to predict the SUT under new SDE changes, we have proposed an ensemble-based modeling technique, and case studies shows that our technique is better than the baseline models. In the future, we want to propose more types of environment-variation-based load tests (e.g., more types of SDE changes). We also want to look into techniques which can dynamically adjust the system environment based on the runtime behavior of the SUT.

To evaluate the effectiveness of different test analysis techniques, we have proposed a framework to compare different load testing analysis techniques under different circumstances. In order to evaluate our framework, we have run 25 load testing runs on three open source systems (DS2, PET and JMS) with different versions. We have found that there is no significant performance difference of techniques built on good test runs and bad test runs. Using two runs data is the best because the performance of the techniques are stable after adding more than two runs of data, while using more data means more time to run tests as well as training the model. Using higher sampling intervals ($\geq$ 180 seconds) to build models yields better results. Generally, CtrlChart, DescStats, and RegTree perform best among all the technique. In the future, we also plan to add additional dimensions to evaluate different test analysis techniques (e.g., model solving time, usefulness to practitioners, etc.). We also want to extend our case study systems in order to verify the generalizability of our current conclusions.

Bibliography

[1] "With Spring Festival Approaching, China's Online Train Ticket System is Still Seriously Broken," https://www.techinasia.com/chinas-12306-online-ticket-purchase-system-falling, visited 2016-10-6.

[2] S. G. Stolberg and M. D. Shear, "Inside the Race to Rescue a Health Care Site, and Obama," 2013, http://www.nytimes.com/2013/12/01/us/politics/inside-the-race-to-rescue-a-health-site-and-obama.html, visited 2015-10-23.

[3] Z. M. Jiang and A. E. Hassan, "A Survey on Load Testing of Large-Scale Software Systems," *IEEE Transactions on Software Engineering*, 2015.

[4] C.-W. Ho, L. Williams, and B. Robinson, "Examining the Relationships between Performance Requirements and "Not a Problem" Defect Reports," in *Proceedings of the 2008 16th IEEE International Requirements Engineering Conference (RE)*, 2008.

[5] A. Avritzer and A. B. Bondi, "Resilience Assessment Based on Performance Testing," in *Resilience Assessment and Evaluation of Computing Systems*, K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, Eds. Springer Berlin Heidelberg, 2012.

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

[7] C. Barna, M. Litoiu, and H. Ghanbari, "Autonomic load-testing framework," in *Proceedings of the 8th ACM international conference on Autonomic computing*, 2011.

[8] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015.

[9] D. A. Menasce, "Load Testing, Benchmarking, And Application Performance Management For The Web," in *Proceedings of the 2002 Computer Management Group Conference (CMG)*, 2002.

[10] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, “Chaos Engineering,” *IEEE Software*, 2016.

[11] “Replication Package,” https://goo.gl/zHFlX5, visited 2016-9-29.

[12] R. Gao, Z. M. J. Jiang, C. Barna, and M. Litoiu, “Replication Package,” https://www.dropbox.com/s/1xzfxc28pf2h284/rep_package.zip?dl=0.

[13] A. Avritzer and E. J. Weyuker, “The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software,” *IEEE Transactions on Software Engineering*, 1995.

[14] M. D. Barros, J. Shiau, C. Shang, K. Gidewall, H. Shi, and J. Forsmann, “Web Services Wind Tunnel: On Performance Testing Large-Scale Stateful Web Services,” in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.

[15] D. Beeby, “Design flaws crashed StatsCan’s census website: documents,” http://www.cbc.ca/news/politics/census-statistics-canada-computers-online-webpage-1.3649989, visited 2016-09-20.

[16] G. Mitchell, “Census 2016: IT experts say Bureau of Statistics should have expected website crash,” http://www.smh.com.au/national/census-

2016-it-experts-say-bureau-of-statistics-should-have-expected-website-crash-20160809-gqosj7.html, visited 2016-09-20.

[17] G. M. Leganza, "The Stress Test Tutorial," in *Proceedings of the 1991 Computer Management Group Conference (CMG)*, 1991.

[18] A. Savoia, "Web Load Test Planning: Predicting how your Web site will respond to stress," *STQE Magazine*, 2001.

[19] J. Zhou, S. Li, Z. Zhang, and Z. Ye, "Position Paper: Cloud-based Performance Testing: Issues and Challenges," in *Proceedings of the 2013 International Workshop on Hot Topics in Cloud Services (HotTopiCS)*, 2013.

[20] M. Yan, H. Sun, X. Wang, and X. Liu, "Building a TaaS Platform for Web Service Load Testing," in *Proceedings of the 2012 IEEE International Conference on Cluster Computing (CLUSTER)*, 2012.

[21] "Shunra Network Virtualization for HP Software," http://media.shunra.com/datasheets/ShunraNVforHP.pdf, visited 2016-09-26.

[22] "BlackBerry fixes critical Enterprise Server flaw," http://www.itpro.co.uk/630034/blackberry-fixes-critical-enterprise-server-flaw, visited 2016-9-12.

[23] "Identifying and resolving performance-related issues caused by the Behavior Monitoring and Device Control," https://success.trendmicro.com/solution/

1056425-identifying-and-resolving-performance-related-issues-caused-by-the-behavior-monitoring-and-device-co, visited 2016-9-12.

[24] A. B. Bondi, "Challenges with Applying Performance Testing Methods for Systems Deployed on Shared Environments with Indeterminate Competing Workloads: Position Paper," in *Companion Publication for ACM/SPEC on International Conference on Performance Engineering (ICPE)*, 2016.

[25] P. Leitner and J. Cito, "Patterns in the Chaos&Mdash;A Study of Performance Variation and Predictability in Public IaaS Clouds," *ACM Transactions on Internet Technology (TOIT)*, 2016.

[26] X. Sun and A. May, "A Comparison of Field-based and Lab-based Experiments to Evaluate User Experience of Personalised Mobile Devices," *Adv. in Hum.-Comp. Int.*, 2013.

[27] "Dell DVD Store Database Test Suite," http://linux.dell.com/dvdstore/, visited 2015-10-23.

[28] "A sample Spring-based application," https://github.com/spring-projects/spring-petclinic, visited 2015-10-23.

[29] "James Project," http://james.apache.org/, visited 2015-10-23.

[30] "Exchange Server 2003 MAPI Messaging Benchmark 3," https://technet.microsoft.com/en-us/library/cc164328%28v=exchg.65%29.aspx?f=255&MSPPError=-2147217396, visited 2015-10-23.

[31] "Playing with Exchange in a Sandbox," https://technet.microsoft.com/en-gb/magazine/2006.08.exchangesandbox.aspx, visited 2015-10-23.

[32] "Apache JMeter," http://jmeter.apache.org/, visited 2015-10-23.

[33] "Performance monitoring tools for Linux," https://github.com/sysstat/sysstat, visited 2015-10-23.

[34] "rsync(1) - Linux man page," http://linux.die.net/man/1/rsync, visited 2016-9-12.

[35] "ClamAV," https://www.clamav.net/, visited 2016-9-12.

[36] "Stress-ng," http://kernel.ubuntu.com/~cking/stress-ng/, visited 2016-9-12.

[37] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?" in *Annual meeting of the Florida Association of Institutional Research*, 2006.

[38] C.-W. Ho, L. Williams, and A. I. Anton, “Improving Performance Requirements Specification from Field Failure Reports,” in *Proceedings of the 2007 15th IEEE International Requirements Engineering Conference (RE)*, 2007.

[39] P. McCullagh and J. A. Nelder, *Generalized linear models*, 1989.

[40] M. Courtois and M. Woodside, “Using Regression Splines for Software Performance Analysis,” in *Proceedings of the 2nd International Workshop on Software and Performance (WOSP)*, 2000.

[41] P. Xiong, C. Pu, X. Zhu, and R. Griffith, “vPerfGuard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments,” in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, 2013.

[42] A. Heiat, “Comparison of artificial neural network and regression models for estimating software development effort,” *Information and software Technology*, 2002.

[43] E. J. Weyuker and F. I. Vokolos, “Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study,” *IEEE Transactions on Software Engineering*, 2000.

[44] "Steve Jobs on MobileMe," http://arstechnica.com/journals/apple.ars/2008/08/05/steve-jobs-on-mobileme-the-full-e-mail, visited 2015-10-23.

[45] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Why you should care about quantile regression," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[46] S. Ghaith, M. Wang, P. Perry, and J. Murphy, "Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems," in *2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*, 2013.

[47] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[48] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing Performance Prediction Robustness by Combining Analytical Modeling and Machine Learning," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2015.

[49] T. Zheng, M. Woodside, and M. Litoiu, "Performance Model Estimation and Tracking Using Optimal Filters," *IEEE Transactions on Software Engineering*, 2008.

[50] M. Litoiu and C. Barna, "A performance evaluation framework for Web applications," *Journal of Software: Evolution and Process*, 2013.

[51] S. Wu and P. Flach, "A scored AUC metric for classifier evaluation and selection," in *Second Workshop on ROC Analysis in ML, Bonn, Germany*, 2005.

[52] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman, "ScottKnott: a package for performing the Scott-Knott clustering algorithm in R," *TEMA (São Carlos)*, 2014.

[53] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[54] J. Dubois, "Spring PetClinic Sample Application," http://blog.ippon.fr/2013/03/11/improving-the-performance-of-the-spring-petclinic-sample-application-part-1-of-5/, visited 2015-10-23.

[55] ——, “Spring PetClinic Sample Application,” https://github.com/jdubois/spring-petclinic, visited 2015-10-23.

[56] “JavaMail,” http://www.oracle.com/technetwork/java/javamail/index.html, visited 2015-10-23.

[57] “Apache ActiveMQ,” http://activemq.apache.org/, visited 2015-10-23.

[58] “Advanced Load Testing Features of Visual Studio Team System,” http://blogs.msdn.com/b/billbar/archive/2006/01/24/advanced-load-testing-features-of-visual-studio-team-system.aspx, visited 2015-10-23.

[59] A. Georges, D. Buytaert, and L. Eeckhout, “Statistically Rigorous Java Performance Evaluation,” 2007.

[60] M. Acharya and V. Kommineni, “Mining Health Models for Performance Monitoring of Services,” in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009.

[61] J. Robbins, K. Krishnan, J. Allspaw, and T. Limoncelli, “Resilience Engineering: Learning to Embrace Failure,” *Queue*, 2012.

[62] M. Grechanik, C. Fu, and Q. Xie, “Automatically Finding Performance Problems with Feedback-directed Learning Software Testing,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[63] P. Zhang, S. Elbaum, and M. B. Dwyer, "Compositional Load Test Generation for Software Pipelines," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[64] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Comparing Software Architecture Recovery Techniques Using Accurate Dependencies," in *Proceedings of the 37th International Conference on Software Engineering (ICSE) - Volume 2*, 2015.

[65] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, 2012.

[66] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *Software Engineering, IEEE Transactions on*, 2007.

[67] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A Mutation Analysis Based Benchmarking Framework for Clone Detectors," in *Proceedings of the 7th International Workshop on Software Clones (IWSC)*, 2013.

[68] C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[69] T.-D. B. Le, D. Lo, and F. Thung, “Should I follow this fault localization tool’s output?” *Empirical Software Engineering*, 2015.