Automated Analysis of Load Testing Results

by

Zhen Ming Jiang

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy in Computer Science

Queen's University

Kingston, Ontario, Canada

January 2013

Abstract

MANY software systems must be load tested to ensure that they can scale up under high load while maintaining functional and non-functional requirements. Studies show that field problems are often related to systems not scaling to field workloads instead of feature bugs. To assure the quality of these systems, load testing is a required testing procedure in addition to conventional functional testing procedures, such as unit and integration testing. Current industrial practices for checking the results of a load test remain ad-hoc, involving high-level manual checks. Few research efforts are devoted to the automated analysis of load testing results, mainly due to the limited access to large scale systems for use as case studies. Approaches for the automated and systematic analysis of load tests are needed, as many services are being offered online to an increasing number of users. This dissertation proposes automated approaches to assess the quality of a system under load by mining some of the recorded load testing data (execution logs). Execution logs, which are readily available yet rarely used, are generated by output statements which developers insert into the source code. Execution logs are hard to parse and analyze automatically due to their free-form structure. We first propose a log abstraction approach that uncovers the internal structure of each log line. Then we propose automated approaches to assess the quality of a system under load by deriving various models (functional, performance and reliability models) from the large set of execution logs. Case studies show that our approaches scale well to large enterprise and open source systems and output high precision results that help load testing practitioners effectively analyze the quality of the system under load.

Dedication

To my dad, mum and Kehan.

Related Publications

The following publications are related to this thesis:

1. **Zhen Ming Jiang**. *Automated Analysis of Load Testing Results*. In Proceedings of the Doctoral Symposium of the 2010 International Conference on Software Testing and Analysis (ISSTA), pages 143-146. Trento, Italy. July, 2010. [Chapter 1]

2. **Zhen Ming Jiang**, Ahmed E. Hassan, Parminder Flora and Gilbert Hamann. *Abstracting Execution Logs to Execution Events for Enterprise Applications*. In Proceedings of the 8th International Conference on Quality Software (QSIC), pages 181-186. Oxford, UK. August, 2008. [Chapter 3]

3. **Zhen Ming Jiang**, Ahmed E. Hassan, Parminder Flora and Gilbert Hamann. *An Automated Approach for Abstracting Execution Logs to Execution Events*. In the Special Issue on "Program Comprehension through Dynamic Analysis" of Wiley's Journal of Software Maintenance and Evolution: Research and Practice, pages 249-267. August, 2008. [Chapter 3]

4. **Zhen Ming Jiang**, Ahmed E. Hassan, Parminder Flora and Gilbert Hamann. *Automatic Identification of Load Testing Problems*. In Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM), pages 307-316. Beijing, China. September, 2008. [Chapter 4]

5. **Zhen Ming Jiang**, Ahmed E. Hassan, Gilbert Hamann and Parminder Flora. *Automatic Performance Analysis of Load Tests*. In Proceedings of the 25th IEEE International

Conference on Software Maintenance (ICSM), pages 125-134. Edmonton, Canada. September, 2009. [Chapter 5]

6. **Zhen Ming Jiang**, Alberto Avritzer, Emad Shihab, Ahmed E. Hassan and Parminder Flora. *An Industrial Case Study on Speeding up User Acceptance Testing by Mining Execution Logs*. In Proceedings of the 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI), pages 131-140. Singapore. June, 2010. [Chapter 6]

Acknowledgments

This thesis would not have been possible without the support of many exceptional people to whom I am grateful.

First of all, I would like to thank my parents, who always believe in me and give me the will to succeed. Thank you for supporting me to study in Canada and for being there every step of the way.

I am greatly indebted to my supervisor Dr. Ahmed E. Hassan for his support and guidance. I met Ahmed when I was an undergraduate research student at the University of Waterloo. Then he co-supervised me for my Master's degree. Ahmed is like a big brother to me. In many occasions, he offered advice and assistance on various aspects of academic and personal life throughout the journey.

I would like to thank my PhD examining committee members: Dr. Saeed Gazor, Dr. Mohammad Zulkernine and Dr. Jerome Rolia for taking time out of their busy schedule to read my thesis and to provide valuable feedback. I would also like to thank Dr. James Cordy and Dr. Patrick Martin for serving on my PhD supervisory committee and providing insightful comments and suggestions.

I would like to thank my collaborators: Dr. Emad Shihab, Dr. Bram Adams, Weiyi Shang, Thanh Nguyen, Haroon Malik, Dharmesh Thakkar, Derek Foo, Dr. Abram Hindle, Dr. Michael Godfrey and Dr. Alberto Avritzer. I also appreciate the help of Mark Syer for proof-reading my depth report, Dr. Stephen Thomas for sharing his thesis template and everyone else from the SAIL lab for their support and encouragement.

I am also very fortunate to work with many great people at the Performance Engineering team in Research In Motion (RIM). Many of the techniques in this thesis were developed to address the daily challenges facing these practitioners. In particular, I want to thank Parminder Flora for providing me with an opportunity to work as an embedded researcher in RIM. I also want to thank Gilbert Hamann, Mohamed Nasser, Terry Green and Denny Chiu for many interesting discussions and feedbacks during the biweekly research meetings.

I want to thank my good friends, who I spent many days and nights working on school projects and have fun: Jian Wang, Yuli Ye, Yang Gao and HaoQing Zhu.

Finally, I am blessed to have a wonderful wife, Kehan Chen. I thank you for your love, support and sacrifice.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

MANY SYSTEMS ranging from e-commerce websites to telecommunication infrastructures must support concurrent access by thousands or millions of users. Studies show that many problems reported in the field are not related to feature bugs, but to systems not scaling well to field workloads [38, 242]. The inability to scale causes catastrophic failures and unfavorable media coverage. Today, there are many instances of service providers not fully load testing or planning their resources carefully for heavy traffic before the launch of their new products (e.g., MobileMe [27]) or new releases (e.g., Firefox website [7]).

The goal of a load test is to assess the quality of a system under load. The objective of analyzing a load test is to assess and quantify the quality of a system under load. The quality assessment involves activities like checking the results of a load test to detect functional and non-functional problems (e.g., performance) or to provide an overall estimate on the system quality under load (e.g., reliability). Analyzing the results of a load test is difficult, due to

the following challenges:

1. **No Documented System Behavior:** Correct and up-to-date documentation of the behavior of a system rarely exists [212].

2. **Monitoring Overhead:** Monitoring or profiling a system has a high overhead on a system and is not suitable for a load test.

3. **Time Pressure:** Load testing is usually the last step in an already delayed release schedule. The time allocated to analyze a test is even more limited, as running a load test usually takes a long time.

4. **Large Volume of Data:** A load test records metrics and logs that are usually hundreds of megabytes or even larger. This data must be analyzed thoroughly to uncover any problems in the load test making in-depth manual analysis not feasible.

Below we briefly describe the related research and practices on load testing:

1.1.1 State of Research on the Analysis of Load Testing Results

Unlike functional testing (e.g., unit testing or integration testing), which focuses on testing a system based on a small number of users, load testing studies the behavior of a system by simulating thousands or millions of users performing tasks at the same time. A typical load test uses one or more load generators that simultaneously send requests to the system under test. A load test can last from several hours to a few days, during which system behavior data like execution logs and various metrics are collected. Execution logs record software activities (e.g., "User authentication successful") and errors (e.g., "Fail to retrieve customer profile"). Execution logs are generated by debug statements that developers insert into the source code to record the run time behavior of the system under test. Metrics can be functional-related (e.g., number of passed/failed requests) or performance-related (e.g., resource usage information like CPU utilization, memory, disk I/O and network traffic or

the end-to-end response time). Some metrics are collected periodically by monitoring tools like PerfMon [21] (for the resource usage metrics). Some other metrics are collected at the end of the tests (e.g., number of passed or failed requests).

A load test consists of three phases: (1) designing a proper load, (2) executing a load test, and (3) analyzing the results of a load test. Most existing load testing research focuses on the first two phases [153]. There is very few work on systematic approaches to automatically analyze the load testing data for assessing the quality of a system under load. Yet, load testing analysis is an important problem, as many field problems are load-related [242]. It would be preferable that load-related problems can be caught early on (i.e., before the system is deployed in the field).

We believe that the current limited research on load testing analysis is mainly due to two reasons: first, there is limited access to large scale multi-user systems and load testing infrastructure, as many of such systems are developed in-house in commercial settings. Second, scalability is not as big of a concern for most prototype systems developed by researchers. However, as more and more services are offered in the cloud for millions of users, research on the analysis of load testing results has become essential.

1.1.2 State of Industrial Practices on the Analysis of Load Testing Results

Current industrial practices for load testing analysis remain ad-hoc, involving high-level manual checks due to the large volume of resulting system behavior data (execution logs and metrics).

– **Analyzing Functional Behavior Under Load**

  Load testing practitioners first check whether the system has crashed, restarted or hung during the load test. Then, they perform a more in-depth analysis by grepping through the log files for specific keywords like “failure” or “error”. Load testing practitioners analyze the context of the matched log lines to determine whether these lines

indicate problems or not.

There are two limitations in current practices for checking functional correctness: First, not all log lines containing terms like “error” or “failure” are worth investigating. A log such as “Failure to locate item in the cache” is likely not a bug. Second, not all errors are indicated in the log file using the terms “error” or “failure”. For example, even though the log line “Internal queue is full” does not contain the words “error” or “failure”, it might be worthwhile investigating, as newly arriving items are possibly being dropped.

– **Analyzing Non-Functional Behavior Under Load**

The main focus of non-functional quality-related analysis is to ensure that the system performance under load is satisfactory. Load testing practitioners first use domain knowledge to check the average response time of a few key scenarios. Then, load testing practitioners examine performance metrics for specific patterns (e.g., memory leaks). Finally, they compare these performance data with prior releases to assess whether there are significant increases in the utilization of system resources.

We believe that current practices of non-functional analysis are not efficient, since it takes hours of manual analysis. Current practices are sufficient either for the following three reasons: First, checking the average response time does not provide a complete picture of the end user experience, as it is not clear how the response time evolves over time or how response time varies according to load. Second, merely reporting symptoms like “system is slowing down” or “high resource utilization” does not provide enough context for developers to reproduce and diagnose the problems. Third, other non-functional quality-related aspects (e.g., reliability) are not assessed.

1.2 Research Hypothesis

Two types of artifacts are recorded during a load test: execution logs and metrics. In this thesis, we focus on mining the collected execution logs to assess the quality of a system under load. We focus on execution logs rather than metrics for two reasons:

1. **Availability:** Execution logs are widely available both in the testing and field environment for large enterprise systems, as logs are used to support problem diagnosis [124], remote issue resolution [250] and to cope with legal acts such as the "Sarbanes-Oxley Act of 2002" [23].

2. **Lack of Research:** Compared to the work done in automated analysis of metrics from load tests [115, 185, 186, 187, 210], there is little work dedicated to analyzing execution logs to uncover load-related problems [153].

Our underlying research hypothesis is as follows:

> *Historical load test repositories, which consists of execution logs from prior load testing, contain valuable, readily available and rarely explored information for the effective and automated analysis of load test results.*

The goal of this thesis is to show the validity of this hypothesis through studying historical load test repositories to assess the quality of a system under load. This thesis will be useful for load testing practitioners and software engineering researchers with interest in testing large scale software systems.

1.3 Thesis Overview

In this section, we present an overview of the works presented in this thesis. This thesis has five main chapters. Since each chapter is geared towards a specific problem, we aim

to make each chapter self-contained, so that readers can read each chapter independently based on their interests. Therefore, despite our efforts to minimize the repetitions in each chapter, some repetition may exist between the various chapters. Related works to each specific problem are examined in the corresponding chapter of the thesis.

– **Chapter 2: A Survey on Load Testing Large Scale Software Systems**

  Unlike many other software testing mechanisms which focus on testing the system based on a small number (one or two) of users; load testing examines the system's behavior based on concurrent access by a large number (thousands or millions) of users. In this chapter, we survey the state of the art literature on load testing research and practice. We compare and contrast current techniques used in the three phases of a load test: (1) designing a proper load, (2) executing a load test, and (3) analyzing the results of a load test. We find that very little test analysis work has been proposed, especially on automated analysis of large volume of load testing data to assess the quality of a system under load.

  Based on this finding, we explore our research hypothesis in the following four chapters as illustrated in Figure 1.1.

– **Chapter 3: Automated Abstraction of Execution Logs**

  Execution logs, generated by output statements that developers insert into the source code, usually use ad-hoc non-standardized logging formats. Automated analysis of such logs is complex due to the loosely-defined structure and a large non-standardized vocabulary of words. The large volume of logs, produced by large software systems, limits the usefulness of manual analysis techniques. Thus, automated techniques are needed to uncover the structure of execution logs. Using the uncovered structure, sophisticated analysis (e.g., statistical or artificial intelligence based analysis), which usually operates on the abstracted data, can be performed. In this chapter, we propose

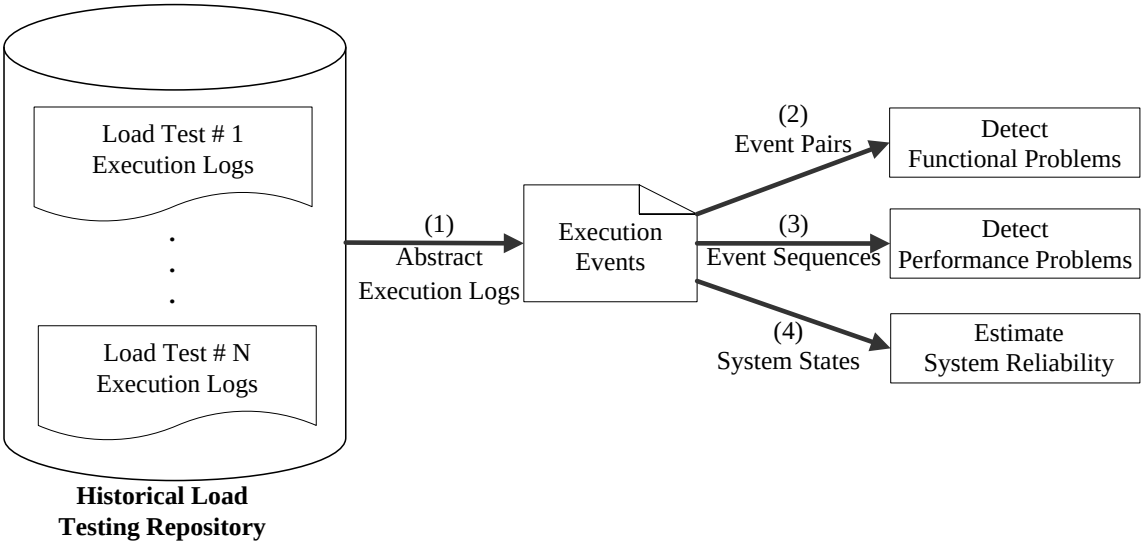


Figure 1.1: Our Approach for Automated Analysis of Load Testing Results

a log abstraction technique that recognizes the execution event of each log line. Using the recovered execution events, log lines can be easily summarized and categorized to help comprehend and investigate the complex behavior of large software applications.

As a system handles concurrent client requests, log lines from different scenarios are intermixed with each other in the execution logs. In the next three chapters, we use different abstract representations (i.e., event pairs, event sequences and system states) for scenarios formed by the execution events to assess various aspects of the system quality under load (i.e., functional, performance and reliability).

– **Chapter 4: Automated Detection of Functional Problems**

Functional problems in a load test can be caused by problems in the load environment, the load generators, or the application under test. It is important to identify and address these problems to ensure that load testing results are correct and that these problems are resolved before field deployment. It is difficult to detect functional problems in a load test due to the large amount of data that must be examined. Current

industrial practices mainly involve time-consuming manual checks which, for example, grep the logs of the application for error messages. In this chapter, we present an approach, which mines the execution logs of an application to uncover the dominant behavior (i.e., the expected behavior) for the application and flags anomalies (i.e., deviations) from the dominant behavior.

– **Chapter 5: Automated Detection of Performance Problems**

Performance problems in a load test refer to the situations where a system suffers from unexpectedly high response time or low throughput. It is difficult to detect performance problems in a load test due to limited in-depth knowledge of system behavior, the absence of formally-defined performance objectives and the large amount of data that must be examined. In addition, we cannot derive the dominant performance behavior from just one load test as we did in the previous chapter: while the non-constant load would still lead to consistent sequences of events, it does not lead to consistent response time throughput a test. A typical workload, which is applied across load tests, usually consists of periods simulating peak usage and periods simulating off-hours usage. In this chapter, we present an approach, which automatically analyzes the execution logs of a load test by systematically comparing against prior tests in order to uncover performance problems.

– **Chapter 6: Automated Estimation of System Reliability**

In previous chapters, we analyze the results of load tests to detect functional and non-functional problems. In this chapter, we use data from several prior runs to obtain an overall quality index. Software reliability is defined as the probability of failure-free operation for a period of time, under certain conditions. The reliability under field load is an important concern for the deployment of large scale mission-critical systems. Reliability estimates must be produced whenever a new version of

a mission-critical system is released. These reliability estimates help customers determine whether a system meets their reliability requirements. However, the general software reliability estimates provided are usually derived from synthetic benchmark workload runs, which is not representative of the actual field usage. An accurate field reliability estimate needs to incorporate data from the actual customer field usage and the system failure scenarios. In this chapter, we propose an approach, which estimates system reliability based on mining the repository of execution logs.

1.4 Thesis Contributions

The major contributions of this thesis are:

1. **A Systematic Survey of the State of Research in Load Testing**

   We present a survey of the state of the art literature on load testing research. To the best of our knowledge, we are the first to systemically compare and contrast the techniques used in all three phases of a load test: (1) designing a load test, (2) executing a load test, and (3) analyzing the results of a load test.

2. **Approaches for Abstracting Execution Logs**

   The abstracted execution events form the basis of enabling the automated analysis of execution logs. Our log event abstraction and sequence recovery techniques can also be used for other types of research like mining customer profiles [231], or analyzing the evolution of software systems [220].

3. **A General Methodology for the Automated Analysis of the Quality of a System under Load**

   We propose methods to infer functional, performance and reliability models from the large volume of load testing data. These methods can be used to systematically assess

the quality of a system under load. Some of our research results (Chapters 3, 4, and 5) are already adopted in practice.

1.5 Thesis Organization

This thesis is organized as follows: Chapter 2 presents our survey on load testing large scale software systems. Chapter 3 explains our log abstraction approach. Chapters 4 and 5 describe our automated approaches for detecting functional and performance problems under load, respectively. Chapter 6 explains our automated approach for estimating the reliability of a system. Chapter 7 concludes the thesis and outlines avenues for future work.

CHAPTER 2

A Survey on Load Testing Large Scale Software Systems

Many software systems must service thousands or millions of concurrent requests. These systems must be load tested to ensure that they can function correctly under load (the rate of the requests submitted to a system). In this chapter, we survey the state of the art literature on load testing research. We compare and contrast current techniques used in the three phases of a load test: (1) designing a proper load, (2) executing a load test, and (3) analyzing the results of a load test. We find that very little work is done on the analysis of load testing results, especially on automated analysis of large volume of load testing data to assess the quality of a system under load.

2.1 Introduction

MANY SYSTEMS ranging from e-commerce websites to telecommunication infrastructures must support concurrent access from thousands or millions of users. Studies show that failures in these systems tend to be caused by inability to scale to meet user demands, as opposed to feature bugs [38, 242]. The failure to scale results in catastrophic failures and unfavorable media coverage (e.g., the meltdown of the Firefox website [7] and the botched launch of Apple's MobileMe [27]). To ensure the quality of these systems, load testing is a required testing procedure in addition to conventional functional testing procedures, like the unit testing and integration testing.

A large scale software system contains thousands or millions of lines of code and possibly

many commercial off-the-shelf components interacting with each other. It is challenging to ensure the quality of such systems, and careful and rigorous cycles of software testing (e.g., unit testing, integration testing, configuration testing and load testing) are required.

This chapter surveys the state of the art literature in load testing research. This survey will be useful for load testing practitioners and software engineering researchers with interests in testing and analyzing large scale software systems. Unlike functional testing, where we have a clear objective (pass/fail criteria), load testing can have one or more functional and non-functional objectives. Based on the typical three phases of load testing illustrated in Figure 2.1, we proposed the following three research questions:

1. **How is a proper load designed?**

   The *Load Design* phase defines the load that will be placed on the system during testing based on the test objectives (e.g., detecting functional and performance problems under load). There are two main schools of load designs: (1) designing a realistic load, which simulates a load that may occur in the field; or (2) designing a fault-inducing load, which may expose load-related problems. Once the load is designed, some optimization and reduction techniques could be applied to further improve various aspects of the load (e.g., reducing the duration of a load test). In this research question, we will discuss various load test case design techniques and explore load design optimization and reduction techniques.

2. **How is a load test executed?**

   In this research question, we will explore the techniques and practices used in the *Load Test Execution* phase. There are three different test execution approaches: (1) using live-users to manually generate load, (2) using load drivers to automatically generate and terminate load, and (3) deploying and executing the load test on special platforms (e.g., a platform which enables deterministic test executions). The Load Test Execution phase can be further broken down into three aspects: (1) setup, which

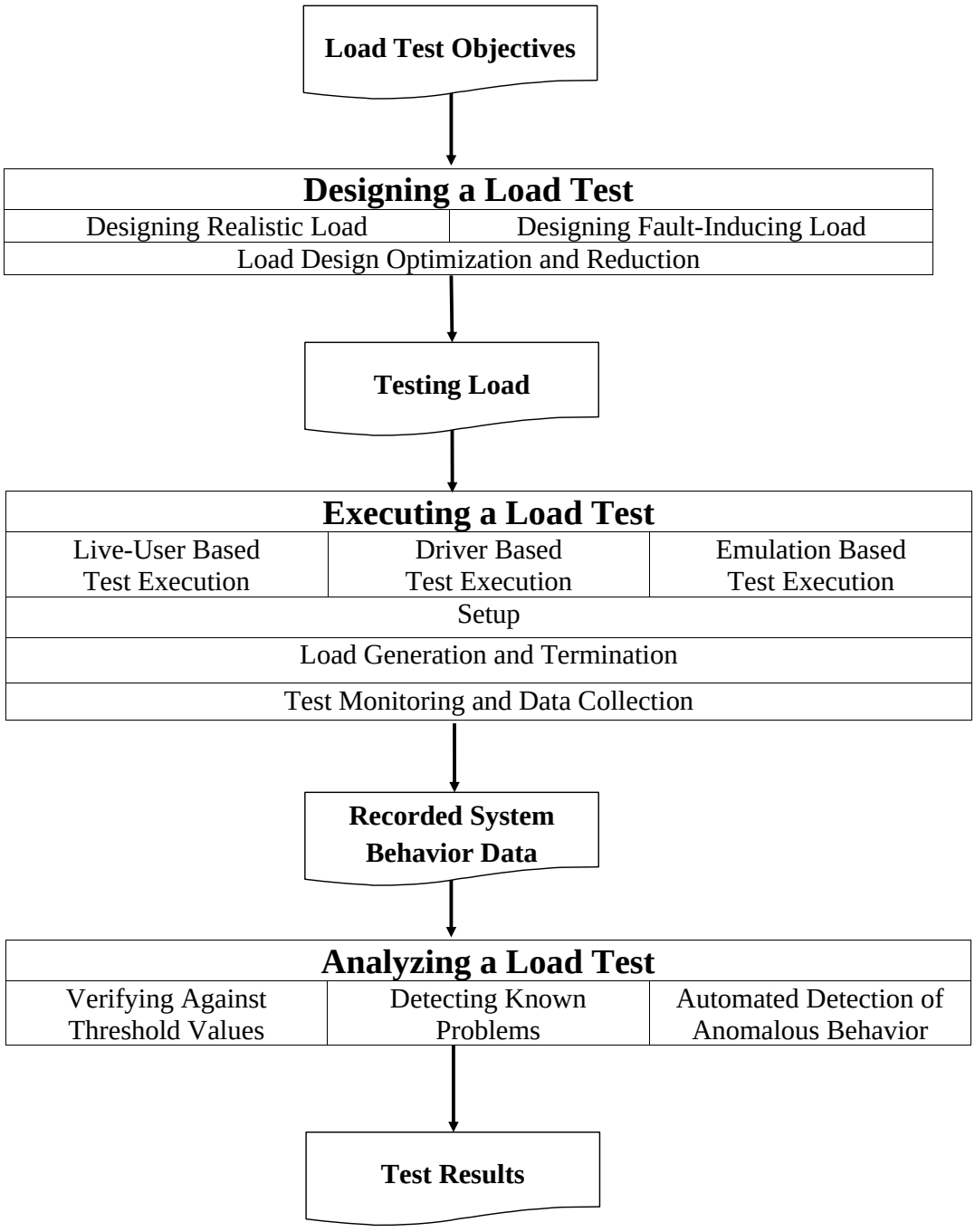


Figure 2.1: Load Testing Process

includes deploying the system and configuring the test infrastructure and the test environment, (2) generating the load and terminating the load once the test is completed, and (3) monitoring the test and recording system behavior to be analyzed in the test analysis phase. The three load test execution approaches share some commonalities and differences in the aforementioned three aspects.

3. **How are the results of a load test analyzed?**

   The system behavior recorded during the test execution phase needs to be analyzed to determine if there are any load-related functional or non-functional problems in the *Load Test Analysis* phase. There are three general load test analysis approaches: (1) verifying against known thresholds (e.g., detecting system reliability problems), (2) checking for known problems (e.g., memory leak detection), and (3) inferring anomalous system behavior.

The structure of this chapter is organized as follows: Section 2.2 provides a background about load testing. Then, based on the flow of a load test, we discuss the techniques used in designing a load (Section 2.3), in executing a load (Section 2.4), and in analyzing the results of a load test (Section 2.5), respectively. Section 2.6 concludes our survey. Appendix A describes our paper selection process.

2.2 Background

Contrary to functional testing, which has clear testing objectives (pass/fail criteria), Ho *et al.* [145, 146] found that load testing objectives (e.g., performance requirements) are not clear in the early development stages and are often defined later on a case-by-case basis. There are many different interpretations of load testing, both in the context of academic research and industrial practices (e.g., [53, 67, 126, 190]). In addition, the term load testing is often used interchangeably with two other terms: performance testing (e.g., [103,

191, 192]) and stress testing (e.g., [66, 67, 247]). In this section, we provide a definition of load testing by contrasting among various interpretations of load, performance and stress testing.

2.2.1 Definitions of Load Testing, Performance Testing and Stress Testing

Table 2.1 outlines the interpretations of load testing, performance testing and stress testing in the existing literature. The table breaks down various interpretations of load, performance and stress testing along the following dimensions:

- **Objectives** refer to the goals that a test is trying to achieve (e.g., detecting performance problems under load);
- **Stages** refer to the applicable software development stages (e.g., design, implementation or testing), when these tests occur;
- **Terms** refer to the terminologies used in the relevant literature (e.g., load testing and performance testing);
- **Is It Load Testing?** indicates whether we consider such cases (performance or stress testing) to be load testing based on our definition of load testing. The criteria for deciding load, performance and stress testing is presented later (Section 2.2.2).

We find that these three types of testing share some common aspects, yet they have their own area of focuses. In the rest of this section, we first summarize the various definitions of the testing techniques. Then we illustrate their relationship with respect to each other. Finally, we present our definition of load testing. Our load testing definition unifies the existing load testing interpretations as well as performance testing and stress testing works, which are also about load testing.

Table 2.1: Interpretations of Load Testing, Performance Testing and Stress Testing

Objectives	Stages	Terms	Is It Load Testing?
Detecting functional problems under load	Testing (After Conventional Functional Testing)	Load Testing [31, 40, 48, 54, 55, 62, 63, 103, 150, 177, 247, 219, 253], Stress Testing [39, 58, 77, 131, 150, 206, 247]	Yes
Detecting performance problems under load	Testing (After Conventional Functional Testing)	Load Testing [53], Performance Testing [53, 238, 242], Stress Testing [87, 161, 251]	Yes
Detecting reliability problems under load	Testing (After Conventional Functional Testing)	Load Testing [48, 53, 54, 55, 161], Reliability Testing [53]	Yes
Detecting stability problems under load	Testing (After Conventional Functional Testing)	Load Testing [53], Stability Testing [53]	Yes
Detecting robustness problems under load	Testing (After Conventional Functional Testing)	Load Testing [53], Stress Testing [39, 53]	Yes
Measuring and/or evaluating system performance under load	Implementation	Performance Testing [141, 142, 143, 144]	Maybe
	Testing (After Conventional Functional Testing)	Performance Testing [52, 60, 74, 125, 126, 190, 191, 192, 197, 215, 245], Load Testing [191, 196], Stress Testing [86, 161, 171, 172, 251, 252]	Maybe
	Maintenance (Regression Testing)	Performance Testing [166], Regression Benchmarking [79, 160]	Maybe
Measuring and/or evaluating system performance without load	Testing (After Conventional Functional Testing)	Performance Testing [84, 85]	No
Measuring and/or evaluating component/unit performance	Implementation	Performance Testing [158]	No
Measuring and/or evaluating various design alternatives	Design	Performance Testing [56, 94, 101, 102], Stress Testing [120, 119, 121, 122]	No
	Testing (After Conventional Functional Testing)	Performance Testing [76]	No
Measuring and/or evaluating system performance under different configurations	Testing (After Conventional Functional Testing)	Performance Testing [147, 215, 224]	No

2.2.1.1 Load Testing

Load testing is the process of assessing the quality of a system under load in order to detect load-related problems. The rate at which different service requests are submitted to the system under test (SUT) is called the *load* [69]. Load testing uncovers load-related functional problems (e.g, such as deadlocks, racing, buffer overflows and memory leaks [55, 62, 63]) and non-functional problems (e.g., high response time and low throughput [62, 63]).

Load testing is conducted on a system (either a prototype or a fully functional systems) rather than on a design or an architectural model. In the case of missing non-functional requirements, the pass/fail criteria are usually derived based on the "no-worse-than-before" principle. The "no-worse-than-before" principle states that the non-functional requirements of the current version should be at least as good as the previous versions [150]. Depending on the objectives, the load can vary from a normal load (the load expected in the field when the system is operational [55, 161]) or a stress load (higher than the expected normal load) to uncover functional or non-functional problems [253].

2.2.1.2 Performance Testing

Performance testing is the process of measuring and/or evaluating performance related aspects of a software system. Examples of performance related aspects include response time, throughput and resource utilizations [62, 63, 128].

Performance testing can focus on parts of the system (e.g., unit performance testing [158] or GUI performance testing [41]), or on the overall system [62, 63, 215]. Performance testing can also study the efficiency of various design/architectural decisions [94, 101, 102], different algorithms [84, 85] and various system configurations [147, 215, 224].

Contrary to load testing, the objectives of performance testing are broader. Performance testing (1) can verify performance requirements [215] or in case of absent performance

requirements, the pass/fail criteria are derived based on the "no-worse-than-previous" principle [150] (similar to load testing); or (2) can be exploratory (no clear pass/fail criteria). For example, one type of performance testing aims to answer the what-if questions like "what is system performance if we change this software configuration option or if we increase the number of users?" [194, 195, 197, 215].

2.2.1.3 Stress Testing

Stress testing is the process of putting a system under extreme conditions to verify the robustness of the system and/or to detect various load-related problems (e.g., memory leaks and deadlocks). Examples of such conditions can either be load-related (putting system under normal [87, 161, 251] or extreme heavy load [103, 150, 161, 177]) or limited computing resources or failures (e.g. disk full or database failure) [40]. In other cases, stress testing is used to evaluate the efficiency of software designs [120, 119, 121, 122].

2.2.2 Relationships Between Load Testing, Performance Testing and Stress Testing

Our unified definition of load testing that is used in this survey is as follows:

> *Load testing is the process of assessing system behavior under load in order to detect load-related problems due to one or both of the following reasons: (1) functional related problems (i.e., functional bugs that appear only under load), and (2) non-functional quality related problems under load (i.e., non-functional attributes that fail to meet the specified requirements under load)*

Comparatively, *performance testing* is used to measure and/or evaluate performance related aspects (e.g., response time, throughput and resource utilizations) of algorithms, designs/architectures, modules, configurations, or the overall systems. *Stress testing* puts

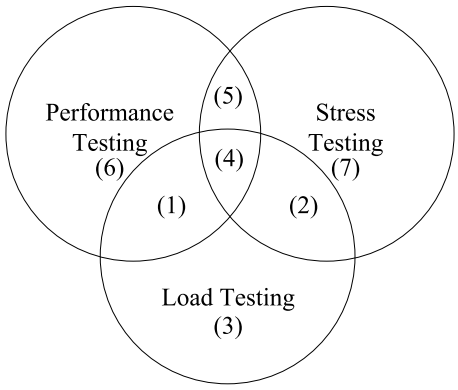


Figure 2.2: Relationships Among Load, Performance and Stress Testing

a system under extreme conditions (e.g., higher than expected load or limited computing resources) to verify the robustness of the system and/or detect various functional bugs (e.g., memory leaks and deadlocks).

There are commonalities and differences among the three types of testing, as illustrated in the Venn Diagram shown in Figure 2.2. We use an online e-commerce system as a working example to demonstrate the relation across these three types of testing techniques.

1. **Scenarios Considered as Both Load Testing and Performance Testing**

   The e-commerce system is required to provide fast response under load (e.g., millions of concurrent client requests). Therefore, testing is needed to validate the system's performance under the expected field workload. Such type of testing is not considered as stress testing as the testing load is only the expected field workload.

2. **Scenarios Considered as Both Load Testing and Stress Testing**

   The e-commerce system must also be be robust under extreme conditions. For example, this system is required to stay up even under bursts of heavy load (e.g., flash crowd [191]). In addition, the system should be free of resource allocation bugs like deadlocks or memory leaks [206].

   This type of testing, which imposes a heavy load on the system to verify the system's

robustness and to detect resource allocation bugs, is considered as both stress testing and load testing. Such testing is not performance testing as the testing objectives do not include performance.

3. **Scenarios Considered as Load Testing Only**

   Although this system is already tested manually using a few users to verify the functional correctness of a service request (e.g., the total cost of a shopping cart is calculated correctly when the customer checks out), the same requests should be verified under hundreds and millions of concurrent users.

   The test, which aims to verify the functional correctness of a system under load is considered only as a load test. This scenario is not performance testing, as the objective is not performance related; nor is this scenario considered as stress testing, as the testing conditions are not extreme.

4. **Scenarios Considered as Load, Performance and Stress Testing**

   This e-commerce website can also be accessed using smartphones. One of the requirements is that the end-to-end service request response time should be reasonable even under poor cellular network conditions (e.g., packet drops and packet delays).

   The type of test used to validate the system's performance requirements under load, given specific computing resources (e.g., network conditions), can be considered as any of the three types of testing.

5. **Scenarios Considered as Performance Testing and Stress Testing**

   Rather than testing the system performance after the implementation is completed. The system architect may want to validate whether a compression algorithm can efficiently handle large image files (processing time and resulting compressed file size). Such testing is not qualified as load testing, as there is no load (concurrent access) applied to the system.

6. **Scenarios Considered as Performance Testing Only**

   In addition to writing unit tests to check the functional correctness of their code, the developers are also required to unit test the code performance. The test to verify the performance of one unit/component of the system is considered only as performance testing.

   In addition, the operators of this e-commerce system need to know the system deployment configurations to achieve the maximal performance throughput and minimal hardware costs. Therefore, performance testing should be carried out to measure the system performance under various database or webserver configurations. The type of test to evaluate the performance of different architectures/algorithms/configurations is only considered as performance testing.

7. **Scenarios Considered as Stress Testing Only**

   Developers have implemented a smartphone application for this e-commerce system to enable users to access and buy items from their smartphones. This smartphone app is required to work under sporadic network conditions. This type of test is considered as stress testing, since the application is tested under extreme network condition. This testing is not considered as performance testing, since the objective is not performance related; nor is this scenario considered as load testing, as the test does not involve load.

2.3 Research Question 1: How is a proper load designed?

The goal of the load design phase is to devise a load, which can uncover functional and/or non-functional (quality related) problems under load. Based on the objectives, there are two general schools of thought for designing a proper load to achieve such objectives:

1. **Designing Realistic Loads**

As the main goal of load testing is to ensure that the system can function correctly once it is deployed in the field, one school of thought is to design loads, which resemble the expected usage once the system is operational in the field. If the SUT can handle such loads without functional and non-functional issues, the SUT would have passed the load test.

Once the load is defined, the SUT executes the load and the system behavior under load is recorded. Load testing practitioners then analyze the system behavior to detect problems. Test durations in such cases are usually not clearly defined and can vary from several hours to a few days depending on the testing objectives (e.g., to obtain steady state estimates of the system performance under load or to verify that system is deadlock-free) and testing budget (e.g., limited testing time). There are two approaches proposed in the literature to design realistic testing loads as categorized in [172]:

(a) **The Aggregate-Workload Based Load Design Approach**

The aggregate-workload based load design approach aims to generate the individual target request rates. For example, an e-commerce system is expected to handle three types of requests with different transaction rates: ten thousand purchasing requests per second, three million browsing requests per second, and five hundred registration requests per second. The resulting load, using the aggregate-workload based load design approach, should resemble these transaction rates.

(b) **The Use-Case Based Load Design Approach**

The use-case (also called *user equivalent* in [172]) based approach is more focused on generating requests that are derived from realistic use cases. For example, in the aforementioned e-commerce system, an individual user would alternate between submitting page requests (browsing, searching and purchasing)

and being idle (reading the web page or thinking). In addition, a user cannot purchase an item before he/she logs into the system.

2. **Designing Fault-Inducing Loads**

   Rather than waiting for the system behavior generated by a realistic load and trying to dig through a large amount of data, another school of thought aims to design loads, which are likely to cause functional or non-functional problems under load. Compared to the realistic load design, the test duration using faulting-inducing loads are usually deterministic and the test results are usually easy to analyze. The test durations in these cases are the time taken for the system to enumerate through the load or the time until the system encounters a functional or non-functional problem.

   There are two approaches proposed in the literature:

   (a) **Deriving Fault-Inducing Loads by Analyzing the Source Code**

   This approach uncovers various functional and non-functional problems under load by systematically analyzing the source code of the system. For example, by analyzing the source code of the aforementioned e-commerce system, load testing practitioners can derive load that exercises these potential functional (e.g., memory leaks) and non-functional (e.g., performance issues) weak spots under load.

   (b) **Deriving Fault-Inducing Loads by Building and Analyzing System Models**

   Various system models abstract different aspects of system behavior (e.g., performance models for the performance aspects). By systematically analyzing these models, potential weak spots, which can lead to load-related problems are reported. For example, load testing practitioners can build performance models in the aforementioned e-commerce system, and discover load that could potentially lead to performance problems (higher than expected response time).

We introduce the following dimensions to compare among various load design techniques as shown in Table 2.2:

- **Techniques** refer to the names of the load design techniques (e.g., step-wise load);
- **Objectives** refer to the goals of the load (e.g., detecting performance problems);
- **Data Sources** refer to the artifacts used in each load design technique. Examples of artifacts can be past field data or operational profiles. *Past field data* could include web access logs, which record the identities the visitors and their visited sites, and database auditing logs, which show the various database interactions. An *Operational Profile* describes the expected field usage once the system is operational in the field [55]. For example, an operational profile for an e-commerce system would describe the number of concurrent requests (e.g., browsing and purchasing) that the system would experience during a day. The process of extracting and representing the expected workload (operational profile) in the field is called *Workload Characterization* [151]. The goal of workload characterization is to extract the expected usage from hundreds or millions hours of user data. Various workload characterization techniques have been surveyed in [82, 111, 151].
- **Output** refers to the types of output from each load design technique. Examples can be workload configurations or usage models. *Workload configuration* refers to one set of workload mix and workload intensity (covered in Section 2.3.1.1). *Models* refer to various abstracted system usage models (e.g., the Markov chain).
- **References** refer to the list of literatures, which propose each technique.

Both load design schools of thought (realistic v.s. fault-inducing load designs) have their advantages and disadvantages: In general, loads resulting from realistic-load based design techniques can be used to detect both functional and non-functional problems. However,

Table 2.2: Load Design Techniques

Techniques	Objectives	Data Sources	Output	References
Realistic Load Design - (1) Aggregate-Workload Based Load Design Approach				
Steady Load	Detecting Functional and Non-functional Quality Related Problems	Operational Profiles, Past Usage Data	One Configuration of Workload Mix and Workload Intensity	[74, 223]
Step-wise Load		Operational Profiles, Past Usage Data	Multiple Configurations of Workload Mixed and Workload Intensities	[39, 43, 87, 114, 140, 200]
Extrapolated Load		Beta-user Usage Data, Interviewing Domain Experts and Competitions Data	One or More Configurations of Workload Mixes and Workload Intensities	[60, 218]
Realistic Load Design - (2) Use-Case Based Load Design Approach				
Testing Loads Derived from UML Models	Detecting Functional and Non-functional Quality Related Problems	UML Use Case Diagrams, UML Activity Diagrams, Operational Profile	UML Diagrams Tagged with Request Rates	[95, 100, 239]
Testing Load Derived from Markov Models		Past Usage Data	Markov Chain Models	[64, 164, 192]
Testing Loads Derived from Stochastic Form-oriented Models		Operational Profile, Business Requirements, User configurations	Stochastic Form-oriented Models	[105, 183]
Fault-Inducing Load Design - (1) Deriving Load from Analyzing the Source Code				
Testing Loads Derived from Data Flow Analysis	Detecting Functional Problems (memory leaks)	Source Code	Testing Loads Lead to Code Paths with Memory Leaks	[247]
Testing Loads Derived from Symbolic Execution	Detecting Functional Problems (high memory usage), Performance Problems (high response time)	Source Code, Symbolic Execution Analysis Tools	Testing Loads Lead to Problematic Code Paths with Performance Problems	[253]
Fault-Inducing Load Design - (2) Deriving Load from Building and Analyzing System Models				
Testing Loads Derived from Linear Programs	Detecting Performance Problems (audio and video not in sync)	Resource Usage Per Request	Testing Loads Lead to Performance Problems (high response time)	[251, 252]
Testing Loads Derived from Genetic Algorithms	Detecting Performance Problems (high response time)	Resource Usage and Response Time Per Task		[133, 213]

the test durations are usually longer and the test analysis is more difficult, as the load testing practitioners have to search through large amounts of data for functional and non-functional problems. Conversely, although loads resulting from fault-inducing load design techniques take less time to uncover potential functional and non-functional problems, the resulting loads usually only cover a small portion of the testing objectives (e.g., only the performance requirements). Thus, there are load optimization and reduction techniques proposed to mitigate the deficiencies of each load design technique.

This section is organized as follows: Section 2.3.1 covers the realistic load design techniques. Section 2.3.2 covers the fault-inducing load design techniques. Section 2.3.3 discusses the test optimization and reduction techniques used in the load design phase. Section 2.3.4 summarizes the load design techniques and proposes a few open problems.

2.3.1 Designing Realistic Loads

In this subsection, we discuss the techniques used to design loads, which resemble the realistic usage once the system is operational in the field. Section 2.3.1.1 and 2.3.1.2 cover the techniques from the Aggregate-Workload and the Use-Case based load design approaches, respectively.

2.3.1.1 Aggregate-Workload Based Load Design Techniques

Aggregate-workload based load design techniques characterize loads along two dimensions: (1) **Workload Intensity**, and (2) **Workload Mix**:

- The *Workload Intensity* refers to the rate of the incoming requests (e.g., browsing, purchasing and searching), or the number of concurrent users;
- The *Workload Mix* refers to the ratios among different types of requests (e.g., 30% browsing, 10% purchasing and 60% searching).

There are three load design techniques proposed to characterize loads with various workload intensity and workload mix:

1. **Steady Load**

   The most straightforward aggregate-workload based load design technique is to devise a steady load, which contains only one configuration of the workload intensity and workload mix throughout the entire load test [74]. A steady load can be inferred from the past data or based on an existing operational profile. This steady load could be the normal expected usage or the peak time usage depending on the testing objectives. Running the SUT using a steady load can be used to verify the system resource requirements (e.g., memory, CPU and response time) [223] and to identify resource usage problems (e.g., memory leaks) [74].

2. **Step-wise Load**

   A system in the field normally undergoes different load characteristics during a normal day. There are periods of light usage (e.g., early in the morning or late at night), normal usage (e.g., during the working hours), and peak usages (e.g., during lunch time). It might not be possible to load test a system using a single type of steady load. Steady-wise load design techniques would devise loads consisting of multiple types of load, to model the light/normal/peak usage expected in the field.

   Step-wise load testing keeps the workload mixes the same throughout the test, while increasing the workload intensity periodically [39, 43, 87, 114, 140, 200]. Step-wise load testing, in essence, consists of multiple levels of steady load. Similar to the steady load approach, the workload mixes can be derived from the past field data or an operational profile. The workload intensity varies from systems to systems. For example, the workload intensity can be the number of users, the normal and peak load usages, or even the amount of results returned from web search engines.

3. **Load Extrapolation Based on Partial or Incomplete Data**

   The steady load and the step-wise load design techniques require an existing operational profile or past field data. However, such data might not be available in some cases: For example, newly developed systems or systems with new features have no existing operational profile or past usage data. Also, some past usage data may not be available due to privacy concerns. To cope with these limitations, loads are extrapolated from the following sources:

   – **Beta-Usage Data**

     Savoia [218] proposes to analyze log files from a limited beta usage and to extrapolate the load based on the number of expected users in the actual deployment field.

   – **Interviews With the Domain Experts**

     Domain experts like system administrators, who monitor and manage deployed systems in the field, generally have a sense of system usage patterns. Barber [60] suggests to obtain a rough estimate of the expected field usage by interviewing such domain experts.

   – **Extrapolation from Using Competitors' Data**

     Barber [60] argues that in many cases, new systems likely do not have beta program due to limited time and budgets and interviewing domain experts might be challenging. Therefore, he proposes an even less formal approach to characterize the load based on checking out published competitors' usage data, if such data exists.

2.3.1.2 Use-Case Based Load Design Techniques

The main problem associated with the aggregate-workload based load design approach is that the loads might not be realistic/feasible in practice, because the resulting requests

might not reflect individual use cases. For example, although the load can generate one million purchasing requests per second, some of these requests would fail due to invalid user states (e.g., some users do not have items added to their shopping carts yet).

However, designing loads reflecting realistic use-cases could be challenging, as there may be too many use cases available for the SUT. For example, continuing with our e-commerce system example, different users can follow different navigation patterns: some users may directly locate items and purchase them. Some users may prefer to check out a few items before buying the items. Some other users may just browse the catalogs without buying. It would not be possible to cover all the combinations of these sequences. Therefore, various usage models are proposed to abstract the use cases from thousands and millions of user-system interactions. In the rest of this subsection, we discuss three load design techniques based on usage models.

1. **Testing Loads Derived from UML Models**

   UML diagrams, like Activity Diagrams and Use Case Diagrams, illustrate detailed user interactions in the system. One of the most straight forward use-case based load design techniques is to tag load information on the UML Activity Diagram [95, 100] and Use Case Diagram [239].

   Realistic Usage Model (RUM) is derived from the UML Use Case Diagrams proposed by Wang *et al.* [239]. Figure 2.3 illustrates the resulting loads for a web-based file system from [239]. As shown in the Use Case Diagram: once a user logs into the system, this user can perform one of four use cases: downloading files ("Download"), navigating in that file system ("Navigate"), uploading more files ("Upload"), and deleting files ("Delete"). Each of the above use cases consist of one or more steps. For example, the "Navigate" use case only has one step and the "Delete" use case consists of three steps: (1) navigating files in the folder, (2) deleting a file (files) in the folder, and (3) navigating this folder again to verify the deletion. The number beside each of the four

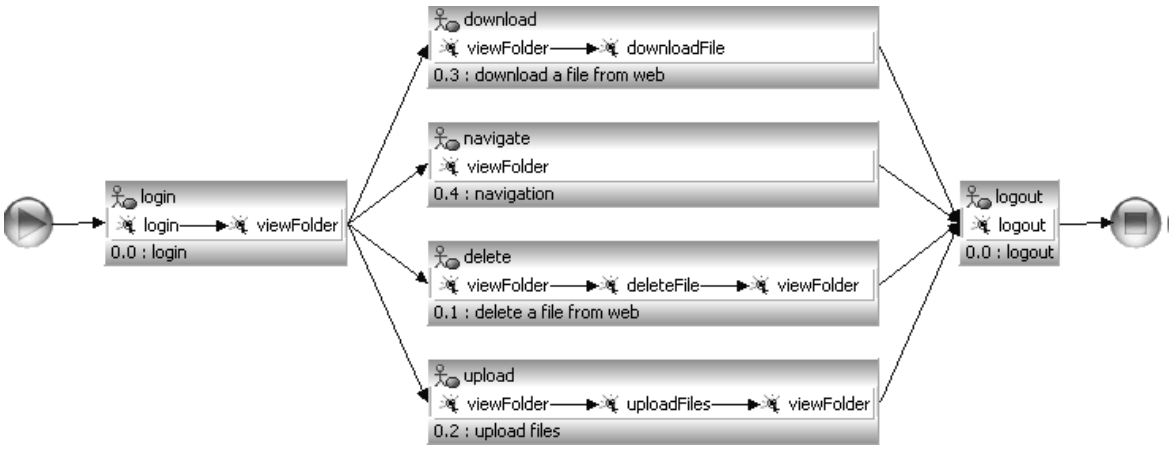


Figure 2.3: Load Design Based on UML Models [191]

use cases is the likelihood that a user triggers that action. For example, a user is more likely to navigate around (40% probability) than to delete a file (10% probability). The sum of all probabilities from these four use cases is 1.

2. **Testing Loads Derived from Markov-Chain Models**

   The problem with the UML-based testing load is that the UML Diagrams may not be available or such information may be too detailed (e.g., hundreds of use cases). Therefore, techniques are needed to abstract load information from other sources.

   A Markov Chain, which is also called the User Behavior Graph [192], consists of a finite number of states and a set of state transition probabilities between these states. Each state has a steady state probability associated with it. If two states are connected, there is a transition probability between these two states.

   Markov Chains are widely used to generate load for web-based e-commerce applications [64, 164, 192], since Markov chains can be easily derived from the past field data (web access logs [192]). Each entry of the log is a URL, which consists of the

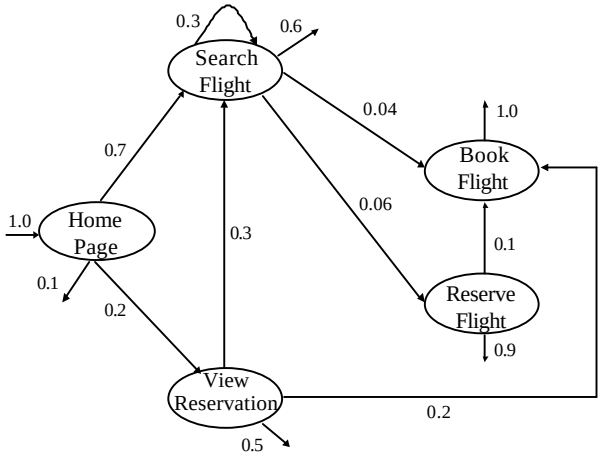


Figure 2.4: A Sample Markov Chain [191]

requested web pages and "parameter name = parameter value" pairs. Therefore, sequences of user sessions can be recovered by grouping sequences of request types belonging to the same session. Each URL requested becomes one state in the generated Markov chain. Transition probabilities between states represent real user navigation patterns, which are derived from the probabilities of one user clicking page B when he/she is on page A.

Figure 2.4 shows a sample Markov Chain from [191]. Each state corresponds to a particular web page in that system. This Markov Chain shows that each user starts at the "Home" page. Afterwards, the user is 20% likely to move to the View Reservation page, 70% likely to move to the Search Flight page, and 10% likely to leave the site.

During the course of a load test, user action sequences are generated based on the probabilities modeled in the Markov chain. The think time between each action is usually generated randomly based on a probabilistic distribution (e.g., a normal distribution or exponential distribution) [164, 192]. As the probability in the Markov chain only reflects the average behavior of a certain period of time, Barros *et al.* [64] recommend the periodically updating of the Markov chain based on the field data in

order to ensure that load testing reflects the actual field behavior.

3. **Testing Load Derived from Stochastic Form-oriented Models**

   Stochastic Form-oriented Model is another technique used to model a sequence of actions performed by users. Compared to the testing loads represented by the Markov Chain models, a *Stochastic Form-oriented model* is richer in modeling user interactions in web-based applications [105]. For example, a user login action can either be a successful login and redirect to the overview page, or a failure login and redirect back to the login page. Such user behavior is difficult to model in a Markov chain [105, 183].

   A form-oriented model consists of pages, actions and transitions. *Pages* are sets of screens, with different screens corresponding to different contents of a page. For example, an Amazon welcome screen is displayed differently based on different user's preferences and purchasing history. Each page contains a number of forms and submissions of these forms are called *actions*. Hyperlinks are like forms but with no data fields. *Transitions* are directed edges between actions and pages. As opposed to Markov chains, which assign probabilities to all states, probabilities in stochastic form-oriented models are assigned only to transitions going from pages to actions. There is no probability going from actions to pages, since the resulting webpage is the consequence of user actions.

   Figure 2.5 is an example of Stochastic Form-oriented Model of a web-based application (from [105]). Ovals are pages (e.g., *Login* and *Menu*) and boxes are actions (e.g., *Verify* and *Confirm Transfer*). The probability of performing a *Make Transfer* action on the *Menu* page is 20%. Edges from boxes to ovals do not have probabilities (e.g. from the *Verify* action to the *Login* page), since probabilities in stochastic form-oriented models are only assigned to transitions from pages to actions. The result of the *Verify* action can be either directed to the *Menu* page (verification success) or back to the

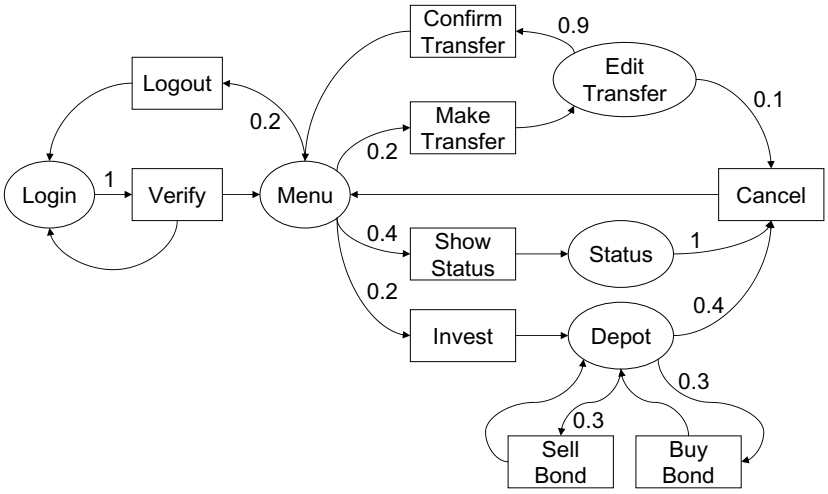


Figure 2.5: An Example Stochastic Form-Oriented Model [105]. Pages are represented as ovals and actions are represented as boxes.

Login page (verification failure).

Cai *et al.* [80, 81] propose a toolset that automatically generates a load for a web application using a three-step process: First, the web site is crawled by a third party web crawler and the website's structural data is recovered. Then, their toolset lays out the crawled web structure using a Stochastic Form-Oriented Model and prompts the performance engineer to manually specify the probabilities between the pages and actions based on an operational profile.

2.3.2 Designing Fault-Inducing Loads

In this subsection, we cover the load design technique from the school of fault-inducing load design. There are two approaches proposed to devise potential fault-inducing testing loads: (1) by looking through the source code (Section 2.3.2.1), and (2) by building and analyzing various system models (Section 2.3.2.2).

2.3.2.1 Deriving Fault-Inducing Loads via Source Code Analysis

There are two techniques proposed to automatically analyze the source code for specific problems. The first technique is trying to locate specific code patterns, which lead to known load-related problems (e.g., memory allocation patterns for memory allocation problems). The second technique uses model checkers to systematically look for memory and performance problems.

1. **Testing Loads Derived from Data Flow Analysis**

   Load sensitive regions are code segments, whose correctness depends on the amount of input data and the duration of testing [247]. Examples of load sensitive regions can be code dealing with various types of resource accesses (e.g., memory, thread pools and database accesses). Yang *et al.* [247] use data flow analysis of the system's source code to generate loads, which exercise the load sensitive regions. Their technique detects memory related faults (e.g., memory allocation, memory deallocation and pointers referencing).

2. **Testing Loads Derived from Symbolic Executions**

   Rather than matching the code for specific patterns (e.g., the resource accesses patterns in [247]), Zhang *et al.* [253] use symbolic test execution techniques to generate loads, which can cause memory or performance problems. Symbolic executions simulate the program execution using symbolic values instead of actual values as inputs. Symbolic values are unknown and will be computed later to satisfy various constraints. Paths are uniquely defined by a set of constraints. Each constraint encodes a branch decision made along the path and is defined as a boolean expression over concrete and symbolic values. For example, the symbolic execution on the following code segment (taken from [253]) would yield two sets of outputs: (1) In order to execute the if-statement, we need to have x and y equal and both x and y greater than 0;

(2) In order to skip the if-statement, we need to have x and y equal and less than and equal to 0. Therefore, by symbolically executing the overall system, the output would contain a set of input values corresponding to different code paths, respectively.

$$y = x;$$

$$if\ (y > 0)\ then\ y++;$$

$$return\ y;$$

Zhang *et al.* use the path information to derive two types of loads:

(a) **Testing Loads Causing Large Response Time**

Zhang *et al.* assign a time value for each step (e.g., 10 for an invoking routing and 1 for other routines). Therefore, 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 constrains form the loads.

(b) **Testing Loads Causing Large Memory Consumptions**

Rather than tracking the time, Zhang *et al.* track the memory usage at each step. The memory footprint information is available through a Symbolic Execution tool, (e.g., the Java Path Finder (JPF)). Zhang *et al.* use the JPF's built-in object life cycle listener mechanism to track the heap size of each path. Paths leading to large memory consumption are identified and values satisfying such code paths form the loads.

2.3.2.2 Deriving Fault-Inducing Loads by Building and Analyzing System Models

Rather than analyzing the source code to explore potential problematic regions/paths, techniques have been proposed to automatically search for potential problematic loads.

1. **Deriving Performance-Problem-Inducing Loads from Linear Programs**

Online multimedia systems have various temporal requirements: (1) *Timing requirements:* audio and video data streams should be delivered in sequence and following strict timing deadlines; (2) *Synchronization requirements:* video and audio data should be in synch with each other; (3) *Functional requirements:* some videos can only be displayed after collecting fee.

Zhang *et al.* [251, 252] propose a two-step technique that automatically generates loads, which can cause a system to violate the synchronization and responsive requirements while satisfying the business requirements. Their idea is based on the belief that timing and synchronization requirements usually fail when the system's resources are saturated: For example, if the memory is used up, the system would slow down due to paging.

(a) **Identify Data Flows using a Petri Net**

A *Petri Net*, which is a technique that models the temporal constraints of a system, consists of a triple: *places*, *transitions*, and *directed arcs*. A petri-net starts with an initial assignment of tokens to their places. Each place may contain zero or more tokens. Directed arcs always run between a place and a transition, but never between places or transitions. An *Input place* has an arc, which runs to a transition. An *Output place* has an arc, which runs from a transition. A transition between two places is enabled when both input places are active. A transition can fire only when it is enabled. When a transition fires, a token is removed from each of its input places and added to its output places. Figure 2.6 shows an example Petri Net from [22], which consists of 4 places $(P1, P2, P3, P4)$, 2 transitions $(T1, T2)$, and 3 tokens. The black dots represent tokens. Many of the places (e.g., $P1$, $P2$, $P3$) are both input and output places and $P4$ is an output place only.

All possible user action sequences can be generated by conducting reachability

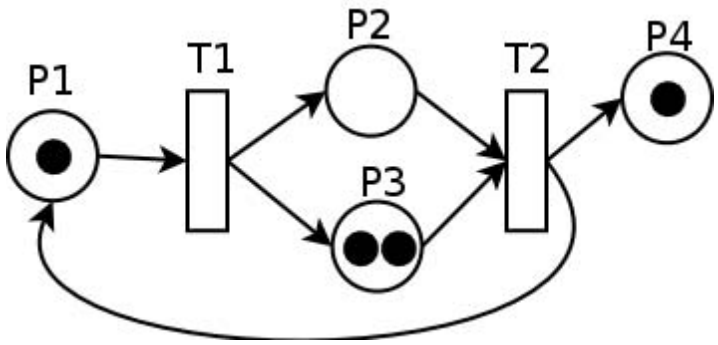


Figure 2.6: A Petri Net Example [22]

analysis, which explores all the possible paths, on the Petri Net. For example, a new video action C cannot be fired until the previous video action A and audio action B are both completed.

(b) **Formulate System Behavior into a Linear Program and Identify Performance Problems**

Linear programming systematically searches for optimal solutions based on certain constraints. A linear program contains the following two types of artifacts: an objective function (the optimal criteria) and a set of constraints. The objective function is to maximize or minimize a linear equation. The constraints are a set of linear equations or inequalities. An example of a linear program is as follows:

$$maximize : 3x_1 + 5x_2$$

$$subject\ to : x_1 + x_2 \leq 5$$

$$x_1 \geq 0, x_2 \geq 0$$

The sequencing of arrival times of these user action sequences are formulated using a set of linear constraints. There are two types of constraint functions: One constraint function ensures the total testing time is within a pre-specified value (the test will not run for too long). The rest of the constraint functions formulate the temporal requirements derived from the possible user action sequences, as

the resource requirements (e.g., CPU, memory, network bandwidth) associated with each multimedia object (video or audio) are assumed to be known. The objective function is set to evaluate whether the arrival time sequence would cause the saturations of one or more system resources (CPU and network).

2. **Deriving Performance-Problem-Inducing Loads from Genetic Algorithms**

   An SLA, *Service Level Agreement*, is a contract with potential users on the non-functional properties like response time and reliability as well as other requirements like costs. Penta *et al.* [213] and Gu *et al.* [133] uses Genetic Algorithms to derive loads causing SLA or QoS (Quality of Service) requirement violations (e.g., response time) in service-oriented systems. Like linear programming, *Genetic Algorithms*, is a search algorithm, which mimics the process of natural evolution for locating optimal solutions towards a specific goal.

   The genetic algorithms are applied twice to derive potential performance sensitive loads:

   (a) Penta *et al.* [213] use the genetic algorithm technique proposed by Canfora *et al.* [83] to identify risky workflows within a service, which is as close to the SLA (high response time) as possible.

   (b) Penta *et al.* [213] apply the genetic algorithm to generate loads that cover the identified risky workflow and violate the SLA.

2.3.3 Load Design Optimization and Reduction Techniques

In this subsection, we discuss two classes of load design optimization and reduction techniques aimed at improving various aspects of load design techniques. Both classes of techniques are aimed at improving the realistic load design techniques.

– **Hybrid Load Optimization Techniques**

The aggregate-workload based techniques focus on generating the desired workload, but fail to mimick realistic user behavior. The user-equivalent based techniques focus on mimicking the individual user behaviour, but fail to match the expected overall workload. The hybrid load optimization techniques (Section 2.3.3.1) aims to combine the strength of the aggregate-workload and use-case based load design approaches. For example, for our example e-commerce system, the resulting load should resemble the targeted transaction rates and mimic real user behavior.

– **Optimizing and Reducing the Duration of a Load Test**

One major problem with loads derived from realistic load testing is that the test durations in these testing loads are usually not clearly defined (i.e., no clear stopping rule). The same scenarios are repeatedly executed over several hours or days. There are two techniques proposed to systematically reduce the overall load test duration.

Table 2.3 compares the various load design optimization and reduction techniques along the following 5 dimensions:

– **Techniques** refer to the load design optimization and reduction techniques used (e.g., the hybrid load optimization techniques).

– **Target Load Design Techniques** refer to the load design techniques that the reduction or optimization techniques are intended to improve. For example, the hybrid load optimization techniques combine the strength of aggregate-workload and use-case based load design techniques.

– **Optimization and Reducing Aspects** refer to the aspects the current load design that the optimization and reduction techniques attempt to improve. One example is to reduce the test duration.

– **References** refer to the list of literatures, which propose each technique.

Table 2.3: Test Reduction and Optimization Techniques Used in the Load Design Phase

Techniques	Target Load Design Techniques	Optimizing and Reducing Aspects	Data Sources	References
Hybrid Load Optimization	All Realistic Load Design Techniques	Combining the strength of aggregate-workload and use-case based load design techniques	Past usage data	[86, 172, 184]
Extrapolation	Step-wise Load Design	Reducing the number of workload intensity levels	Step-wise testing loads	[174, 191, 192]
Deterministic State	All Realistic Load Design Techniques	Reducing repeated execution of the same scenarios	Realistic testing loads	[48, 54, 55]

2.3.3.1 Hybrid Load Optimization Techniques

Hybrid load optimization techniques aim to better model the realistic load for web-based e-commerce systems [172, 184]. It consists of the following three steps:

– **Step 1 - Extracting Realistic Individual User Behavior From Past Data**

Most of the e-commerce systems record past usage data in the form of web access logs. Each time a user hits a web page, an entry is recorded in the web access logs. Each log entry is usually a URL (e.g., the browse page or the login page), combined with some user identification data (e.g., session IDs). Therefore, individual user action sequences, which describe the step-by-step user actions, can be recovered by grouping the log entries with user identification data.

– **Step 2 - Deriving Targeted Aggregate Load By Carefully Arranging the User Action Sequence Data**

The aggregate load is achieved by carefully arranging and stacking up the user action sequences (e.g., two concurrent requests are generated from two individual user action sequences). There are two techniques proposed to calculate user action sequences:

1. **Matching the Peak Load By Compressing Multiple Hours Worth of Load**

   Burstiness refers to short uneven spikes of requests. One type of burstiness is caused by the *flash crowd*. The phenomenon where a website suddenly experiences a heavier than expected request rate. An example of flash crowd includes when many users flocked to the news sites like CNN.com during the 9/11 incident, or during the World CUP period, the FIFA website was often more loaded when a goal was scored. During the flash crowd incident, the load could be several times higher than the expected load. Incorporating realistic burstiness into load testing is important to verify the capacity of a system [201].

   Maccabee and Ma [184] squeeze multiple one-hour user action sequences together into one-hour testing load to generate a realistic peak load, which is several times higher than the normal load.

2. **Matching the Specific Request Request Rates By Linear Programs**

   Maccabee and Ma's [184] technique is simple and can generate higher than normal load to verify the system capacity and guard against problems like a flash crowd. However, their technique has problems like coarse-grained aggregate load, which cannot reflect the normal expected field usage. For example, the individual requests rates (e.g., browsing or purchasing rates) might not match with the targeting request rates. Krishnamurthy *et al.* [171] use linear programming to systematically arranging user action sequences, which match with the desired workload.

   A linear program in this case is formulated as follows:

   - The variables (x_1, x_2, ...) correspond to the number of individual user action sequences, respectively.
   - Assuming that we have a desired load specified beforehand, each workload request in that load is specified as one constraint in the linear program. The

coefficients in each constraint are the number of corresponding workload requests in that session. The linear combination of real sessions would not exceed the pre-specified number of requests. For example, if there are two individual user action sequences, one with two browsing requests and the other with three browsing requests, and the overall desired browsing request is 20. The constraints would be specified as: $2x_1 + 3x_2 \leq 20$.

- The objective function measures the overall similarity between the set of user action sequences and the desired workload. The output/goal of this linear program is to generate the number of individual user action sequences that most closely match the desired workload.

– **Step 3 - Specifying the Inter-arrival Time Between User Actions**

There is a delay between each user action, when the user is either reading the page or thinking about what to do next. This delay is called the "think time" or the "inter-arrival time" between actions. The think time distribution among the user action sequences is specified manually in [171, 172]. Casale et al. [86] extend the technique in [171, 172] to create realistic burstiness. They use a burstiness level metrics, called the *Index of Dispersion* [201], which can be calculated based on the inter-arrival time between requests. They use the same constraint functions as [171, 172], but a different non-linear objective function. The goal of the objective function is to find the optimal session mix, whose *index of dispersion* is as close to the real-time value as possible.

The hybrid technique outputs the exact individual user actions during the course of the actions. The advantage of such output is to avoid some of the expected system failures from other techniques like the Markov chains. Krishnamurthy *et al.* [172] outline one such case in the e-commerce systems: A user cannot purchase an item unless he/she logs in and adds that item into the shopping cart. Krishnamurthy *et al.* [172] explain that test

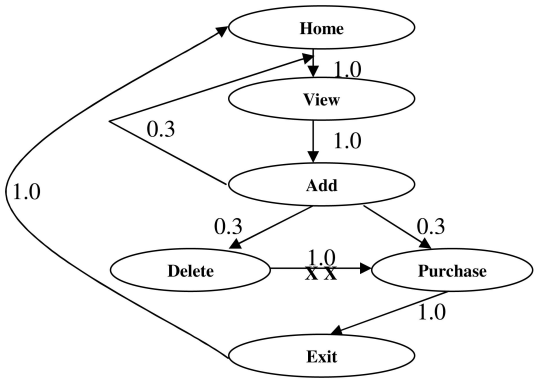


Figure 2.7: An example of a Markov Chain that fails to capture the inter-request dependencies from [172]

cases generated by the resulting Markov chain in the web-based systems may not satisfy the such requirements. Their example is illustrated in Figure 2.7. Suppose a Markov chain is derived based on two traces from two different customers: Customer A performs the following actions: $[Home, View, Add, View, Add, Delete, Purchase]$ and Customer B performs the following actions: $[Home, View, Add, Purchase]$. The resulting Markov chain, shown in Figure 2.7, may generate the following incorrect synthetic sequences: [Home, View, Add, Delete, Purchase]. This sequence could result in an error, as there would be no items to purchase in one user's shopping chart.

Special load generators are required to take such input and generate the testing load [172]. The scalability of the approach would be limited by the machine's memory, as the load generators need to read in all the input data (testing user actions at each time instance) at once.

2.3.3.2 Optimizing and Reducing the Duration of a Load Test

There are two techniques proposed to systematically reduce the load test duration for the realistic-load-based techniques. One technique aims at reducing a particular load design technique (step-wise load testing). The other technique aims at optimizing and reducing

the realistic load design techniques by adding determinism.

1. **Load Test Reduction By Extrapolation**

   Load testing needs to be conducted at various load levels (e.g., number of user levels) for step-wise load testing. Rather than examining the system behavior under all load levels, Menasce *et al.* [191, 192] propose to only test a few load levels and extrapolate the system performance at other load levels.

   Furthermore, Leganza [174] proposes to extrapolate the load testing data from the results conducted on a lower number of users onto the actual production workload (300 users in testing versus $1,500$ users in production) to verify whether the current SUT and hardware infrastructure can handle the desired workload.

2. **Load Test Optimization and Reduction By Deterministic States**

   Rather than repeatedly executing a set of scenarios over and over, like many of the aggregate-workload based load design techniques (e.g., steady load and Markov-chain), Avritzer *et al.* [48, 54, 55] propose a load optimization technique, called the *Deterministic State Testing*, which ensures each type of load is only executed once.

   Avritzer *et al.* characterizes the testing load using states. Each state measures the number of different active processing jobs at the moment. Each number in the state represents the number of active requests of a particular request. Suppose our e-commerce system consists of four scenarios: registration, browsing, purchasing and searching. The state $(1, 0, 0, 1)$ would indicate that currently only there is one registration request and one search request active and the state $(0, 0, 0, 0)$ would indicate that the system is idle. The probability of these states, called "Probability Mass Coverage", measures the likelihood that the testing states is going to be covered in the field. These probabilities are calculated based on the production data. The higher the probability of one particular state, the more likely it is going to happen in the field.

Load test optimization can also be achieved by making use of the probability associated with each state to prioritize tests. If time is limited, only a small set of states with a high probability of occurrence in the field can be selected. However, the "Probability Mass Coverage" metric fails to capture the likelihood of transitions between states, which can be important for measuring performance impacts and identifying resource allocation faults.

In addition to reducing the test durations, deterministic state testing is very good at detecting and reproducing resource allocation failures (e.g., memory leaks and deadlocks). However, such testing cannot be used to verify some other non-functional aspects (e.g., performance).

2.3.4 Summary and Open Problems

There are two schools of thought for load design: (1) Designing loads, which mimic realistic usage; and (2) Designing loads, which are likely to trigger functional and non-functional failures. Realistic Load Design techniques are more general, but the resulting loads can take a long time to execute. Results of a load test are harder to analyze (due to the large volume of data). On the contrary, Fault-Inducing Load Design techniques are more narrowly focused on a few objectives (i.e., you will not detect unexpected problems), but the test duration is usually deterministic and shorter. The test results are usually easier to analyze.

However, a few issues are still not explored thoroughly:

- **Optimal Test Duration for the Realistic Load Design**

  One unanswered question among all the realistic load design techniques is to identify the optimal test duration, which is the shortest test duration while still covering all the test objectives.

- **Benchmarking & Empirical Studies of the Effectiveness of Various Techniques**

Among the load design techniques, the effectiveness of these techniques, in terms of scale and coverage, is not clear. In large scale industrial systems, which are not web-based systems, can we still apply techniques like Stochastic Form-oriented Models? A benchmark suite (like the Siemens benchmark suite for functional bug detection [4]) is needed to systematical evaluate the scale and coverage of these techniques.

– **Test Coverage Metrics**

Unlike functional testing suites, which have various metrics (e.g., code coverage) to measure the test coverage. There are few load testing coverage metrics other than the "Probability Mass Coverage" metric, which is proposed by Avritzer *et al.* [48, 54, 55]. Unfortunately, this metric fails to capture other aspects of the system under load (e.g., transition among different states and the input load level).

– **Load Design Optimization and Reduction Techniques**

Currently, load optimization and reduction techniques focus on realistic load design techniques. There are no techniques aimed at improving the fault-inducing-based load design techniques. For example, there are no work to combine the strength of various fault-inducing techniques, so that the overall testing load can achieve multiple objectives.

– **Testing Load Evolution and Maintenance**

There is no existing work aimed at maintaining and evolving the resulting load. Below we provide two examples where the evolution of the load is likely to play an important role:

1. **Realistic Loads:** As the users get more familiar with the system, the usage patterns are likely to change. How much change would merit an update to a realistic-based testing load?

2. **Fault-Inducing Loads:** For fault-inducing techniques, as the system evolve over time, can we improve the model building by incrementally analyzing the system internals (e.g., changed source code or the change features)?

2.4 Research Question 2: How is a load test executed?

Once a proper load is designed, a load test is executed. The load test execution phase consists of the following three main aspects: (1) *Setup*, which includes system deployment and test execution setup; (2) *Load Generation and Termination*, which consists of generating the load according to the configurations and terminating the load when the load test is completed; and (3) *Test Monitoring and Data Collection*, which includes recording the system behavior (e.g., execution logs and performance metrics) during execution. The recorded data is then used in the Test Analysis phase.

As shown in Table 2.4, there are three general approaches of load test executions:

1. **Live-User Based Executions**

   A load test examines a system's behavior when the system is simultaneously used by many users. Therefore, one of the most intuitive load test execution approach is to execute a load test by employing a group of human testers [31, 174]. Individual users (testers) are selected based on the testing requirements (e.g., locations and browsers).

   The live-user based execution approach reflects the most realistic user behaviors. In addition, this approach can obtain real user feedbacks on aspects like acceptable request performance (e.g., whether certain requests are taking too long) and functional correctness (e.g., a movie or a figure is not displaying properly). However, the live-user based execution approach cannot scale well, as the approach is limited by the number of recruited testers and the test duration [174]). Furthermore, the approach cannot explore various timing issues due to complexity of manual coordination of

many testers.

2. **Driver Based Executions**

   To overcome the scalability issue of the live-user based approach, the driver based execution approach is introduced to automatically generate thousands or millions of concurrent requests for a long period of time. Compared to the live-user based executions, where individual testers are selected and trained, driver based executions require setup and configuration of the load drivers. Therefore, a new challenge in driver based execution is the configuration of load drivers to properly produce the load. In addition, some system behavior (e.g., the movie or image display) cannot be easily tracked, as it is hard for the load driver to judge the audio or video quality.

   Different from existing load driver surveys [68, 214, 235], which focus on comparing the capabilities of various load drivers, our survey of driver based execution focuses on the techniques used by the load drivers. Comparing the load driver techniques, as opposed to capabilities, has the following two advantages in terms of knowledge contributions: (1) **Avoid Repetitions:** Tools from different vendors can adopt similar techniques. For example, WebLoad [33] and HP LoadRunner [10] both support the store-and-replay test configuration technique. (2) **Tool Evolution:** The evolution of such load drivers is not tracked in the tool-based surveyed. Some tools get decommissioned over time. For example, tools like Microsoft's Web App Stress Tool surveyed in [235], no longer exists. New features (e.g., supported protocols) are constantly added into the load testing tools over time. For example, Apache JMeter [3] has recently added support for model-based testing (e.g., Markov-chain models).

   There are three categories of load drivers:

   (a) **Benchmark Suite** is a specialized load driver, designed for one type of system.

For example, LoadGen [18] is a load driver specified used to load test the Microsoft Exchange MailServer. Benchmark suites are also used to measure and compare the performance of different versions of software and/or hardware setup (called *Benchmarking*). Practitioners specify the rate of requests as well as test duration. Such load drivers are usually customized and can only be used to load test one type of system [18, 204].

In comparison to benchmark suites, the following two categories of load drivers (centralized and peer-to-peer load drivers) are more generic (applicable for many systems).

(b) **Centralized Load Drivers** refer to a single load driver, which generates the load [10, 33].

(c) **Peer-to-peer Load Drivers** refer to a set of load drivers, which collectively generate the target testing load. Peer-to-peer load drivers usually have a controller component, which coordinates the load generation among the peer load drivers [107, 246].

Centralized load drivers are better at generating targeted load, as there is only one single load driver to control the traffic. Peer-to-peer load drivers can generate larger scale load (more scalable), as centralized load drivers are limited by processing and storage capabilities of a single machine.

3. **Emulation Based Executions**

The previous two load test execution (live-user based and driver based execution) approaches require a fully functional system and conduct load testing in the field or in a field-like environment. The emulation based load test execution approach performs the load testing on special platforms. In this survey, we focus on two types of special platforms:

(a) **Special Platforms Enabling Early and Continuous Examination of System Behavior Under Load**

In the development of large distributed software systems (e.g., service-oriented systems), many components like the application-level entities and the infrastructure-level entities are developed and validated in different phases of the software lifecycle. This creates the *serialized-phasing* problem, as the end-to-end functional and quality-of-service (QoS) aspects cannot be evaluated until late in the software life cycle (e.g., at the system integration time) [141, 142, 143, 144]. Emulation based execution can emulate parts of the system that are not readily available. Such execution techniques can be used to examine the system's functional and non-functional behavior under load throughout the software development lifecycle, even before the system is completely developed.

(b) **Special Platforms Enabling Deterministic Execution**

Reporting and reproducing problems like deadlocks or high response time is much easier on these special platforms, as these platforms can provide fine-grained controls on method and thread inter-leavings. When problems occur, such platforms can provide more insights on the exact system state.

Live-user based and driver based executions require deploying the system and running the test in the field or field-like environment. Both approaches need to face the challenge of setting up realistic test environment (e.g., with proper network latency mimicking distributed locations). Running the system on special platforms avoids such complications. However, emulation based executions usually focus on a few test objectives (e.g., functional problems under load), which are not general purposes like the live-user based and driver based executions. In addition, like driver based executions, emulation based executions use load drivers to automatically generate load.

Among the three main aspects of the load test execution phase, Table 2.4 outlines

the similarities and differences among the aforementioned three load test execution approaches. For example, there are two distinct setup activities in the Setup aspect: System Deployment and Test Execution Setup. Some setup activities would contain different aspects for the three test execution approaches (e.g., during the test execution setup activity). Some other activities would be similar (e.g., the system deployment activity is the same for the live-user and driver based executions).

In the next three subsections, we compare and contrast the different techniques applied in the three aspects of load execution phases: Section 2.4.1 explains the setup techniques, Section 2.4.2 discusses the load generation and termination techniques and Section 2.4.3 describes the test monitoring and data collection techniques. Section 2.4.4 summaries the load test execution techniques and lists some open problems.

2.4.1 Setup

As shown in Table 2.4, there are two setup activities in the Setup aspect:

- **System Deployment** refers to deploying the system in the proper test environment and making the system operational. Examples can include installing the SUT and configuring the associated third party components (e.g., the mail server and the database server).

- **Test Execution Setup** refers to setting up and configuring the load testing tools (for driver based and emulation based executions), or recruiting and training testers (for live-user based executions) and configuring the test environment to reflect the field environment (e.g., network latency for long distance communication).

2.4.1.1 System Deployment

The system deployment process is the same for the live-user based and the driver based executions, but different from the emulation based executions.

Table 2.4: Load Execution Techniques

	Load Test Execution Approaches	Live-user based Execution	Driver based Execution	Emulation based Execution
Aspect 1. Setup				
Setup Activities	**System Deployment**	System installation and configuration in the field/field-like/lab environment	System installation and configurations in the field/field-like/lab environment	System deployment on the special platforms
	Test Execution Setup	Tester Recruitment and Training, Test Environment Configurations	Load Driver Installation and Configurations, Test Environment Configurations	Load Driver Installation and Configurations
Aspect 2. Load Generation and Terminations				
Options for Load Generation and Termination	**Static Configurations**	✓	✓	✓
	Dynamic	x	✓	x
	Deterministic	x	x	✓
Aspect 3. Test Monitoring and Analysis				
Types of System Behavior Data	**Functional Problems**	✓	✓	✓
	Execution Logs	✓	✓	✓
	Performance Metrics	✓	✓	x
	System Snapshots	x	✓	x

– **System Installation and Configuration for the Live-user based and Driver based Executions**

For live-user based and driver based executions, it is recommended to perform the load testing on the actual field environment, although the testing time can be limited and there could be high cost associated [218]. However, in many cases, load tests are conducted in a lab environment due to accessibility and cost concerns (difficult and costly to access the actual production environment) [60, 174, 193, 218]. Therefore, extra efforts are required to configure the lab environment to reflect the most relevant field characteristics. The system deployment process for emulation based execution is different from the other two approaches.

The SUT and its associated components (e.g., database and mail servers) are deployed in a field-like setting. One of the important aspects mentioned in the load testing literature is creating realistic databases, which have a size and structure similar to the field setting. It would be ideal to have a copy of the field database. However, sometimes no such data is available or the field database cannot be directly used due to security or privacy concerns.

There are two proposed techniques to create field like test databases:

– **Importing Raw Data**

One technique is to import a set of raw data, which shares the same characteristics (e.g., size and structure) as the field data.

Bainbridge *et al.* [58] describe their experience of creating a realistic database for load testing digital library systems, which are required to index and query millions or billions of records. Such data usually contains large volumes of text, images and metadata (e.g., bibliography). Bainbridge *et al.* setup the synthetic database by importing a large amount of newspaper data (20 GB of raw text, 50 GB of metadata and 570 GB and of images) into the database.

– **Data Sanitization**

In many cases, raw data that shares similar characteristics with the field data, may not be available. Furthermore, importing such large volumes of raw data would take a long time. Another approach is to transform the field database into a test database [240], but to remove certain sensitive information. Such process is called *data sanitization* or *data anonymization*.

Barros *et al.* [64] first identify the confidential data from the field database. Then, they compute the hash value of the confidential data using a secure one-way hash function, called SHA-1, and replace the confidential data in the database with the computed hash value. For different confidential records, a hash function will provide different hash values. A hash function is called "secure" if it is computationally impossible to find the key corresponds to a specific hash value. In this way, the resulting test database preserves data integrity (e.g., primary keys and uniqueness) as well as data relations.

One limitation with Barros *et al.*'s hashing approach is that the hashed values may lead to different code paths and code coverage, since the application logic may treat different database field values differently. To overcome this limitation, Grechanik *et al.* [129] propose to identify the database field and values, which impact the control flow of the application. Grechanik *et al.* identify the database field and values by applying existing test cases on the instrumented SUT and recording the resulting control flows. Then they calculate the range of database field values, which satisfy the path conditions in these control flows. Finally, these database field values are generalized while still satisfying the control flow during the data sanitization process. For example, all ages between $40 - 60$ are all generalized to 50, as this age group follows same code path.

– **System Deployment for the Emulation based Executions**

For the emulation based executions, the SUT needs to be deployed on the special platforms, in which the load test is to be executed. The deployment techniques for the two types of special platforms mentioned above are different:

– **Automated Code Generation for the Incomplete System Components**

The automated code generation for the incomplete system components is achieved by the model-driven engineering platforms. Rather than implementing the actual system components via programming, developers can work at a higher level of abstraction in model-driven engineering (e.g., using domain-specific modeling languages or visual representations). Concrete code artifacts and system configurations are generated based on the model interpreter [141, 142, 143, 144] or the code factory [71]. The overall system is implemented using a model-based engineering framework in Domain-specific modeling languages. For the components, which are not available yet, the framework interpreter will automatically generate mock objects (method stubs) based on the model specifications. These mock objects, which conform to the interface of the actual components, emulate the actual component functionality. In order to support a new environment (e.g., middleware or operating system), the model interpreter needs to be adapted for various middleware or operating systems, but no change to the upper level model specifications is required.

– **Special Profiling and Scheduling Platform**

In order to provide more detailed information on the system's state when a problem occurs (e.g., deadlocks or racing), special platforms (e.g., the CHESS platform [206]), which control the inter-leaving of threads are used. The SUT needs to be run under a development IDE (Microsoft Visual Studio) with a specific scheduling in CHESS. In this way, the CHESS scheduler, rather than the operating system, can control the inter-leaving of threads.

2.4.1.2 Test Execution Setup and Configuration

The test execution setup and configuration includes two parts: (1) setting up and configurating the test components: testers (for live-user based executions) or load drivers (for driver based and emulation based executions); (2) configuring the test environment.

Setting Up and Configuring the Test Components

Depending on the execution approaches, the test components for setup and configuration are different:

- **Tester Recruitment, Setup and Training (Live-user based executions)**

  For live-user based executions, the three main steps involved in the test execution setup and configuration aspects [31, 174] are:

  1. **Tester Recruitment**

     Testers are hired to perform load tests. There are specific criteria to select live users depending on the testing objectives and type of system. For example, for web-applications, individual users are picked based on factors like geographical locations, languages, operating systems and browsers;

  2. **Tester Setup**

     Necessary procedures are carried out to enable testers to access the systems (e.g., network permission, account permission, monitoring and data recording software installation);

  3. **Tester Training**

     The selected testers are trained to be familiar with the target system and their testing scenarios.

- **Load Driver Deployment (Driver based and emulation based executions)**

  Deploying the load drivers involves the installing and configuring of load drivers:

1. **Load Driver Installation**

   The installation of load drivers is usually straight-forward [10, 33], except for the peer-to-peer load drivers. Dumitrescu *et al.* [107] implement a framework to automatically push the peer load drivers to different machines for load testing GRID systems. The framework picks one machine in the GRID to act as a controller. The controller pushes the peer load driver to other machines, which are responsible for requesting web services under test.

2. **Load Driver Configurations**

   The configuration of load drivers is the process of encoding the load as inputs, which the load drivers can understand. There are currently four general load driver configuration techniques:

   (a) **Simple GUI Configuration**

   Some load drivers (especially the benchmark suites like [18]) provide a simple graphical user interface for load test practitioners to specify the rate of the requests as well as test durations.

   (b) **Programable Configuration**

   Many of the general purpose load drivers let load test practitioners encode the testing load using programming languages. The choice of programming languages vary between load drivers. For example, the language could be generic programming languages like C++ [10], Javascript [33] and Java [198]; or domain specific languages, which enable easy specifications of test environment like the setup/configration of database, network and storage [108] and or for specialized systems (e.g., TTCN-3 for telecommunication systems [219]).

   (c) **Store-and-replay Configuration**

   Rather than directly encoding the load via coding, many load drivers support

store-and-replay to reduce the programming efforts. Store-and-replay load driver configuration techniques are used in web-based applications [10, 33, 100] and wireless mobile applications [225, 226]. This configuration technique consists of the following three steps:

i. **The Storing Phase**:

During the storing phase, load test practitioners perform a sequence of actions for each scenario. For example, in a web-based system, a user would first login to the system, browse a few catalogs then logout. A probe, which is included in the load drivers, is used to capture all incoming and outgoing data. For example, all HTTP requests can be captured by either implementing a probe at the client browser side (e.g., browser proxy in WebLoad [10, 33]) or at the network packet level using a packet analyzer like Wireshark [35]. The recorded scenarios are encoded in load-driver specific programming languages (e.g., C++ [10] and Javascript [33]).

Rich Internet Applications (RIA) dynamically update parts of the web page based on the user actions. Therefore, the user action sequences cannot be easily use in record-and-replay via URL editing. Instead, The store-and-replay is achieved via using GUI automation tools like Selenium [24] to record user actions instead.

ii. **The Editing Phase:**

The recorded data needs to be edited and customized by load test practitioners in order to be properly executed by the load driver. The stored data is usually edited to remove runtime-specific values (e.g., session IDs and user IDs).

iii. **The Replaying Phase**: Once load test practitioners finish editing, they

need to identify the rates of these scenarios, the delay between individual requests and the test duration.

(d) **Model Configuration**

Section 2.3.1.2 explains realistic load design techniques via usage models. There are two approaches to translate the usage models into load driver inputs: on one hand, many load drivers can directly take usage models as their inputs. On the other hand, works have been proposed to automatically generate load driver configuration code based on the usage models.

i. **Readily Supported Models**:

Test cases formulated in Markov chain can be directly used in load test execution tools like LoadRunner [10] and Apache JMeter [3] (through plugin) or research tools like [222].

ii. **Automated Generation of Load Driver Configuration Code**

Many techniques have been proposed to automatically generate load driver configuration code from usage models. Silveira *et al.* [95] automatically generate LoadGen scripts based on UML activity diagram. The Stochastic Form Charts can be automatically encoded into JMeter scripts [80, 81].

Configuring the Test Environment

As mentioned above, live-user based and driver based executions usually take place in a lab environment. Extra care is needed to configure the test environment to be as realistic as possible.

First, it is important to understand the implication of the hardware platforms. Netto *et al.* [209] and White *et al.* [243] evaluate the stability of the generated load under virtualized environments (e.g., virtual machines). They find that the system throughput sometimes might not produce stable load on virtual machines. Second, additional operating system

configurations might need to be tuned. For example, Kim [167] reported that extra settings need to be specified in Windows platforms in order to generate hundreds or millions of concurrent connections. Last, it is crucial to make network behavior as realistic as possible. This is covered in two aspects:

1. **Network Latency**

   Many load-driver based test execution techniques are conducted within a local area network, where packets are delivered swiftly and reliably. The case of no/little packet latency is usually not applicable in the field, as packets may be delayed, dropped or corrupted. IP Network Emulator Tools like Shunra [25, 188] are used in load testing to create a realistic load testing network environment [177].

2. **Network Spoofing**

   Routers sometimes try to optimize overall network throughput by caching the source and destination. If the requests come from the same IP address, the network latency measure won't be as realistic. In addition, some systems perform traffic controls based on requests from different network addresses (IP addresses) for purposes like guarding against Denial of Service (DoS) attacks or providing different Quality of Services. **IP Spoofing** in a load test refers to the practice of generating different IP addresses for workload requests coming from different simulated users. IP Spoofing is needed to properly load test some web-based systems using the driver based executions, as these systems usually deny large volume of requests from the same IP addresses to protect against the DoS attacks. IP spoofing is usually configured in supported load drivers (e.g. [10]).

2.4.2 Load Generation and Termination

This subsection covers three categories of load generation and termination techniques: manual load generation and termination techniques (Section 2.4.2.1), load generation and

termination based on static configurations(Section 2.4.2.2), and load generation and termination techniques based on dynamic system feedback (Section 2.4.2.3).

2.4.2.1 Manual Load Generation and (Timer-based) Termination Techniques

Each user repeatedly conducts a sequence of actions over a fixed period of time. Sometimes, actions among different live users need to be coordinated in order to reach the desired load.

2.4.2.2 Static-Configuration-Based Load Generation and Termination Techniques

Each load driver has a controller component to generate the specified load based on the configurations [10, 33, 64]. If the load drivers are installed on multiple machines, the controller needs to send messages among distributed components to coordinate among the load drivers to generate the desired load [107].

Each specific request is either generated based on a random number during runtime (e.g., 10% of the time user A is doing browsing) [10, 64] or based on a specific pre-defined schedule (e.g., during the first five minutes, user B is doing browsing) [171, 172].

There are four types of load termination techniques based on pre-defined static configurations. The first three techniques (continuous, timer-driven and counter-driven) exist in many existing load drivers [226]. The fourth technique (statistic-driven) was recently introduced [188, 223] to ensure the validity or accuracy of the data collected.

1. **Continuous**: A load test runs continuously until the load test practitioners manually stop it [226];

2. **Timer-Driven**: A load test runs for a pre-specified test duration then stops [226].

3. **Counter-Driven**: A load test runs continuously until a pre-specified number of requests have been processed or sent [226].

4. **Statistic-Driven:** A load test is terminated once the performance metrics of interest (e.g., response time, CPU and memory) is statistically stable. This means the metrics of interest yield high confidence interval to estimate such value or have small standard deviations among the collected data points [188, 223].

2.4.2.3 Dynamic-Feedback-Based Load Generation and Termination Techniques

Rather than generating and terminating a load test based on static configurations, techniques have been proposed to dynamically steer the load based on the system feedback [62, 63, 66, 67].

Depending on the load testing objectives, the definition of important inputs can vary. For example, one goal is to detect memory leaks [66]. Thus, input parameters that significantly impact the system memory usage, are considered as important parameters. Other goals can be to find/verify the maximum number of users that the system can support before the response time degrades [66] or to locate software bottleneck [62, 63]. Thus, important inputs are the ones that significantly impact the testing objectives (e.g., performance objectives like the response time or throughput).

There are two techniques proposed to locate the important inputs.

1. **System Identification Technique**

   Bayan and Cangussu calculate the important inputs using the System Identification Technique [159, 181]. The general idea is as follows: the metric mentioned in the objectives is considered as the output variable (e.g., memory or response time). Different combinations of input parameters lead to different values in the output variable. A series of random testing, which measures the system performance using randomly generated inputs, would create a set of linear equations with the output variable on one side and various combinations of input variables on the other side. Thus, locating the resource impacting inputs is equivalent to solving these linear equations and

identifying the inputs, which are large (sensitive to the resources of interest).

2. **Analytical Queuing Modeling**

   Compared with the above System Identification Technique, which calculates the important inputs before load test execution starts, Branal *et al.* dynamically model the software system using a two-layer queuing model and use analytical techniques to find the workload mixes that change the bottlenecks in the systems. Branal *et al.* iteratively tune the analytical queuing model based on the system performance metrics (e.g., CPU, disk and memory). Through iteratively driving load, their model gradually narrows down the bottleneck/important inputs.

Once these important inputs are identified, the load driver automatically generate the target load to detect memory leaks [66], to verify system performance requirements [67] or to identify software bottlenecks [62, 63].

2.4.2.4 Deterministic Load Generation and Termination Techniques

Even though all of these load test execution techniques manage to inject many concurrent requests into the system, none of those techniques can guarantee to explore all the possible inter-leavings of threads and timing of asynchronous events. Such system state information is important, as some thread inter-leaving and events could lead to hard to catch and reproduce problems like deadlocks or racing conditions.

As we mentioned in the beginning of this section, the CHESS platform [206] can be used to deterministically execute a test based on all the possible event inter-leavings. The deterministic inter-leaving execution is achieved by the scheduling component, as the actual scheduling during the test execution is controlled by the tool scheduler rather than the OS scheduler.

The CHESS scheduler understands the semantics of all non-deterministic APIs and provides an alternative implementation of these APIs. The alternative API implementations

insert a call to the CHESS scheduler whenever there is a call to these non-deterministic APIs. In addition, the CHESS scheduler creates a lock for each thread.

Upon receiving concurrent requests, whenever there is a call to the non-deterministic API, the calling thread queries the tool scheduler to see if the calling thread should block. If the CHESS scheduler decides the calling thread should block, the calling thread blocks on its lock and the CHESS scheduler releases the next thread supposedly to run next. Therefore, by picking different threads to block at different execution points, the CHESS scheduler is able to deterministically explore all the possible inter-leavings. The test stops when the scheduler explores all the task inter-leavings.

The CHESS platform automatically reports when there is a deadlock or race conditions, along with the exact execution context (e.g., thread interleaving and events).

2.4.3 Test Monitoring and Data Collection

The system behavior under load is monitored and recorded during the course of the load test execution. There is a tradeoff between the level of monitoring details and monitoring overhead. Detailed monitoring has a huge performance overhead, which may slow down the system execution and may even alter the system behavior [207]. Therefore, probing techniques for load testing are usually light weight and are intended to impose minimal overhead to the overall system.

In general, there are four categories of collected data in the research literature: Metrics, Execution Logs, Functional Failures, and System Snapshots.

2.4.3.1 Monitoring and Collecting Metrics

In general, there are two types of metrics getting monitored and collected during the course of the load test execution phase: Throughput Metrics (“Number of Pass/Fail Requests”) and Performance Metrics (“End-to-End Response Time” and “Resource Usage Metrics”). These

metrics are getting tracked by recruited testers in the live-user based executions [31, 173], by load drivers in the driver based and emulation based executions [10, 33, 72, 190, 191, 196, 245] or by light weight system monitoring tools PerfMon or pidstats [173].

1. **Number of Passed and Failed Requests**

   Once the load is terminated, the number of passed and failed requests are collected from live users. This metric can either be recorded periodically (the number of pass and fail requests at this interval) or recorded once at the end of the load test (the total number of pass and failed requests).

2. **End-to-End Response Time**

   The end-to-end response time (or just response time) is the time it takes to complete one individual request.

3. **Resource Usage Metrics**

   System resource usage metrics like CPU, memory, disk and network usage, are collected for the system under load. These resource usage metrics are usually collected and recorded at a fixed time interval. Similar as the end-to-end metrics, depending on the specifications, the recorded data can either be aggregated values or a sampled value at that particular time instance. System resource usage metrics can either be collected through system monitoring tools like PerfMon in Windows or pidstats in Unix/Linux [173]. Such resource usage metrics are usually collected both for the SUT and its associated components (e.g., databases and mail servers).

2.4.3.2 Instrumenting and Collecting Execution Logs

Execution logs are generated by the instrumentation of code that developers insert into the source code. There are three types of instrumentation mechanisms: (1) ad-hoc debug statements, like printf or System.out, (2) general instrumentation frameworks like Log4j [116]

and (3) through specialized instrumentation frameworks like ARM (Application Response Measurement) [132]:

1. **Ad-hoc Logging**: The ad-hoc logging mechanism is the most commonly used, as developers insert output statements (e.g., printf or System.out) into the source code for debugging purposes [141]. However, extra care is required to (1) minimize the amount of information generated, and to (2) to make sure the statements are not garbled as multiple logging threads are trying to write to the same file concurrently.

2. **General Instrumentation Framework**: General instrumentation frameworks like Log4j [116] address the two limitations in the ad-hoc mechanism. The instrumentation framework provides a platform to support thread-safe logging and fine-grained control of information. *Thread-safe logging* makes sure that each logging thread serially accesses the single log file for multi-threaded systems. *Fine-grained logging control* enables developers to specify logging at various levels. For example, there can be many levels of logging suited for various purposes, like information level logs for monitoring and legal compliances [23], and debug level logs for debugging purposes. During load tests and actual field deployments, only higher level logging (e.g., at the information level) is generated to minimize overhead.

3. **Specialized Instrumentation Framework**: Specialized instrumentation frameworks like ARM (Application Response Measurement) [132] can facilitate the process of gathering performance information from running programs.

2.4.3.3 Monitoring and Collecting Functional Failures

Live-user based and emulation based executions record functional problems, whenever the failure occurs. For each request that a live user executes, he/she records whether the request has completed successfully. If not, he/she will note the problem areas (e.g., flash content is

not displayed properly [31]). For the deterministic emulation based execution (the CHESS platform), the detailed system state is recorded when the deadlock or race conditions occur.

2.4.3.4 Monitoring System Behavior and Collecting System Snapshots

Rather than capturing information throughout the course of the load testing, Bertolino *et al.* [72] propose a technique that captures a snapshot of the entire test environment as well as the system state when a problem arises. Whenever the system's overall QoS is below some threshold, all network requests as well as snapshot of the system state is saved. This snapshot can be replayed later for debugging purposes.

2.4.4 Summary and Open Problems

There are three general load test execution approaches: (1) the live-user based executions, where recruited testers manually generate the testing load; (2) the driver based executions, where the testing load is automatically generated; and (3) the emulation based executions, where the SUT is executed on top of special platforms. Live-user based executions provide the most realistic feedback on the system behavior, but suffer from scalability issues. Driver based executions can scale to large testing load and test durations, but require substantial efforts to deploy and configure the load drives for the targeted testing load. Emulation based executions provide special capacities over the other two execution approaches: (1) early examination of system behavior before system is fully implemented, (2) easy detection and reporting of load problems. However, emulation based execution techniques can only focus on a small subset of the load testing objectives.

Here, we list two open problems, which are still not explored thoroughly:

– **Encoding Testing Loads into Testing Tools**

  It is not straight-forward to translate the designed load into inputs used by load drivers. For example, the load resulted from hybrid load optimization techniques [172]

is in the form of traces. Therefore, load drivers need to be modified to take these traces as inputs and replay the exact order of these sequences. However, if the size of traces become large, the load driver might not be able to handle traces. Similarity, testing load derived from deterministic state testing [51, 53] is not easily realized in existing load drivers, either.

– **System Monitoring Details and Load Testing Analysis**

On one hand, it is important to minimize the system monitoring overhead during the execution of a load test. On the other hand, the recorded data might not be sufficient (or straight-forward) for load testing analysis. For example, recorded data (e.g., metrics and logs) can be too large to be examined manually for problems. Additional work is needed to find proper system monitoring data suited for load testing.

2.5 Research Question 3: How is the result of a load test analyzed?

During the load test execution phase, the system behavior (e.g., logs and metrics) is recorded. Such data must be analyzed to decide whether the SUT has met the test objectives. Different types of data and analysis techniques are needed to validate different test objectives.

As discussed in Section 2.4.3, there are four categories of system behavior data: metrics, execution logs, functional failures and system snapshots. All of the research literature focuses on the analysis and reporting techniques used for working with metrics and execution logs. (It is relatively straight-forward to handle the functional failure data by reporting them to the development team, and there is no further discussion on how to analyze system snapshots [72]).

There are three categories of load testing analysis approaches:

1. **Verifying Against Threshold Values**

Some system requirements under load (especially non-functional requirements) are defined using threshold values. One example is the system resource requirements. The CPU and memory usage cannot be too high during the course of a load test, otherwise the request processing can hang and system performance can be unstable [117]. Another example is the reliability requirement for safety critical and telecommunication systems [51, 53]. The reliability requirements are usually specified as "three-nines" or "five-nines", which means the system reliability cannot be lower than 99.9% (for "three-nines") and 99.999% (for "five-nines"). The most intuitive load test analysis technique is to summarize the system behavior into one number and verify this number against a threshold. The usual output for such analysis is simply pass/fail.

2. **Detecting Known Problems**

   Another general category of load test analysis is examining the system behavior to locate patterns of known problems; as some problems are buried in the data and cannot be found based on threshold values, but can be spotted by known patterns. One example is to check the memory growth trend over time for memory leaks. The usual output for such analysis is a list of detected problems.

3. **Automated Detection of Anomalous Behavior**

   Unfortunately, not all problems can be specified using patterns and certainly not all problems have been detected previously. In addition, the volume of recorded system behavior is too big to examine manually. Therefore, automated techniques have been proposed to systematically analyze the system behavior to uncover anomalous behavior. These techniques usually apply statistical or artificial intelligence methods to automatically derive "normal/expected behavior" and flag "anomalous behavior" from the data. However, the accuracy of such techniques might not be as high as the above two approaches, as the "anomalous behavior" are merely hints of potential system problems under load. The output for such analysis is usually the anomalous

behavior and some reasoning/diagnosis on the potential problematic behavior.

All three aforementioned techniques can analyze different categories of data to verify a range of objectives (detecting functional problems and non-functional problems). These load test analysis techniques can be used individually or together based on the types of data available and the available time. For example, if time permitted, load testing practitioners can verify against known requirements based on the threshold, locate problems based on specific patterns and run the automated anomaly detection techniques just to check if there are any more problems. We categorize the load test analysis technique into the following six dimensions as shown in Table 2.5.

- **Approaches** refer to one of the above three load test analysis approaches.
- **Techniques** refer to the load test analysis technique like memory leak detection.
- **Data** refers to the types of system behavior that the test analysis technique can analyze. Examples are execution logs and performance metrics like response time.
- **Test Objectives** refer to the goal or goals of load test objectives (e.g., detecting performance problems), which the test analysis technique achieve.
- **Reported Results** refer to the types of reported outcomes, which can simply be pass/fail or detailed problem diagnoses.
- **References** refer to the list of literatures, which propose each technique.

This section is organized as follows: The next three subsections describe the three categories of load testing analysis techniques respectively: Section 2.5.1 explains the techniques of verifying load test results against threshold values, Section 2.5.2 describes the techniques of detecting known problems, and Section 2.5.3 explains the techniques of automated anomaly detection. Section 2.5.4 summarize the load test analysis techniques and propose some open problems.

Table 2.5: Load Test Analysis Techniques

Approaches	Techniques	Data	Test Objectives	Reported Results	References
Verifying Against Threshold Values	Straight-forward Comparison	Performance metrics	Detecting performance and scalability problems	Pass/Fail	[104, 232, 249]
	Comparison Against Processed Data (Max, medium or 90-percentile values)	Periodic sampling metrics	Detecting performance problems		[104, 107, 117, 127, 130, 141, 142, 143, 144, 191, 251]
	Comparison Against Derived (Threshold and/or target) Data	Number of pass/fail requests, past performance metrics	Detecting performance and reliability problems		[109, 150, 191, 192]
Detecting Known Problems	Memory Leak Detection	Memory usage metrics	Detecting functional problems	Pass/Fail	[66, 74]
	Locating Error Keywords	Execution logs	Detecting functional problems	Error log lines and error types	[154]
	Detecting Throughput Problems Using Queuing Theory	Throughput, response time metrics	Detecting functional and scalability problems	Pass/Fail	[188]
Automated Detection of Anomalous Behavior	Detecting Anomalous Performance Behavior using Performance and Resource Usage Metrics	Performance and resource usage metrics	Detecting performance problems	Anomalous performance metrics	[115, 185, 186, 187, 210, 229, 230]

2.5.1 Verifying Against Threshold Values

The threshold-based test analysis approach can be further broken down into three techniques based on the availability of the data and threshold values.

2.5.1.1 Straight-forward Comparison

When the data is available and the threshold requirement is clearly defined, load testing practitioners can perform a straight-forward comparison between the data and the threshold values. One example is throughput analysis. Throughput, which is the rate of successful requests completed, can be used to compare against the load to validate whether the system's functionality can scale under load [104, 232, 249].

2.5.1.2 Comparison Against Processed Data

If the system resources, like CPU and memory utilization are too high, the system performance may not be stable [117] and user experience could degrade (e.g., slow response time) [43, 52, 114, 177, 199].

There can be many formats of system behavior. One example is resource usage data, which is sampled at a fixed interval. Another example is the end-to-end response time, which is recorded as response time for each individual request. These types of data need to be processed before comparing against threshold values. On one hand, as Bondi pointed out [74], system resources may fluctuate during the startup time for warmup and cooldown period. Hence, it is important to only focus on the system behavior once the system reaches a stabilized state. On the other hand, a proper data summarization technique is needed to describe these many data instances into one number. There are three types of data summarization techniques proposed in the literature. We use response time analysis as an example to describe the proposed data summarization techniques:

1. **Maximum Values**

For online distributed multi-media systems, if any video and audio packets are out of synch or not delivered in time, it is considered a failure [251]. Therefore, the inability of the end-to-end response time to meet a specific threshold (e.g., video buffering period) is considered as a failure.

2. **Average or Medium Values**

   The average or medium response time summarizes the majority of the response times during the load test and is used to evaluate the overall system performance under load [104, 107, 141, 142, 143, 144, 191].

3. **90-percentile Values**

   Some researchers advocate the 90-percentile response time is a better measurement than the average/medium response time [117, 127, 130], as 90-percentile response time accounts for most of the peaks, while eliminating the outliers.

2.5.1.3 Comparison Against Derived Data

In some cases, either the data (e.g., system reliable under load) to compare or the threshold value is not directly available. Extra steps need to be taken to derive this data before analysis.

– **Deriving Threshold**

  Some other threshold values for non-functional requirements are informally defined. One example is the "no-worse-than-before" principle when verifying the overall system performance. The "no-worse-than-before" principle states that the average response time (system performance requirements) for the current version should be at least as good as prior versions [150].

– **Deriving Target Data**

  There are two method of deriving the target data to be analyzed:

– **Through Extrapolation:** As mentioned in Section 2.3.3.2, due to time or cost limitations, sometimes it is not possible to run the targeted load, but we might run tests with lower load levels (same workload mix but different intensity). Based on the performance of these lower workload intensity level tests, load test practitioners can extrapolate the performance metrics at the targeted load [192, 191, 109]. If certain resource metrics are higher than the hardware limits (e.g., requires more memory than provided or CPU is greater than 100%) based on the extrapolation, scalability problems are noted.

– **Through Bayesian Network:** Software reliability is defined as the probability of failure-free operation for a period of time, under certain conditions. Mission critical systems usually have very strict reliability requirements. Avritzer *et al.* [55, 51] uses the Bayesian Network to estimated the system reliability from the load test data. Avritzer *et al.* use the failure probability of each type of load (workload mix and workload intensity) and the likelihood of these types of load occurring in the field. Load test practitioners can then use such reliability estimates to track the quality of the SUT across various builds and decide whether the SUT is ready for release.

2.5.2 Detecting Known Problems

There are three known load-related problems, which can be analyzed using patterns: memory leak detection(Section 2.5.2.1), searching for error keywords(Section 2.5.2.2), and detecting throughput problems using queuing theory(Section 2.5.2.3).

2.5.2.1 Memory Leak Detection

Memory leaks can cause long running systems to crash. Memory leak problems can be detected if one of the two scenarios are present: (1) an upward trend of the memory footprint

throughout the course of load testing [157]; or (2) the number of processed requests drops with the same memory footprint as the load test progresses [66, 74].

2.5.2.2 Locating Error Keywords

Execution logs, generated by code instrumentations, provide textual descriptions of the system behavior during runtime. Compared to system resource usage data, which are structural and easy to analyze, execution logs are more difficult to analyze, but provide more in-depth knowledge.

One of the challenges of analyzing execution logs is the size of the data. At the end of a load test, the size of execution logs can be several hundred megabytes or gigabytes. Therefore, automatic log analysis techniques are needed to scan through logs to detect problems.

Load testing practitioners can search for specific keywords like "errors", "failures", "crash" or "restart" in the execution logs [154]. Once these log lines are found, load test practitioners need to analyze the context of the matched log lines to determine whether they indicate problems or not.

2.5.2.3 Detecting Throughput Problems Using Queuing Theory

Mansharamani *et al.* [188] uses Little's Law from Queuing Theory to validate the load test results:

$$Throughput = \frac{Number\ of\ users}{Response\ Time + Average\ Think\ Time}$$

If there is a big difference between the calculated and measured throughput, there could be failure in the transactions or load variations (e.g., during warm up or cool down) or load generation errors (e.g., load generation machines cannot keep up with the specified loads).

2.5.3 Automated Detection of Anomalous Behavior

Current automated anomaly detection approaches are focuses on analyzing the resource usage and response time metrics. There are five techniques proposed to derive the "expected/normal" behavior and flag "anomalous" behavior based on response time or resource usage data:

2.5.3.1 Deriving and Comparing Clusters

As noted by Georges *et al.* [123, 207], it is important to execute the same tests multiple times to gain a better view of the system performance due to issues like system warmup and memory layouts. Bulej *et al.* [79] propose the use of statistical techniques to detect performance regressions (performance degradations in the context of regression testing). Bulej *et al.* repeatedly execute the same tests multiple times. Then, they group the response time for each request into clusters and compare the response time distributions cluster-by-cluster. They have used various statistical tests (Student-t test, Kolmogorov-Smirnov Test, Wilcoxon test, Kruskal-Wallis test) to compare the response time distributions between the current release and prior releases. The results in their case studies show that these statistical tests yield similar results.

2.5.3.2 Deriving Clusters and Finding Outliers

Rather than comparing the resulting clusters as in [79], Syer et al. [229, 230] use a hierarchical clustering technique to identify outliers, which represent threads with deviating behavior in a thread pool. A thread pool, which is a popular design pattern for large scale software systems, contains a collection of threads available to perform the same type of computational tasks. Each thread in the thread pool performs similar tasks and should exhibit similar behavior with respect to resource usage metric, such as CPU and memory usage. Threads with performance deviations likely indicate problems, such as deadlock or

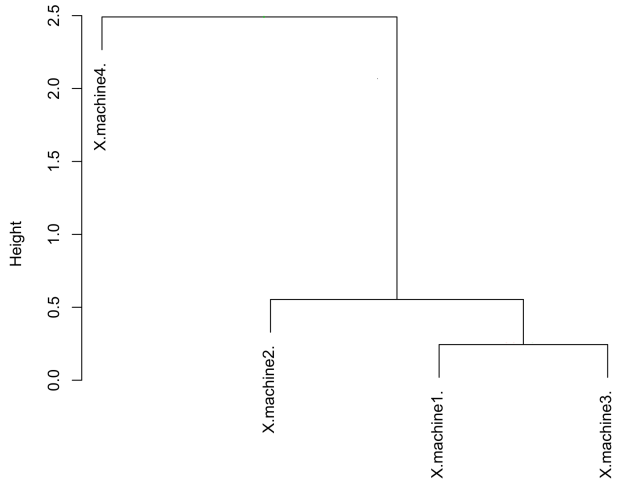


Figure 2.8: An Example of a Hierarchical Clustering Diagram from [229]

memory leaks.

Figure 2.8 shows an example of a hierarchical clustering diagram (from [229]), which consists of a set of nested clusters organized as a tree. Each leaf node represents a thread. If two threads' performance are similar, they are joined closer to the bottom of the tree (e.g., X.machine1 and X.machine3). If two threads' performance are very different, they are very far apart and only joined at the top of the tree (e.g., X.machine1 and X.machine4). In this diagram, the performance from X.machine4 is different from all other threads.

2.5.3.3 Deriving Performance Ranges Using Control Charts

As shown in Figure 2.9, a control chart consists of three parts: a control line (center line), a lower control limit (LCL) and an upper control limit. If a point lies outside the controlled regions (between the upper and lower limits), the point is counted as a violation. Control charts are used widely in the manufacturing process to detect anomalies.

Nguyen *et al.* [210] use control charts to flag anomalous resource usage metrics. For each recorded resource usage metrics, Nguyen *et al.* [210] derive the "expected behavior" in the form of control chart limits based on prior good tests. Then current test data is

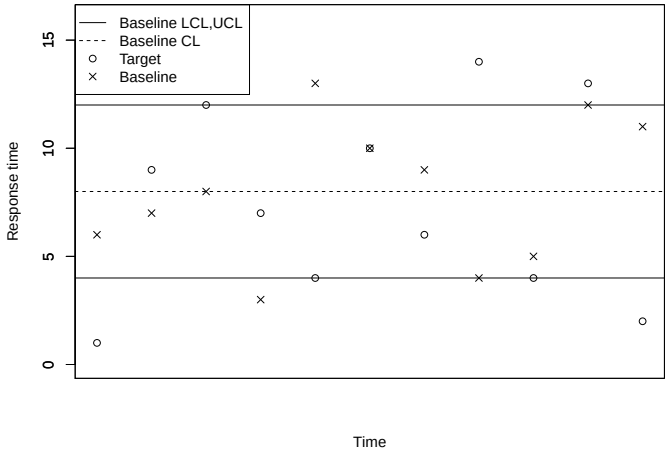


Figure 2.9: An example of a Control Chart from [210] with lower and upper control limits shown as solid lines (LCL and UCL), control line shown as dotted lines (CL), and the data points of the performance metric shown as cycles (target)

overlayed on the control chart. If the examined performance metric (e.g., subsystem CPU) has a high number of violations, this metric is flagged as an anomaly and is reported to the development team for further analysis.

2.5.3.4 Deriving Performance Rules Using AI Techniques

Nguyen *et al.* [210] treat each metric separately and derive range boundary values for each of these metrics. However, in many cases, these metrics are correlated with each other. For example, when the system is processing a large number of requests, the CPU usage and memory usage could be high.

Foo *et al.* [115] build performance rules, and flag metrics, which violates these rules. A performance rule groups a set of correlating metrics. For example, a large number of requests imply high CPU and memory usage. For all the past tests, Foo *et al.* first categorize each metrics into one of high/medium/low categories, then derive performance rules by applying an artificial intelligence technique, called Association Rule mining. The performance rules (association rules) are derived by finding frequent co-occurred metrics.

For example, if high browsing requests, high Database CPU and high web server memory footprint always appear together, *Browsing/DB CPU/Web Server Memory* form a set (called "frequent-item-set"). Based on the frequent-item-set, association rules can be formed (e.g., high browsing requests and high web server memory implies high database CPU). Metrics from the current test are matched against these rules. Metrics (e.g., low database CPU), which violates these rules, are flagged as "anomalous behavior".

2.5.3.5 Deriving Performance Signatures Using Statistical Techniques

Rather than finding and grouping related metrics by AI techniques like in [115], Malik *et al.* [185, 186, 187] use statistical techniques to select the most important metrics among hundreds or thousands of metrics and group these metrics into relevant groups, called "Performance Signature".

The statistical technique, which Malik *et al.* used, is called Principal Component Analysis (PCA). First, Malik *et al.* normalize all metrics into values between 0 and 1. Then PCA is applied to show the relationship between metrics. PCA groups metrics into groups, called Principle Components (PC). Each group has a value called variance, which explains the importance/relevance of the group to explain the overall data. The higher the variance values of the groups, the more relevant these groups are. Furthermore, each metric is a member of all the PCs, but the importance of the metrics within one group varies. The higher the Eigen-value of a metric within one group, the more important the metric is to the group. Malik *et al.* select first N Principle Components with then largest variance. Then within each Principle Component, Malik *et al.* select important counters by calculating pair-wise correlations between counters. These important counters forms the "Performance Signatures". The performance signatures are calculated on the past good tests and the current test, respectively. The discrepancies between the performance signatures are flagged as "Anomalous Behavior".

2.5.4 Summary and Open Problems

Depending on the types of data and test objectives, there are different load test analysis techniques that have been proposed. There are three general test analysis approaches: verifying the test data against fixed threshold values, searching through the test data for known problem patterns and automated detection of anomalous behaviors.

Below are a few open problems:

- **Limited Load Test Analysis Work**

  Load testing analysis is challenging (due to the volume of data and limited time), yet very little test analysis work has been proposed, especially on automated anomaly detection. Most of the load test analysis work is done within our research team [115, 185, 186, 187, 210], which focus on analyzing the metrics. There is little work on automated anomaly detection on execution logs.

- **Can we use system monitoring techniques to analyze load test data?**

  Many research ideas in production system monitoring may be applicable for load testing analysis. For example, works [40, 90, 152, 136] have been proposed to build performance signatures based on the past failures, so that whenever such symptoms occur, the problems can be detected and notified right away. Analogously, we can formulate our performance signature based on mining the past load testing history and use these performance signatures to detect recurrent problems in load tests. A promising research area is to explore the applicability and ease of adapting system monitoring techniques for the analysis of load tests.

2.6 Conclusion

To ensure the quality of large scale systems, load testing is required in addition to conventional functional testing procedures. Furthermore, load testing is becoming more important, as an increasing number of services are being offered in the cloud to millions of users. However, as observed by Visser [236], load testing is a difficult task requiring a great understanding of the system under test. In this chapter, we surveyed techniques used in the three phases of a load test: the load design phase, the load execution phase, and the load test analysis phase. We compared and contrasted these techniques and provided a few open research problems in each phase. One of the key findings highlighted in this survey is that little research has been done on the analysis of the behavior of the system under load, especially on automated analysis of large volume of load testing data to assess the system quality under load. This finding motivates our research proposed in subsequent part of this thesis (Chapters 3, 4, 5 and 6).

CHAPTER 3

Automated Abstraction of Execution Logs

Motivated by Chapter 2's finding, we set out to explore automated approaches to analyze the results of load tests (execution logs). Execution logs are generated by output statements which developers insert into the source code. Execution logs are widely available and are helpful in monitoring, remote issue resolution, and system understanding of complex enterprise applications. There are many proposals for standardized log formats such as the W3C and SNMP formats. However, most applications use ad-hoc nonstandardized logging formats. Automated analysis of such logs is complex due to the loosely defined structure and a large non-fixed vocabulary of words. The large volume of logs, produced by enterprise applications, limits the usefulness of manual analysis techniques. Automated techniques are needed to uncover the structure of execution logs. Using the uncovered structure, sophisticated analysis of logs can be performed.

In this chapter, we propose a log abstraction technique which recognizes the internal structure of each log line. Using the recovered structure, log lines can be easily summarized and categorized to help comprehend and investigate the complex behavior of large software applications. Our proposed approach handles free-form log lines with minimal requirements on the format of a log line. Through a case study using log files from four enterprise applications, we demonstrate that our approach abstracts log files of different complexities with high precision and recall.

3.1 Introduction

CHAPTER 2 reports that few research efforts has been devoted on the automated analysis of large volume of load testing data. Motivated by this finding, we set out to explore this important and practical research topic in the remaining part of

this thesis. There are two types of artifacts recorded during a load test: execution logs and metrics. We focus on analyzing the collected execution logs, since logs are widely available and there is little research on logs compared to metrics [115, 185, 186, 187, 210].

Execution logs are generated by output statements, which developers insert into the source code. Logs record application events at run-time. Logs help in monitoring, remote issue resolution and system understanding of complex enterprise applications. Tracing and execution logs are two types of logs that are commonly used for dynamic analysis. Tracing logs are generated by instrumenting or monitoring an application using a variety of technologies such as the Java Virtual Machine Profiler Interface (JVMPI). In contrast, execution logs are inserted by developers during the development of an application. Execution logs are at a higher level of abstraction than tracing logs. Developers use execution logs to track domain level events (e.g., "Login Verified"), instead of tracking implementation level events (e.g., "Function CheckPassword() Called"). There is an abundance of execution logs in the field, whereas tracing logs must be produced for specific scenarios. Producing tracing logs may not be possible in a production setting due to the performance overhead, the lack of system knowledge, and the inaccessibility of the source code. The availability of execution logs continues to increase at a rapid rate due to recent legal acts. For example, the Sarbanes-Oxley Act of 2002 [23] stipulates that the execution of telecommunication and financial applications must be logged.

Execution logs are hard to parse and analyze automatically since they are free-form with no strict format and with a large vocabulary of words. Several standard log formats (e.g., [2, 19, 26]) have been proposed to ease the automated analysis of logs. However, such standards are rarely used in practice. Modifying a legacy application to follow a particular standard is usually not feasible nor economically possible due to the lack of resources, the limited system knowledge or the inaccessible source code for off-the-shelf applications.

In this chapter, we present an approach which can uncover the structure of log lines. The approach examines each log line and abstracts it to its corresponding execution event.

Table 3.1: Example log lines

1.	User checkout for accountId(*tom*), item=*100*
2.	User checkout for accountId(*jerry*), item=*100*
3.	Item shipped for accountId(*tom*), item=*100*
4.	User checkout for accountId(*john*), item=*103*

Although execution logs may not follow a strict format, they have one general structure: a log line is a mixture of static and dynamic information. Each log line contains static information indicating the execution event, and dynamic information which is specific to the particular occurrence of the execution event. The dynamic information causes the same execution event to result in different log lines. Looking at Table 3.1, the italic tokens are dynamic information generated at runtime. The rest of a line is static information. Our approach would identify that logs lines in Table 3.1 correspond to two execution events: "User checkout" and "Item Shipped". For example, the first log line would be abstracted to the "User checkout" execution event.

Once a log file is processed and the execution event corresponding to each log line is identified, a developer can use the recovered log file structure to understand and investigate the dynamic behaviour of a software application. Our abstraction approach reduces the volume of data that a developer needs to investigate. Moreover the abstracted events can be used to reason about important events in a log file. Depending on the type of information recorded and the logging details, different types of analysis can be conducted. For example, the log file of one application in our case study (*see* Section 3.5) contains more than 1.6 million log lines. More than $23,000$ lines in this log file, contain the word "fail" or "failure". A developer investigating these failures would need to manually go through each one of these lines. Alternatively, our approach abstracts these 1.6 million log lines to 319 execution events. Among these 319 events, there are 12 types of failure events which range from external protocol communication failures to internal synchronization failures. Using the frequency of failure events, the developer can prioritize his work by tackling the most frequently occurring failures first.

Organization of the Chapter

This chapter is organized as follows. Section 3.2 overviews related work in the field and places our contributions relative to prior work. Section 3.3 gives an overview of the process for abstracting log lines. We use precision and recall, two traditional information retrieval metrics, to measure the performance of log abstraction approaches. Section 3.4 presents our approach of abstracting log lines to execution events. Section 3.5 demonstrates the effectiveness of our approach through a case study using log files from four enterprise software applications. We also discuss lessons learned from our study. Section 3.6 concludes the chapter.

3.2 Related Work

Uncovering the structure of free-form text is commonly referred to as the grammar inference problem [96, 99]. Prior approach for inferring the grammar of execution logs (i.e., abstracting log lines to execution events) could be grouped under three general approaches: **Rule-based**, **Codebook-based** and **AI-based** approaches.

Rule-based approaches [97, 138, 175, 234] use a set of hard coded rules for abstracting log lines to execution events. These approaches are commonly used in practice since they are very accurate. However these approaches require a substantial effort for encoding and updating the rules. For the logs shown in Table 3.1, a rule-based approach would define two regular expressions to map each log line to one of the two possible execution events: the "user checkout" or "item shipped" events.

Codebook-based approaches [134, 168, 217] are similar to the rule-based approach. However codebook approaches process a subset of execution events ("alarms") instead of all events. The subset of events, which forms the codebook, is used in real-time to match the observed symptoms. For the logs shown in Table 3.1, a codebook-based approach may consider only tracking the "item shipped" events so an alarm for that event would be created

Table 3.2: Summary of related work

Approach	Interpretability	Needed Knowledge	Required Effort	Coverage
Rule-based [97, 138, 175, 234]	Y	High	High	N
Codebook-based [134, 168, 217]	Y	Medium	High	N
AI-based [135, 148, 149, 178, 221, 227, 233, 244]	N	Low	Low	N
Our approach	Y	Low	Low	Y

using a regular expression.

AI-based approaches [135, 148, 149, 178, 221, 227, 233, 244] use various types of artificial intelligent techniques such as, Bayesian networks, frequent-itemset mining, to abstract execution logs to execution events. For the logs shown in Table 3.1, a frequent-itemset approach would recognize the high repetition of the "user checkout" event. However, the approach would not recognize the "item shipped" event since it does not occur that frequently.

Our approach, presented in Section 3.4, is a mixture of rule-based and AI-based approaches. Our approach requires less system knowledge and effort than other approaches. Rather than encoding rules to recognize specific execution events, our approach uses a few general heuristics to recognize static and dynamic information in log lines. Log lines with identical static information are then grouped together to abstract log lines to execution events.

We define several criteria (Table 3.2) to summarize the difference between these four log abstraction approaches.

1. **Interpretability**: Whether a user can easily understand the rationale for abstracting a log line to a particular execution event? For example, in a neural-network AI approach, the user cannot determine the rationale for abstracting a log line to a particular event. We desire an approach with high transparency so users would trust it and adopt it.

2. **Needed system knowledge**: What is the amount of knowledge needed about the system for the approach to work? For example, in a rule-based approach a domain expert is needed to encode all the rules.

3. **Required effort**: What is the amount of effort required for the approach to work properly? Rule-based and cookbook-based approaches require a large amount of human effort to encode the rules or alarms. These encodings must be updated for every version of a software system.

4. **Coverage**: Is each log line abstracted to an appropriate execution event? For example, some AI approaches only abstract log lines, which occur above a particular threshold.

3.3 Measuring the Performance of Approaches for Abstraction Logs

Given the four log lines shown in Table 3.1, a log abstraction approach would determine that log lines 1, 2, and 4 correspond to the execution event: "User checkout for accountId($v), item=$v", where $v indicates the dynamic parts of an execution event. Log line 3 corresponds to another execution event, namely "Item shipped for accountId($v), item=$v". Due to the simple structure of the log lines used in our example, a log abstraction approach

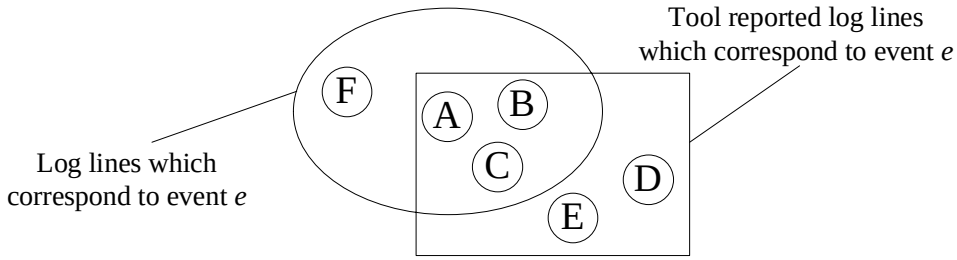


Figure 3.1: Measuring the performance of a log abstraction approach

could easily perform the abstraction. However, in practice the structure of log lines is not as simple and the performance of each approaches differs between applications.

We use precision and recall, two traditional information retrieval metrics, to measure the performance of different approaches for abstracting log lines to execution events. We first measure the performance of an approach for each execution event, then we sum up the performance for each log line to determine the overall performance of the approach. Given a single execution event (e), we know that log lines (A, B, C, F) correspond to e (see Figure 3.1). On the other hand, the log abstraction approach abstracted log lines (A, B, C, D, E) to event e. Therefore, the log abstraction approach correctly classified log lines: A, B, C, incorrectly classified log lines: D, E, and missed classifying event F. For event e, we define the number of log lines classified correctly as PC_e, the number of lines classified incorrectly as PF_e, the number of missed lines as PM_e. Using the information from Figure 3.1, we have for event e:

$$PC_e = \{A, B, C\},\ PF_e = \{D, E\},\ and\ PM_e = \{F\}$$

We define precision and recall as:

$$precision = \frac{PC_e}{PC_e + PF_e},\ and\ recall = \frac{PC_e}{PC_e + PM_e}$$

Therefore, the approach used in our example would have a recall of $\frac{3}{3+2}$ or 60%, and a

precision of $\frac{3}{3+1}$ or 75%.

In the ideal situation, PF_e and PM_e are empty and then both precision and recall would reach their maximum value. The maximum value for precision and recall is 1. We desire an approach with high precision and high recall.

3.3.1 Average Performance

The above formulas measure performance for a particular execution event (e). To measure the overall performance of an approach, we define the average precision and the average recall which combine the precision and recall measures for all unique k events (e_1, e_2, ... , e_k) in a log file as follows:

$$Average\ Precision = \frac{1}{k} \times \sum_{1}^{k} precision_{e_i},\ and$$

$$Average\ Recall = \frac{1}{k} \times \sum_{1}^{k} recall_{e_i}$$

We use the average precision and recall measures in the rest of the chapter to compare log abstraction approaches. We desire an approach which maximizes precision and recall to reduce the need for manual analysis of log lines.

3.4 Our Log Abstraction Approach

In this section, we present our log abstraction approach. Our approach uses clone detection techniques to uncover common tokens in logs lines and to parameterize each log line. We first report on our experience in using an off-the-shelf clone detection tool to perform the log abstraction then present our approach in detail.

linux-2.6.16.13\drivers\net\ne2.c : 285 - 295

```
#ifndef MODULE
struct net_device * __init ne2_probe(int unit)
{
        struct net_device *dev = alloc_ei_netdev();
        int err;

        if (!dev)
                return ERR_PTR(-ENOMEM);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_ne2_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(A)

linux-2.6.16.13\drivers\net\lne390.c: 152 - 162

```
#ifndef MODULE
struct net_device * __init lne390_probe(int unit)
{
        struct net_device *dev = alloc_ei_netdev();
        int err;

        if (!dev)
                return ERR_PTR(-ENOMEM);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_lne390_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(B)

linux-2.6.16.13\drivers\net\lance.c: 437 - 447

```
#ifndef MODULE
struct net_device * __init lance_probe(int unit)
{
        struct net_device *dev = alloc_etherdev(0);
        int err;

        if (!dev)
                return ERR_PTR(-ENODEV);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_lance_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(C)

Figure 3.2: Clone example taken from Linux kernel version 2.6.16.13

3.4.1 Clone Detection

Log lines generated due to the same execution event tend to have high textual similarities (see lines 1, 2, and 4 in Table 3.1). The process of abstracting log lines to execution events can be considered as detecting and grouping similarities among log lines. This intuition leads us to consider using clone detection approaches for abstracting log lines to events.

Code clones refer to identical or similar segments of source code. Code clones are created by copy-and-paste practices adopted by developers to reuse certain design patterns, or to minimize risks [91]. Figure 3.2 shows a code clone example taken from the driver code of the Linux kernel version 2.6.16.13. The example shows three code snippets. The source of each snippet is shown at the top of the Figure. The cloned areas are coloured in grey with the identical code tokens in black and the variation points in red.

Clone detection approaches uncover clones using different measures of similarity. For example, some approaches compare the Abstract Syntax Tree (AST) for two code segments [65]. While other approaches compare a set of data flow and control metrics [169] or measure the text similarities of code segments and report code segments as clones if the

measurement of similarities exceeds a threshold [106].

We expect a clone detection approach to determine that lines 1, 2 and 4, in Table 3.1 are clones of each other with slight differences due to parameterization (i.e., *tom* v.s. *jerry* v.s. *john* and *100* v.s. *100* v.s. *103*). To verify our intuition, we experimented with using the Kamiya $et\ al.$ algorithm to abstract logs. We experimented with Kamiya $et\ al.$'s CCFinder [162] tool to abstract log lines. CCFinder, which uses a parameterized token matching algorithm, detect similarities in multiple programming languages and plain text. The tool is easy to use and is fully automated. The only input required is the name of the files to be analyzed, their type and number of similar tokens. The input files could be C, C++, COBOL, Java or plain text. The number of similar tokens is a user-specified threshold which is used to determine whether two code segments are similar. If the similarities between two code segments exceed this threshold, they will be reported as clones. The tool scales well to handle large software systems, like Apache, FreeBSD, NetBSD, and Linux [155, 162, 165, 180], with thousands or millions of lines of code.

After running CCFinder on a log file, we discovered that CCFinder is not suitable for our purposes. In particular, CCFinder has the following limitations:

1. **Threshold**: CCFinder has a configuration parameter which specifies the minimum number of tokens that two segments of code should have in common to qualify as a clone pair. However, each execution event contains a different number of words. Therefore, it is not possible to give a single threshold number for all execution events. A large threshold would only process execution event with a large number of tokens and ignore execution events with a small number of tokens. A large threshold will increase the precision of the approach and decrease its recall. Conversely, a small threshold will increase the recall but decrease the precision. A percentage threshold would be more suitable for our purposes.

2. **File size**: CCFinder could only handle small-sized log files. Large files resulted in

CCFinder crashing due to excessive use of memory. We require a tool that can process log files with thousands or millions of lines.

3. **Delimiters**: CCFinder performs clone detection across multiple lines. However, this is not ideal for abstracting log lines, since we want the output to stop at the line boundary.

Although CCFinder works well on large source code bases, it is not able to process large log files. We believe this is due to the following reasons:

1. Source code and plain text wrap around lines but have delimiters for each statement (like ";", "." or "!"); whereas a log line does not use similar delimiters. Therefore, CCFinder cannot find the end of each log line and treats all log lines as one large chunk.

2. Source code contains control keywords like if, else, for, while, etc. These keywords are the static parts in the source code and are used by CCFinder to mark the static parts of the source code. As log lines have a less strict grammar and a non-fixed vocabulary; CCFinder cannot mark up any specific parts as static when processing log lines.

3.4.2 The Steps of Our Approach

Based on our experience using CCFinder, we implemented our own tool to detect similarities among log lines, then to parameterize and abstract log lines. Our approach addresses the following problems:

1. **Threshold**: Our approach minimizes the impact of threshold for deciding whether two log lines are similar to each other.

2. **File size**: Our approach scales up to process log files, which contain thousands or millions of log lines.

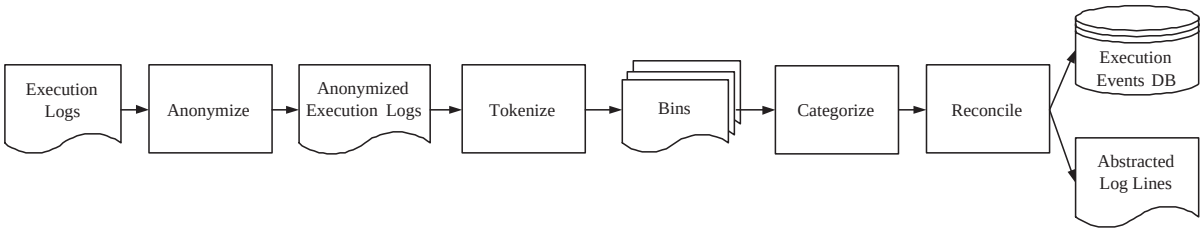


Figure 3.3: Our approach for abstracting execution logs to execution events

Table 3.3: Log lines used as a running example to explain our approach

1.	Start check out
2.	Paid for, item=bag, quality=1, amount=100
3.	Paid for, item=book, quality=3, amount=150
4.	Check out, total amount is 250
5.	Check out done

3. **Delimiters**: Our approach uses flexible heuristics to mark the dynamic and static information for each log line. We treat end of line characters as the delimiter for each log line.

As shown in Figure 3.3, our approach consists of four steps: Anonymize, Tokenize, Categorize and Reconcile. In the rest of this section, we demonstrate our approach using a small running example that is shown in Table 3.3. The example has 5 log lines.

The Anonymize step

The Anonymize step uses heuristics to recognize dynamic tokens in log lines. Once dynamic tokens are recognized they are replaced with a generic token ($v). Heuristics can be added or removed from this step based on domain knowledge. The following are two heuristics to recognize dynamic parts in log lines:

1. Assignment pairs like "$word = value$";
2. Phrases like "$is[are|was|were]\ value$"

Table 3.4: Running example logs after the Anonymize step

1.	Start check out
2.	Paid for, item=$v, quality=$v, amount=$v
3.	Paid for, item=$v, quality=$v, amount=$v
4.	Check out, total amount=$v
5.	Check out done

Table 3.4 shows the sample log lines after the Anonymize step. In the second and third lines, tokens after the equal signs (=) are replaced with the generic term $v. In the fourth line, the phrase "is 250" is replaced with the term "=$v". There are no changes made to the first line and last line.

The Tokenize step

The Tokenize step separates the anonymized log lines into different groups (i.e., bins) according to the number of words and estimated parameters in each log line. The use of multiple bins limits the search space of the following step (i.e., the categorize step). The use of bins permits us to process large log files in a timely fashion using a limited memory footprint since the analysis is done per bin instead of having to load up all the lines in the log file. We estimate the number of parameters in a log line by counting the number of generic terms (i.e., $v). Log lines with the same number of tokens and parameters are placed in the same bin.

Table 3.5 shows the sample log lines after the Anonymize and Tokenize steps. The left column indicates the name of a bin. Each bin is named with a tuple: number of words and number of parameters that are contained in the log line associated with that bin. The right column in Table 6 shows the log lines. Each row shows the bin and its corresponding log lines. The second and the third log lines contain 8 words and are likely to contain 3 parameters. Thus the second and third log lines are grouped together in the $(8, 3)$ bin. Similarly, the first and last log lines are grouped together in the $(3, 0)$ bin since they both contain 3 words and are likely to contain no parameters.

Table 3.5: Running example logs after the Tokenize step

Bin Names (# of words, # of parameters)	**Log Lines**
(3, 0)	1. Start check out 5. Check out done
(5, 1)	4. Check out, total amount = $v
(8, 3)	2. Paid for, item=$v, quality=$v, amount=$v 3. Paid for, item=$v, quality=$v, amount=$v

The Categorize step

The Categorize step compares log lines in each bin and abstracts them to the corresponding execution events. The inferred execution events are stored in an execution events database for future references. The algorithm used in the Categorize step is shown below. Our algorithm goes through the log lines bin by bin. After this step, each log line should be abstracted to an execution event. Table 3.6 shows the results of our working example after the Categorize step.

```
for each bin b_i
  for each log line l_k in bin b_i
    for each execution event e_(b_i,j) corresponding to b_i in the execution events DB
      perform word by word comparison between e_(b_i,j) and l_k
      if (there is no difference) then
        l_k is of type e_(b_i,j)
        break
      end if
    end for // advance to next e_(b_i,j)
    if ( l_k does not have a matching execution event) then
      l_k is a new execution event
      store an abstracted l_k into the execution events DB
    end if
```

Table 3.6: Running example logs after the Categorize step

Execution Events (word_parameter_id)	**Log Lines**
3_0_1	1. Start check out
3_0_2	5. Check out done
5_1_1	4. Check out, total amount = $v
8_3_1	2. Paid for, item=$v, quality=$v, amount=$v
8_3_1	3. Paid for, item=$v, quality=$v, amount=$v

```
  end for // advance to the next log line

end for // advance to the next bin
```

We now explain our algorithm using the running example. Our algorithm starts with the (3, 0) bin. Initially, there are no execution events which correspond to this bin yet. Therefore the execution event corresponding to the first log line becomes the first execution event namely 3_0_1. The _1 at the end of 3_0_1 indicates that this is the first execution event to correspond to the bin which has 3 words and no parameters (i.e., bin 3_0). Then the algorithm moves to the next log line in the (3, 0) bin which contains the fifth log line. The algorithm compares the fifth log line with all the existing execution events in the (3, 0) bin. Currently, there is only one execution event: 3_0_1. As the fifth log line is not similar to the 3_0_1 execution event, we create a new execution event 3_0_2 for the fifth log line. With all the log lines in the (3, 0) bin processed, we can move on to the (5, 1) bin. As there are no execution events which correspond to the (5, 1) bin initially, the fourth log line gets assigned to a new execution event 5_1_1. Finally, we move on to the (8, 3) bin. First, the second log line gets assigned with a new execution event 8_3_1 since there are no execution events corresponding to this bin yet. As the third log line is the same as the second log line (after the Anonymize step), the third log line is categorized as the same execution event as the second log line. Table 3.6 shows the sample log lines after the Categorize step. The left column is the abstracted execution event. The right column shows the line number together with the corresponding log lines.

Table 3.7: Sample logs which the Categorize step would fail to abstract

Event IDs	Execution Events
5_0_1	Start processing for user Jen
5_0_2	Start processing for user Tom
5_0_3	Start processing for user Henry
5_0_4	Start processing for user Jack
5_0_5	Start processing for user Peter

The Reconcile step

Since the Anonymize step uses heuristics to identify dynamic information in a log line, there is a chance that we might miss to anonymize some dynamic information. The missed dynamic information will result in the abstraction of several log lines to several execution events which are very similar. Table 3.7 shows an example of dynamic information that was missed by the Anonymize step. The Table shows 5 different execution events. However, the user names after "for user" are dynamic information and should have been replaced by the generic token "$v". All the log lines shown in Table 3.7 should have been abstracted to the same execution event after the Categorize step. The Reconcile step addresses this situation. All execution events are re-examined to identify which ones are to be merged. Execution events are merged if:

1. They belong to the same bin;
2. They differ from each other by one token at the same positions;
3. There exists a few of such execution events. We used a threshold of 5 events in our case studies. Other values are possibly based on the content of the analyzed log files. The threshold prevents the merging of similar yet different execution events, such as "Start processing" and "Stop processing", which should not be merged.

Looking at the execution events in Table 3.7, we note that they all belong to the "5_0" bin and differ from each other only in the last token. Since there are 5 of such events, we merged them into one event. Table 3.8 shows the execution events from Table 3.7 after the

Table 3.8: Sample logs after the Reconcile step

Event IDs	Execution Events
5_0_1	Start processing for user $v

Table 3.9: Size of the log files of the four studied applications

Application	App 1	App 2	LoadSim	Blue Gene/L
Number of Log Lines	723,608	1,688,876	67,651	2,994,986

Reconcile step. Note that if the "5_0" bin contains another execution event: "Stop processing for user John"; it will not be merged with the above execution events since it differs by two tokens instead of only the last token.

3.5 Case Study

We conducted a case study to evaluate the effectiveness of our approach in abstracting the logs for enterprise applications. We used log files from four different applications: App 1, App 2, LoadSim, and Blue Gene/L. App 1 is a large scale enterprise application developed by Research In Motion (RIM). App 2 is a medium scale enterprise application developed by Research In Motion (RIM). Loadsim is a medium scale enterprise application developed by Microsoft. These three applications are deployed and used by millions of users in thousands of enterprises worldwide. Blue Gene/L logs [28] are from an application running on the Blue Gene/L supercomputer [211]. Table 3.9 tabulates the applications and the size of studied log files. Table 3.10 shows sample log lines taken from LoadSim and Blue Gene/L.

Table 3.10: Sample Log Lines from LoadSim and Blue Gene/L

Sample LoadSim Logs	Sample Blue Gene/L Logs
Browse Mail: Read 1, Deleted 1	RAS KERNEL INFO program interrupt
Sent oups4k.msg (4 recipient(s))	RAS KERNEL INFO generating core.406
Browse Mail: Read 1, Moved 1	RAS KERNEL INFO program interrupt
There are 6 rules: Added 1 rule	RAS KERNEL FATAL data TLB error interrupt
Sent oups2k.msg (3 recipient(s))	RAS KERNEL FATAL data TLB error interrupt

3.5.1 Other Approach for Log Abstraction

In the case study, we want to compare the performance of our approach to other approaches for log abstraction. Since we have limited knowledge of the studied software applications, we could not use rule-based or codebook-based approaches. We could only use an AI-based approach. There exist two AI tools against which we could compare our approach. The tools are: teirify [227] and SLCT (Simple Logfile Clustering Tool) [233]. The teirify tool uses a bio-informatics algorithm [29] to detect line patterns, whereas the SLCT tool uses frequent-itemset mining techniques to cluster similar log lines. Unfortunately, teirify requires a large amount of memory and cannot handle log files, exceeding $10,000$ log lines. In the case study, we compared the performance of our approach against the result obtained from SLCT, which scaled to handle large files. We ran SLCT with the *-s <support threshold>* and *-j* options. The *-s <support threshold>* option specifies a support threshold value. Each line pattern exceeding this threshold will be outputted. For each outputted line pattern, SLCT also shows the support count associated with this pattern, i.e., the number of input lines that correspond to this pattern. Without the *-j* option, a log line will only be counted towards the support count of a single pattern. With the *-j* option specified, if a log line matches multiple line patterns then this line will be counted multiple times for the support count for each matched line patterns. Table 3.11 shows an example of the SLCT output. The left column shows 5 sample input log lines. The right column shows the results of several runs for SLCT using these input log lines. For each run, the table shows the used SLCT options and the corresponding output. If the support threshold is 4, the "In Checkout, user is *" pattern is outputted. All five input lines match the outputted pattern so the support for this pattern is 5. If the support threshold is 3, the "In Checkout, user is *" and "In Checkout, user is Tom" patterns are outputted. The "In Checkout, user is Tom" is a sub-pattern of "In Checkout, user is *", so all lines that are mapped to the "In Checkout, user is Tom" pattern are mapped to the "In Checkout, user is *" pattern as well. If the support threshold is 2, the

Table 3.11: An example of SLCT output

Log Lines	SLCT Runs	
	Options	SLCT Output
In Checkout, user is Tom	-s 4 -j	In Checkout, user is * (5)
In Checkout, user is Jerry	-s 3 -j	In Checkout, user is * (5)
In Checkout, user is Tom		In Checkout, user is Tom (3)
In Checkout, user is Tom	-s 2 -j	In Checkout, user is Tom (3)
In Checkout, user is Jerry		In Checkout, user is Jerry (2)

"In Checkout, user is Tom" and "In Checkout, user is Jerry" patterns are outputted.

For our case study, we wrote a *Perl* script which processes the SLCT output to ease our comparison process. Since a long line can correspond to several line patterns, our script matches each log line to the pattern with the largest support value. For example, using the output from "-s 3 -j", the log line "In Checkout, user is Tom" will be matched with the "In Checkout, user is *" pattern. We also explored mapping a line to the pattern with the smallest support value. The precision values using the smallest support mapping are similar to the large support mapping. However, the recall values are slightly lower when mapping to the smallest support value.

3.5.2 Measuring the Performance of an Approach

To measure the performance of a log abstraction approach, we need to know the correct mapping between every log line and its corresponding execution event. Acquiring such mapping is challenging, even if the source code of an application is available. Simply searching for "LOG" statements (e.g., "printf" statements in C/C++ or "System.out" statements in Java) is not sufficient since in many instances an execution event may be generated dynamically using several output statements in the source code. For example, the log line, shown in Figure 3.4, is generated by three output statements: one statement is outside the loop, and another two statements are inside the loop.

For our case study, we used two techniques to determine the gold standard (i.e., the accurate mapping of each log line to its corresponding execution event). For application

Log Lines	Source Code

User Shopping Basket contains: 2, 3, 5	LOG("User Shopping Basket contains: ");
	for (int i=0; i<shoppingBasket.size(); i++)
	{
	itemId = shoppingBasket[i];
	if(i > 0) {
	LOG(" , ” + itemId);
	} else {
	LOG(itemId);
	}
	}

Figure 3.4: An example of an execution event generated by multiple output statements

App 1, we used an internationalization file to determine the mappings. The application was internationalized and part of the internationalization efforts involved the manual mapping of each log line to an execution event. The execution events are stored in a separate file which is translated to different languages. This file acted as the gold standard in our performance evaluation for App 1. For the other three applications (App 2, LoadSim and Blue Gene/L), we performed random sampling to measure the performance of the log abstraction approaches. We randomly picked 100 log lines and measured the performance of SLCT and our approach on these log lines. Such sample size is sufficient to ensure a confidence level of 90% and a confidence interval of $\pm 8.2\%$ for the measured precision and recall results. We believe that a standard corpus, based on the logs of several enterprise applications, would be very valuable for studying and comparing the performance of log abstraction approaches. For this work, our random sampling and the internationalization file were the only possible options instead of a complete manual analysis of the large log files.

Table 3.12 tabulates the performance results for the two approaches. We note that since SLCT uses a frequent item set technique, it requires a minimum support count as a parameter. We experimented with different support count: 10, 50, 100, 150, 200, 250, 500, 750 and 1000. Due to page limitations, we do not discuss the details of these experiments.

Table 3.12: Performance of both approaches on the studied applications

Application	SLCT Precision (%)	SLCT Recall (%)	Our Precision (%)	Our Recall (%)
App 1	31.7	13.4	100	92.6
App 2	23.1 ± 8.2	7.5 ± 8.2	84.2 ± 8.2	82.4 ± 8.2
LoadSim	9.1 ± 8.2	9.1 ± 8.2	87.9 ± 8.2	85.3 ± 8.2
Blue Gene	33.3 ± 8.2	28.3 ± 8.2	100 ± 8.2	100 ± 8.2

However, our experiments show that the differences in performance due to different support counts is not statistically significant (within 2% for recall and within 1% for precision) using an ANOVA analysis with an alpha of 0.05. For reported results, we use a support count of 100.

The results reported in Table 3.12 show that our approach abstracts log lines to events with high precision and recall across the studied applications. On the other hand, the performance of SLCT is is not as high. The low precision and recall values for SLCT are due to the following two reasons.

1. Many log lines are not abstract to any execution events by SLCT since these lines do not occur often enough for a frequent pattern to emerge.

2. Although SLCT outputs a higher support count for the more general line pattern with the *-j* option, it does not attempt to abstract patterns further. For example as shown in Table 3.11 for support count 2, SLCT would output these two similar patterns "In Checkout, user is Tom" and "In Checkout, user is Jerry". SLCT does not attempt to merge and abstract identified patterns further since it may lead to all patterns being abstracted to a very general "*" pattern.

Our approach does not suffer from the problem of limited frequencies and subpattern merging since we map log lines to events even if the line occurs a single time and our Reconcile step identifies similar yet different by one token patterns (i.e., events) and merges them.

Adjusting Our Log Abstraction Heuristics

Our approach uses heuristics to recognize the dynamic information in a log line. These heuristics are based on coding conventions and observations made by examining log lines. However, these heuristics are not necessary complete nor applicable across applications. For different software systems, even different versions of a system, we might need to adjust our heuristics accordingly:

1. The anonymization rules depend on the application and might need to be adjusted for each application. For the Blue Gene/L logs, the parameter values are printed in "name:value" style rather than the "name=value" style. For both App 1 and App 2 logs, we needed to anonymize email addresses. For the LoadSim logs, we needed to anonymize different message file names ("*.msg"). These were simple changes which required minimal effort.

2. The Reconcile step, the final step in our approach, merges similar events which may have been missed by earlier anonymization rules. If there are multiple execution events which differ by one word in the same position and there are at least 5 of these events, then these execution events are merged. This heuristic performs relatively well on logs from the above four applications. However, this heuristic might need to be adjusted for other types of applications.

3.5.3 Studying the Characteristics of Log Files

To gain a better understanding of the performance of our approach and the other log abstraction approach, we examine the properties of the studied log files. We want to determine the effect of the complexity of a log file on the performance of a log abstraction approach. We assess the complexity of a log file by calculating the Shannon Entropy metric for that file [228]. The entropy metric assesses the amount of information contained in each log

file. The metric measures the uncertainty which is related to information. For example, suppose we monitored the output of an application which emitted 4 execution events, A, B, C, or D. As we wait for the next execution event, we are uncertain as to which event the application will produce (i.e., we are uncertain about the distribution of the output). Once we see an event outputted, our uncertainty decreases. We now have a better idea about the distribution of the output; this reduction of uncertainty has given us information. Shannon proposed to measure the amount of uncertainty using entropy. Shannon's entropy H_n is defined as:

$$H_n(P) = -\sum_{k=1}^{n} (p_k \times log_2 p_k)$$

where $p_k \geq 0$, $\forall k \in 1, 2, 3, \ldots, n$, ($n$ is the number of events), and $\sum_{k=1}^{n} p_k = 1$.

By defining the amount of uncertainty in a distribution, H_n describes the minimum number of bits required to uniquely distinguish the distribution. In other words, it defines the best possible compression for the distribution (i.e., the output of the application).

We measure the information contained in a log file by calculating Shannon's entropy on the distribution of execution events as reported by our approach. p_k is the probability that a log line will be assigned to the execution event k. To compare log files with a different number of events, we use the normalized shannon entropy which divides the entropy value by the log of the number of events (i.e., $\frac{H_n(P)}{log_2 n}$). If all log lines belong to different execution events ($p_k = \frac{1}{n}$), we achieve maximum entropy. On the other hand, if all log lines belong to the same execution event i (i.e., $p_k = 0$ for $k \neq i$), we achieve minimal entropy. The larger the entropy, the more complex the log file is. Table 3.13 tabulates the normalized entropy for the four log files (0 is minimal entropy and 1 is maximal entropy). The Table shows that the Blue Gene/L and LoadSim log files are the least complex files, while App 1 log file is the most complex. A comparison of the entropy results shown in Table 3.13 and the performance of both approaches, in Table 3.12, indicates that the performance of an

Table 3.13: Shannon entropy for the log files of the studied applications

Application	App 1	App 2	LoadSim	Blue Gene/L
Entropy	0.71	0.52	0.39	0.40

Table 3.14: Detailed analysis of the content of the log lines for the studied applications

System	**Log Lines**	**Execution Events**	**Top Event (%)**	**Top 10 Events (%)**
App 1	723,608	396	5.85	41.1
App 2	1,668,876	338	12.8	70.5
LoadSim	67,651	163	60.5	86.3
Blue Gene/L	2,994,986	2,918	35.9	74.3

approach is independent of the complexity of a log file.

In addition to measuring the entropy of each log file, we calculated the number of log lines in each log file, the number of identified execution events, the percentage of log lines which are abstracted to the most occurring execution event (Top Event), and the percentage of log lines which are abstracted to the top ten most occurring execution events (see Table 3.14). The metrics shown in the Table help explain the entropy values shown earlier in Table 3.13. In particular, the higher the top ten column in Table 3.14, the lower is the entropy due to the limited variability in the occurrence of execution events. For example, for the LoadSim log file, which has the lowest entropy, the top ten column indicates that 86% of all log lines are abstracted to just ten execution events. A closer analysis of the log file shows that LoadSim outputs a particular log line "$v: User is keeping a total of $v messages open (Max = $v)" each time the system status changes.

We also note that even though the Blue Gene/L log file contains more execution events, it is less complex than the App 1 logs. A manual analysis of the log files for both application indicated that the differences are due to two different logging styles: Blue Gene/L logs record the faults from different types of sources (e.g., disks, networks, printers, etc.). When a fault happens (e.g., disk full), the fault will persist for a period of time until either the fault is resolved or the process is aborted. Therefore, we expect to see hundreds of adjacent log lines (e.g., disk full) which report the same message within a time period. The App 1 logs record the execution of different scenarios of a complex application, therefore, we

expect a higher variability in the execution events.

3.6 Conclusion

Much of the research in dynamic analysis focuses on instrumenting an application and producing tracing logs. All too often instrumenting an application is not possible in a production setting due to performance overhead, lack of system knowledge, and the inaccessibility of the source code. On the other hand, execution logs are commonly available for most enterprise applications and could be used to study the dynamic behavior of complex applications.

It is difficult to analyze the log files produced by enterprise applications. Such logs are very large with millions of lines in them. The logs rarely follow standard logging formats. Techniques are needed to preprocess log files and identify the internal structure of each log line. Abstraction log lines abstracted to execution events helps to comprehend and summarize log files

In this chapter, we developed an approach that recognizes the internal structure of logs lines. The approach uses clone detection techniques to abstract each log line to its corresponding execution event. We conducted a case study using logs from four enterprise applications that are developed by three different organizations. Our case study shows that our approach performs well on large log files and has very high precision and recall.

As large scale systems are used by millions of users simultaneously, log lines from different scenarios and users are intermixed with each other in the execution logs. In the next three chapters, we will link related execution context from the logs and form different event representations (i.e., event pairs, event sequences and system states) to assess various aspects of the system quality under load (i.e., functional, performance and reliability).

CHAPTER 4

Automatic Detection of Functional Problems

Many software applications must provide services to hundreds or thousands of users concurrently. These applications must be load tested to ensure that they can function correctly under high load. Functional problems in load testing are due to problems in the load environment, the load generators, and the application under test. It is important to identify and address these problems to ensure that load testing results are correct and these problems are resolved. It is difficult to detect such problems in a load test due to the large amount of data which must be examined. Current industrial practice mainly involves time-consuming manual checks which, for example, grep the logs of the application for error messages.

In this chapter, we present an approach which mines the execution logs of an application to uncover the dominant behavior for the application and flags functional anomalies (i.e., deviations) from the dominant behavior. Using a case study of two open source and two large enterprise software applications, we show that our approach can automatically identify functional problems in a load test. Our approach flags $< 0.01\%$ of the log lines for closer analysis by domain experts. The flagged lines indicate load testing problems with a relatively small number of false alarms. Our approach scales well for large applications and is currently used daily in practice.

4.1 Introduction

MANY SYSTEMS ranging from e-commerce websites to telecommunications must support concurrent access by hundreds or thousands of users. To assure the quality of these systems, load testing is a required testing procedure in addition to conventional functional testing procedures, such as unit and integration testing.

Load testing, in general, refers to the practice of assessing the system behavior under *load* [70]. *Load* refers to the rate of the incoming requests to the system. A load test usually lasts for several hours or even a few days. Load testing requires one or more load generators which mimic clients sending thousands or millions of concurrent requests to the application under test. During the course of a load test, the application is monitored and performance data along with execution logs are stored. Performance data record resource usage information such as CPU utilization, memory, disk I/O and network traffic. Execution logs record the run time behavior of the application under test.

As observed by Visser [34], load testing is a difficult task requiring a great understanding of the application under test. Problems in the application under test (e.g., bugs), the load generator or the load environment are usually the sources of functional problems under load. However, as discussed in Chapter 1, looking for functional problems in a load testing is a time-consuming and difficult task, due to the challenges like no documented system behavior, monitoring overhead, time pressure and large volume of data. Most practitioners look for the functional problems under load using ad-hoc manual searches for specific keywords like “failure”, or “error”. Then load testing practitioners analyze the context of the matched log lines to determine whether they indicate functional problems or not. Depending on the length of a load test and the volume of generated data, it takes load testing practitioners several hours to perform these checks.

We believe this current practice is not efficient since it takes hours of manual analysis, nor is it sufficient since it may miss possible problems. On one hand, not all log lines containing terms like “error” or “failure” are worth investigating. A log such as “Failure to locate item in the cache” is likely not a bug. On the other hand, not all errors are indicated in the log file using the terms “error” or “failure”. For example, even though the log line “Internal queue is full” does not contain the words “error” or “failure”, it might also be worthwhile investigating it, since newly arriving items are possibly being dropped.

In this chapter, we introduce an approach for automatically uncovering the dominant

behavior of an application by mining logs generated during a load test. We use the recovered dominant behavior to flag any deviation, i.e., anomalies, from the dominant behavior. The main intuition behind our work is that a load test repeatedly executes a set of scenarios over a period of time. Therefore, the applications should follow the same behavior (e.g. generate the same logs) each time the scenario is executed. Therefore, the dominant behavior is probably the normal (i.e., correct) behavior and the minority (i.e. deviated) behaviors are probably troublesome and worth investigating. The main contributions of our work is as follows:

1. This work is the first work to propose a systematic approach to detect functional problems in a load test;

2. Unlike other existing log analysis work which require a user specified model (e.g., [44]), our approach is self-learning, requiring little domain knowledge about an application and little maintenance to update the models over releases. The model for the dominant behavior is created automatically;

3. Case studies show that our approach flags a very small percentage of log lines that are worth investigating. The approach produces very few false alarms (precision $> 50\%$) with many of the flagged lines indicating load testing problems;

4. Our proposed approach is easy to adopt and scales well to large scale enterprise applications. Our approach is currently used daily in practice for analyzing the results of load tests of large enterprise applications.

Organization of the Chapter

This chapter is organized as follows: Section 4.2 explains the types of problems that can occur during a load test. Section 4.3 describes our anomaly detection approach. Section 4.4 presents a case study of our anomaly detection approach. We have applied our approach

to uncover functional problems in load tests for two open source and two large enterprise applications. Section 4.5 discusses our current approach and present some of its limitations. Section 4.6 describes related work. Section 4.7 concludes this chapter.

4.2 Functional Problems in a Load Test

Load testing involves the setup of a complex load environment. The application under test should be setup and configured correctly. Similarly, the load generators must be configured correctly to ensure the validity of the load test. The results of a load test must be analyzed closely to discover any problems in the application under test (i.e., load related problems), in the load environment, or in the load generation. We detail the various types of functional problems that occur during load testing.

Bugs in the Application Under Test

The main purpose of a load test is to uncover *load sensitive* errors. Load sensitive errors are problems which only appear under load or extended execution. For example, memory leaks are not easy to spot under light-load with one or two clients, or during a short-run. However, memory leaks usually exhibit a clear trend during extended runs. Another example of load sensitive errors are deadlock or synchronization errors which show up due to the timing of concurrent requests.

Problems with the Load Environment

Problems with the load testing environment can lead to invalid test results. These problems should be identified and addressed to ensure that the load test is valid. Examples of load environment problems are:

Mis-configuration: The application under test or its run-time environment, e.g., databases

or web servers, may be mis-configured. For example, the number of concurrent connections allowed for a database may be incorrect. A small number of allowed connections may prevent the login of several users and would lead to a lower load being applied on the application under test.

Hardware Failures: The hardware running the application and the load test may fail. For example, the hard disks may fill up due to the tester forgetting to clean up the data from an older run. Once the disk is full, the application under test may turn-off specific features. This would lead to an incomplete load test since some of the functionalities of the application have not been fully load tested.

Software Interactions: A load test may exhibit problems due to intervention from other applications. For example, during a long running load test, an anti-virus software may start up and and intervene with the running load test. Or the operating system may apply updates and reboot itself.

Problems with the Load Generation

Load generators are used to generate hundreds or thousands of concurrent requests trying to access the application. Problems in the load generators can invalidate the results of a load test. Examples of possible load generation problems.

Incorrect Use of Load Generation Tools: Some of the generic load testing tools [32] require load testing practitioners to first record the scenarios, edit the recordings and replay them. This is an error-prone process. Edited recordings may not trigger the same execution paths as expected. For example, in a web application, the recorded URLs have a session ID which must be consistent for each request by the same user otherwise the application would simply return an error page instead of performing the expected operations.

Buggy Load Generators: The load generation tools are software applications which may themselves have *load sensitive* problems or bugs. For example, rather than sending requests to the application under test in a uniform rate, many load generation tools allow load testing practitioners to specify different distributions. However, the requests may not follow that distribution during a short run.

It is important to identify and remedy these functional problems. However, identifying these functional problems is a challenging and time-consuming task due to the large amount of generated data and the long running time of load tests. The motivation of our work is to help practitioners identify these functional problems.

4.3 Our Anomaly Detection Approach

The intuition behind our approach is that load testing involves the execution of the same operations a large number of times. Therefore, we would expect that the application under test would generate similar sequences of events a large number of times. These highly repeated sequences of events are the dominant behavior of the application. Variations from this behavior are anomalies which should be closely investigated since they are likely to reveal load testing problems.

We cannot instrument the application to derive the dominant behavior of the application, as instrumentation may affect the performance of the application and the software behavior won't be comparable with the deployed application. Fortunately, most large enterprise applications have some form of logging enabled for the following reasons:

1. to support remote issue resolution when problems occurs and
2. to cope with recent legal acts such as the "Sarbanes-Oxley Act of 2002" [23] which stipulate that the execution of telecommunication and financial applications must be logged.

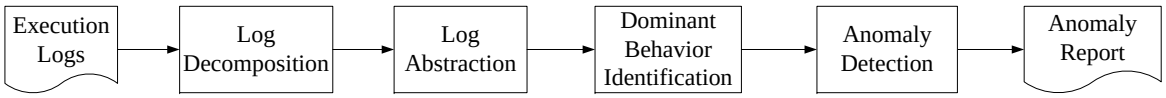


Figure 4.1: Our anomaly detection approach

Such logs record software activities (e.g. "User authentication successful") and errors (e.g. "Fail to retrieve customer profile"). We can mine the dominant behavior of the application from these commonly available logs. In this section we present an approach to detect anomalies in these logs. These anomalies are good indicators of problems in a load test.

As shown in Figure 4.1, our anomaly detection approach takes a log file as input and goes through four steps: Log Decomposition, Log Abstraction, Identification of the Dominant Behavior, and Anomaly Detection. Our approach produces an HTML anomaly report. We explain each step in detail in the following subsections.

4.3.1 Log Decomposition

Most modern enterprise applications are multi-threaded applications which process thousands of transactions concurrently. The processing of all these transactions is logged to a log file. Related log lines do not show up continuously in the log file, instead they may be far apart. The log decomposition step processes the log file and groups related log lines together. Log lines could be related because they are processed by the same thread or because they are related to the same transaction. Most of the enterprise applications have a standard format for logging the transaction information (e.g. header part of a log line), as this information is important for remote issue resolution. For example, in a web application, each log line contains a session or customer ID. Or in a multi-threaded application, each log line contains a thread ID. Or in a database application, each log line might contain the transaction ID. Sometimes, a log line might contain multiple types of IDs. For example, in an e-commerce application, a log line can contain both the session and customer IDs. Depending on the granularity of the analysis, one or multiple of these IDs are used to group

Table 4.1: Example log lines

#	Log lines	Group
1.	accountId(*Tom*) User purchase, item=*100*	Tom
2.	accountId(*Jerry*) User purchase, item=*100*	Jerry
3.	accountId(*Tom*) Update shopping cart, item=*100*	Tom
4.	accountId(*John*) User purchase, item=*103*	John
5.	accountId(*Tom*) User checkout	Tom
6.	accountId(*Jerry*) Update shopping cart, item=*100*	Jerry
7.	accountId(*John*) User purchase, item=*105*	John

Table 4.2: Example execution events

Event ID	Event Template
E_1	User purchase, item=$v
E_2	Update shopping cart, item=$v
E_3	User checkout

related lines together.

Table 4.1 shows a log file with 7 log lines. If the log file is decomposed using the accountId field, the log decomposition step would produce 3 groups (Tom, Jerry and John). This step requires domain knowledge by the load testing practitioner to decide which field to use to decompose the logs.

4.3.2 Log Abstraction

Each log line is a mixture of dynamic and static information. Log lines containing the same static information belong to the same execution event. We want a technique that would recognize that two log lines are due to the same event. We call this process the log abstraction problem. In Chapter 3, we have proposed a technique which can uniquely map each log line to an execution event. The technique parameterizes log lines using a similar process as token-based code cloning techniques. The log lines in Table 4.1 would be abstracted to only 3 execution events as shown in Table 4.2. The "$v" sign indicates a runtime generated parameter value.

Based on the log decomposition and abstraction steps, the sample log file in Table 4.1 would result in the grouping of events shown in Table 4.3.

Table 4.3: The sample log file after the log decomposition and the log abstraction steps

Group	Event ID	Log line #
Tom	E_1	1
	E_2	3
	E_3	5
Jerry	E_1	2
	E_2	6
John	E_1	4
	E_1	7

4.3.3 Identification of the Dominant Behavior

In this step, we identify the dominant behavior in the logs. We achieve this by analyzing the *execute-after* relations for each event E. The **execute-after** relation for an event E, denote by (E, *), refers to the occurrences of all the event pairs with the leading event E. Two events E_1 and E_2 form an *event pair*, if

1. E_1 and E_2 belong to the same group; and
2. E_2 is the next event that directly follows E_1.

In the event pair (E_1, E_2), E_1 is referred to as the leading event. The execute-after pair for event E_1 is formed by aggregating all the event pairs which have E_1 as the leading event. Table 4.4 shows all the execute-after pairs in Table 4.3. There are two execute-after pairs: one for E_1 and one for E_2. For each execute-after pair, the table shows the event pairs, the number of occurrences for each event pair, and a sample log lines corresponding to the first occurrence of each event pair. There are two types of events which are executed after the *User purchase* event (E_1). E_1 could be followed with another E_1. This is generated by John's session from log lines 4 and 7. Or E_1 could be followed with *Update shopping cart* (E_2). There are two occurrences of (E_1,E_2) which are attributed to Tom's and Jerry's sessions. The first and third log lines correspond to the first occurrence of the event pair (E_1,E_2). Event pairs are grouped by the *execute-after* relations. For example, (User purchase, *) includes all the event pairs which start with the *User purchase* event. Thus, the event pairs (User purchase, Update shopping cart) and (User purchase, User purchase) are grouped under the *execute-after* relations for the *User purchase* event, (User purchase, *).

Table 4.4: Log file after the identification of the dominant behavior step

$(\boldsymbol{E}, *)$	Event Pair	Occurrences	Sample Line #
$(E_1, *)$	$(\boldsymbol{E_1}, \boldsymbol{E_2})$	2	1, 3
	(E_1, E_1)	1	4, 7
$(E_2, *)$	$(\boldsymbol{E_2}, \boldsymbol{E_3})$	1	3, 6

Table 4.5: Summary of execute-after pairs

$(\boldsymbol{E}, *)$	Event Pair	Frequency
(User purchase, *)	**(User purchase, Update cart)**	1,000
	(User purchase, User purchase)	1
(User signin, *)	**(User signin, Browse catalog)**	100
	(User signin, Update account)	20
	(User signin, Browse history)	10
(Browse catalog, *)	**(Browse catalog, User purchase)**	500
	(Browse catalog, Update account)	500
	(Browse catalog, Search item)	100

The dominant behavior for $(E, *)$ refers to the largest event pair(s) which starts with E. The dominant behavior pairs for each *execute-after* relation are shown in bold in Table 4.4. The dominant behavior for $(E_1, *)$ is (E_1, E_2). The dominant behavior for $(E_2, *)$ is (E_2, E_3). Sample line numbers show the first occurrences of the event-pair in the log file. The sample log line numbers are displayed later in the anomaly report.

4.3.4 Anomaly Detection

The previous step identifies the dominant behavior in the logs. In this step, we mark any deviations, i.e., anomalies, from the domination behavior for closer investigation. As load testing practitioners have limited time, we need a way to rank anomalies to help load testing practitioners prioritize their investigation. We use a statistical metric called *z-stats*. Recent work by Kremenek and Engler [170] shows that the z-stats metric performs well in ranking deviation from dominant behaviors when performing static analysis of source code. The z-stats metric measures the amount of deviation of an anomaly from the dominant behavior. The higher the z-stats is, the stronger the probability that the majority behavior is the expected behavior. Therefore the higher the z-stats value, the higher the chance that a deviation, i.e. low frequency pairs, are anomalies that are worth investigating. The

#	Z-Stat	Kinds	Min	Max	Total	Event
E_1	10.44	2	1	1,000	1,001	accountId(Tom) User purchase, item=100
E_3	-4.97	3	10	100	130	accountId(Tim) User signin, user=Tim
E_4	-49.25	3	100	500	1,100	accountId(John) Browse catalog, catalog=book

Figure 4.2: An example anomaly report

formula to calculate z-stats is as follows: $z(n,m) = \frac{(\frac{m}{n} - p_0)}{\sqrt{\frac{p_0 \times (1-p_0)}{n}}}$, where n is total number of occurrences of event E, m is the occurrences of the dominant event pairs which starts with E, and p_0 is the probability of the errors. p_0 is normally assigned a value of 0.9 [112] for error ranking.

We illustrate the use of z-stats using the example shown in Table 4.5 with dominant behavior marked in bold. The dominant behavior for (User purchase, *) is (User purchase, Update cart). Thus the z-stats for (User purchase, *) is calculated as follows ($m = 1000$, $n = 1001$): $z(1001, 1000) = \frac{(\frac{1000}{1001} - 0.9)}{\sqrt{\frac{0.9 \times (1-0.9)}{1001}}} = 10.44$, and the z-stats for (User signin, *) is z($m = 100$, $n = 130$) = -4.97. The dominant behavior for (Browse catalog, *) is (Browse catalog, Purchase item) or (Browse catalog, Update account). Thus the z-stats for (Browse catalog, *) is z($m = 500$, $n = 1100$) = -49.25. (User purchase, *) has a higher z-stats score than (User signin, *) and (Browse catalog, *). This indicates that low frequency event pairs in the group of (User purchase, *) are likely anomalies that should be investigated closely. Normally, each purchase is followed by an update. The missing *Update cart* event suggests that the system might miss information about items selected by a customer.

4.3.5 Anomaly Report

To help a load testing practitioner examine the anomalies, we generate an anomaly report. The report is generated in dynamic-HTML so testers can easily attach it to emails that are sent out while investigating a particular anomaly.

#	Z-Stat	Kinds	Min	Max	Total	Event
E_1	10.44	2	1	1,000	1,001	accountId(Tom) User purchase, item=100
E_3	-4.97	3	10	100	130	accountId(Tim) User signin, user=Tim
E_4	-49.25	3	100	500	1,100	accountId(John) Browse catalog, catalog=book

Freq	Sample	Details (Sort by Freq)
1,000 (99%)	log.txt, line 20 log.txt, line 23	E_1 --> accountId(Tom) User purchase, item=100 E_2 --> accountId(Tom) Update shopping cart, item=100
1 (<1%)	log.txt, line 104 log.txt, line 108	E_1 --> accountId(John) User purchase, item=103 E_1 --> accountId(John) User purchase, item=105

Figure 4.3: An expanded anomaly report

Figure 4.2 shows the generated report for our running example. Our anomaly report is a table with each row corresponding to one *execute-after* relation. Rows are sorted by decreasing z-stats score. *Execute-after* relations with high z-stats value are more likely to contain anomalies that are worth investigating. The first row in Figure 4.2 corresponds to the execute-after pair for the *User purchase* event (E_1). There are in total two types of event pairs with *User purchase* as the leading event. One event pair occurs $1,000$ times and the other event pair occurs just once. In total, all the event pairs, with *User purchase* as the leading event, appear $1,001$ times during the course of this load test. A sample line for this event (E_1) is also shown.

Each sample line is a clickable hyperlink. Once a user clicks the hyperlink, the report shows detailed information about the execute-after pairs for that event. Figure 4.3 shows the screenshot of the anomaly report after clicking the sample line for Event E_1. Event pairs for (User purchase, *) are sorted with decreasing frequency. The topmost event pair (User purchase, Update cart(E_2)) is the dominant behavior. (User purchase, Update cart) occurs 99% $(1,000)$ of the time. The first occurrence of this event pair is in log file *log.txt* lines 20 and 23. The other event pair (User purchase, User purchase) is a deviated behavior. It occurs only once ($< 1\%$). It is recorded in log file *log.txt* lines at 104 and 108.

4.4 Case Studies

We have conducted three case studies on four different applications. The application are: the Dell DVD Store (DS2), the JPetStore application (JPetStore), and two large enterprise software applications. Table 4.6 gives an overview of these four case studies. Based on our experience, z-stats lower than 10 are likely noise. Thus, we only output event pairs with z-stats score larger than 10 in these four experiments. The table summarizes the types of the applications, the duration of the load test, size of logs, and our anomaly detection results. For example, Dell DVD Store is an open source web applications implemented using JSP. The load test was 5 hours long and generated a log file with $147,005$ log lines. Our anomaly detection approach takes less than 5 minutes to process the log file. We have discovered 23 abstract event types. There are 4 anomalies detected and 18 log lines are flagged, that is less than 0.01% of the whole log file. Among these four anomalies, two of them are actual problems in the load test. Our precision is 50%. Among these two problems: one is a bug in the application under test, the other is a bug in the load generator. We did not detect any problems with the load environment. The percentage of flagged lines is the total number of log lines shown in the anomaly report. As shown in Figure 4.4, there are total 9 event pairs $(3 + 2 + 2 + 2)$. Thus our approach has flagged $9 \times 2 = 18$ lines. The processing time for our approach is measured using a laptop with $2G$ memory, $7,200$ RPM hard-drive and a Dual Core 2.0 GHz processor.

The rest of this section covers the details of our case studies. For each application, we present the setup of the load test then we discuss the results of applying our approach to identify problems. The goal of the studies is to measure the number of false positive (i.e. precision) reported by our approach. A false positive is a flagged anomaly that did not point to a load testing problem. We cannot measure the recall of our approach since we do not know the actual number of problems.

Table 4.6: Overview of our case studies

Applications	DS2	JPetStore	App 1	App 2
Application Domain	Web	Web	Telecom	Telecom
License	Open Source	Open Source	Enterprise	Enterprise
Source Code	JSP, C#	J2EE	C++, Java	C++, Java
Load Test Durations	5 hr	5 hr	8 hr	8 hr
Number of Log lines	$147,005$	$118,640$	$2,100,762$	$3,811,771$
% Flagged	$\frac{18}{147005}(< 0.01)\%$	$\frac{8}{118640} < 0.01\%$	$< 0.01\%$	$< 0.01\%$
Number of Events	23	22	> 400	> 400
Application Size	$2.3M$	$4.9M$	$> 300M$	$> 400M$
Precision	$\frac{2}{4}(or\ 50\%)$	$\frac{2}{2}(or\ 100\%)$	56%	100%
Processing Time	< 5 min	< 5 min	< 15 min	< 15 min
Break Down of Problems (Application/Environment/Load)	1/0/1	2/0/0	Y/Y/N	Y/N/N

4.4.1 DELL DVD Store

The DVD Store (DS2) application is an online web application [6]. DS2 provides basic e-commerce functionality, including: user registration, user login, product search, and item purchase. DS2 is an open source application and is used to benchmark Dell hardware, and for database performance comparisons [20]. DS2 comes in different distribution package to support various web platforms (e.g. Apache Tomcat, or ASP .NET) and database vendors (MySQL, Microsoft SQL Server, and Oracle).

Experiment Setup

DS2 contains a database, a load generator and a web application. For a load test, the database is populated with entries using provided scripts. The web application consists of four JSP pages which interact with the database and display dynamic content. The DS2 load generator supports a range of configuration parameters to specify the workload. Table 4.7 shows the parameters used in our experiment. Note that "Think Time" refers to the time the user takes between different requests. We use a small database, which by default contains $20,000$ users. Parameter values marked with a "*" indicate that we use the default value. In this experiment, we use MySQL as the backend database and the Apache Tomcat as our web server engine. We increase the number of allowed concurrent connections in MySQL to enable a large number of concurrent access. For this configuration, The web application

Table 4.7: Workload configuration for DS2

Parameter	Value
Duration	5 hours
Number of driver threads	50
Startup request rate	5
Think time	50 sec
Database size	Small
Percentage of new customers	20%
Average number of searches per order	3*
Average number of items returned in each search	5*
Average number of items per order	5*

layer is implemented in JSP and the load generator is implemented in C#.

Each action from the user (login, registration, browse, purchase) results in a separate database connection and transaction. Since DS2 has no logs, we manually instrument its four JSP pages so that logs are output for each database transaction. Each log line also contains the session ID and customer ID.

Analysis of the Results of the Load Test

The load test generated a log file with $147,005$ log lines for 23 execution events. Our approach takes about 2 minutes to process the logs.

Figure 4.4 shows the screenshot of the anomaly report for the DS2 application. The report shows 4 anomalies. We cross examine the logs with the source code to determine whether the flagged anomalies are actual load testing problems. The precision of this report is 50%. Two out of four anomalies are actual problems in the load tests. We briefly explain these two problems.

Figure 4.4 shows the details of the first anomaly. The first execute-after pair is about a customer trying to add item(s) into their shopping cart (E_{13}). About 99% ($87,528$) of the time, the customer's shopping cart is empty. Therefore, a shopping cart is created in the database along with the purchased item information (E_{14}). Less than 1% ($1,436$) of the time, the customer adds more item(s) into their existing shopping cart (E_{13}). For the other 358 ($< 1\%$) cases, the customer directly exits this process without updating the order

information (E_{15}).

The first event pair is the dominant behavior. The second event pair refers to the cases that customers purchases multiple items in one session. However, the last event pair (E_{13}, E_{15}) looks suspicious. A closer analysis of the DS2 source code reveals that this is a bug in the web application code. DS2 pretty prints any number if it is larger than 999. For example, 1000 would be outputted as $1,000$. However, the pretty printed numbers are concatenated into the SQL statement which are used for updating (or inserting) the customer's information. The additional comma results in incorrect SQL code since a comma in the SQL statements means different columns. For example, a SQL statement like: "INSERT into DS2.ORDERS (ORDERDATE, CUSTOMERID, NETAMOUNT, TAX, TOTALAMOUNT) ('2004-01-27', 24, 888, 313.24, 1,200)" will cause an SQL error, since SQL treats a value of $1,200$ for TOTALAMOUNT as two values: 1 and 200.

The second and third anomalies are not problems. They are both due to the nature of the applied load. For example, the second anomaly is because we only have a few new customers in our experiment (20% new customers in Table 4.7). The expected behavior after each customer login is to show their previous purchases. There are a few occurrences where DS2 does not show any previous purchase history. These occurrences are due to newly registered customers who do not have any purchase history.

The fourth anomaly is due to a problem with the load generator. The load generator randomly generates a unique ID for a customer. However, the driver does not check whether this random number is unique across all concurrent executing sessions. The shown anomaly is due to one occurrence, in a 5 hour experiment, where two customers were given the same customer ID.

#	Z-Stat	Kinds	Min	Max	Total	Event
E_{13}	79.61	3	358	87,528	89,322	SessionID=19420, Entering purchase for simple quantity queries
E_6	39.96	2	1	14,393	14,394	SessionID=11771, Login finish for existing user
E_{19}	34.73	2	317	16,273	16,590	SessionID=14128, End of purchase process
E_{22}	20.65	2	1	3,857	3,858	SessionID=12067, Purchase complete

Freq	Sample	Details (Sort by Freq)
87,528 (98%)	ds2logs.txt 688 ds2logs.txt 689	E_{13} --> SessionID=19420, Entering purchase for simple quantity queries E_{14} --> SessionID=19420, Initial purchase, update cart
1,436 (<1%)	ds2logs.txt 2,484 ds2logs.txt 2,488	E_{13} --> SessionID=16242, Entering purchase for simple quantity queries E_{13} --> SessionID=16242, Entering purchase for simple quantity queries
358 (<1%)	ds2logs.txt 10,020 ds2logs.txt 10,021	E_{13} --> SessionID=13496, Entering purchase for simple quantity queries E_{15} --> SessionID=13496, Finish purchase before commit

Figure 4.4: DS2 Anomaly Report

4.4.2 PetStore

We used our approach to verify the results of a load test of another open source web application software called JPetStore [11]. Unlike Sun's original version of Pet Store [16] which is more focused on demonstrating the capability of the J2EE platform, JPetStore is a re-implementation with a more efficient design [13] and is targeted for benchmarking the J2EE platform against other web platforms such as .Net.

JPetStore is a larger and more complex application relative to DS2. Unlike DS2 which embeds all the application logic into the JSP code, JPetStore uses the "Model-View-Controller" framework [15] and XML files for object/relational mappings.

Experiment Setup

We deployed JPetStore application on Apache Tomcat and use MySQL as the database backend. As JPetStore does not come with a load generator, we use Webload [32], an open source web load testing tool, to load test the application. Using webload we recorded four different customer scenarios for replay during load testing. In Scenario 1, a customer only browses the catalog without purchasing. In Scenario 2, a new customer first registers for an account, then purchase one item. In Scenario 3, a new customer first purchases an item,

Table 4.8: Workload configuration for JPetStore

Parameter	Value
Duration	300 minutes (5 hours)
Request rate	5 - 150 (random distribution)
Think time	50 sec
Scenario 1/2/3/4	25% / 25% / 25% / 25%

then register for an account then checkout. In Scenario 4, an existing customer purchases multiple items.

In this load test, we ran two WebLoad instances from two different machines sending requests to the JPetStore web application. For each WebLoad instances, we added in $5,000$ users. Table 4.8 shows the workload configuration parameters for JPetStore load test. Note that WebLoad can specify the distribution of the generated request rate. In this experiment, we specify a random distribution for the user's requests with minimum rate 5 requests/sec and maximum rate 150 requests/sec.

Analysis of the Results of the Load Test

The load test generated a log file with 118,640 log lines. It takes our approach around 2 minutes to process the logs. Two anomalies are reported and they are both application problems.

The first problem is a bug in the registration of new users. We have two load generators running concurrently. Each load generator has an input file with randomly generated customer IDs. These customer IDs are used to generate web requests for scenarios (2 and 3). There are a some user IDs which are common to both WebLoad instances. If a user tries to register an ID which already exists in the database, PetStore does not gracefully report a failure. Rather, PetStore will output a stack of JSP and SQL errors.

The second problem reveals that JPetStore does not process page requests when it is under a heavy load. There is one instance out of $22,330$ instances where the header JSP page is not displayed. The error logs for the WebLoad tool indicate that the PetStore application timed out and could not process the request for the header JSP page on time.

4.4.3 Large Enterprise Applications

We applied our approach on two large enterprise applications, which can handle thousands of user requests concurrently. Two applications are both tested for 8 hours. It takes our approach about 15 minutes to process the log files. Table 4.6 shows the precision of our approach (56% - 100%). We have found bugs in development versions of the applications (App 1 and App 2). One of the bugs in the applications shows the SQL statement was corrupted due to a memory corruption. Further investigation leads to a memory corruption problems in the systems. In addition, our approach detected problems with the load environment due to the complexity of the load environment for the enterprise applications. The false positives in App 1 are mainly due to some rare events at the start of the application. When using our approach in practice, load testing practitioners commented that:

1. Our approach considerably speeds up the analysis work for a load test from several hours down to a few minutes.

2. Our approach helps uncover load testing problems by flagging lines that do not simply contain keywords like “error” or “failure”.

3. Our approach helps load testing practitioners communicate more effectively with developers when a problem is discovered. The generated HTML report can be emailed to developers for feedback instead of emailing a large log file. Moreover the report gives detailed examples of the dominant and the deviated behaviors. These simple examples are essential in easing the communication between the testers and the development team.

4.5 Discussions and Limitations

Our approach assumes that load testing is performed after the functionality of the application is well tested. Thus, the dominant behavior is the expected behavior and the minority

deviated behavior is the anomalies. However, this might not be a valid assumption. For example, if the disk of an application fills up one hour into a ten hour load test, then the majority of the logs will be the error behavior. That said, our approach would still flag this problem and the expert analyzing the logs would recognize that dominant behavior is the problematic case.

Our approach processes the logs for the whole load test at once. This whole processing might cause our approach to miss problems. For instance, if the disk for the database fills up halfway during a load test, the application under test will report errors for all the incoming requests which arrives afterwards. Normal and erroneous behavior may have equal frequencies. Our statistical analysis would not flag such a problem. However, if we segment the log files into various chunks and process each individual chunk separately, we can detect these types of anomalies by comparing frequencies across chunks.

Finally, our anomaly report contains false positives. Anomalies can be flagged due to the workload setup. For example, our report for the DS2 case study contains two false positives which are due to the workload. Also in a threaded application when a thread is done processing a particular request and starts processing a new request, the pair of events: event at end of a request and event at start of a request may be incorrectly flagged as an anomaly. We plan on exploring techniques to reduce with these false positives. For now, load testing practitioners are able to specify a false positive pair in a separate exclusion file. These pairs are used to clean up the results of future log file analysis.

4.6 Related Work

Much of the work in literature focuses on identifying bugs in software applications. Our work is the first, to our knowledge, that tackles the issue of identifying problems in load tests. These problems may be due to problems in the application, load generation or load environment.

Table 4.9: Summary of Related Work

Category	References	Key Idea	Technique	Challenges
Static	[112]	Inferring source code deviations	Statistics (z-stats)	Requires templates
	[176]	Inferring call sequences inconsistency	Frequent Item-Set mining	Too many reported violations, precision unknown
Dynamic	[92]	Comparing pass and failure runs	Finite State Machine	Scalability
	[137]	Inferring invariant violations	Invariants Confidence	Scalability
	[179]	Inferring control flow abnormality	Hypothesis Testing	Scalability
	[241]	Inferring error handling policy	Statistics	Coverage
	[248]	Inferring programme properties	Heuristics	Requires heuristics for interesting properties
Hybrid	[93]	Inferring invariant violations	Testing	Scalability

The work in the literature closest to our approach is all the work related to inferring dominant properties in a software application and flagging deviations from these properties as possible bugs. Such work can be divided into three types of approaches: 1) **Static approaches** which infer program properties from the source code and report code segments which violate the inferred properties; 2) **Dynamic approaches** infer program properties from program execution traces; 3) **Hybrid approaches** combine both approaches. Table 4.9 summarizes the related work. For each work, the table shows the main idea of the approach, the used techniques, and the challenge of directly adopting this approach to load testing.

Dawson et al. [112] gather statistics about the frequency of occurrence for coding patterns such as: pointer deference and lock/unlock patterns. They then use z-stats to detect and rank the errors. Li et. al. [176] use frequent item-set mining techniques to mine the call graph for anomalies. However, this approach produces many violations and the authors only evaluate a few violations thus the overall precision is unknown.

Hangal et al. [137] use dynamically inferred invariants to detect programming errors. Csallner et. al. [93] further improve the precision of bug detection techniques by executing the program using automatically generated test cases that are derived from the inferred

invariants. Liu et. al. [179] use hypothesis testing on code branches to detect programming errors. The above three techniques cannot be applied to load testing since the detailed instrumentations would have an impractical performance overhead.

Weimer et al. [241] detect bugs by mining the error handling behavior using statistical ranking techniques. Their approach only works for Java applications which have try-catch blocks and requires good knowledge of the source code which is not applicable for load testing practitioners..

Yang et al. [248] instrument the source code and mine the sequences of call graphs (pairs) to infer various programme properties. They look at function calls which are directly adjacent to each other as well as gapped function pairs. Due to the large size of inferred explicit properties, they use heuristics to select interesting patterns (e.g. lock/unlock). Their approach requires a great deal of manual work and the instrumentations has a high performance overhead.

Cotroneo et al. [92] produce a finite state machine based on profiled data. Then a failed workload is compared against the finite state machine to infer the failure causes. Profiling during a load test is infeasible due to inability to collect performance data. In addition, inferring a deterministic finite machine is not possible in a complex workload due to the large number of events that are generated using random load generators.

4.7 Conclusion

In this chapter, we present an approach to automatically identify functional problems under load. Our approach mines the logs of an application to infer the dominant behavior of the application. The inferred dominant behavior is used to flag anomalies. These anomalies are good indicators of load testing problems. Our case study on four applications shows that our approach performs with high precision and scales well to large systems. In the next chapter, we will explore an automated approach to detect performance problems under load.

CHAPTER 5

Automatic Detection of Performance Problems

Once the system is verified to function correctly under load, the next step is to determine if there are any non-functional problems (e.g., performance problems). Performance problems refer to the situations where a system suffers from unexpectedly high response time or low throughput. It is difficult to detect performance problems in a load test due to the absence of formally-defined performance objectives and the large amount of data that must be examined.

In this chapter, we present an approach which automatically analyzes the execution logs of a load test for performance problems. We first derive the system's performance baseline from previous runs. Then we perform an in-depth performance comparison against the derived performance baseline. Case studies show that our approach produces few false alarms (with a precision of 77%) and scales well to large industrial systems.

5.1 Introduction

LOAD TESTING, in general, refers to the practice of assessing the quality of a system under *load* [70]. A load is typically based on an operational profile, which describes the expected workload of the system once it is operational in the field [48, 55]. A load consists of the types of executed scenarios and the rate of these scenarios. For example, the load of an e-commerce website would contain information such as: browsing (40%) with a min/average/max rate of 5/10/20 requests/sec, and purchasing (40%) with a min/average/max rate of 2/3/5 requests/sec. A load test usually lasts for several hours

or even a few days.

The goal of a load test is to uncover functional and non-functional problems under load. Functional problems are often bugs, which do not surface during the functional testing process. Deadlock is an example of functional problems under load. Typical non-functional problems are performance problems like high response time or low throughput under load. As discussed in Chapter 1, current practice of load testing analysis involves with manual high level checks. We believe this current practice is not efficient since it takes hours of manual analysis, nor is it sufficient since it might miss problems. Our previous work (Chapter 4) flags possible functional problems by mining the execution logs of a load test to uncover dominant execution patterns and to automatically flag functional deviations from this pattern within a test. In this chapter, we introduce an approach that automatically flags possible performance problems in a load test.

We cannot derive the dominant performance behavior from just one load test as we did in Chapter 4, since the load is not constant. A typical workload usually consists of periods simulating peak usage and periods simulating off-hours usage. The same workload is usually applied across load tests, so that the results of prior load tests are used as an informal baseline and compared against the current run. If the current run has scenarios which follow a different response time distribution than the baseline, this run is probably troublesome and worth investigating. The main contributions of this chapter are as follows:

1. To the best of our knowledge, our approach is the first work to automatically detect performance problems in the load testing results.

2. Our approach makes use of readily available execution logs, which avoids the need of performance impacting instrumentation [61, 88] .

3. Our approach automatically reports scenarios with performance problems and pinpoints the performance bottlenecks within these scenarios. Case studies show that our

approach scales well to large systems and produces few false alarms (with a precision of 77%).

Organization of the Chapter

The chapter is organized as follows: Section 5.2 shows an example of how a load testing practitioner can use our performance analysis report to analyze the performance of a load test. Section 5.3 describes our performance analysis approach. Section 5.4 presents three case studies on our performance analysis approach: two on open source applications and one on a large enterprise application. Section 5.5 presents some discussions. Section 5.6 describes related work. Section 5.7 concludes the chapter.

5.2 Performance Analysis Report

The previous section discussed the challenges and limitations of the current performance analysis practices. In response, we have developed an automatically generated report, which can uncover potential performance problems. The report extracts information from the readily available execution logs and uses a previous run as an informal performance baseline to compare against.

Consider the following example. Jack, a load testing practitioner, is asked to load test a new version of an online application. He needs to determine whether the application can scale to hundreds of concurrent client requests just like the earlier version did. The application is an online pet store which supports operations like login/logout, browse, purchase and registration for new users. We now demonstrate how Jack can use our performance analysis report to uncover performance problems based on the run he did on an early release.

Throughput	Deviation	Event Sequences
5,000 5,002	0.8	E1→E2→ E3→E4
4,500 4,498	0.5	E1→E3→ E5→E7
4,000 4,010	0.3	E1→E3→ E5→E7→ E2 →E4

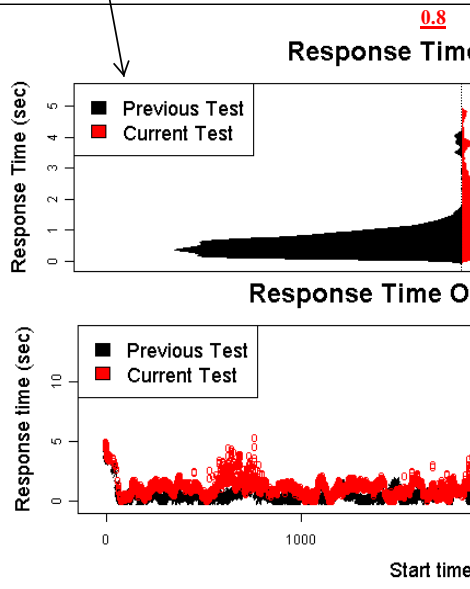
(a) Overall Performance Summary

Throughput	Deviation	Event Sequences
5,000 5,002	0.8	E1→E2→E3→E4
4,500 4,498	0.5	E1→E3→E5→E7
4,000 4,010	0.3	E1→E3→E5→E7→E2 →E4

Average	Deviation	Event Sequences
		E1: sessionId=2, time=1, user=Tom, login successfully
1.0 1.1	0.1	E2: sessionId=2, time=1, user=Tom, browse catalog, catalog=bird
0.5 1.8	0.9	E3: sessionId=2, time=2, user=Tom, add item, item=parrot, quantity=1
0.5 0.5	0.1	E4: sessionId=2, time=3, user=Tom, user checkout successfully

(c) Stepwise Performance Diagnosis

Throughput	Deviation	Event Sequences
5,000 5,002	0.8	E1→E2→ E3→E4
4,500 4,498	0.5	E1→E3→E5→E7
4,000 4,010	0.3	E1→E3→E5→ E7→E2 →E4



(b) Visual Performance Comparison

Figure 5.1: An example performance analysis report

5.2.1 Overall Performance Summary

Since extra instrumentation or profiling of the application may slow it down and affect the performance measurement, Jack has to work with readily available execution logs and performance metrics.

As shown in Figure 5.1(a), from millions of log lines in the execution logs, our report flags 3 scenarios whose performance is statistically different than the previous run. Each row in the table corresponds to one performance deviated scenario. Among these three scenarios, there are two scenarios which are worse than the previous run (shown in red) and one scenario which is better (shown in blue). Scenarios are sorted by the degree of performance deviations from the previous run so that Jack can make best use of his time by working from the top.

Looking at the first row, Jack discovers that the first scenario has around the same throughput ($5,000$ versus $5,002$ requests per second) from both runs. However, he notices that the current scenario performance is worse than the previous run, indicated by the red colour under the deviation value 0.8. The event sequence triggered by this scenario is also displayed. Each event is abbreviated using its abstracted event id (E_1, E_2, E_3, and E_4).

5.2.2 Visual Performance Comparison

After gaining an overall impression about the system performance, Jack clicks the hyperlink under the deviation value to dig deeper into the performance of the first scenario. As shown in Figure 5.1(b), the report expands and shows a visual comparison of this scenario's performance between the two runs. He examines the two graphs seeking to answer the following two questions:

1. How does the scenario performance differ?

The top graph in Figure 5.1(b) is a beanplot [163], which visualizes the response time distributions of the same scenario in the previous and current runs side-by-side. The left side shows the response time distribution from the previous run and the right side shows the current run. The width of the plot indicates the frequency. For example, most of the instances in the current run take about 1 second and around 0.5 second in the previous run.

After examining the beanplot, Jack now has a better idea of why this scenario is flagged: First, the majority of the cases from the current run are slower than the previous run (1 second versus 0.5 second). Second, the maximum response time from the current run is 5 seconds compared with 4 seconds from the previous run.

2. How does the performance evolve over time?

The bottom graph in Figure 5.1(b) shows whether the system performance has degraded over the course of the load test. The horizontal axis indicates the start time of a scenario instance. The vertical axis indicates the response time of the scenario instances triggered at this moment. If more than one instances from this scenario are trigged at the same moment, the average response time from these instances is used.

For the first scenario, there are no performance degradations even though the response time fluctuates over time. Most of the time, the current run (in red) is above the previous run (in black), which again indicates that the current run is worse (i.e. slower).

5.2.3 Stepwise Performance Diagnosis

Once Jack gets a visual comparison of the performance differences for the first scenario, Jack clicks the link below the event sequences to find out the exact cause for the performance deviations (Figure 5.1(c)). Sample log lines from the execution logs are shown along with the average durations between the adjacent steps. In addition, deviations from

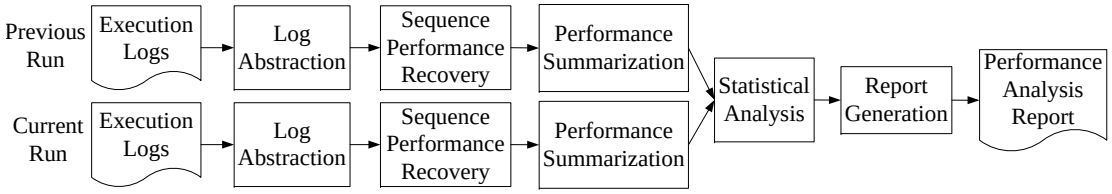


Figure 5.2: Our Approach to Generate the Performance Analysis Report

the adjacent event pairs between the previous and current runs are also shown.

For the investigated scenario, Jack concludes that the scenario performance degradation is mostly caused by a slow down between the browsing (E_2) and purchasing (E_3) events. The average duration of this event pair jumps from 0.5 second in the previous run to 1.8 seconds in the current run. The deviation between these two events, flagged (in red), is the highest deviation (0.9) among all the event pairs in this scenario. Jack then clicks the hyperlink below the 0.9 deviation to get a visual comparison of the performance differences in the $E_2 \rightarrow E_3$ pair. The report is again expanded to display similar graphs as in Figure 5.1(b) but for specific event pairs.

5.2.4 Reporting

Based on the analysis of the first scenario, Jack concludes that there is a performance problem between the browsing and purchasing events. He then performs similar analysis on the other two scenarios. Jack can now compose an email, which explains the performance problems. The email is sent to the appropriate developers with the performance analysis report attached.

5.3 Our Approach to Create the Performance Analysis Report

We now present our approach to generate the performance analysis report discussed in Section 5.2. As shown in Figure 5.2, our approach consists of five phases. For both tests, we

conduct log abstraction, recovery the performance of sequences and performance summarization. Then we flag the performance deviating scenarios by statistically analyzing the recovered performance data. Finally, a performance analysis report, ranked by the degree of performance deviation, is generated.

We explain our approach using the running example shown in Table 5.1.

Phase 1. Log Abstraction

Since additional instrumentation or profiling of the system slows down its execution, these techniques are not feasible for load testing analysis. As a result, our approach analyzes the readily available execution logs. Table 5.1(a) shows the first 8 log lines from an execution log file. We use the log abstraction technique proposed in Chapter 3 to automatically transform the execution logs into execution events as shown in Table 5.1(b).

Phase 2. Recovery the Performance of Sequences

As the system handles concurrent client requests, log lines from different scenarios are intermixed with each other in the execution logs. As shown in Table 5.1(a), scenarios related to users Tom, John and Mike are mangled together. Furthermore, some log lines (e.g., the 8th log line), which only output the system status information, are not related to any customer scenarios. These log lines should be filtered from our performance analysis.

In order to recover the performance from all the customer scenarios, we need to first recover the event sequences. We recover the sequences by linking the appropriate parameter values.

The first phase abstracts each log line into an execution event. Here, we use the dynamic values ($v) as well as their parameter names in the execution event. The parameter name-value pairs are used to link the related log lines into sequences. For example, the first line contains 3 parameter name-value pairs: sessionId with value 1, time with value 1, and user

with value $John$.

Table 5.1(c) shows the results after extracting and linking information from the log lines using the sessionId values of the log lines in Table 5.1(a). For example, the second (E_1), third (E_2), fourth (E_3), and sixth (E_4) log lines all share the same sessionId (2), so we group them together. The 8th log line in Table 5.1(a) corresponds to a periodic health check event, which has no sessionId and is used to determine if the system is alive. This log line is filtered out.

The performance of the recovered sequence is calculated by taking the time difference between the first and the last events. In our running example, the overall response time of the sequence ($E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$) with $sessionId = 2$ is 2 seconds. Since there is only one log line with sessionId 3, there is no duration information for this session.

The durations between every adjacent event pair are also calculated. The pairwise duration information is used later for stepwise performance diagnosis. The time duration between the first two events (E_1 from the 2nd log line and E_2 from the 3rd log line) is 0 seconds, and so on.

Phase 3. Performance Summarization

The previous phase has recovered the performance from the individual sequences. Identical event sequences correspond to the same scenario. As a load test repeatedly executes scenarios over and over, the sequence $E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$ can appear multiple times. In the third phase, we summarize the performance of each scenario.

Tables 5.1(a), 5.1(b) and 5.1(c) only process the first 8 log lines from the logs. Execution logs usually contain millions of lines. Table 5.1(d) shows an example of the summarized scenario performance using the entire execution logs. In total, we have 2 scenarios. The first scenario ($E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$) occurs 300 times. Among them, 100 times occurs with a duration of 2 seconds and 200 times with 3 seconds. The performance of the adjacent

event pairs from each scenario is also summarized. For example, the event pair $E_2 \rightarrow E_3$ from the first scenario takes on average 1.7 seconds. Among the total 300 occurrences of this event pair in this scenario, it takes 1 second in 100 times and 2 seconds in the other 200 times.

Phase 4. Statistical Analysis

Phrase 3 uncovers the performance of all the scenarios from the previous and current runs. The fourth phase evaluates the overall performance of each scenario and pin-points the performance deviating event-pairs.

We compare each scenario's performance in the previous and current runs using a statistical test. We use the unpaired student-t test to compare the scenario response time distributions from the previous run against the current run. Furthermore, we use the type of student-t test [151], which outputs a confidence interval. Compared with hypothesis testing, whose output only answers whether the two distributions are statistically the same, a confidence interval also provides possible ranges. Depending on the relative position of the confidence interval to zero, we can tell which run has better performance (i.e. timing) for this scenario. We only show the scenarios, whose performance is statistically different between the two runs, in our report.

Once the scenarios with deviated performance are flagged, we need to pin-point the event pairs which cause the performance deviations. We achieve this by applying the same statistical test on all the adjacent event pairs. For example, for the flagged scenario ($E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$), we compare the performance of $E_1 \rightarrow E_2$, $E_2 \rightarrow E_3$, and $E_3 \rightarrow E_4$ between the previous and the current runs.

Phase 5. Report Generation

As the load testing practitioner has limited time, we rank the potentially troublesome scenarios to help him prioritize his time. We rank each scenario by the degree of the performance deviation between the previous and current runs.

Cosine distance, which measures the degree of similarity between two distributions, outputs a value between 0 and 1. If the two distributions are very similar, the cosine distance is close to 1. If they are very different, the cosine distance is close to 0. As deviation is the opposite of similarity, we use the following formula to calculate the deviation:

$$deviation(P, C) = 1 - cosine(P, C) \tag{5.1}$$

$$cosine(P, C) = \frac{\sum_x P(x)C(x)}{\sqrt{\sum_x P(x)^2}\sqrt{\sum_x C(x)^2}} \tag{5.2}$$

Note that P(x) and C(x) correspond to the number of instances in the previous and current runs which have response time x for a particular scenario/ For example, if the cosine distance of the first scenario in Table 5.1(d) is 0.2, their deviation value is 0.8.

5.4 Case Studies

We conducted 3 case studies on 3 different systems (2 open source applications and 1 large enterprise application). We seek to verify whether our performance analysis report can help load testing practitioners in their usual tasks. As our approach only flags the performance deviated scenarios, it is up to his knowledge to decide whether the deviated performance leads to performance problems.

We use precision to evaluate the performance of our approach. The precision is defined as follows:

Table 5.1: The Running Example

#	Log Lines
1.	sessionId=1, time=1, user=John, login successfully
2.	sessionId=2, time=1, user=Tom, login successfully
3.	sessionId=2, time=1, user=Tom, browse catalog, catalog=bird
4.	sessionId=2, time=2, user=Tom, add item, item=parrot, quantity=1
5.	sessionId=1, time=3, user=John, browse catalog, catalog=dog
6.	sessionId=2, time=3, user=Tom, user checkout successfully
7.	sessionId=3, time=4, user=Mike, user search item, item=rattlesnake
8.	time=4, health check

(a) Example Execution Log Lines

Event ID	Event Template	#
E1	sessionId=$v, time=$v, user=$v, login successfully	1, 2
E2	sessionId=$v, time=$v, user=$v, browse catalog, catalog=$v	3, 5
E3	sessionId=$v, time=$v, user=$v, add item, item=$v, quantity=$v	4
E4	sessionId=$v, time=$v, user=$v, user checkout successfully	6
E5	sessionId=$v, time=$v, user=$v, user search item, item=$v	7
E6	time=$v, health check	8

(b) Abstracted Execution Events and Corresponding Log Lines

sessionId	Event Sequences	Durations
1	E1→E2	2
	⇨ E1→E2	2
2	E1 →E2→E3→E4	2
	⇨ E1→E2	0
	⇨ E2→E3	1
	⇨ E3→E4	1
3	E5	

(c) Recovered Sequences with Durations Calculated

Event Sequences	Durations
E1 →E2→E3→E4	2 (100), 3 (200)
⇨ E1→E2 (0)	0 (300)
⇨ E2→E3 (1.7)	1 (100), 2 (200)
⇨ E3→E4 (1)	0 (100), 1 (100), 2 (100)
E1 →E3→E5→E7→E4	3 (200)
⇨ E1→E3 (0.5)	0 (100), 1 (100)
⇨ E3→E5 (1)	1 (200)
⇨ E5→E7 (1)	1 (200)
⇨ E7→E9 (0.5)	0 (100), 1 (100)

(d) Summarized Response Time for Each Scenario

$$precision(\%) = (1 - \frac{\#\ of\ false\ alarm\ scenarios}{Total\ \#\ of\ flagged\ scenarios}) \times 100\%$$

Multiple scenarios with deviating performance can be caused by the same performance problem, which can be an indication of the impact of this problem. If a flagged scenario does not lead to a performance problem, then this scenario is considered as a false alarm.

A load testing practitioner conducts and analyzes a load test of an evolving system seeking to accomplish many of the following tasks: to recommend optimal system settings, to certify software/hardware platforms, and to study the impact of design changes. We use this to structure our case study. For each study, we first give a brief description of the studied system and an overview of the task which the load testing practitioner needs to complete. Then we explain the steps to conduct the experiments and data collection procedure. Finally, we present our results and conclusions.

Our performance analysis prototype is written in Perl and uses R [30] to generate the graphs.

5.4.1 Task 1: Recommending Optimal Configurations

Enterprise systems interact often with other software components like databases or mail servers. Sub-optimal configurations of these components can impact the performance of the system. A load testing practitioner needs to explore these configurations and provide recommendations for deployment. In this task, we use the Dell DVD Store (DS2) as our case study system.

Studied System: the Dell DVD Store

The Dell DVD Store (DS2) application is an open source online application used for benchmarking Dell hardware. [6]. DS2 provides basic e-commerce functionality, including: user registration, user login, product search, and item purchase.

DS2 contains a database, a load generator and a web application. It comes in different distribution packages which support various web platforms (e.g. Apache Tomcat or JBoss) and database vendors (MySQL, Microsoft SQL Server, and Oracle). The web application consists of four JSP pages which interact with the database and display dynamic contents. The DS2 load generator supports a range of configuration parameters to specify the workload. In this task, we use MySQL as the backend database and Apache Tomcat as our web server engine.

Goal: Recommending Optimal MySQL Configuration

We seek to evaluate the performance impact due to the following two software configuration options for the MySQL database:

Caching: Whether to cache the query results or not;

Storage Engines: Whether to use MyISAM or InnoDB as the storage engine.

Experiments and Data Collections

DS2 has no logs, thus we manually instrumented its four JSP pages. We run 4 one-hour load tests: InnoDB with and without query caching, and MyISAM with and without caching. All 4 runs are exercised under the same workload and are all conducted on a single core CPU machine with 1 GB of RAM and a $5,400$ rpm hard disk. Each test generated over $120,000$ log lines.

Result Analysis and Conclusions

We conduct the performance analyses on the results of these 4 runs.

To determine the performance impact of caching, we perform two comparisons. First, we compare the results from InnoDB with/without caching, then MyISAM with/without

caching. We use the session ids to recover the sequences. Both of our performance analysis reports have flagged 18 performance deviated scenarios. Examining the stepwise performance diagnosis sub-tables across all the event pairs, we find events, which invoke database operations, perform better with caching enabled.

To determine the performance impact of different storage engines, we compare the results from MySQL configured with MyISAM engine against those with the InnoDB engine. Since our previous analysis shows that enabling caching yields better performance, we only compare the two runs with caching enabled. Again, 18 scenarios are flagged. By investigating stepwise performance diagnosis sub-tables, we find database-related events perform better with the InnoDB engine. Our finding agrees with prior benchmark studies [14].

Using our performance analysis reports, we conclude that DS2 should be deployed with the following MySQL configurations to achieve optimal performance: InnoDB as the storage engine and cache-enabled. The precision is 100% among all three performance analysis reports.

5.4.2 Task 2. Certifying Software/Hardware Platforms

Nowadays, systems support various software platforms (e.g., operating systems) as well as hardware platforms. A load testing practitioner needs to check whether a system under test can make optimal use of the available services and resources. In this task, we use the JPetStore as our case study system.

Studied System: JPetStore

JPetStore [11] is a larger and more complex open source web application relative to DS2. Unlike Sun's original version of Pet Store which is more focused on demonstrating the capability of the J2EE platform, JPetStore is a re-implementation with a more efficient design [13] and is targeted on benchmarking the J2EE platform against other web platforms

such as .Net. Unlike DS2 which embeds all the application logic into the JSP code, JPetStore uses the "Model-View-Controller" framework and XML files for object/relational mappings. In this task, we have deployed the JPetStore application on Apache Tomcat and used MySQL as the database backend.

Goal: Certifying a Multicore Server

In this case study, we seek to certify the hardware platform by checking whether there is a performance gain by migrating JPetStore from a single core machine to a more powerful multi-core server.

Experiments and Data Collection

As JPetStore does not come with a load generator, we use Webload [32], an open source web load testing tool, to load test the application. Using Webload, we have recorded a single customer scenario for replay during load testing. In addition, we configure the Webload so that it incrementally increases the workload as the load test progresses.

On each hardware platform, we conduct one one-hour run. The first run uses one machine which has a single CPU with 1G of memory and one $5,400$ rpm hard-disk. The second run uses another machine which has a Quad-Core CPU with 8G of memory and one $7,200$ rpm hard-disk. Both runs generate over $770,000$ log lines.

Result Analysis and Conclusions

Figure 5.3 shows the performance analysis report for JPetStore. The report has flagged our only scenario. The colour of deviation (blue) shows that the run on the Quad-core machine statistically out-performs the run on the single CPU machine. However, the visual comparison of the two runs reveals that:

1. The contrast of the maximum and minimum response time from both runs is very

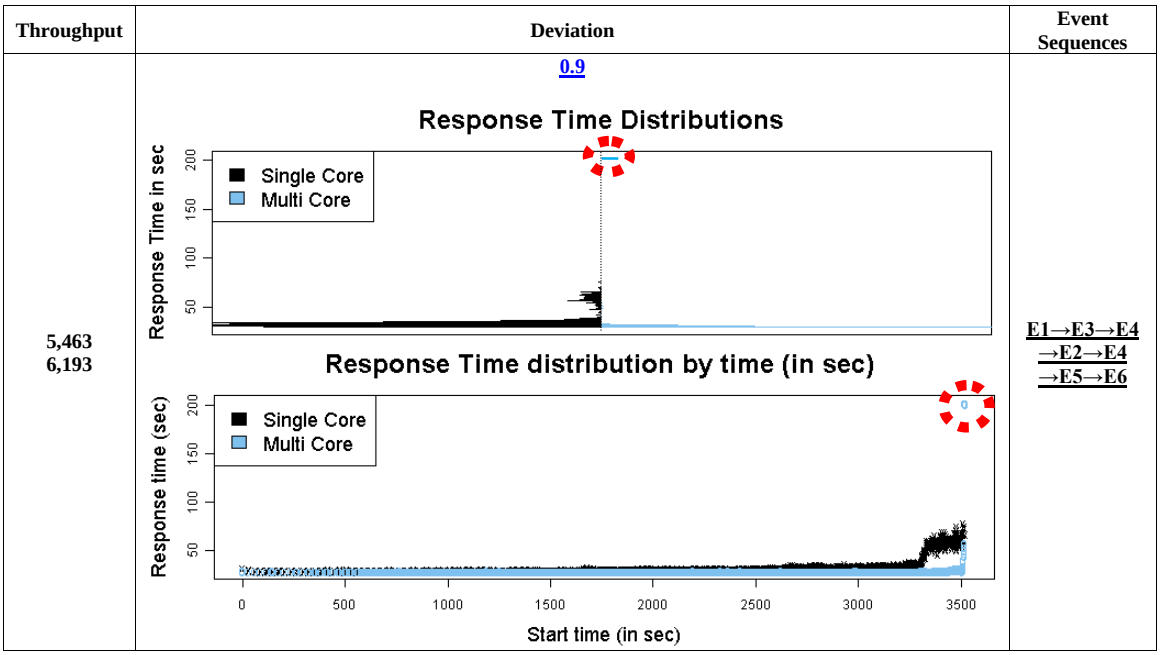


Figure 5.3: The performance analysis report on JPetStore

large, especially for the run on the multi-core server. As shown in the beanplot of Figure 5.3, the minimum response time for the Quad-core machine is around 30 seconds and maximum is around 200 seconds.

2. As shown in the lower graph of Figure 5.3, both runs slow down as the load test progresses due to the steady increase in the workload. However, the response time on the Quad-core machine during the last few seconds is about 3 times longer than the single core machine during heavy load (around 200 seconds on the Quad-core machine versus around 80 seconds on the single core machine).

We repeat both runs and still obtain the same patterns. Through close examination of the stepwise performance diagnosis sub-table in our report, we reveal a MySQL performance bug. The MySQL InnoDB storage engine has trouble scaling to multi-core machines. Our finding is confirmed by MySQL developer postings [9].

Using our performance analysis approach, we conclude that the JPetStore performance improves on a more powerful machine. However, there is a MySQL performance bug which prohibits the efficient use of the multi-core hardware architecture under heavy load. The precision of this report is 100%.

5.4.3 Task 3. Studying the Impact of System Changes

Systems are constantly undergoing maintenance changes to fix bugs and to cope with new standards or new interfaces. Changes can cause suboptimal system performance. In this task, we use a large enterprise application as our case study system.

Studied System: A Large Enterprise Application

The system we studied is a large distributed enterprise application. This application supports a large number of scenarios which are used by thousands of users simultaneously.

Goal: Evaluating Different Software Designs

There is a new software build which uses a different communication mechanism. We want to determine the performance impact of the new design.

Experiments and Data Collection

A load testing practitioner has conducted the experiments. Two 8-hour load tests are run on different software builds under the same load. Both runs generate over 2 million log lines.

Result Analysis and Conclusions

This is a large enterprise application with readily available execution logs. After briefly going through the log lines, we find three main parameters which can be used to link log

lines to form scenarios.

Our report flags around 30 performance deviating scenarios. Most of the scenarios are executed more than $4,000$ times. Our report reveals that the current build performs worse than the previous build. Examining the step-wise performance diagnosis sub-tables, this load testing practitioner is able to pin-point the performance deviating steps and to subsequently identify the performance problems of the new design.

Our performance analysis report has a precision of 77%. A few flagged sequences are not scenarios. Thus, false alarms are caused by the random time intervals between the adjacent event pairs.

We conclude that our performance analysis report has helped the load testing practitioner evaluate the performance impact of different designs. The precision of our performance analysis report is 77%.

5.5 Discussion of Results

In this section, we present some discussions.

Performance Comparisons

Our approach uses results from a prior run as an informal performance baseline. The performance baseline can also be derived from the data in one or more runs. For example, we can form the performance baseline by combining the results from 10 runs conducted last year. The large resulting data is more desirable to infer the past common performance behavior.

We believe both the performance improving and degrading scenarios are worth investigating for two reasons. First, a load testing practitioner needs to verify the performance bug fixes by comparing the current run against a test which suffers from performance problems. In this case, the performance improvement cases are also of interest to him. Second,

we do not want to miss potential performance problems. Take our JPetStore case study for example. Performance on a powerful server is generally better but suffers from severe slow down under stress conditions.

Scenario Recovery

Our approach requires some manual work at the beginning as we need to find what are the identifying parameters in the logs. This is a one-time effort and requires minimal effort. These parameters can either be obtained by asking a domain expert or by skimming through the logs.

Statistical Analysis

Our approach uses a student-t test to compare the response time distributions. A student t-test is a parametric statistical test which assumes that the response time is normally distributed. Non-parametric statistical tests, like the Kolmogorov-Smirnov test [75], can also be used in our analysis. These tests hold no assumptions about the distribution of the data. Parametric tests are less strict than non-parametric tests. In other words, non-parametric tests tend to say there are no statistical differences between two data sets whereas the parametric tests would indicate a statistical significant difference. We choose to use the less strict student t-test, since we want to find all the possible performance problems. Furthermore, we have also tried using the Kolmogorov-Smirnov test to verify the findings in our case studies. We find that Kolmogorov-Smirnov test results agree with the student-t test. This finding confirms with the results reported by Bulej et al. [79].

Analyzing System Performance Using Execution Logs

Our approach uses readily available execution logs and infers system performance based on the time differences between log lines. Therefore, our approach is limited by the granularity

of the log lines. Furthermore, we assume that logs are relatively stable over time. This assumption holds true for many industrial systems. Since the logs are already processed by many other tools, log changes are usually minimized. In the future, we plan to apply approximate matching on different sequences between runs.

New Scenarios

Since there are no formal performance objectives or data from previous runs for new scenarios, our current approach will not analyze them. In the future, we plan to infer the expected scenario performance based on existing scenarios.

5.6 Related Work

We discuss two areas of related work.

Load Testing

Most existing load testing research focuses on the automatic generation of load test suites [48, 54, 55, 66, 78, 121, 251]. The previous chapter (Chapter 4) focuses on automatically uncovering functional problems in a load test. Bulej et al. [79] propose the concept of regression benchmarking as a variant of regression testing for performance regression. Regression benchmarking compares the performance across different versions of the systems using the same benchmarking suite. Our work is an improvement over the regression benchmarking in four aspects. First, our approach recovers the scenarios automatically without specification. Second, our approach is more fine-grained, as we analyze the performance of the scenarios as well as the individual steps in each scenario. Third, our approach minimizes the amount of manual processing. Fourth, the reported problems in our performance analysis report are ranked so that a load testing practitioner can prioritize his efforts and make optimal use of his time.

Automated Performance Monitoring and Analysis of Production Systems

Automated performance monitoring and analysis of production systems can be divided into two subcategories: analyzing the performance metrics and analyzing the logs.

The following work analyzes the performance metrics: Avritzer et al. [50, 49] propose various algorithms to detect the need for software rejuvenation by monitoring the changing values of various system metrics. Mi et al. [89, 202] and Cohen et al. [90, 254] develop application signatures based on the various system metrics (like CPU, memory). The application metrics are further used for efficient capacity planning and anomaly detection. The main difference between these approaches and ours is that we use execution logs for our analysis. Compared with system metrics, execution logs provide more in-depth domain specific information.

The following work analyzes the readily logs with no additional instrumentations: Aguilera et al. [42, 216] developed various algorithms to perform black-box performance debugging on distributed systems. They use the header information on the TCP packet traces (source, destination and time) to infer the dominant causal paths through a distributed system. Unfortunately, the accuracy of the inferred causal paths decreases as the degree of parallelism increases. This is not ideal for load testing analysis as there are many message exchanges occurring simultaneously. Marwede et al. [189] use the timing anomaly to automatically uncover functional problems.

5.7 Conclusion

It is difficult to conduct a performance analysis of load testing results due to the absence of a documented performance baseline, time pressure, monitoring overhead and large volume of data. In this chapter, we propose an approach which automatically flags possible performance problems by adopting a previous run as a performance baseline and comparing against it. Our approach is easy to adopt and scales well to large systems with high

precision (77%). In the next chapter, we will take a different perspective of the analysis of system behavior under load: rather than detecting load-related problems for a single load test, we will propose an automated approach to estimate the overall system quality by using data from several load tests.

CHAPTER 6

Automatic Estimation of System Reliability

The current agile continuous delivery process requires each version of the software system to pass the quality criteria before the system can be released to the field. However, current release criteria mainly focuses on the functionality at a small user level (e.g., passing the functional tests) but not on load. In this chapter, we propose to use reliability as a quality index to assess the quality of a system under load in the continuous delivery process. Software reliability is defined as the probability of failure-free operation for a period of time, under certain conditions. Many large commercial systems are marketed to have very high software reliability requirements (e.g., "three-nine" reliability). However, the general software reliability estimates provided are usually derived from synthetic benchmark workload runs, which is not representative of the actual field usage. The discrepancy in usage patterns could lead to inaccurate reliability estimates for each field deployment.

In this chapter, we propose an approach, which provides a customized reliability estimate for each deployment field based on mining repositories of execution logs. Rather than focusing on one specific load test like we did in our previous two chapters, we use data from many load tests to estimate the system reliability. Our approach abstracts the system behavior into system states, which represents the current active requests. We extract the customer usage, which is the occurrence information of systems states, from the field pre-deployment logs. Then, we extract the failure information for the same system states from load testing and other pre-deployment logs. By combining the usage and failure information of these states, we can derive a customized reliability.

6.1 Introduction

NOWADAYS, many software systems employ the agile software development process, in which software systems are developed and released continuously to the field as soon as the systems pass the quality criteria [46, 110]. However, most

of these quality criteria is focused on functionality (e.g., passing all the the unit testing, integration testing, and end-to-end functional testing). Quality-related metrics, which describe the system performance under load, are seldom included. However, as studies show that many field problems are not due to feature bugs, but rather due to systems not scaling to field workloads [38, 242]. It is important to include the software performance under load as a quality measure into the release criteria.

Software reliability is defined as the probability of failure-free operation for a period of time, under certain conditions [113, 205][1]. In the traditional software development process, software reliability is used as a quality index to monitor and assess the quality of the large mission-critical systems under load [55]. These systems can range from web applications to telecommunication infrastructures, and they must support concurrent access by thousands or millions of users while functioning over a long period of time. In this chapter, we propose the use of software reliability as a quality metric in the agile continuous delivery process.

However, the major challenge of adopting reliability measurements in the continuous delivery process is that one size does not fit all. The general reliability estimates are usually derived based on workloads from synthetic benchmark load runs and/or early field deployments of the system. These benchmark workloads rarely match the actual field workload, leading to estimates that do not match the expected field reliability of the system [73, 237]. This means that the general reliability estimates might not be realistic, i.e., not reflective of the actual field reliability [47]. For example, if deployment field A does not use one buggy feature as much, then the expected reliability of the system will be much higher than the estimate provided by the general reliability estimate.

The previous two chapters (Chapters 4 and 5) use history of prior tests to detect functional and performance problems. This chapter uses history of all prior tests to generate a

[1]There are also other definitions of reliability (e.g., [36, 37]). In this chapter, we use the definition from Musa et al. [113, 205].

quality index using software reliability for the system under test using execution logs. Our approach analyzes rarely used, yet readily available execution logs to produce empirically validated and customized reliability estimates for mission-critical systems. Basically, our approach uses logs generated from customer sites (e.g., user acceptance testing or previous customer logs) to identify the usage patterns ("system states") that are important for a customer, and the "occurrence probability". Then, our approach uses logs derived from thousands of hours of execution from prior deployments and earlier tests of the same version of the system to provide a good estimate of how often the identified system states fail ("failure probabilities"). By combining the occurrence probabilities from the previous pre-deployment testing with the failure probabilities from log repositories, we can produce an accurate, customized reliability estimate using less resources and time.

The main contributions of this chapter are as follows:

- Our approach requires no additional instrumentation or profiling, instead it leverages widely available, yet rarely used execution logs.
- Our approach provides a useful enhancement for the existing agile continuous delivery process with minimal disruptions.

Organization of the Chapter

The remainder of this chapter is organized as follows: Section 6.2 provides an example to motivate the benefits of using deployment-specific reliability estimates. Section 6.3 first presents an overview of our reliability estimation approach, then describes the three phases (Sections 6.3.1, 6.3.2 and 6.3.3) involved in our approach. Section 6.4 evaluates our approach on a large mission-critical system. Section 6.5 presents the threats to validity. Section 6.6 presents related work. Section 6.7 concludes this chapter.

6.2 Motivating Example

In this section, we present an example to motivate our approach. Jack's company is developing a software system, which is an application platform enabling various e-commerce business. Jack's software system can be deployed either in the cloud (cloud deployment) or on the customer premise (on-prem deployment). Jack's system, which is developed using an agile process and produces a new software version every other week, currently serves many e-commerce vendors across the world. These versions pass the quality criteria, which includes passing all the unit, integration and end-to-end functional testing. Rather than rolling out the new release to all the vendors, Jack decides to follow a more cautious approach: only upgrading the system whose performance under load is satisfactory. However, the challenge that Jack faces is how to accurately assess the software performance under load for each vendor, as each vendor has different usage patterns and deployment configurations (on-prem or cloud) may vary. One approach is to conduct on-site user acceptance testing (field load testing) for the on-prem deployment and customer workload specific load testing for the cloud deployment. However, such specialized load testing is costly and time consuming.

Tom, one of the developers of the system, proposes to use software reliability as a quality index to assess the software performance under load and only upgrade the vendors, whose estimated reliability reaches the advertised "three-nines" (0.999). A reliability of 1 means that the system never fails. A reliability of 0.999 indicates that 99.9% of the time the system will function correctly with no crashes and no Service Level Agreement (SLA) violations. Tom further added that he has an approach to calculate the reliability of the system for each vendor efficiently and accurately.

Tom's approach is based on work done by Avritzer [57] to model the reliability of telecommunication systems. Tom explained his approach by drawing the analysis of one

Table 6.1: System State and Failure Profile Derived from Synthetic Runs and Other Deployments

System States (Search, Browse, Purchase)	Occurrence Probability	Failure Probability
(0,0,0)	0.40	0
(0,1,0)	0.30	0
(1,1,0)	0.20	0
(1,1,1)	0.10	0.10

Table 6.2: System State Profile Recovered from Previous Customer Logs

System States (Search, Browse, Purchase)	Occurrence Probability
(0,0,0)	0.15
(0,1,0)	0.15
(1,1,0)	0.10
(1,1,1)	0.60

vendor, vendor A, as an example. Instead of simply capturing the workload using a black-box approach (e.g., by measuring metrics like the number of transactions per second), Tom uses a white-box approach that captures the internal state of the system as it processes the workload. These states are influenced by the deployment environment and workload. Tom defines the system state for the system as a 3-value tuple that captures the usage of the system in terms of the following three currently executing scenarios: browsing, purchasing and searching. Tom opts for this simple high-level definition, although other more complex low-level definitions are also possible. For example, the browsing scenarios could be further divided into browsing catalogs and browsing recommendations.

Using this system state model, Tom samples the execution of the system at run-time and determines which states it resides in. He defines each value in the tuple to be the number of active scenarios at the moment. For example, the system state (0, 0, 0) indicates the system is in the idle state. The state (1, 1, 1) indicates that the system is currently processing 1 search, 1 browse and 1 purchase scenario concurrently. Other models, such as those including the percentage of utilization, are possible as well.

By continuously sampling the data from the in-house testing and other deployments,

Tom derives a system state profile for the system, i.e., an overview of the system states that occur together with their frequency. Table 6.1 shows the system state profile derived from synthetic runs and other deployments. The profile indicates that the system is in idle state (0,0,0) 40% of the time. At each sample, Tom also determines if there are any reported failures of the system (e.g., crashes or SLA violations). He can then calculate the reliability for each system state. Looking back at Table 6.1, we find that no failures are reported in the occurrences of state (0,0,0), while 10% of the occurrences of state (1,1,1) exhibit some type of failure. Using the sampled failure distribution and the system state profile, Tom can derive a reliability estimate for the system. By combining data from multiple deployments, Tom concludes that based on the lab testing and other deployments, the general reliability for this system is 0.99 (failure occurs only in $0.40 * 0 + 0.30 * 0 + 0.20 * 0 + 0.10 * 0.10 = 1\%$ of the cases).

However, this in-house reliability estimate does not consider the differences in customer usage and deployment patterns in different deployments. For example, based on last Monday's logs from vendor A (cloud deployment), Tom notices that state (1,1,1) occurs at a much higher frequency (60% in Table 6.2 showing the occurrence probability of system states) compared to the data used to calculate the in-house reliability estimate (10% in Table 6.1). It is clear that vendor A's deployment spends more time in state (1,1,1), which has a high failure probability (based on in-house testing). This knowledge should be used to customize the reported in-house reliability estimate. The customized reliability estimate for vendor A's is 0.94 (failure occurs in $0.15*0+0.15*0+0.10*0+0.60*0.10 = 6\%$ of the cases) instead of the general in-house estimate of 0.99. Based on our customized reliability calculation, the new upgrade should not be rolled to vendor A's site. This deployment estimate is derived from the occurrence probabilities of states in a single day of testing (second column of Table 6.2), and the failure probabilities of states in hundreds of hours of execution in the lab and other deployments (third column of Table 6.1). Therefore, upgrading vendor A should put on hold until later versions.

We note three novel contributions of our approach:

1. It is important to capture the internal system states and their failure probability when we estimate the system reliability. The system state profile is influenced by both the customer usage pattern and the deployment characteristics. Given two deployments with the same usage pattern, they might still end up with different system state profiles because of different deployment characteristics. For example, if one of the deployments has a much slower database, the system might not be able to process searches as fast, causing the system states to have a higher number of active "Search" requests. The deployment with a slower database will likely have lower reliability even under the same workload.

2. Our approach makes use of the execution logs of the system to calculate the system state profiles, instead of instrumenting or profiling the system during runtime. Sampling a system during runtime is not feasible in a production setting due to the high overhead [59, 182]. Execution logs are readily available and are often used for remote issue resolution and for legal compliance purposes (e.g., "Sarbanes-Oxley Act of 2002" [23]). By sampling the logs at a constant frequency (e.g., once a second), we can create a system state profile and failure profile without impacting the performance of a system.

3. The use of execution logs permits Tom to continuously improve the reliability estimate of the system, since he can keep on integrating new logs coming from field deployments in an automated fashion. As for each vendor, he would need to provide a log that captures the expected usage patterns. To provide such a log, vendors could provide logs from past execution or logs from a limited deployment of the new version of the system (e.g., user acceptance testing logs). For example, based on vendor A's past usage data, Tom can provide a reliability estimate based on hundreds of deployments that have been running over the past six months.

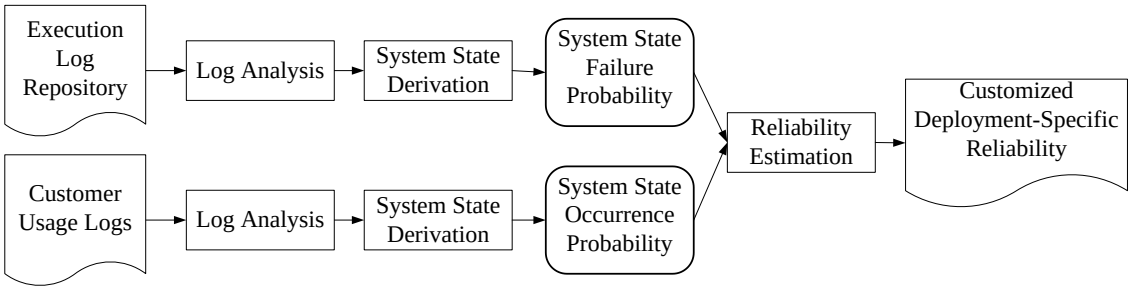


Figure 6.1: An Overview of Our Deployment-specific Reliability Estimation Approach

6.3 Approach Overview

Figure 6.1 gives an overview of our approach, which consists of the following three phases:

1. *Log Analysis:* Recover the executed scenario instances and identify the reported errors from execution logs.

2. *System State Derivation:* Derive the system states and calculate the occurrence probability and failure probability for each system state.

3. *Reliability Estimation:* Estimate the deployment-specific reliability using Bayesian Networks.

The *Log Analysis* and *System State Derivation* phases both analyze two sources of execution logs:

– An *execution log repository*, which stores the load test and other field deployment logs for the new version of the system. We analyze the execution log repository to get a collection of all possible system states as well as their *failure probability*.

– *Customer usage logs*, which can be user-acceptance testing or past usage data containing the customer usage profiles. We analyze these logs to obtain the *occurrence probability* distribution of those system states that occur in practice at the deployment site.

Table 6.3: Example log lines

#	**Log Lines**
1	time=1, thread=1, session=1, receiving new user registration request
2	time=1, thread=1, session=1, inserting user information to the database
3	time=1, thread=2, session=2, user=Jack, browse catalog=novels
4	time=1, thread=2, session=2, user=Jack, sending search queries to the database
5	time=3, thread=1, session=1, user=Tom, registration completed, sending confirmation email to the user
6	time=3, thread=2, session=2, database connection error: session timeout
7	time=4, thread=1, session=1, fail to send the confirmation email, number of retry = 1
8	time=6, thread=2, session=2, user=Jack, successfully retrieved data from the database
9	time=7, thread=2, system health check
10	time=8, thread=1, session=1, registration email sent successfully to user=Tom
11	time=9, thread=2, session=3, user=Tom, browse catalog=travel
12	time=10, thread=2, session=3, user=Tom, sending search queries to the database
13	time=10, thread=3, session=4, user=Jim, updating user profile
14	time=11, thread=3, session=4, user=Jim, database error: deadlock

Then, we calculate the deployment-specific reliability by matching the occurrence probability distribution of the system states obtained from the customer usage logs with the corresponding failure probability distribution of these system states obtained from the execution log repository.

In the next three subsections, we will use logs from a small online bookstore as a running example to explain the aforementioned three steps of our approach.

6.3.1 Log Analysis

Instead of instrumenting an system, we make use of the readily available execution logs to recover the needed information for our analysis. The techniques that we use for abstracting the logs into events, recovering scenario instances and their timing information from logs are explained in Chapters 3 and 5, but here we briefly summarize the technique using a sample execution log. Table 6.3 shows the first fourteen log lines from the log repository of an online bookstore. These log lines record software activities (e.g., line one), system health (e.g., line nine) as well as errors (e.g., line fourteen).

Each usage scenario in the logs consists of a sequence of steps. For example, as shown in Table 6.3, a user registration scenario consists of the following steps:

1. A user sends a request to the web server;

Table 6.4: Abstracted Execution Events and Corresponding Log Lines

ID	Event Template	#
E1	time=$v, thread=$v, session=$v, receiving new user registration request	1
E2	time=$v, thread=$v, session=$v, inserting user information to the database	2
E3	time=$v, thread=$v, session=$v, user=$v, browse catalog=$v	3, 11
E4	time=$v, thread=$v, session=$v, user=$v, sending search queries to the database	4, 12
E5	time=$v, thread=$v, session=$v, user=$v, registration completed, sending confirmation email to the user	5
E6	time=$v, thread=$v, session=$v, database connection error: session timeout	6
E7	time=$v, thread=$v, session=$v, fail to send the confirmation email, number of retry=$v	7
E8	time=$v, thread=$v, session=$v, user=$v, retrieving data successfully from the database	8
E9	time=$v, thread=$v, system health check	9
E10	time=$v, thread=$v, session=$v, registration email sent successfully to user=$v	10
E11	time=$v, thread=$v, session=$v, user=$v, updating user profile	13
E12	time=$v, thread=$v, session=$v, user=$v, database error: deadlock	14

2. The web server processes the request and stores the user information in a database server;

3. A confirmation email is sent out to the user.

Furthermore, each step in these scenarios shares certain identification values such as session and user ids. We need to know the frequency of different scenarios as well as how long each of them takes. Therefore, we recover the scenario instances by first abstracting log lines into execution events. Then we link related log lines (events) into sequences, whose frequency and duration can then be determined easily.

Step 1. Log Abstraction

Log lines are the output of the debug statements that developers insert into the source code. Each log line is a mixture of static and dynamic information. The static information describes the execution event (i.e., the context), whereas the varying (i.e., dynamic) parts are parameter values generated at run-time. Different values for the latter parameters cause the same execution event to result in different log lines. For example, the fourth and the twelfth log lines are generated from the same code location, but they are different since they are generated by the execution of different sessions.

In Chapter 3, we have proposed an approach that automatically abstracts log lines into

Table 6.5: Recovered Scenario Instances

Session	Log lines	(Start, End)	Keywords
1	1,2,5,7,10	(1,8)	Register
2	3,4,6,8	(1,6)	Browse
3	11,12	(9,10)	Browse
4	13,14	(10,11)	Update

execution events and marks the dynamic and static parts, such that log lines that are related to the same session can be grouped into sequences later on (step 2). Furthermore, the abstracted events can also be used to identify failure events. Table 6.4 shows the results of log abstraction in our running example.

Step 2. Scenario Sequence Recovery

As the system handles concurrent client requests, log lines from different scenarios are intermixed with each other in the execution logs. The log lines in Table 6.3 are generated as a result of the activities of different users: Tom, Jack and Jim. Furthermore, there are two scenarios related to Tom: user registration and catalog browsing. We recover the sequences and the timing information by linking the appropriate parameter values using the same techniques presented in Chapter 5. In our running example, we use session ids to automatically link related log lines. The results are shown in Table 6.5. In addition, the third column of the table also shows the timestamp of the first and last steps of each recovered sequence. For example, session two started at time one and ended at time six.

6.3.2 State Derivation

We can now derive the system states and estimate the occurrence probability and failure probability associated with these system states based on the recovered scenario sequences from the log analysis.

We first categorize the scenario sequences into groups and identify any failures. Then, we derive the system states by taking a snapshot of the system's scenarios at a fixed time

interval. The associated occurrence probability and failure probability for each state are also calculated.

Step 1. Sequence Labeling

Executing one scenario can exercise different code paths, therefore, resulting in different sequences. Hence, at the end of our Scenario Sequence Recovery step, there can be hundreds of sequences corresponding to only a handful of scenarios. We need to further reduce the amount of sequence data by properly categorizing (labeling) sequences into scenarios. We label each sequence with keywords specified by a domain expert. This process is done once for a system using keyword matching in the corresponding log entries. For new versions of a system, the domain expert might need to update some of the keyword mappings.

Our online bookstore example supports four types of operations: register, browse, purchase and update. These are the keywords used to label the sequences. When we match the keywords against each execution event in the scenario sequences, each word from the event's log entry is mapped into its root form (i.e. word stemming). For example, words like "browsing" and "browsed" will be mapped to the same root form "browse". The last column of Table 6.5 shows the labeled scenario names for each sequence.

Step 2. Failure Identification

We identify two types of failures in the logs based on domain knowledge. We identify and categorize failures as follows:

1. *Functional Failures (or Severity 1 failures)* are associated with system-wide outages. Examples of functional failures are system crashes, system restarts and system deadlocks. A domain expert is used to identify severity 1 failures by marking specific keywords (e.g., "thread dump") in the log files. For the example shown in Table 6.3, there is one functional error which occurs at line 14. It is a deadlock error.

Table 6.6: Derived System States and Their Failure Occurrences (Sampling Interval == 1)

Time (t)	**States ($S^r(t)$) (Register, Browse, Purchase, Update)**	**Failure (-/x)**
0	(0, 0, 0, 0)	-
1	(1, 1, 0, 0)	-
2	(1, 1, 0, 0)	-
3	(1, 1, 0, 0)	-
4	(1, 1, 0, 0)	-
5	(1, 1, 0, 0)	-
6	(1, 1, 0, 0)	x
7	(1, 0, 0, 0)	x
8	(1, 0, 0, 0)	x
9	(0, 1, 0, 0)	-
10	(0, 1, 0, 1)	-
11	(0, 0, 0, 1)	x

2. *Performance Failures (or Severity 2 failures)* are associated with performance slowdowns. Performance failures impact the user experience. Examples of performance failures are Service Level Agreement (SLA) violations. We basically compare the duration for each scenario's instances and see if it takes longer than the time specified by the SLA. If the SLA for the online bookstore states that all the scenarios should be executed within 5 seconds, there are two severity 2 errors at time 6 for session 1 and session 2 (as both sessions started at time 1). In Chapter 5, we presented an approach and a tool to provide automated support for rapidly identifying such failures.

Step 3. System States Derivation

We derive the system states by taking a snapshot at a fixed time interval based on the scenario durations. The snapshot interval must be smaller than the response time of any scenario, to ensure we do not miss any scenario information. We define the system state from the log repository at time t $S^r(t)$ to be: $S^r(t) = (s_1, s_2, \ldots, s_n)$, where s_i is the number of active scenarios of type i at time t and n represents the total number of scenarios. The state $(0, 0, \ldots, 0)$ refers to the idle state. The state is denoted as $S^r(t)$ if it is derived from the log repository, and as $S^a(t)$ for customer usage logs.

Table 6.7: Estimated Occurrence Probability and Failure Probability for Each System State

States S_i^r	**Occurrences**	**Failure Occurrences**	**Failure Probability** $p_f(S_i^r)$
(0, 0, 0, 0)	2000	0	0
(0, 0, 0, 1)	500	100	0.2
(0, 1, 0, 0)	250	0	0
(0, 1, 0, 1)	400	0	0
(1, 0, 0, 0)	40	20	0.5
(1, 1, 0, 0)	500	50	0.1
(1, 1, 0, 1)	250	0	0
(2, 2, 2, 0)	60	0	0

In our running example, there are four types of scenarios: Register, Browse, Purchase and Update. Therefore, the system state, $S^r(t)$, is a four-dimensional state vector (s_1, s_2, s_3, s_4).

Table 6.6 shows the derived system states from the recovered scenario instances. For example, at time 2, we have one register scenario and one browse scenario that are executing simultaneously (see Table 6.5). Therefore, the system state at time 2 is $S^r(2) = (1, 1, 0, 0)$. In addition, Table 6.6 also keeps track of the failure information at each time instance. As shown in the last column of this table, if there is at least one identified system failure at time t, the corresponding system state $S^r(t)$ is marked with an "x". If there is no failure at time t, the state is tagged with a "-". The states at time 6, 7 and 8 contain errors because of the SLA violation in sessions 1 and 2.

Step 4. Occurrence Probability and Failure Probability Calculations

Table 6.6 shows the system states derived from the first 14 log lines. In practice, the log repository would contain thousands or millions of log lines. The second column of table 6.7 shows the aggregated occurrences of all system states based on a large execution log repository.

As mentioned earlier, we need to calculate failure probabilities for the system states from the log repository and occurrence probabilities for the system states from the customer

usage logs. The failure probability for each state, $p_f(S_i^r)$ is calculated using the following formula:

$$p_f(S_i^r) = \frac{\#\ Failure\ Occurrences\ of\ S_i^r}{\#\ Occurrences\ of\ S_i^r} \tag{6.1}$$

For example, the failure probability for state $(1,1,0,0)$ is calculated as: $p_f((1,1,0,0)) = \frac{50}{500} = 0.1$. The 4th column of Table 6.7 shows the failure probabilities for each system state based on the execution log repository.

To calculate the occurrence probability of a system state, $p(S_i^a)$, from the usage data, we use the following formula:

$$p(S_i^a) = \frac{\#\ Occurrences\ of\ S_i^a\ in\ the\ usage\ data}{Total\ \#\ occurrences\ of\ all\ states\ in\ the\ usage\ data} \tag{6.2}$$

If we assume that the data from Table 6.7 comes from a customer usage log, the 2nd column of Table 6.7 would show the number of occurrences of each system state in the customer usage log. The total number of occurrences of all the system states in the customer usage logs is $4,000$ (sum of entries in second column). The occurrence probability of the idle state is then calculated as $p((0,0,0,0)) = \frac{2000}{4000} = 0.5$.

6.3.3 Deployment-Specific Reliability Estimation

In this section, we present our technique to provide a deployment-specific reliability estimate using Bayesian Networks. Our technique consists of the following three steps:

1. *System States Selection* - Identifying the system states that are common between the customer usage logs and the log repository;

Table 6.8: System States Derived from the Usage Data

System States	Occurrence Probability
(0, 0, 0, 0)	0.5
(0, 1, 0, 1)	0.25
(1, 1, 0, 0)	0.125
(2, 3, 2, 0)	0.125

2. *Test Coverage Calculation* - Calculating the test coverage of the log repository on the customer usage logs;

3. *Reliability Estimation* - Estimating the deployment-specific reliability using Bayesian Networks.

Step 1. System States Selection

The system states $S_i^{r=a}$ that are common between the log repository and the customer usage data are selected. Based on the occurrence probability in the real world (customer usage data) and the past failure probability (Log Repository) for these states, we can calculate the deployment-specific reliability estimates in the next step.

Table 6.8 shows the system states and their occurrence probabilities after analyzing usage data. After comparing the states from the log repository (Table 6.7) and the customer usage data (Table 6.8), there are three states in common: (0,0,0,0), (0,1,0,1) and (1,1,0,0).

Step 2. Test Coverage Calculation

For system states that are tested in the lab or in other field deployments, we know the likelihood of failure. For system states that do *not* show up in the repository, we have no prior knowledge about their failure probability. In order to obtain a lower bound estimate of the system reliability, we need to take into account for how many system states we have failure data, i.e., we need to measure test coverage.

The Test Coverage (TC) is calculated using the following formula:

$$TC(S) = \sum_{S_i^{r=a} \in S} p(S_i^{r=a}) \tag{6.3}$$

where $S_i^{r=a}$ are the common states between the log repository and the deployment logs; $p(S_i^{r=a})$ denotes the occurrence probability of state $S_i^{r=a}$ based on data from the customer usage data, and S is the set of covered system states. In our example, there are three states in common. Therefore, the test coverage is calculated as: $TC = 0.5 + 0.25 + 0.125 = 0.875$. The test coverage in the motivating example of Section 6.2 was 1.0, as all states that were observed in the customer usage data were also observed in the log repository.

Step 3. Reliability Estimation

Once the failure and occurrence probability for each state are calculated, we use Bayesian Networks to estimate the deployment-specific reliability $R(TC, S)$. Since we have no prior knowledge about the reliability of states that are not included in the log repository, we assume that all previously unseen system states derived from the customer usage data contain failures. Hence, the maximum possible reliability is $TC(S)$. This leads to a lower bound (worse case) estimate of the system reliability based on test coverage. $R(TC, S)$ is calculated as follows:

$$R(TC, S) = TC(S) - \sum_{S_i^{r=a} \in S} p(S_i^{r=a}) * p_f(S_i^{r=a}) \tag{6.4}$$

where $R(TC, S)$ denotes the estimated reliability given the field test coverage TC and the set of covered system states S; $p(S_i^{r=a})$ denotes the occurrence probability of state $S_i^{r=a}$

in the customer usage data; and $p_f(S_i^{r=a})$ denotes the failure probability of state $S_i^{r=a}$ based on data from the log repository.

In our running example, the deployment-specific reliability is calculated as: $0.875-(0.5*0+0.25*0+0.125*0.1) = 0.8625$. Thus, the estimated deployment reliability is 0.8625. Once the reliability is estimated, deployers can match it with their targeted reliability threshold and decide whether or not to upgrade this new version of the system.

6.4 Industrial Case Studies

In this section, we present two case studies to demonstrate our log-based reliability estimation approach. The first case study (Section 6.4.1) validates the correctness of our approach on a large mission critical system. The second case study (Section 6.4.2) demonstrate the usefulness of our approach by applying our reliability estimates for enhancing pre-release field testing.

Our case study system is a large mission-critical system. It is a telecommunication system that is responsible for processing thousands of simultaneous client requests and has very stringent reliability requirements. This system has been deployed into hundreds of customer sites (on-prem), which have different numbers of users, usage characteristics and reliability requirements.

6.4.1 Providing Deployment Specific Reliability Estimates

In this case study, we aim to validate our approach by comparing our estimated reliability against a five day (work-week) reliability that has been produced for a large mission-critical system.

For our case study, we used a repository with around 125 GB of log data. The repository contains logs derived from load and stress tests of the system and from other field

deployments of the system. For customer usage data, we use data from the User Acceptance Testing of two different deployments, which have different configuration and usage patterns. Both deployments were tested over five days using the same version of the system. One deployment generated 4 GB of logs while the other generated 23 GB of logs. We measure the reliability of the system for both deployments using:

- five days worth of logs (full User Acceptance Testing);
- our approach, which uses just one day worth of logs and estimates the reliability using the log repository.

We then compare both reliability estimates. Due to confidentiality, we cannot list the actual estimate values. However, we note that our estimation error relative to the five-day estimate is 2.5% for the first deployment and 3.6% for the second deployment. As explained in section 6.3, the error is an over-estimate, i.e., the estimate is a safe estimate to use when deciding to deploy since the system is likely to have a higher reliability than reported by our estimate.

After examining the system state of both deployments, we note that for both deployments a small portion of the system states (5% and 8.8%) covers the majority (90%) of the occurrence probability of system states from both deployments. These small portions of system states never fail. The system states that suffer from performance slow-down are very rare, leading to a high reliability. However, if the deployed system spends considerable time in failure-prone states, we expect that our approach will provide a better estimate than the estimates based on the full acceptance testing logs. The reason for this is that the log repository contains much more accurate failure probability information than the logs from even the full acceptance testing. We draw an analogy to coin flipping to make this more clear. The more we flip a coin, the better the empirically estimated probability that a coin falls on its head side. As the log repository contains weeks or even months of system behavior, the failure probability estimated based on the log repository will be closer to the real

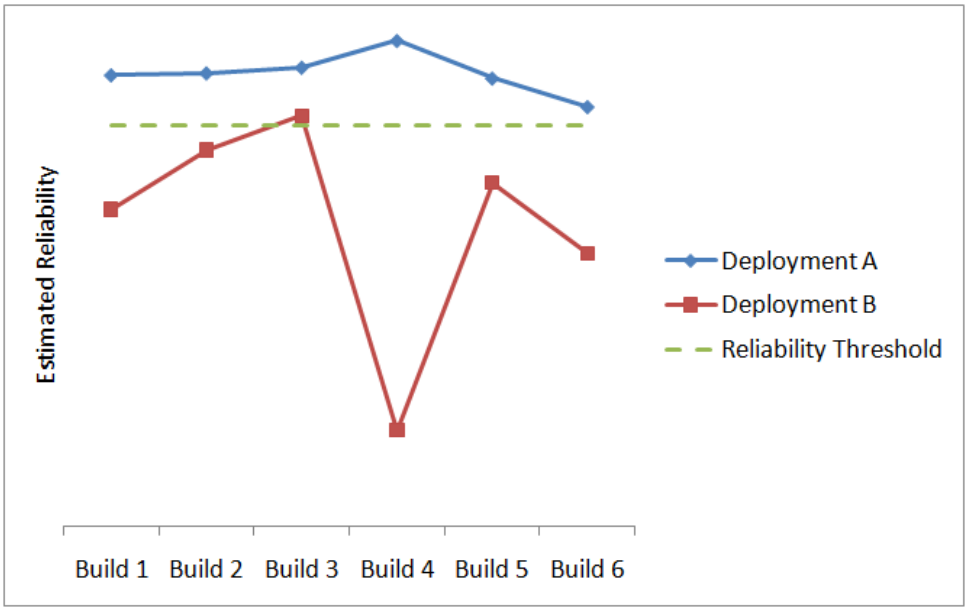


Figure 6.2: Estimated Reliability for six Different Software Builds

failure probability, and therefore providing a better reliability estimate in the long run.

In summary, our deployment-specific reliability estimates using just one day of User Acceptance Testing provide a relatively good estimate with little loss in accuracy, while providing substantial savings in consulting and testing time.

6.4.2 Speeding up Pre-release Field Testing

In this case study, we want to show that our reliability estimates can be used to speed up the development process by picking builds that are most suitable for limited test deployment prior to the release of an system (e.g., alpha testing). Alpha testing involves deploying the system in the field to explore its use by real users. Potential alpha candidate builds often support the main functionality, but might contain bugs in the system or miss certain features.

Figure 6.2 shows the reliability estimates for two internal deployments (A and B). The dashed line in the Figure shows the minimum acceptable reliability of the alpha build of

the system. For deployment A, any of the builds could be safely deployed, whereas for deployment B only the third build could be deployed safely. Using this information, the development team could start gathering user feedback much sooner by deploying the system at site A first. In addition, the team should have waited until the third build to deploy at site B and should have avoided upgrading from the currently installed version of the software in order to avoid frustrating the users due to a high chance of failure. By deploying the system earlier at site A and only at the appropriate time at site B, we are able to gain a better understanding of the reliability and usage characteristics of the system in a real-life setting. This information would help improve future builds and speed up the development process.

6.5 Threats to Validity

This section discusses various possible threats to the validity of our approach.

6.5.1 Construct Validity

6.5.1.1 Snapshot Interval

To accurately capture the change of system states, we need to pick a snapshot interval smaller than the shortest response time of any scenario in the large scale system used in our case study. Shorter intervals lead to more accurate system states. However, the length of the snapshot interval is limited by the logging interval.

In our industrial case study, the timestamp in the execution logs is accurate up to milliseconds. However, if we pick the snapshot interval to be every millisecond or every 10 milliseconds, the state derivation step will lead to a huge number of repeated states with no changes between them. For this reason, we picked a one-second interval. Our state derivation step and our reliability estimation finish within one hour on a Quad-core machine. We

cannot pick any interval longer than 1 second, as most of the scenarios in the system finish within 1 to 2 seconds.

6.5.1.2 System States

As there can be an infinite number of combinations of different workload requests, there can be an infinite number of system states. However, in our large industrial study, there are only a limited number of system states. Furthermore, the distribution of system states follows the Pareto-principle: a small percentage (5% and 8.8%) of states covers the majority (90%) of the system behavior. However, many of the states in the remaining 10% of the system behavior can be equivalent, so we plan in the future to apply fuzzy-clustering techniques on these states to group them and further reduce the number of system states.

In this chapter, we only consider the failure probability and occurrence probability of system states, since we believe that a system fails or slows down due to the current state (i.e., heavy workload). In the future, we plan to look into transitions between system states to see whether they might be the reason for system failure. In particular, a state might be a failure state just because of the previous states that eventually lead it to fail.

6.5.2 Internal Validity

6.5.2.1 Customer Usage Logs

In this chapter, we use previous user acceptance testing or past customer logs for our sources of customer usage data. However, using logs from different versions of the system brings many challenges to our analysis, in particular:

1. **Contingency of Workload**

   Our deployment-specific reliability estimates are based on the workload in the field deployment at a particular point in time. In reality, this workload might shift over

time leading to different reliability estimates. Our approach could be used to track such shifts in workload and warn about the impact of such a shift on the reliability of the system.

One method of checking if the system's behavior changes over time, is to check the distribution of system states over time. If the distribution of system states changes over time, then the system reliability will likely change as well. The distribution of system states remained stable in both of our case studies (the alpha testing and the User Acceptance Testing in two different sites).

2. **Performance improvement**

   The new version might run faster under the same workload than the older version due to architectural and design changes. Therefore, the resulting occurrence probability distribution will shift with the system spending more or less time in different states. In this case, we cannot directly match the distribution from the old logs with the failure information from the log repository for the new version. To overcome this concern, we can apply our performance analysis technique 5 to detect if there are any performance problems before we apply our reliability estimation.

3. **Addition of new features**

   Because there is no information about the usage information in the old system for new features, we are not able to match the system states from the old logs.

6.5.2.2 Limitation of Execution Logs

In the studied mission-critical system, there is a dedicated component that monitors the overall health. Thus, business level information as well as error messages are logged. Our current approach may not work if the studied system does not log all the necessary system information. In addition, we assume that all the errors are recorded in the logs, which is

usually true for large mission-critical systems. Last but not the last, not all errors reported in the logs are operational or performance-related. Errors from which the system recovers are not considered as failures. Examples of such errors are temporary communication failures that do not impact the overall system health or SLA. In this chapter, we manually verified all failures included in our analysis.

6.5.3 External Validity

In this chapter, we introduced a novel approach to estimate the reliability based on customer usage context. We evaluate our approach against a large mission-critical telecommunication system and showed that our estimate lies within 4% of the more traditional longer User Acceptance testing. To show the general applicability of our approach, we should evaluate it on other large mission-critical systems. However, such systems are usually developed by large commercial companies and their data is hard to obtain due to legal and confidentiality concerns.

6.6 Related Work

In this section, we discuss two areas of work related to our log-based empirical reliability estimation approach: approaches that estimate the system quality in the field and approaches that use log analysis.

Empirical Estimation of Software Availability and Reliability

Mockus [203] uses information from operational customer support systems to estimate the availability of a large telecommunication system. Information from customer support systems is more straight-forward to obtain than extracting failure information from the execution logs. However, the customer support systems may not contain all failure information,

as they miss externally unnoticeable problems and problems that are not reported by customers. Nagappan et al. [98, 208] provide a reliability estimate based on information from the static source code metrics and dynamic test coverage. Their approach is implemented as an Eclipse IDE plugin to provide rapid feedback for unit testing.

Avritzer et al. [51, 55, 57] have proposed several approaches for generating test suites based on the operational profile and for estimating the reliability of mission-critical systems. In [55], Avritzer et al. use Markov chains to generate load testing suites based on an operational profile. In [51, 57], Avritzer et al. introduce a transient analysis of the failure-based Markov chain to model reliability decay as a function of time. Our approach is similar to [51, 57], as we incorporate the occurrence probability distribution from the deployment and the failure probability from the repository. However, rather than using the failure information from vendors, we use the log framework and a large data set of field data to empirically derive failure and occurrence probability distributions. In addition, since we have the actual logs to estimate the occurrence probability of system states, we do not need Markov chains to model workload usage.

Log Analysis

In general, there are two sources of non-invasive data that can be used to understand, monitor, and analyze the various aspects of the system behavior: execution logs and performance logs.

Execution logs are generated by output statements that developers insert into the source code. Execution logs are widely available and are often used for remote issue resolution and for legal compliance purposes (e.g., "Sarbanes-Oxley Act of 2002" [23]). Aguilera et al. [42, 216] developed various algorithms to perform black-box performance debugging on distributed systems. They use the header information of the TCP packet traces (source,

destination and time) to infer the dominant causal paths through a distributed system. Unfortunately, the accuracy of the inferred causal paths decreases as the degree of parallelism increases. Marwede et al. [189] use timing anomalies to automatically uncover functional problems. Hassan et al. [139] propose a light-weight approach to extract customer operational profiles from the execution logs. Chapters 4 and 5 propose automated log analysis approaches to detect functional and performance problems in the load tests. The log analysis approach presented in this chapter is related to the approach presented in [53], because in both papers we abstract the log information into a set of different system states. However, in [53] the state definition used was the set of rules that were fired as a result of a change in object memory, while in this paper the system state is defined as a set of active scenarios present in the system at a particular moment in time.

Performance logs, which are generated by third party monitoring tools like PerfMon [21], record the system resource utilizations like CPU, memory and disk. Avritzer et al. [50, 49] propose algorithms to detect the need for software rejuvenation by monitoring the changing values of performance metrics. Mi et al. [89, 202] and Cohen et al. [90, 254] develop application signatures based on various system metrics (like CPU and memory usage). The application signatures are further used for efficient capacity planning and anomaly detection. The main difference between these approaches and ours is that we use execution logs for our analysis. Execution logs provide more in-depth domain-specific information.

6.7 Conclusion

Studies show that many field failures are due to load problems rather than functional problems. Software reliability is a useful quality index to determine the quality of a system under load. Large software systems are deployed to many customers, which could have different usage patterns. Therefore, one general reliability cannot accurately reflect the quality of all customer deployments. In this chapter, we propose an automated approach that estimates

the deployment specific reliability based on mining the large sets of execution logs. Different from previous two chapters, which focus on detecting load-related problems from a single load test, we use data from many load tests to estimate the overall quality of a system under load. Case studies show that our approach is accurate and enhances the existing continuous delivery process.

CHAPTER 7

Conclusions and Future Work

LARGE SOFTWARE SYSTEMS must be load tested to ensure they can support a large number of concurrent users. Analyzing a load test is challenging, because it is hard to build models from the large set of load testing data. Current industrial practice consists mainly of high-level manual checks. Such practice is not efficient, nor is it sufficient. Existing load testing research focuses on the test design and execution. There is very little research on analysis of the load testing results. In this dissertation, we propose automated approaches to assess various aspects (i.e., functional, performance and reliability) of the quality of a system under load by analyzing the recorded load testing data. Some of our research results (Chapters 3, 4 and 5) are already adopted in practice.

This chapter is organized as follows: First, we summarize our thesis findings and contributions (Section 7.1). Then, we present some future works (Section 7.2). Finally, we provide our closing remarks (Section 7.3).

7.1 Thesis Findings and Contributions

– **A Survey on Load Testing Large Scale Software Systems**

We have provided a taxonomy of load testing in terms of defining the phases of a load test (design, execution and analysis) as well as an in-depth survey of approaches inside each phases (e.g., realistic v.s. fault-inducing load design approaches in the load design phase).

– **Automated Abstraction of Execution Logs**

Execution logs are unstructured and have an infinite set of vocabulary. Since many statistical and artificial intelligence techniques operate on the structured data, there is a need to abstract execution logs into structure data to enable automated analysis. Existing log abstraction approaches, which work for generic log formats, either cannot scale to large log files [227] or cannot uniquely map one log line to one execution event [233]. Log lines generated by the same output statements will have identical static information and the same structure of dynamic information. Our approach is influenced by source code clone detection techniques to automatically recognize the static and dynamic parts of log lines. The resulting abstracted forms are called execution events. Case studies show that our approach can handle large log file sizes with satisfying results.

– **Automated Detection of Functional Problems**

We can detect functional problems of a system by analyzing its execution logs. As a load test repeatedly executes a set of scenarios and is conducted after the functional testing is complete, the execution of the same scenario should generate identical event sequences. Any variations of these sequences might indicate potential problems. We have proposed an approach that derives the pair-wise temporal relations out of abstracted execution logs. Unlike many temporal specification mining approaches

(e.g., [45, 118, 248]), which profile and analyze the systems at the method-invocation level, our analysis examines the execution logs to avoid big performance overhead during a load test. Compared to method-invocation level data, execution logs are hard to abstract and group into scenario sequences. Our approach automatically uncovers the dominant behavior and flags deviations from execution logs by analyzing event pairs. Case studies show that our approach detects various types of problems in the load testing environment, load generators, the system under test, and scales well to large enterprise systems.

– **Automated Detection of Performance Problems**

We can detect performance problems of a system by comparing the current test against other load tests. As similar loads are applied on load tests, performance data should be comparable across tests and informal performance baselines can be derived. We evaluate the performance of a system in terms of the end-user experience (response time). Response time throughout a test is not constant, as a typical workload usually consists of periods simulating peak usage and periods simulating off-hours usage. Therefore, we need to evaluate the end-user experience by examining the entire response time distribution instead of merely comparing the average response time. If the current run has scenarios that follow a different response time distribution than the baseline, this run is probably troublesome and worth investigating. We have presented an approach that automatically flags scenarios with response time problems by comparing the durations of execution sequences from the current test against prior test(s). We recover execution sequences from logs and calculate the duration of these sequences using the time stamps associated with each log line. Then, we compare the duration of various sequences using statistical techniques and visualize the problems in a report. Our approach not only reports scenarios with performance problems but also pin-points the steps with performance bottlenecks within these scenarios. Case

studies show that our approach produces few false alarms and scales well to large industrial systems. Unlike [90], our approach detects performance problems without specifications like SLAs (Service Level Agreements). In contrast to [89], our approach provides more context for developers to reproduce and diagnose problems.

– **Automated Estimation of System Reliability**

Nowadays, many software systems adopt the agile continuous delivery process. One of the challenges is to decide when the system is ready to release to the field. Current quality measurement in the agile process does not include any metrics related to the system performance under load. We have proposed to use software reliability as a useful quality index to summarize the quality of a system under load. We can estimate the software reliability by mining the execution log repository, which contains the in-house testing and customer usage data. Accurate reliability estimates can help to determine whether a system has a reliability problem (i.e., not meeting the target reliability requirement). The general reliability estimates are usually derived from synthetic benchmark workload runs. These benchmark workloads rarely match the actual field workload, leading to estimates that do not match the expected field reliability of the system. We have proposed an approach, which derives an empirically-validated and customized field reliability based on mining the execution log repositories. First, we recover the customer usage patterns ("system states") from the deployment load testing data. Then, we use logs generated from thousands of hours of execution (in-house testing and other customer deployment load testing) of the same version of the system to provide a good estimate of how often the identified system states fail ("failure probabilities"). By combining the customer usage and failure information of these system states, we can produce an accurate and customized reliability estimate.

7.2 Future Work

7.2.1 Enhancing Execution Logs for Effective Load Testing Analysis

Execution logs, which are generated from debug statements, are free-form, unstructured text. The anomalous behavior, obtained by mining large volumes of data, could potentially lead to load-related problems, but cannot provide more insight into the root cause of the problem (e.g., the input causing the problems). One possible avenue of future work is to explore the possibility of enriching the execution logs to ease the root cause analysis of a load test without imposing high overhead of the system. This log enhancement process could be an iterative process, as developers or load testing professionals actively investigates the problems.

7.2.2 Building An Efficient Repository for Load Testing

Like code and bug repositories, load testing repositories contain rich software historical data, which records the software behavior under load for different versions. In this thesis, we have demonstrated that effective and automated load testing analysis can be performed by mining such data. However, current state of storing load testing data (execution logs, metrics and analysis results from the load testing professionals) are ad-hoc. Building an efficient load testing repository, which provides fast search and retrieval of load testing data as well as linkage to other repositories (e.g., code, bug and mailing lists), could greatly enhance the automated load testing analysis and software quality as a whole.

7.2.3 Benchmarks for Evaluating Approaches Which Analyze the Results of Load Tests

The goal of load testing analysis is to detect various load-related problems. In this thesis, we use execution logs to detect functional, performance and reliability problems under

load. There are works proposed to detect load-related problems using metrics [115, 154, 156, 157, 185, 186, 187, 210]. As more and more load testing analysis techniques have been proposed, a benchmark, similar as the Siemens benchmark suite for functional bug detection [4], is needed to evaluate the effectiveness of various approaches to detect load-related problems.

7.2.4 Leveraging Big-Data Analysis Infrastructures for Analyzing the Results of Load Tests

As load tests generate large volume of data, the load test analysis techniques need to be scalable and efficient. However, as data grows larger (e.g., bigger than one machine's hard-drive to store), we might need to look into using the big-data analysis infrastructures (e.g., Hadoop) for analyzing the results of load tests. However, most of these infrastructures provide limited mechanisms to express the problems (e.g., only Map/Reduce for Hadoop). Existing analysis techniques may need to be adapted or even re-written to work on these infrastructures.

7.3 Closing Remarks

To ensure the quality of large scale systems, load testing is required in addition to conventional functional testing procedures. Load testing is becoming more important, as an increasing number of services are being offered in the cloud to millions of users. Load testing is a challenging area, in which industry has invested large amount of resources. Yet, there are few academic research efforts devoted to load testing. In this thesis, we have proposed automated approaches to assess the system quality under load by analyzing large sets of readily-available execution logs. We hope that this thesis will raise awareness and highlight the lack of research in this important and practical research topic.

Bibliography

[1] ACM Portal. http://portal.acm.org/, visited 2012-10-24. (221)

[2] Apache Custom Log Format. http://tinyurl.com/9aqstlq, visited 2012-10-24. (83)

[3] Apache JMeter. http://jakarta.apache.org/jmeter/, visited 2012-10-24. (48, 59)

[4] Aristotle Analysis System – Siemens Programs, HR Variants. http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/, visited 2012-10-24. (46, 184)

[5] CompleteSearch DBLP. http://www.dblp.org/search/index.php, visited 2010-10-24. (219)

[6] Dell DVD Store. http://linux.dell.com/dvdstore/, visited 2012-10-24. (120, 141)

[7] Firefox download stunt sets record for quickest meltdown. http://blogs.siliconvalley.com/gmsv/2008/06/firefox-download-stunt-sets-record-for-quickest-meltdown.html, visited 2008-08-24. (1, 11)

[8] Google Scholar. http://scholar.google.com/, visited 2012-10-24. (219, 221)

[9] Heikki Tuuri Innodb answers - Part I. http://www.mysqlperformanceblog.com/2007/10/26/heikki-tuuri-innodb-answers-part-i/, visited 2012-10-24. (145)

[10] HP LoadRunner. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451, visited 2012-10-24. (48, 49, 57, 58, 59, 60, 61, 65)

[11] iBATIS JPetStore. http://sourceforge.net/projects/ibatisjpetstore/, visited 2012-10-24. (123, 143)

[12] IEEE Explore. http://ieeexplore.ieee.org/Xplore/guesthome.jsp, visited 2012-10-24. (221)

[13] Implementing the Microsoft .NET Pet Shop using Java. http://www.clintonbegin.com/downloads/JPetStore-1-2-0.pdf, visited 2012-10-24. (123, 143)

[14] InnoDB vs MyISAM vs Falcon benchmarks. http://www.mysqlperformanceblog.com/2007/01/08/innodb-vs-myisam-vs-falcon-benchmarks-part-1/, visited 2012-10-24. (143)

[15] Jakarta Struts. http://struts.apache.org/, visited 2012-10-24. (123)

[16] Java Pet Store. http://java.sun.com/developer/releases/petstore/, visited 2009-06-13. (123)

[17] Microsoft Academic Search. http://academic.research.microsoft.com/, visited 2012-10-24. (219, 221)

[18] Microsoft Exchange Load Generator (LoadGen). http://www.microsoft.com/en-us/download/details.aspx?id=14060, visited 2012-10-24. (49, 57)

[19] Microsoft IIS W3C Extended Log File Format. http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/

676400bc-8969-4aa7-851a-9319490a9bbb.mspx?mfr=true, visited 2012-10-24. (83)

[20] MySQL Wins Prestigious International Database Contest. http://www.mysql.com/news-and-events/press-release/release_2006_35.html, visited 2009-06-13. (120)

[21] PerfMon Sample. http://msdn.microsoft.com/en-us/library/aa645516(VS.71).aspx, visited 2012-10-24. (3, 177)

[22] Petri Net Example. http://en.wikipedia.org/wiki/Petri_net, visited 2012-10-24. (xi, 36, 37)

[23] Sarbanes-Oxley Act of 2002. http://www.soxlaw.com/, visited 2012-10-24. (5, 66, 83, 112, 158, 176)

[24] Selenium - Web Browser Automation. http://seleniumhq.org/, visited 2012-10-24. (58)

[25] Shunra. http://www.shunra.com/, visited 2012-10-24. (60)

[26] SNMP Logs. http://www.ietf.org/rfc/rfc1157.txt, visited 2012-10-24. (83)

[27] Steve Jobs on MobileMe. http://arstechnica.com/journals/apple.ars/2008/08/05/steve-jobs-on-mobileme-the-full-e-mail, visited 2012-10-24. (1, 11)

[28] Supercomputer event logs. Available at: http://www.cs.sandia.gov/~jrstear/.logs-alpha1/, visited 2012-10-24. (98)

[29] Teiresias. http://cbcsrv.watson.ibm.com/Tspd.html, visited 2012-10-24. (99)

[30] The R Project for Statistical Computing. http://www.r-project.org/, visited 2012-10-24. (141)

[31] uTest - Load Testing Services. http://www.utest.com/load-testing. (16, 47, 56, 65, 67)

[32] WebLoad. http://www.webload.org/, visited 2012-10-24. (111, 123, 144)

[33] WebLOAD product overview. http://www.radview.com/DownloadCenter/productdemo.aspx, visited 2012-10-24. (48, 49, 57, 58, 61, 65)

[34] Willem Visser's Research. http://www.visserhome.com/willem/, visited 2009-11-14. (108)

[35] Wireshark - Go Deep. http://www.wireshark.org/, visited 2012-10-24. (58)

[36] IEEE standard definitions for use in reporting electric generating unit reliability, availability, and productivity. *ANSI/IEEE Std 762-1987*, 1987. (153)

[37] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990. (153)

[38] Applied Performance Management Survey, Oct 2007. (1, 11, 153)

[39] S. Abu-Nimeh, S. Nair, and M. Marchetti. Avoiding denial of service via stress testing. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, AICCSA '06, pages 300–307, Washington, DC, USA, 2006. IEEE Computer Society. (16, 25, 27)

[40] 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 '09, pages 409–420, Washington, DC, USA, 2009. IEEE Computer Society. (16, 18, 80)

[41] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth. Automated gui performance testing. *Software Quality Control*, 19(4):801–839, December 2011. (17)

[42] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 74–89, New York, NY, USA, 2003. ACM. (150, 176)

[43] M. Andreolini, M. Colajanni, and P. Valente. Design and testing of scalable web-based systems with performance constraints. In *Proceedings of the 2005 Workshop on Techniques, Methodologies and Tools for Performance Evaluation of Complex Systems*, FIRB-PERF '05, pages 15–25, Washington, DC, USA, 2005. IEEE Computer Society. (25, 27, 72)

[44] J. Andrews. Testing using log file analysis: Tools, methods, and issues. In *Proceedings of the 13th IEEE international conference on Automated software engineering*, ASE '98, pages 157–166, Washington, DC, USA, 1998. IEEE Computer Society. (109)

[45] B. Anton, M. Leonardo, and P. Fabrizio. Ava: Automated interpretation of dynamically detected anomalies. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 237–248, New York, NY, USA, 2009. ACM. (181)

[46] C. AtLee, L. Blakk, J. O'Duinn, and A. Z. Gasparian. Firefox release engineering. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications, Volume II: Structure, Scale and a Few Fearless Hacks*. CreativeCommons, 2012. (152)

[47] M. Audris, P. Zhang, and P. L. Li. Predictors of customer perceived software quality. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 225–233, New York, NY, USA, 2005. ACM. (153)

[48] A. Avritzer and B. Avritzer. Load testing software using deterministic state testing. In *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing*

and analysis, ISSTA '93, pages 82–88, New York, NY, USA, 1993. ACM. (16, 40, 44, 46, 129, 149, 220, 222)

[49] A. Avritzer, A. Bondi, M. Grottke, K. S. Grottke, and E. J. Weyuker. Performance assurance via software rejuvenation: Monitoring, statistics and algorithms. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 435–444, Washington, DC, USA, 2006. IEEE Computer Society. (150, 177)

[50] A. Avritzer, A. Bondi, and E. J. Weyuker. Ensuring stable performance for systems that degrade. In *Proceedings of the 5th international workshop on Software and performance*, WOSP '05, pages 43–51, New York, NY, USA, 2005. ACM. (150, 177)

[51] A. Avritzer, F. P. Duarte, a. Rosa Maria Meri Le E. de Souza e Silva, M. Cohen, and D. Costello. Reliability estimation for large distributed software systems. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, CASCON '08, pages 157–165, New York, NY, USA, 2008. ACM. (68, 69, 74, 176, 221)

[52] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker. Software performance testing based on workload characterization. In *Proceedings of the 3rd international workshop on Software and performance*, WOSP '02, pages 17–24, New York, NY, USA, 2002. ACM. (16, 72)

[53] A. Avritzer, J. P. Ros, and E. J. Weyuker. Reliability testing of rule-based systems. *IEEE Software*, 13(5):76–82, 1996. (14, 16, 68, 69, 177)

[54] A. Avritzer and E. J. Weyuker. Generating test suites for software load testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 44–57, New York, NY, USA, 1994. ACM. (16, 40, 44, 46, 149)

[55] 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*, 21(9):705–716, Sept. 1995. (16, 17, 24, 40, 44, 46, 74, 129, 149, 153, 176, 220, 221, 222)

[56] A. Avritzer and E. J. Weyuker. Deriving workloads for performance testing. *Software - Practice & Experience*, 26(6):613–633, June 1996. (16)

[57] A. Avritzer and E. J. Weyuker. The automated generation of test cases using an extended domain based reliability model. In *Proceedings of the ICSE Workshop on Automation of Software Test, 2009 (AST 2009)*, pages 44–52, 2009. (155, 176)

[58] D. Bainbridge, I. H. Witten, S. Boddie, and J. Thompson. *Research and Advanced Technology for Digital Libraries*, chapter Stress-Testing General Purpose Digital Library Software, pages 203–214. Springer, 2009. (16, 53)

[59] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. (158)

[60] S. Barber. Creating effective load models for performance testing with incomplete empirical data. In *Proceedings of the Web Site Evolution, Sixth IEEE International Workshop*, WSE '04, pages 51–59, Washington, DC, USA, 2004. IEEE Computer Society. (16, 25, 28, 53)

[61] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 259–272, Berkeley, CA, USA, 2004. USENIX Association. (130)

[62] C. Barna, M. Litoiu, and H. Ghanbari. Autonomic load-testing framework. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 91–100, New York, NY, USA, 2011. ACM. (16, 17, 62, 63)

[63] C. Barna, M. Litoiu, and H. Ghanbari. Model-based performance testing (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 872–875, New York, NY, USA, 2011. ACM. (16, 17, 62, 63)

[64] 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 '07, pages 612–617, Washington, DC, USA, 2007. IEEE Computer Society. (25, 30, 31, 54, 61)

[65] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, Washington, DC, USA, 1998. IEEE Computer Society. (90)

[66] M. S. Bayan and J. W. Cangussu. Automatic stress and load testing for embedded systems. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 229–233, 2006. (15, 62, 63, 71, 75, 149, 222)

[67] M. S. Bayan and J. W. Cangussu. Automatic feedback, control-based, stress and load testing. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC)*, pages 661–666, 2008. (14, 15, 62, 63, 222)

[68] T. Bear. Shootout: Load Runner vs The Grinder vs Apache JMeter, June 2006. http://blackanvil.blogspot.com/2006/06/shootout-load-runner-vs-grinder-vs.html, visited 2012-10-24. (48)

[69] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, March 1984. (17)

[70] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, March 1984. (108, 129)

[71] A. Bertolino, G. Angelis, A. Marco, P. Inverardi, A. Sabetta, and M. Tivoli. A framework for analyzing and testing the performance of software services. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 206–220. Springer Berlin Heidelberg, 2009. (55)

[72] A. Bertolino, G. D. Angelis, and A. Sabetta. Vcr: Virtual capture and replay for performance testing. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 399–402, Washington, DC, USA, 2008. IEEE Computer Society. (65, 67, 68)

[73] L. Bertolotti and M. C. Calzarossa. Models of mail server workloads. *Performance Evaluation*, 46(2-3):65–76, 2001. (153)

[74] A. B. Bondi. Automating the analysis of load test results to assess the scalability and stability of a component. In *Proceedings of the 2007 Computer Management Group Conference (CMG)*, pages 133–146, 2007. (16, 25, 27, 71, 72, 75)

[75] S. Boslaugh and D. P. A. Watters. *Statistics in a Nutshell A Desktop Quick Reference*. O'Reilly, 2008. (148)

[76] E. Bozdag, A. Mesbah, and A. van Deursen. Performance testing of data delivery techniques for ajax applications. *Journal of Web Engineering*, 8(4):287–315, 2009. (16)

[77] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1021–1028, New York, NY, USA, 2005. ACM. (16)

[78] L. C. Briand, Y. Labiche, and M. Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2):145–170, 2006. (149)

[79] L. Bulej, T. Kalibera, and P. Tma. Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 60(1-4):345–358, 2005. (16, 76, 148, 149)

[80] Y. Cai, J. Grundy, and J. Hosking. Experiences integrating and scaling a performance test bed generator with an open source case tool. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE '04, pages 36–45, Washington, DC, USA, 2004. IEEE Computer Society. (33, 59)

[81] Y. Cai, J. Grundy, and J. Hosking. Synthesizing client load models for performance engineering via web crawling. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 353–362, New York, NY, USA, 2007. ACM. (33, 59)

[82] M. Calzarossa and G. Serazzi. Workload characterization: a survey. *Proceedings of the IEEE*, 81(8):1136–1150, Aug 1993. (24)

[83] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1069–1075, New York, NY, USA, 2005. ACM. (38)

[84] J. W. Cangussu, K. Cooper, and W. E. Wong. Reducing the number of test cases for performance evaluation of components. In *Proceedings of the Nineteenth International*

Conference on Software Engineering & Knowledge Engineering (SEKE), pages 145–150, 2007. (16, 17)

[85] J. W. Cangussu, K. Cooper, and W. E. Wong. A segment based approach for the reduction of the number of test cases for performance evaluation of components. *International Journal of Software Engineering and Knowledge Engineering*, 19(4):481–505, 2009. (16, 17)

[86] G. Casale, A. Kalbasi, D. Krishnamurthy, and J. Rolia. Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, pages 1–20, New York, NY, USA, 2009. Springer-Verlag New York, Inc. (16, 40, 42)

[87] A. Chakravarty. Stress testing an ai based web service: A case study. In *Seventh International Conference on Information Technology: New Generations (ITNG 2010)*, pages 1004–1008, Los Alamitos, CA, USA, April 2010. IEEE Computer Society. (16, 18, 25, 27)

[88] M. Y. Chen, A. Accardi, E. Kiciman, J. Kiciman, D. Kiciman, A. Fox, and E. Brewer. Path-based faliure and evolution management. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 23–23, Berkeley, CA, USA, 2004. USENIX Association. (130)

[89] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *IEEE International Conference on Dependable Systems and Networks*, pages 452–461. IEEE Computer Society, 2008. (150, 177, 182)

[90] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the twentieth*

ACM symposium on Operating systems principles, SOSP '05, pages 105–118, New York, NY, USA, 2005. ACM. (80, 150, 177, 182)

[91] J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 196–205, Washington, DC, USA, 2003. IEEE Computer Society. (90)

[92] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore. Investigation of failure causes in workload-driven reliability testing. In *Fourth international workshop on Software quality assurance (SOQUA)*, pages 78–85, 2007. (127, 128)

[93] C. Csallner and Y. Smaragdakis. Dsd-crasher: a hybrid analysis tool for bug finding. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 245–254, New York, NY, USA, 2006. ACM. (127)

[94] P. Csurgay and M. Malek. Performance testing at early design phases. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, pages 317–330, 1999. (16, 17)

[95] M. B. da Silveira, E. de M. Rodrigues, A. F. Zorzo, L. T. Costa, H. V. Vieira, and F. M. de Oliveira. Reusing functional testing in order to decrease performance and stress testing costs. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE 2011),*, pages 470–474, 2011. (25, 29, 59)

[96] R. Dale, H. L. Somers, and H. Moisl, editors. *Handbook of Natural Language Processing*. Marcel Dekker, Inc., New York, NY, USA, 2000. (85)

[97] C. V. Damásio, P. Fröhlich, W. Nejdl, L. M. Pereira, and M. Schroeder. Using extended logic programming for alarm-correlation in cellular phone networks. *Applied Intelligence*, 17(2):187–202, 2002. (85, 86)

[98] M. Davidsson, J. Zheng, N. Nagappan, L. Williams, and M. Vouk. Gert: An empirical reliability estimation and testing feedback tool. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 269–280, 2004. (176)

[99] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005. (85)

[100] I. de Sousa Santos, A. R. Santos, and P. de Alcantara dos S. Neto. Generation of scripts for performance testing based on uml models. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE 2011),*, pages 258–263, 2011. (25, 29, 58)

[101] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. In *Proceedings of the 4th international workshop on Software and performance*, WOSP '04, pages 94–103, New York, NY, USA, 2004. ACM. (16, 17)

[102] G. Denaro, A. Polini, and W. Emmerich. *Performance Testing of Distributed Component Architectures. In Building Quality into COTS Components: Testing and Debugging*, chapter Performance Testing of Distributed Component Architectures. Springer-Verlag, 2005. (16, 17)

[103] B. Dillenseger. Clif, a framework based on fractal for flexible, distributed load testing. *Annals of Telecommunications*, 64:101–120, 2009. (15, 16, 18)

[104] G. Din, I. Schieferdecker, and R. Petre. Performance test design process and its implementation patterns for multi-services systems. In *Proceedings of the 20th IFIP TC*

6/WG 6.1 international conference on Testing of Software and Communicating Systems (TestCom/FATES), pages 135–152, 2008. (71, 72, 73)

[105] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of web applications. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 57–70, Washington, DC, USA, 2006. IEEE Computer Society. (xi, 25, 32, 33)

[106] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, pages 109–118, Washington, DC, USA, 1999. IEEE Computer Society. (91)

[107] C. Dumitrescu, I. Raicu, M. Ripeanu, and I. Foster. Diperf: An automated distributed performance testing framework. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID)*, pages 289–296, 2004. (49, 57, 61, 71, 73)

[108] S. Dunning and D. Sawyer. A little language for rapidly constructing automated performance tests. In *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ICPE '11, pages 371–380, New York, NY, USA, 2011. ACM. (57)

[109] S. Duttagupta and M. Nambiar. Performance extrapolation for load testing results of mixture of applications. In *2011 Fifth UKSim European Symposium on Computer Modeling and Simulation (EMS)*, pages 424–429, November 2011. (71, 74)

[110] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007. (152)

[111] S. Elnaffar and P. Martin. Characterizing computer systems' workloads. Technical report, Queen's University, 2002. (24)

[112] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM. (117, 127)

[113] W. W. Everett, J. D. Musa, A. F. Ackerman, and G. A. Wilson. *Software Reliability Engineering*. John Wiley & Sons Inc., 2002. (153)

[114] B. L. Farrell, R. Menninger, and S. G. Strickland. Performance testing & analysis of distributed client/server database systems. In *Proceedings of the 1998 Computer Management Group Conference (CMG)*, pages 910–921, 1998. (25, 27, 72)

[115] K. C. Foo, Z. M. Jiang, B. Adams, Y. Z. Ahmed E. Hassan, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 32–41, Washington, DC, USA, 2010. IEEE Computer Society. (5, 71, 78, 79, 80, 83, 184)

[116] T. A. S. Foundation. Log4j. http://logging.apache.org/log4j/1.2/, visited 2012-10-24. (65, 66)

[117] E. M. Friedman and J. L. Rosenberg. Web load testing made easy: Testing with wcat and wast for windows applications. In *Proceedings of the 2003 Computer Management Group Conference (CMG)*, pages 57–82, 2003. (69, 71, 72, 73)

[118] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM. (181)

[119] V. Garousi. Empirical analysis of a genetic algorithm-based stress test technique. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 1743–1750, New York, NY, USA, 2008. ACM. (16, 18)

[120] V. Garousi. A genetic algorithm-based stress test requirements generator tool and its empirical evaluation. *Transactions on Softare Engineering*, 36(6):778–797, Nov. 2010. (16, 18)

[121] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed systems based on uml models. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 391–400, New York, NY, USA, 2006. ACM. (16, 18, 149)

[122] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed real-time systems based on uml models using genetic algorithms. *Journal of Systems and Software*, 81(2):161–185, 2008. (16, 18)

[123] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. 42(10):57–76, 2007. (76)

[124] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. (5)

[125] G. Gheorghiu. More on performance vs. load testing, April 2005. http://agiletesting.blogspot.com/2005/04/more-on-performance-vs-load-testing.html, visited 2012-10-24. (16)

[126] G. Gheorghiu. Performance vs. load vs. stress testing, February 2005. http://agiletesting.blogspot.com/2005/02/performance-vs-load-vs-stress-testing.html, visited 2012-10-24. (14, 16)

[127] A. L. Glaser. Load testing in an ir organization: Getting by 'with a little help from my friends'. In *Proceedings of the 1999 Computer Management Group Conference (CMG)*, pages 686–698, 1999. (71, 73)

[128] I. Gorton. *Essential Software Architecture*. Springer, 2000. (17)

[129] M. Grechanik, C. Csallner, C. Fu, and Q. Xie. Is data privacy always good for software testing? In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE),*, ISSRE '10, pages 368–377, Washington, DC, USA, 2010. IEEE Computer Society. (54)

[130] D. Grossman, M. C. McCabe, C. Staton, B. Bailey, O. Frieder, and D. C. Roberts. Performance testing a large finance application. *IEEE Software*, 13(5):50–54, September 1996. (71, 73)

[131] C. D. Grosso, G. Antoniol, M. D. Penta, P. Galinier, and E. Merlo. Improving network applications security: a new heuristic to generate stress testing data. In *Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO)*, pages 1037–1043, New York, NY, USA, 2005. ACM. (16)

[132] T. O. Group. Application Response Measurement - ARM. http://www.opengroup.org/tech/management/arm/, visited 2012-10-24. (66)

[133] Y. Gu and Y. Ge. Search-based performance testing of applications with composite services. In *Proceedings of the 2009 International Conference on Web Information Systems and Mining (WISM)*, pages 320–324, 2009. (25, 38)

[134] M. Gupta and M. Subramanian. Preprocessor algorithm for network management codebook. In *ID'99: Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring*, pages 93–102, Berkeley, CA, USA, 1999. USENIX Association. (85, 86)

[135] D. W. Gurer, I. Khan, R. Ogier, and et al. An artificial intelligence approach to network fault management. In *SRI International*, 1996. (86)

[136] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, ICSE 2012, pages 145–155, Piscataway, NJ, USA, 2012. IEEE Press. (80)

[137] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM. (127)

[138] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. In *LISA '93: Proceedings of the 7th USENIX conference on System administration*, pages 145–152, Berkeley, CA, USA, 1993. USENIX Association. (85, 86)

[139] A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield, and D. Dietz. An industrial case study of customizing operational profiles using log compression. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 713–723, New York, NY, USA, 2008. ACM. (177)

[140] R. Hayes. How to load test e-commerce applications. In *Proceedings of the 2000 Computer Management Group Conference (CMG)*, pages 275–282, 2000. (25, 27)

[141] J. Hill, D. Schmidt, J. Edmondson, and A. Gokhale. Tools for continuously evaluating distributed system qualities. *IEEE Software*, 27(4):65–71, July 2010. (16, 50, 55, 66, 71, 73)

[142] J. H. Hill. An architecture independent approach to emulating computation intensive workload for early integration testing of enterprise dre systems. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems (OTM)*, pages 744–759, Berlin, Heidelberg, 2009. Springer-Verlag. (16, 50, 55, 71, 73)

[143] J. H. Hill, D. C. Schmidt, A. A. Porter, and J. M. Slaby. Cicuts: Combining system execution modeling tools with continuous integration environments. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS '08, pages 66–75, Washington, DC, USA, 2008. IEEE Computer Society. (16, 50, 55, 71, 73)

[144] J. H. Hill, S. Tambe, and A. Gokhale. Model-driven engineering for development-time qos validation of component-based software systems. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 307–316, Washington, DC, USA, 2007. IEEE Computer Society. (16, 50, 55, 71, 73)

[145] 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)*, pages 79–88, Washington, DC, USA, 2007. IEEE Computer Society. (14)

[146] 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 ’08, pages 135–144, Washington, DC, USA, 2008. IEEE Computer Society. (14)

[147] D. S. Hoskins, C. J. Colbourn, and D. C. Montgomery. Software performance testing using covering arrays: efficient screening designs with categorical factors. In *Proceedings of the 5th international workshop on Software and performance (WOSP)*, pages 131–136, 2005. (16, 17)

[148] J. Huard. Probabilistic reasoning for fault management on xunet. Technical report, AT&T Bell Labs, 1994. (86)

[149] J. Huard and A. Lazar. Fault isolation based on decision-theoretic troubleshooting. Technical report, Center for Telecommunications Research, Columbia University, 1996. (86)

[150] F. Huebner, K. S. Meier-Hellstern, and P. Reeser. Performance testing for ip services and systems. In *Performance Engineering, State of the Art and Current Trends*, pages 283–299, 2001. (16, 17, 18, 71, 73)

[151] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, 1991. (24, 138)

[152] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward. Detection and diagnosis of recurrent faults in software systems by invariant analysis. In *Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium (HASE)*, pages 323–332, 2008. (80)

[153] Z. M. Jiang. A Survey on Load Testing Large Software Systems (Depth Report), August 2012. (3, 5)

[154] Z. M. Jiang, A. E. Hassa, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *Journal Software Maintenance Evolution*, 20:249–267, July 2008. (71, 75, 184)

[155] Z. M. Jiang and A. E. Hassan. A framework for studying clones in large software systems. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 203–212, Washington, DC, USA, 2007. IEEE Computer Society. (91)

[156] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, pages 307–316. IEEE Computer Society, 2008. (184, 220)

[157] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, pages 125–134. IEEE Computer Society, 2009. (75, 184, 220, 221)

[158] M. J. Johnson, C.-W. Ho, M. E. Maximilien, and L. Williams. Incorporating performance testing in test-driven development. *IEEE Software*, 24:67–73, May 2007. (16, 17)

[159] J.-N. Juang. *Applied System Identification*. Prentice Hall, first edition, 1993. (62)

[160] T. Kalibera, L. Bulej, and P. Tuma. Automated detection of performance regressions: the mono experience. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 183–190, 27-29 2005. (16)

[161] M. Kalita and T. Bezboruah. Investigation on performance testing and evaluation of prewebd: a .net technique for implementing web application. *IET Software*, 5(4):357 –365, August 2011. (16, 17, 18)

[162] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002. (91)

[163] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, 2008. (134)

[164] K. Kant, V. Tewary, and R. Iyer. An internet traffic generator for server architecture evaluation. In *Proceedings of Workshop Computer Architecture Evaluation Using Commercial Workloads*, pages 117–134. Springer-Verlag, 2001. (25, 30, 31)

[165] C. Kapser and M. W. Godfrey. Improved tool support for the investigation of duplication in software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 305–314, Washington, DC, USA, 2005. IEEE Computer Society. (91)

[166] A. F. Karr and A. A. Porter. Distributed performance testing using statistical modeling. In *Proceedings of the 1st international workshop on Advances in model-based testing (A-MOST)*, pages 1–7, 2005. (16)

[167] G.-B. Kim. A method of generating massive virtual clients and model-based performance test. In *Proceedings of the Fifth International Conference on Quality Software*, QSIC '05, pages 250–254, Washington, DC, USA, 2005. IEEE Computer Society. (60)

[168] S. Klinger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *Proceedings of the fourth international symposium on Integrated network management IV*, pages 266–277, London, UK, UK, 1995. Chapman & Hall, Ltd. (85, 86)

[169] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 44–54, Washington, DC, USA, 1997. IEEE Computer Society. (90)

[170] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 295–315, Berlin, Heidelberg, 2003. Springer-Verlag. (116)

[171] D. Krishnamurthy, J. Rolia, and S. Majumdar. Swat: A tool for stress testing session-based web applications. In *Computer Measurement Group Conference*, 2003. (16, 41, 42, 61)

[172] D. Krishnamurthy, J. Rolia, and S. Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering*, 32(11):868–882, 2006. (xi, 16, 22, 40, 42, 43, 61, 67)

[173] G. M. Leganza. Coping with stress tests: Managing the application benchmark. In *Proceedings of the 1990 Computer Management Group Conference (CMG)*, pages 1018–1026, 1990. (65)

[174] G. M. Leganza. The stress test tutorial. In *Proceedings of the 1991 Computer Management Group Conference (CMG)*, pages 994–1004, 1991. (40, 44, 47, 53, 56)

[175] W. Ley and U. Ellerman. http://www.cert.dfn.de/eng/logsurf/, visited 2008-12-03. (85, 86)

[176] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 306–315, New York, NY, USA, 2005. ACM. (127)

[177] B. Lim, J. Kim, and K. Shim. Hierarchical load testing architecture using large scale virtual clients. pages 581–584, 2006. (16, 18, 60, 72)

[178] A. Lin. A hybrid approach to fault diagnosis in network and system management. Technical report, HP Technical Report, 1998. (86)

[179] C. Liu, X. Yan, and J. Han. Mining control flow abnormality for logic error isolation. In *SDM*, pages 106–117, 2006. (127, 128)

[180] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Analysis of the linux kernel evolution using code clone coverage. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 22–, Washington, DC, USA, 2007. IEEE Computer Society. (91)

[181] L. Ljung. *System Identification: Theory for the user.* Prentice Hall, 1987. (62)

[182] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. (158)

[183] C. Lutteroth and G. Weber. Modeling a realistic workload for performance testing. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC)*, pages 149–158, 2008. (25, 32)

[184] M. M. Maccabee and S. Ma. Web application performance: Realistic work load for stress test. In *Proceedings of the 2002 Computer Management Group Conference (CMG)*, pages 353–362, 2002. (40, 41)

[185] H. Malik, B. Adams, and A. E. Hassan. Pinpointing the subsystems responsible for the performance deviations in a load test. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 201–210, Washington, DC, USA, 2010. IEEE Computer Society. (5, 71, 79, 80, 83, 184)

[186] H. Malik, Bram, Adams, A. E. Hassan, P. Flora, and G. Hamann. Using load tests to automatically compare the subsystems of a large enterprise system. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference*, COMPSAC '10, pages 117–126, Washington, DC, USA, 2010. IEEE Computer Society. (5, 71, 79, 80, 83, 184)

[187] H. Malik, Z. M. Jiang, B. Adams, P. Flora, and G. Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 222–231, Washington, DC, USA, 2010. IEEE Computer Society. (5, 71, 79, 80, 83, 184)

[188] R. K. Mansharamani, A. Khanapurkar, B. Mathew, and R. Subramanyan. Performance testing: Far from steady state. In *IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW 2010)*, pages 341 –346, July 2010. (60, 61, 62, 71, 75)

[189] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society. (150, 177)

[190] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea. Performance Testing Guidance for Web Applications - patterns & practices, September 1997. http://msdn.microsoft.com/en-us/library/bb924375.aspx, visited 2012-10-24. (14, 16, 65)

[191] D. A. Menasce. Load testing, benchmarking, and application performance management for the web. In *Proceedings of the 2002 Computer Management Group Conference (CMG)*, pages 271–281, 2002. (xi, 15, 16, 19, 30, 31, 40, 44, 65, 71, 73, 74, 221, 222)

[192] D. A. Menasce. Load testing of web sites. *IEEE Internet Computing*, 6(4):70–74, July 2002. (15, 16, 25, 30, 31, 40, 44, 71, 74, 221)

[193] D. A. Menasce. Workload characterization. *IEEE Internet Computing*, 7(5):89–92, 2003. (53)

[194] D. A. Menasce and V. A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. (18)

[195] D. A. Menasce, V. A. Almeida, and L. W. Dowd. *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. (18)

[196] D. A. Menasce, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of e-commerce sites. In *Proceedings of the 1st ACM conference on Electronic commerce (EC)*, pages 119–128, 1999. (16, 65)

[197] D. A. Menasce and A. F. A. Virgilio. *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000. (16, 18)

[198] X. Meng. Designing approach analysis on small-scale software performance testing tools. In *2011 International Conference on Electronic and Mechanical Engineering and Information Technology (EMEIT)*, pages 4254–4257, August 2011. (57)

[199] J. K. Merton. Performance testing in a client-server environment. In *Proceedings of the 1997 Computer Management Group Conference (CMG)*, pages 594–601, 1997. (72)

[200] J. K. Merton. Evolution of performance testing in a distributed client server environment. In *Proceedings of the 1994 Computer Management Group Conference (CMG)*, pages 118–124, 1999. (25, 27)

[201] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in multi-tier applications: symptoms, causes, and new models. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 265–286, New York, NY, USA, 2008. Springer-Verlag New York, Inc. (41, 42)

[202] N. Mi, L. Cherkasova, K. M. Ozonat, J. Symons, and E. Smirni. Analysis of application performance and its change via representative application signatures. In *Network Operations and Management Symposium*, pages 216–223, 2008. (150, 177)

[203] A. Mockus. Empirical estimates of software availability of deployed systems. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 222–231, New York, NY, USA, 2006. ACM. (175)

[204] M. Murth, D. Winkler, S. Biffl, E. Kuhn, and T. Moser. Performance testing of semantic publish/subscribe systems. In *Proceedings of the 2010 international conference on On*

the move to meaningful internet systems, OTM'10, pages 45–46, Berlin, Heidelberg, 2010. Springer-Verlag. (49)

[205] J. D. Musa, A. Iannino, and K. Okumoto. *Software reliability: Measurement, prediction, application*. McGraw-Hill, 1987. (153)

[206] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association. (16, 19, 55, 63)

[207] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI)*, pages 187–197, 2010. (64, 76)

[208] N. Nagappan, L. Williams, and M. Vouk. "good enough" software reliability estimation plug-in for eclipse. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, eclipse '03, pages 30–34, New York, NY, USA, 2003. ACM. (176)

[209] M. A. S. Netto, S. Menon, H. V. Vieira, L. T. Costa, F. M. de Oliveira, R. Saad, and A. F. Zorzo:. Evaluating load generation in virtualized environments for software performance testing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 993–1000, May 2011. (59)

[210] T. H. D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora. Automated verification of load tests using control charts. In *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, APSEC '11, pages 282–289, Washington, DC, USA, 2011. IEEE Computer Society. (xi, 5, 71, 77, 78, 80, 83, 184)

[211] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on*

Dependable Systems and Networks, pages 575–584, Washington, DC, USA, 2007. IEEE Computer Society. (98)

[212] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. (2)

[213] M. D. Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno. Search-based testing of service level agreements. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO)*, pages 1090–1097, New York, NY, USA, 2007. ACM. (25, 38)

[214] A. Podelko. Load Testing Tools. http://alexanderpodelko.com/PerfTesting.html#LoadTestingTools, visited 2012-10-24. (48)

[215] B. A. Pozin and I. V. Galakhov. Models in performance testing. *Programming and Computer Software*, 37:15–25, 2011. 10.1134/S036176881101004X. (16, 17, 18)

[216] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 347–356, New York, NY, USA, 2006. ACM. (150, 176)

[217] M. E. Y. Y. S. A. Yemini, S. Sliger and D. Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, pages 82–90, 1996. (85, 86)

[218] A. Savoia. Web load test planning: Predicting how your web site will respond to stress. *STQE Magazine*, 2001. (25, 28, 53)

[219] I. Schieferdecker, G. Din, and D. Apostolidis. Distributed functional and load tests for web services. *International Journal on Software Tools for Technology Transfer (STTT)*, 7:351–360, 2005. (16, 57)

[220] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 335–344, Washington, DC, USA, 2011. IEEE Computer Society. (9)

[221] J. W. Sheppard and W. R. Simpson. Improving the accuracy of diagnostics provided by fault dictionaries. In *VTS '96: Proceedings of the 14th IEEE VLSI Test Symposium (VTS '96)*, pages 180–185, Washington, DC, USA, 1996. IEEE Computer Society. (86)

[222] S. Shirodkar and V. Apte. Autoperf: an automated load generator and performance measurement tool for multi-tier software systems. In *Proceedings of the 16th international conference on World Wide Web (WWW)*, pages 1291–1292, 2007. (59)

[223] N. Snellman, A. Ashraf, and I. Porres. Towards automatic performance and scalability testing of rich internet applications in the cloud. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011)*, pages 161–169, September 2011. (25, 27, 61, 62)

[224] M. Sopitkamol and D. A. Menascé. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *Proceedings of the 5th international workshop on Software and performance (WOSP)*, pages 53–64, 2005. (16, 17)

[225] N. Stankovic. Distributed tool for performance testing. In *Software Engineering Research and Practice*, pages 38–44, 2006. (58)

[226] N. Stankovic. Patterns and tools for performance testing. In *2006 IEEE International Conference on Electro/information Technology*, pages 152–157, 2006. (58, 61)

[227] J. Stearley. Towards informatic analysis of syslogs. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 309–318, Washington, DC, USA, 2004. IEEE Computer Society. (86, 99, 180)

[228] S.Weaver. *The mathematical theory of communication*. Urbana: University of Illinois Press, 1949. (103)

[229] M. D. Syer, B. Adams, and A. E. Hassan. Identifying performance deviations in thread pools. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 83–92, Washington, DC, USA, 2011. IEEE Computer Society. (xi, 71, 76, 77, 221)

[230] M. D. Syer, B. Adams, and A. E. Hassan. Industrial case study on supporting the comprehension of system behaviour under load. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 215–216, Washington, DC, USA, 2011. IEEE Computer Society. (71, 76)

[231] D. Thakkar, Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Retrieving relevant reports from a customer engagement repository. In *In Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, pages 117–126. IEEE Computer Society, 2008. (9)

[232] P. Tran, J. Gosper, and I. Gorton. Evaluating the sustained performance of cots-based messaging systems. *Software Testing, Verification and Reliability*, 13(4):229–240, 2003. (71, 72)

[233] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 2003 IEEE Workshop on IP Operations and Management*, pages 119–126, 2003. (86, 99, 180)

[234] R. Vaarandi. Simple event correlator for real-time security log monitoring. *Hakin9 Magazine*, 6(1):28–39, 2006. (85, 86)

[235] C. Vail. Stress, load, volume, performance, benchmark and base line testing tool evaluation and comparison, 2005. http://www.vcaa.com/tools/loadtesttoolevaluationchart-023.pdf, visited 2012-10-24. (48)

[236] W. Visser. Who really cares if the program crashes? In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 5–5, Berlin, Heidelberg, 2009. Springer-Verlag. (81)

[237] J. Voas. Will the real operational profile please stand up? *IEEE Software*, 17(2):87–89, Mar. 2000. (153)

[238] F. I. Vokolos and E. J. Weyuker. Performance testing of software systems. In *Proceedings of the 1st international workshop on Software and performance*, WOSP '98, pages 80–87, New York, NY, USA, 1998. ACM. (16)

[239] X. Wang, B. Zhou, and W. Li. Model based load testing of web applications. In *2010 International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 483–490. IEEE Computer Society, September 2010. (25, 29)

[240] Y. Wang, X. Wu, and Y. Zheng. *Trust and Privacy in Digital Business*, chapter Efficient Evaluation of Multifactor Dependent System Performance Using Fractional Factorial Design, pages 142–151. Springer, 2004. (54)

[241] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476, Berlin, Heidelberg, 2005. Springer-Verlag. (127, 128)

[242] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, Dec. 2000. (1, 3, 11, 16, 153)

[243] J. White and A. Pilbeam. A survey of virtualization technologies with performance testing. *CoRR*, abs/1010.3233, 2010. (59)

[244] H. Wietgrefe and K. Tochs. Using neural networks for alarm correlation in cellular phone networks. In *International Workshop on Applications of Neural Networks in Telecommunications*, pages 248–255, 1997. (86)

[245] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society. (16, 65)

[246] J. Xie, X. Ye, B. Li, and F. Xie. A configurable web service performance testing framework. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 312–319, 2008. (49)

[247] C.-S. D. Yang and L. L. Pollock. Towards a structural load testing tool. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '96, pages 201–208, New York, NY, USA, 1996. ACM. (15, 16, 25, 34, 221)

[248] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM. (127, 128, 181)

[249] X. Yang, X. Li, Y. Ji, and M. Sha. Crownbench: a grid performance testing system using customizable synthetic workload. In *Proceedings of the 10th Asia-Pacific web*

conference on Progress in WWW research and development (APWeb), pages 190–201, 2008. (71, 72)

[250] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 143–154, New York, NY, USA, 2010. ACM. (5)

[251] J. Zhang and S. C. Cheung. Automated test case generation for the stress testing of multimedia systems. *Software - Practice & Experience*, 32(15):1411–1435, December 2002. (16, 18, 25, 36, 71, 73, 149)

[252] J. Zhang, S.-C. Cheung, and S. T. Chanson. Stress testing of distributed multimedia software systems. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, FORTE XII / PSTV XIX '99, pages 119–133, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V. (16, 25, 36)

[253] P. Zhang, S. G. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society. (16, 17, 25, 34)

[254] S. Zhang, I. Cohen, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 644–653, Washington, DC, USA, 2005. IEEE Computer Society. (150, 177)

APPENDIX A

Our Paper Selection Process

Our paper selection process is illustrated in Figure A.1, which consists of the following three steps:

1. **Picking the most suitable search engine(s):** There are three types of search engines available: General research article search engines (e.g., Microsoft Academic Search [17], and Google Scholar [8]), specific organizational search engine (e.g., search engine from ACM Portal and IEEE Explore), and online indexing database search engines (e.g. CiteSeer and DBLP searches [5]).

   We have decided against specific organizational paper repositories like ACM Portal and IEEE Explore due to concerns on the search efforts and search quality:

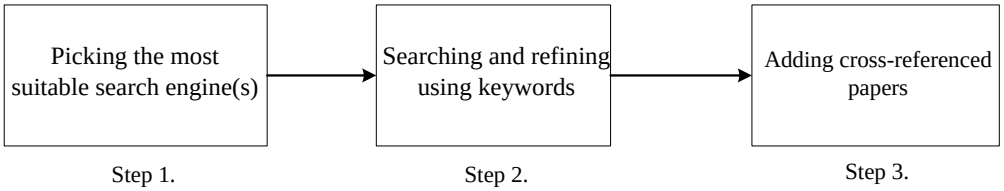


Figure A.1: Our Paper Selection Process

First, some papers appear in only one organization's repository or all of them. For example, papers like [55] appear in both IEEE Explore and ACM Portal, but some other papers like [156, 157] only appear in the IEEE Explore repository and papers like [48] only appears in the ACM repositories. Furthermore, these organizational search results are more focused on the search quantity rather than search quality. A single search on the term "load testing" from the IEEE Explore returns $17,504$ records as of June 27th, 2010; yet many of these entries are false positives, such as "Power minimization technique for induction motor load test". Although the IEEE portal provides search refinement by Authors, by Affiliations, by Publication Title and by Subject, these lists are not exhaustive. For example, the refinement by Publication title tab only lists the 25 top journal titles with the terms "load test" and misses the most relevant venues like "IEEE Transactions on Software Engineering", which contains [55]. ACM Portal suffers from similar search quality and search refinement problems. Finally, the terms "load test" or "load tests" are not considered the same by the ACM or IEEE search engines, since the search results for "load tests" will not contain papers with "load test" in them.

Therefore, we conduct our survey mainly based on the results of the General scholarly article search engines, which provide a more user friendly and higher quality search results. For example, a search with the terms like "load test" in DBLP would return not only all the papers whose titles contain "load test", but other terms like "load testing" or "test ... load".

2. **Keyword-based search and refinement:** Our initial search with the term "load test" returns 100 papers in DBLP on March 28, 2012. We have filtered 74 irrelevant papers based on the paper titles, publication venues and abstracts. For example, results like "Test front loading in early stages of automotive software development based on AUTOSAR" are filtered out.

After reading through some papers, we realized that other terms are also used interchangeably with "load testing". In particular, the terms "performance testing" (e.g. [192, 191]) and "stress testing" (e.g. [247]) are used as synonyms for the term load testing. Therefore, we expand our search keywords into: "load test", "performance test", and "stress test".

Unfortunately, there is no standard definition of load test, which all papers agree on. Each interpretation has its limitations and/or vagueness. Furthermore, not all performance tests and stress test papers are related to load testing. Therefore, we have provided our unified definition of load testing (see Section 2.2), which combines all the different load testing interpretations in a concise view. Based on our interpretations, we first removed performance and stress testing papers that are not related to software (e.g., "Backdrive Stress-Testing of CMOS Gate Array Circuits"). Then, we filter out performance and stress testing papers unrelated to our interpretations of load testing.

Table A.1 shows the number of papers obtained from the above three keywords before and after filtering. Some of these papers can appear in more than one search result. For example, [157] appears both in "load test" and "performance test".

3. **Adding cross-referenced papers and tools:** DBLP only searches through the paper titles, but not the actual paper contents or tools for the above three keywords. Therefore, we add more papers based on the related work sections from relevant load testing papers. Finally, we also include relevant papers that cite these papers, based on the "Cited by" feature from Microsoft Academic Search [17], Google Scholar [8], ACM Portal [1] and IEEE Explore [12]. For example, papers like [51, 229] is included, because they cite [55] and [157], respectively.

In the end, we have surveyed a total of 120 papers and tools between the year 1993 – 2011 as shown in Table A.1. To verify the completeness of the surveyed papers, the final

Table A.1: Number of Papers Found at Each Step

Steps	**Number of Papers and Tools**		
Keyword-based Search and Refinement	**Keyword**	**Initial**	**Refined**
	Load test	100	26
	Performance test	669	60
	Stress test	139	14
Adding Cross-Referenced Papers and Tools	120		

results include all the papers we knew beforehand to be related to load testing [48, 55, 66, 67, 191].