Looking at Execution Logs Beyond Execution Events

Enriching Execution Events to Compare the Behaviour of Large-Scale Software Systems Against Their Historical Behaviour

by

Mark D. Syer

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
October 2016

Abstract

Failures in large-scale software systems are often associated with performance issues. Therefore, performance testing has become essential to ensure the problem-free operation of these systems. Performance analysts must understand how their system behaves during performance testing and how such behaviour differs from its historical behaviour (i.e., the system's behaviour during a previous performance test or in the field). However, documentation describing the historical behaviour of a system is rarely up-to-date. Fortunately, execution logs, which record notable events at runtime, are readily available in most large-scale software systems to support remote monitoring, issue resolution and legal compliance. However, execution logs are typically treated as a sequence of events without fully leveraging additional sources of valuable information associated with such events (e.g., the dynamic information within the execution logs and performance counters that are collected during the system's execution). Therefore, in this thesis, we propose approaches

for enriching execution events with additional sources of valuable information to compare a system's current behaviour against its historical behaviour. Through case studies on large-scale software systems, including open-source and enterprise systems, we show that our approaches are scalable and can help performance analysts to 1) detect and diagnose performance regressions and 2) improve their performance tests to be more representative of the field.

Statement of Originality

I, Mark D. Syer, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Co-Authorship

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

- *Enriching the Execution Events by Leveraging the Time Stamps and Worker IDs in the Execution Logs (Chapter 3)*

  Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M. and Flora, P. (2014), Continuous Validation of Load Test Suites, in "Proceedings of the International Conference on Performance Engineering", pp. 232-241.

  Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E. (2016), "Continuous Validation of Performance Test Workloads", *Automated Software Engineering*, *to appear*.

- *Enriching the Execution Events by Combining Execution Logs and Performance Counters (Chapter 5)*

  Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M. and Flora, P. (2013), Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues, in "Proceedings of the International Conference on Software Maintenance", pp. 110-119.

Dedication

To my wife, Dr. Stefanie Syer. This thesis would not have been possible without your continuous love, support, encouragement and occasional (very) late nights.

Acknowledgments

I am deeply grateful to my Ph.D supervisor, Dr. Ahmed E. Hassan, for his continuous guidance and support throughout my Ph.D. Ahmed's mentorship has had an immense impact on my professional development and personal growth. I look forward to our continued friendship and future opportunities.

I would like to thank my Ph.D supervisory and examination committee members, Dr. Dorothea Blostein, Dr. Robin Dawes, Dr. Troy Day, Dr. Juergen Dingel, Dr. Nicholas Graham, Dr. Hossan Hassanein, Dr. Scott Yam and Dr. Andy Zaidman, for their valuable feedback at each stage of my Ph.D.

I consider myself very lucky to have worked with such an amazing group of researchers in the Software Analysis and Intelligence Lab (SAIL) at Queen's University. In particular, Dr. Zhen Ming Jiang, Dr. Weiyi Shang and Dr. Tse-Hsun Chen have provided me with valuable advice on both research and life.

I would like to thank Queen's University and the School of Computing for providing an unparalleled environment to work and study and for their generous support.

I would like to thank the Performance Engineering team at BlackBerry for their generous support. Working with this team as an embedded researcher has given me a deep appreciation for the current challenges facing performance analysts in practice. In particular, Parminder Flora, Mohamed Nasser, Denny Chiu, Terry Green and Gilbert Hamann have provided valuable feedback on my work during our biweekly research meetings.

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for their generous support.

Finally, I would like to express my deepest thanks to my friends and family, especially my parents Frank and Laura Syer, for their support, encouragement and understanding while I completed my thesis.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements (Woodside et al., 2007). These performance requirements describe the responsiveness (i.e., the ability to respond to user requests in a timely manner) and scalability (i.e., the ability to retain responsiveness despite increasing user demand) of a software system (Smith and Williams, 2002). Software performance engineering is particularly relevant to today's large-scale software systems (e.g., Amazon.com and Google's GMail). These systems are deployed across thousands of machines, require near-perfect up-time and support millions of concurrent connections and operations. Failures in such systems are more often associated with performance issues, than with feature bugs

(Compuware, 2006; Dean and Barroso, 2013; Simic and Conklin, 2012; Weyuker and Vokolos, 2000). These performance issues have led to several high-profile failures, including:

- Scalability issues during 1) Apple's release of MobileMe (Cheng, 2008), 2) Mozilla's release of Firefox 3.0 (Murrell, 2008) and 3) the United States government's release of healthcare.gov (Bataille, 2013).
- Costly performance (i.e., concurrency) defects during the NASDAQ's initial public offering of Facebook's shares (Benoit, 2012).
- Service outages lasting several hours and affecting tens of millions of users at Amazon Web Services (Williams, 2012) and Facebook (Johnson, 2010).

Such failures have significant financial and reputational repercussions (Ausick, 2012; Coleman Parkes, 2011; Harris, 2011; Jr. and Dinan, 2014).

Performance testing is the primary software performance engineering approach that is used to prevent these failures (Woodside et al., 2007). Performance analysts conduct performance tests that study the behaviour of a system under various workloads. Such tests are designed to satisfy a particular test objective. For example, tests may be designed to determine the maximum operating capacity of a system, validate non-functional performance requirements or uncover bottlenecks in the system.

The primary goal of performance testing is to understand how a system behaves during performance testing and how such behaviour differs from its historical behaviour (i.e., the system's behaviour during a previous performance test or the system's behaviour in the field). Understanding how a system's behaviour differs from

its historical behaviour provides performance analysts with valuable insights into 1) whether there are any performance regressions (i.e., whether the system's performance has degraded after a change to its source code or configuration) and 2) whether the performance test is representative of the field (i.e., whether the system's behaviour during performance testing is similar to the system's behaviour in the field). However, the system's behaviour is based on the behaviour of thousands or millions of users interacting with the system and continuously evolves as 1) the user base changes, 2) features are added, modified or removed and 3) user feature preferences change (Abad et al., 2012; Bertolotti and Calzarossa, 2001; Gill et al., 2007; Kavulya et al., 2010; Voas, 2000). Further, documentation describing the expected behaviour of a system is rarely up-to-date (Ernst et al., 2015; Holvitie et al., 2014; Parnas, 1994). Therefore, performance analysts increasingly need support to enable performance testing within their cost and time constraints (Barber, 2004).

The system's behaviour during performance testing or in the field can be described through execution logs. Execution logs record notable events at runtime. They are generated by output statements that developers manually insert into the source code of the system. These output statements are triggered by specific events (e.g., starting, queuing or completing a job) and errors within the system. Execution logs are used by developers (to debug the system) and operators (to monitor the operation of the system) (Cinque et al., 2013; Fu et al., 2014; Shang et al., 2011, 2014; Yuan et al., 2012). Therefore, execution logs are readily available in most large-scale systems to support remote monitoring, issue resolution and legal compliance. However, execution logs are not typically designed for automated analysis (Jiang et al., 2008a). Each occurrence of an execution event results in a

slightly different log line because each log line contains static components as well as dynamic information (which may be different for each occurrence of a particular execution event) (Bac; Pecchia et al., 2015). For example, `Processing item 1 / 1` and `Processing item 1 / 1,000` contain static components (`Processing item ___ / ___`) that represent the execution event and dynamic information (`1 / 1` and `1 / 1,000`) that represent additional information about the execution event. However, little work has been done to fully leverage the information in the execution logs because prior work tends to view execution logs as only as execution events (i.e., prior work discards the dynamic information in the execution logs). Further, little work has been done to enrich the execution events by combining them with additional sources of valuable information such as performance counters.

Therefore, in this thesis, we survey the state-of-the-art of performance testing large-scale software systems with a focus on how execution logs are used to 1) characterize a system's behaviour and 2) compare a system's behaviour to its historical behaviour. We found that prior efforts only leverage the execution events in the execution logs (i.e., the static components of each log line). Hence, we present novel approaches for enriching the execution events with additional sources of valuable information to characterize and compare the system's current behaviour to its historical behaviour (i.e., the system's behaviour during a previous performance test or the system's behaviour in the field).

We provide an overview of our 1) literature survey, 2) research hypothesis and 3) thesis contributions below.

1.1 Literature Survey

We surveyed the state-of-the-art of performance testing of large-scale software systems with a focus on how execution logs are used to 1) characterize a system's behaviour and 2) compare a system's behaviour to its historical behaviour.

1.1.1 Characterizing a System's Behaviour

The purpose of characterizing a system's behaviour (i.e., workload characterization or operational profiling) is to build a "model" of the system's behaviour (Jain, 1991; Menascé, 2003). These models vary considerably in how they model the system's behaviour. Hence, we classify these models into two groups: 1) models of aggregate user behaviour and 2) models of individual user behaviour.

Current approaches for characterizing a system's behaviour do not fully leverage the execution logs because the dynamic information in the logs is discarded when the logs are abstracted to events. Therefore, novel approaches are needed for characterizing a system's behaviour by fully leveraging all the information that is available in the execution logs as well as additional sources of information about the system's behaviour (e.g., performance counters).

1.1.2 Comparing a System's Behaviour

The purpose of comparing a system's behaviour to its historical behaviour is to 1) identify anomalous system behaviour or 2) detect performance regressions. These comparisons are performed by analyzing either 1) the execution logs or 2) the performance counters.

Current approaches for comparing a system's behaviour to its historical behaviour do not fully leverage the execution logs because the dynamic information in the logs is discarded when the logs are abstracted to static execution events. Further, these approaches do not simultaneously consider multiple sources of information (i.e., they consider either the execution logs *or* the performance counters, but *not both*). A survey of performance analysts notes that combining multiple source of information (e.g., execution logs and performance counters) is a major challenge (Simic and Conklin, 2012). Therefore, novel approaches are needed for comparing a system's behaviour to its historical behaviour that enrich the execution events with additional sources of valuable information (i.e., performance counters) and fully leverage the information in the execution logs (i.e., *both* the static components of the execution logs *and* dynamic information in the execution logs).

1.2 Research Hypothesis

Motivated by our literature survey, prior research and experience, we propose the following research hypothesis.

While there has been much work on characterizing a system's behaviour and comparing the system's current behaviour to its historical behaviour, prior work does not fully leverage the execution logs nor does it enrich such logs with additional sources of valuable information. Systematic approaches to comprehensively characterize a system's behaviour and compare this richer characterization of a system's current behaviour to its historical behaviour are needed to provide performance analysts with a deeper understanding of the behaviour of their system.

1.3 Thesis Contributions

The main contribution of our thesis is to present novel approaches to fully leverage the information in the execution logs and enrich the execution events with additional sources of valuable information. Our approaches 1) *characterize* a system's behaviour using execution events that have been enriched with additional sources of valuable information and then 2) *compare* this richer characterization of a system's current behaviour to its historical behaviour to identify the most significant differences. Our approaches leverage multiple sources of information (i.e., the static components *and* the dynamic information in the execution logs as well as the performance counters) to enrich the execution events and to provide performance analysts with a deeper understanding of their system's behaviour. Through case studies on large-scale software systems, including open-source and enterprise systems, we show that our approaches are scalable and can help performance analysts to understand the differences between their system's behaviour and its historical behaviour. Knowledge of such differences should help performance analysts to 1) detect and diagnose performance regressions and 2) improve their performance tests to be more representative of the field.

1.4 Organization of the Thesis

This thesis is organized as follows. Chapter 2 provides a literature survey of relevant research for 1) characterizing a system's behaviour and the 2) comparing a system's behaviour to its historical behaviour. Chapter 3 presents our first approach for enriching the execution events by leveraging *both* the static components of the

execution logs *and* the time stamps and worker IDs in the execution logs. Chapter 4 presents our approach for enriching the execution events by leveraging *both* the static components of the execution logs *and* the dynamic information in the execution logs. Chapter 5 presents our approach for enriching the execution events with information from the performance counters that are collected during the system's execution. Finally, Chapter 6 concludes the thesis and presents promising opportunities for future work.

CHAPTER 2

Literature Survey

Failures in today's large-scale software systems are typically associated with performance issues, rather than with feature bugs (Compuware, 2006; Dean and Barroso, 2013; Simic and Conklin, 2012; Weyuker and Vokolos, 2000). Therefore, performance testing has become essential to the problem-free operation of these systems. However, performance issues are often difficult to detect, diagnose and fix (Zaman et al., 2011, 2012b).

To address the challenges facing performance analysts, we propose novel approaches for enriching execution events with additional sources of valuable information to compare a system's current behaviour against its historical behaviour (i.e., the system's behaviour during a previous performance test or the system's behaviour in the field). Our approaches first characterize the system's behaviour using

execution events that have been enriched with additional sources of valuable information and then compare the system's current behaviour to its historical behaviour. Therefore, this literature survey focuses on two areas: 1) characterizing the system's behaviour and 2) comparing the system's current behaviour to its historical behaviour.

Additional information about our Literature Survey is available in the appendices of this thesis. Appendix A presents the criteria that we used to conduct our literature survey. Appendix B presents a brief summary of each paper that we included in our literature survey.

2.1 Characterizing a System's Behaviour

The primary goal of performance testing is to understand how a system behaves during performance testing and how such behaviour differs from its historical behaviour. To satisfy this goal, performance analysts design a test workload to exercise their system and then monitor their system's behaviour under the workload. Test workloads are usually derived from an existing benchmark or by creating a new benchmark by characterizing a previous workload (i.e., alpha or beta testing data or actual production data).

2.1.1 Benchmarks

Performance analysts have proposed several benchmark workloads that are designed to represent how a system will be used in the field. Such benchmarks have been proposed for many different types of systems including 1) cloud-based data

storage systems (Ren et al., 2012, 2014), 2) cloud-based transaction processing systems (Abad et al., 2012; Chen et al., 2012; Cooper et al., 2010; Saletore et al., 2013), 3) data centres (Jia et al., 2013), 4) databases (Buda, 2013; Chays et al., 2000; Dayarathna and Suzumura, 2014) 5) email server systems (Bertolotti and Calzarossa, 2001), 6) multimedia streaming systems (Costa et al., 2004; Yu et al., 1992), 7) regular expression matching systems (Becchi et al., 2008) and 8) web sites (Amza et al., 2002; Nagpurkar et al., 2008; Poggi et al., 2010; Sarhan et al., 2008; Stewart et al., 2008; Xi et al., 2011). Such benchmarks most often use execution logs to determine the number of executed events or event sequences per unit of time. For example, a benchmark for a cloud-based data storage systems may specify the number of executed `file download` requests per second (i.e., an event per second benchmark) or the number of executed `log in` $\rightarrow$ `download file` $\rightarrow$ `log out` request sequences per second (i.e., an event sequence per second benchmark). However, such benchmarks may not reflect how users perceive the system's performance (Dilley, 1996; Zaman et al., 2012a). Further, benchmarks rarely capture the full variability of the system's behaviour (Voas, 2000) and it is unlikely that up-to-date benchmarks for every type of system could be developed and maintained by performance analysts. Finally, existing benchmarks may not generalize (e.g., from one e-commerce system to another). Therefore, benchmarks should be validated by comparing the system's current behaviour to its historical behaviour under the same benchmark workload.

Several tools have also been proposed to allow performance analysts to manually create benchmarks for 1) cloud storage systems (Zheng et al., 2012, 2013), 2) network intrusion detection systems (Antonatos et al., 2004) and 3) peer-to-peer

systems (de Almeida et al., 2010). However, manually created benchmarks may not be representative of the field because performance analysts' intuition about the system's expected usage is often inaccurate (Chen et al., 2012) and field workloads often cannot be represented by well-defined distributions (Xi et al., 2011). Yet, specifying an up-to-date benchmark is important to derive a good understanding of the system's behaviour (Dean and Barroso, 2013; Morin et al., 2005; Stets et al., 2002; Zhang et al., 2013). Therefore, performance analysts should create benchmarks by characterizing the system's behaviour rather than using manually created benchmarks.

2.1.2 Workload Characterization

The purpose of workload characterization (also known as operational profiling) is to build a "model" of the system's behaviour (Jain, 1991; Menascé, 2003). These models are then used to generate a workload during performance testing. Performance tests based on workload characterization provide more insight into the system's behaviour than performance tests based on manually created benchmarks because generating a workload from a model built by characterizing the system's historical behaviour is more representative of the system's historical behaviour than a manually created benchmark (Avritzer et al., 2002; Barros et al., 2007). Therefore, performance tests based on workload characterization are more likely to help performance analysts to detect and diagnose functional and performance issues in their systems (Avritzer et al., 2002; Barros et al., 2007).

Workload characterization models are built using event traces. Event traces and execution logs both record notable events at runtime. However, event traces are

generated by automated instrumentation tools (e.g., JProfiler (JProfiler, 2016) or the JVM Tool Interface (JVM Tool Interface, 2013)) that view the system as a black-box whereas execution logs are generated by output statements that developers manually insert into the system's source code (Shang et al., 2011, 2014). Therefore, event traces record implementation-level events (e.g., `authenticate_login() called`) whereas execution logs tend to record domain-level events (e.g., `login successful`) (Jiang et al., 2008a). Further, automated instrumentation is often not feasible as it imposes a heavy overhead on the instrumented system (Bernat and Miller, 2011; Laurenzano et al., 2015; Maplesden et al., 2015; Uh et al., 2006). Hence, execution logs are often the most sensible data available to characterize a system's workload.

Workload characterization approaches that use execution logs have two steps (see Section 2.2.1 for more details). First, the execution logs are abstracted to execution events by removing implementation and instance-specific details (i.e., dynamic information) from the execution logs. Second, the execution events are represented by a higher-level model (e.g., a finite-state machine). Workload characterization models vary considerably in how they model the system's behaviour. Therefore, we classify workload characterization models into two groups: 1) models of aggregate user behaviour and 2) models of individual user behaviour.

2.1.2.1 Aggregate User Behaviour Models

Aggregate user behaviour models measure the entire workload (i.e., the total number of requests, session or events) across all users. Delimitrou et al. (2011, 2012) characterize the workload in terms of I/O (e.g., average read or write size). Their

models can vary in granularity between characterizing the workload per-thread to characterizing the workload per-data centre. Goseva-Popstojanova et al. (2006), Kurz et al. (2005) and Tian et al. (2004) characterize the workload in terms of the average user behaviour (e.g., session length, number of requests per session and number of sessions per day or hour). Jia et al. (2014) characterize the workload in terms of the microarchitecture behaviour (e.g., percentage of load or store instructions or percentage cache hits). Menascé (2003) characterize the workload in terms of the average number of events (e.g., the number of purchases on an e-commerce website) over time and the different type of sessions (e.g., sessions where users browse a catalogue and sessions where users purchase an item).

Aggregate user behaviour models (by design) fail to capture the individual user behaviour. Yet, the system's users may exhibit considerably variability in their behaviour. Further, a high-level characterization of the system's workload may not reflect how users perceive the system's performance (Dilley, 1996; Zaman et al., 2012a).

2.1.2.2 Individual User Behaviour Models

Individual user behaviour models measure the workload generated by a single user (i.e., how a single user interacts with the system). These models can be used to simulate the behaviour of a single user and a workload is created by simultaneously simulating thousands or millions of users. State-based models are the most common approach for modelling individual user behaviour (Avritzer and Larson, 1993; Avritzer and Weyuker, 1994, 1995; Ballocca et al., 2002; Barros et al., 2007; Cai et al., 2007; Costa et al., 2004; Draheim et al., 2006; Krishnamurthy et al., 2006;

Luo et al., 2009; Menascé et al., 1999). These models are composed of 1) states and 2) transitions (the probability of moving from one state to another). For example, in a state-based model of a web site, each web page would be a state and moving from one page to another would be a transition. Transition probabilities would be prespecified by a performance analyst based on domain knowledge or mined from the system's execution logs or performance counters.

Individual user behaviour models (by design) fail to capture the aggregate user behaviour. Yet, the aggregate user behaviour has an impact on the system's performance because responding to each user's requests requires resources (e.g., CPU, memory and bandwidth). Further, individual user behaviour models do not scale well (Hall, 2008).

A special class of individual user behaviour models are replay scripts. Replay scripts record the behaviour of real users in the field then play back the recorded behaviour during testing. In theory, replay scripts can be used to accurately replicate the conditions in the field (Krishnamurthy et al., 2006). However, replay scripts require complex software to concurrently simulate the millions of users and billions of requests recorded in the field. Therefore, replay scripts are less scalable than other models (e.g., state-based models) of individual user behaviour (Meira et al., 2012).

Current approaches for characterizing a system's behaviour do not fully leverage the execution logs. In particular, the dynamic information in the logs is discarded when the logs are abstracted to events. We believe that the rich information available in execution logs should be fully leveraged.

2.2 Comparing a System's Behaviour

The second area of related research is comparing the system's current behaviour to its historical behaviour. The system's behaviour may be compared to its historical behaviour by analyzing *either* the execution logs *or* the performance counters (combining these two sources of information is a major challenge for performance analysts (Simic and Conklin, 2012)). Such work attempts to address issues such as identifying anomalous system behaviour and detecting performance regressions by comparing the system's current behaviour to its historical behaviour (i.e., the system's behaviour during a previous performance test or the system's behaviour in the field). Table 2.1 provides an overview of the literature on comparing a system's current behaviour to its historical behaviour.

2.2.1 Execution Log Analysis

Execution logs record notable events at runtime. They are used by developers (to debug the system) and operators (to monitor the operation of the system) (Cinque et al., 2013; Fu et al., 2014; Shang et al., 2011, 2014; Yuan et al., 2012). Execution logs are generated by output statements that developers manually insert into the source code of the system. These output statements are triggered by specific events (e.g., starting, queuing or completing a job) and errors within the system. Execution logs are readily available in most large-scale systems to support remote monitoring, issue resolution and legal compliance whereas performance counters require explicit monitoring tools (e.g., PerfMon (PerfMon, 2016) and Pidstat (SYSSTAT, 2016)) to be collected.

Table 2.1: Summary of the Literature on Comparing a System's Current Behaviour to its Historical Behaviour

Category	Purpose	References
Execution Log Analysis	Detect event sequences that take longer during performance testing than they have historically taken	Jiang et al. (2009); Tan et al. (2008)
	Detect event sequences that occur during performance testing, but have not occurred historically or event signatures that occurred historically, but did not occur during performance testing.	Cotroneo et al. (2007); Fu et al. (2009); Jiang et al. (2005, 2007); Lou et al. (2010); Shang et al. (2013)
	Detect event sequences that occur during performance testing and have also occurred historically	Hellerstein et al. (2002); Larsson and Hamou-Lhadj (2013)
Performance Counter Analysis	Detect performance counters whose value during performance testing differs from their historical values	Breitgand et al. (2009); Cherkasova et al. (2008, 2009); Duttagupta and Nambiar (2011); Huebner et al. (2001); Mi et al. (2008); Nguyen et al. (2011, 2012, 2014); Sandeep et al. (2008)
	Detect performance counters whose relationship during performance testing differs from their historical relationships	Foo et al. (2010); Malik et al. (2010, 2013)

Execution logs are not typically designed for automated analysis (Jiang et al., 2008a). Each occurrence of an execution event results in a slightly different log line, because log lines contain static components as well as dynamic information (which may be different for each occurrence of a particular execution event). Dynamic information includes, but is not limited to, user names, IP addresses, URLs, message contents, job IDs and queue sizes. For example, `Queuing item 1 / 1` and `Queuing`

`item 1 / 1,000` contain static components (`Queuing item ___ / ___`) that represent the execution event and dynamic information (`1 / 1` and `1 / 1,000`) that represent additional information about the execution event. Dynamic information must be removed from the log lines prior to identify similar execution events. We refer to the process of identifying and removing dynamic information from a log line as "abstracting" the log line.

Execution log analysis approaches have three steps. First, the execution logs are abstracted to execution events by removing implementation and instance-specific details from the execution logs. Second, the execution events from a previous performance or from the field (i.e., the baseline) are represented by a higher-level model (e.g., a finite-state machine). Third, these higher-level models are used to flag anomalies (e.g., event sequences that occur during performance testing, but have not historically occurred) in the execution events from the performance test.

The most common application of execution log analysis is to detect event sequences that occur in the field, but do not occur during performance testing (Cotroneo et al., 2007; Fu et al., 2009; Jiang et al., 2005, 2007; Lou et al., 2010; Shang et al., 2013). Event sequences that occur in the field, but not during performance testing indicate untested functionality. Such sequences may also indicate issues with the system's behaviour. For example, Shang et al. (2013) identified event sequences that occurred in the field, but not during performance testing. These sequences were caused by a variety of issues including 1) machine failures, 2) missing libraries and 3) disk space errors.

Hellerstein et al. (2002) and Larsson and Hamou-Lhadj (2013) mine event sequences from the execution logs of a system. These sequences are given to performance analysts to help them understand the system's behaviour. For example, event sequences mined by Larsson and Hamou-Lhadj (2013) were given to performance analysts to reduce the manual effort required to identify relevant events when debugging their systems.

Jiang et al. (2009) and Tan et al. (2008) mine event sequences from the execution logs of a system. These sequences are annotated with information about the sequences performance (e.g., sequence or event duration). The annotated sequences are used to flag anomalies where the same sequence occurs in both the performance test and the field, but the sequences take much longer in the field compared to the test.

Several other execution log analysis approaches also exist that do not require a baseline (Fu et al., 2013; Jiang et al., 2008c; Salfner and Tschirpke, 2008; Xu et al., 2009a,b, 2010). These approaches mine execution logs to determine the dominant (expected) behaviour of the system and flag anomalous deviations from the dominant behaviour. However, these approaches have two limitations. First, these approaches cannot be used to compare performance tests to the field. Second, these approaches assume that the dominant behaviour is the expected (correct) behaviour.

Execution log analysis is faced with a major challenge. The first step in execution log analysis is to remove dynamic information from the execution logs in order to identify similar execution events (i.e., "abstracting" the log line). However, log abstraction discards all dynamic information (Fu et al., 2009; Jiang et al.,

2008a,b; Kunz, 1997; Nagappan and Vouk, 2010; Xu et al., 2009a). Therefore, information that may be highly relevant for understanding whether the system's behaviour differs from the system's historical behaviour is neglected. For example, `files in queue = 1` and `files in queue = 10,000` are both abstracted to `files in queue = ___` although the exact number of `files in queue` may be relevant for understanding whether the system's behaviour differs from the system's historical behaviour.

Execution log analysis approaches have one major limitation. These approaches do not consider performance counters and assume that differences between the system's behaviour and its historical behaviour are associated with anomalous (i.e., uncommon) sequences. Therefore, these approaches cannot detect differences in resource usage. For example, memory leaks often do not lead to an anomalous sequence until the system's memory is exhausted, such leaks will simply increase the net memory consumption of the sequence.

2.2.2 Performance Counter Analysis

Performance counters record the system's resource usage (e.g., CPU usage, memory consumption and network IO), performance (e.g., response time and throughput) and reliability (e.g., mean time-to-failure). They are collected through explicit monitoring tools (e.g., PerfMon (PerfMon, 2016) and Pidstat (SYSSTAT, 2016)). Performance counters can be easily collected using these tools whereas execution logs must be manually inserted into the code by developers and require pre-processing (i.e., abstraction).

The most common application of performance counter analysis is to detect performance counters whose value in the field exceed their value during performance testing (e.g., a system that requires 10GB of memory in the field, but only 4GB of memory during performance testing) (Breitgand et al., 2009; Cherkasova et al., 2008, 2009; Duttagupta and Nambiar, 2011; Huebner et al., 2001; Mi et al., 2008; Nguyen et al., 2011, 2012, 2014; Sandeep et al., 2008). These approaches often assume that the performance counters are independent. However, such an assumption rarely holds because performance counters are often highly correlated (Malik et al., 2010, 2013). Such correlations between the performance counters can significantly reduce the usefulness of performance counter analysis. For example, memory, CPU and incoming request rate may all simultaneously exceed their historical thresholds, yet a control chart cannot detect whether these exceptions are caused by the same issue (Nguyen et al., 2011).

Foo et al. (2010) and Malik et al. (2010, 2013) have proposed approaches to performance counter analysis that leverage the correlations between the performance counters. These approaches measure the correlation between the performance counters in the field and the correlation between performance counters in the test. The correlations in the field are then compared to the correlations in the test to identify performance issues (e.g., performance regressions and performance anomalies). Such approaches have been successful at *detecting* performance issues. However, they cannot *detect* the underlying cause.

Similar to execution log analysis approaches, several other performance counter analysis approach also exist that do not require a baseline (Syer et al., 2011a,b).

However, these approaches cannot be used to compare the system's current behaviour to its historical behaviour.

Performance counter analysis has two limitations. First, performance counters capture the system's perspective of performance and not the user's perspective. However, the user's perspective of performance may differ significantly from the aggregate performance (Dilley, 1996; Zaman et al., 2012a). For example, the system's average response time for 1,000 users may be 100 milliseconds, but the response time for some users may be significantly higher than other users. Second, performance counter analysis approaches do not consider execution logs. Therefore, these approaches can identify *if* the system's behaviours differs its historical behaviour, but they cannot identify *why* they differ.

Current approaches for comparing a system's current behaviour to its historical behaviour do not fully leverage the execution logs because the dynamic information in the logs is discarded when the logs are abstracted to events. Further, these approaches do not simultaneously consider multiple sources of information (i.e., they consider either the execution logs *or* the performance counters, but not both). Therefore, novel approaches are needed for comparing a system's current behaviour to its historical behaviour that enrich the execution events with additional sources of valuable information (i.e., performance counters) and fully leverage the information in the execution logs (i.e., *both* the static components of the execution logs *and* dynamic information in the execution logs).

CHAPTER 3

Enriching the Execution Events by Leveraging the Time Stamps and Worker IDs in the Execution Logs

The rise of large-scale software systems poses many new challenges for the software performance engineering field. Failures in these systems are often associated with performance issues, rather than with feature bugs. Therefore, performance testing has become essential to ensuring the problem-free operation of these systems. However, the performance testing process is faced with major challenges. One such challenge is that evolving field workloads, in terms of evolving feature sets and usage patterns, often lead to "outdated" tests that are not representative of the field. Hence performance analysts must continually validate whether their tests are still representative of the field. Such validation may be performed by comparing execution logs from the performance test and the field. However, the size and

unstructured nature of execution logs makes such a comparison unfeasible without automated support.

In this chapter, we propose an approach to enrich the execution events with information about *when* the event occurred and *who* caused the event by leveraging the dynamic information in the execution logs. Our approach then uses the enriched execution events to validate whether a performance test resembles the field workload and, if not, determines how they differ. Performance analysts can then update their tests to eliminate such differences, hence creating more realistic tests. We perform six case studies on two large systems: one open-source system and one enterprise system. Our approach identifies differences between performance tests and the field with a precision of 92% compared to only 61% for the state-of-the-practice and 19% for a conventional statistical comparison.

3.1 Introduction

The rise of large-scale software systems (e.g., Amazon.com and Google's GMail) poses new challenges for the software performance engineering field (Software Engineering Institute, 2006). These systems are deployed across thousands of machines, require near-perfect up-time and support millions of concurrent connections and operations. Failures in such systems are often associated with performance issues, rather than with feature bugs (Compuware, 2006; Dean and Barroso, 2013; Simic and Conklin, 2012; Weyuker and Vokolos, 2000). These performance issues have led to several high-profile failures. Such failures have significant financial and reputational repercussions.

Performance testing has become essential in ensuring the problem-free operation of such systems. Performance tests are usually derived from the field (i.e., alpha or beta testing data or actual production data). The goal of such tests is to examine how the system behaves under realistic workloads to ensure that the system performs well in the field. However, ensuring that tests are "realistic" (i.e., that they accurately represent the current field workloads) is a major challenge. Field workloads are based on the behaviour of thousands or millions of users interacting with the system. These workloads continuously evolve as 1) the user base changes, 2) features are added, modified or removed and 3) user feature preferences changee (Abad et al., 2012; Gill et al., 2007; Kavulya et al., 2010). For example, the failure rate for Hadoop jobs on Yahoo's M45 cluster dropped from 70% in April 2008 to 10% in Many 2008 and to 3% in June 2008 as the Hadoop applications using the cluster matured (Kavulya et al., 2010). Such evolving field workloads often lead to performance tests that are not representative of the field (Bertolotti and Calzarossa, 2001; Voas, 2000). Yet the system's behaviour depends significantly on the field workload (Dean and Barroso, 2013; Zhang et al., 2013). Therefore, "outdated" performance tests may not allow performance analysts to ensure that the system will perform well in the field.

Performance analysts monitor the impact of field workloads on the system's performance using performance counters (e.g., response time and memory usage) and reliability counters (e.g., mean time-to-failure). Performance analysts must determine the cause of any deviation in the counter values from the specified or expected range (e.g., response time exceeds the maximum response time permitted by the service level agreements or memory usage exceeds the average historical memory

usage). These deviations may be caused by changes to the field workloads (Dean and Barroso, 2013; Zhang et al., 2013). Such changes are common and may require performance analysts to update their tests (Bertolotti and Calzarossa, 2001; Voas, 2000). This has led to the emergence of "continuous testing," where tests are continuously updated and re-run even after the system's deployment.

A major challenge in the continuous testing process is to ensure performance tests accurately represent the current field workloads. However, documentation describing the expected system behaviour is rarely up-to-date (Ernst et al., 2015; Holvitie et al., 2014; Parnas, 1994). Fortunately, execution logs, which record notable events at runtime, are readily available in most large-scale systems to support remote issue resolution and legal compliance. Further, these logs contain developer and operator knowledge (i.e., they are manually inserted by developers) whereas instrumentation tends to view the system as a black-box (Shang et al., 2011, 2015). Hence, execution logs are the best data available to describe and monitor the behaviour of the system under a realistic workload. Therefore, we propose an automated approach to validate performance tests by comparing system behaviour across tests and the field. Our approach enriches the execution events with information about *when* the event occurred and *who* caused the event. We then derive workload signatures from the enriched execution event, then use statistical techniques to identify differences between the workload signatures of the performance test and the field.

Such differences can be broadly classified as feature differences (i.e., differences in the exercised features), intensity differences (i.e., differences in how often each feature is exercised) and issue differences (i.e., new errors appearing in the field).

These identified differences can help performance analysts improve their tests in the following two ways. First, performance analysts can tune their performance tests to more accurately represent current field workloads. For example, the performance test workloads can be updated to eliminate differences in how often features are exercised (i.e., to eliminate intensity differences). Second, new field errors, which are not covered in existing testing, can be identified based on the differences. For example, a machine failure in a distributed system may raise new errors that are often not tested.

This chapter makes three contributions:

1. We present an approach to enrich the execution events with information about *when* the event occurred and *who* caused the event by leveraging the dynamic information in the execution logs.

2. We demonstrate an automated approach to validate the representativeness of a performance test by comparing the system behaviour between tests and the field. Our approach identifies important execution events that best explain the differences between the system's behaviour during a performance test and in the field.

3. Through six case studies on two large systems, one open-source system and one enterprise system, we show that our approach is scalable and can help performance analysts validate their tests.

3.1.1 Organization of the Chapter

This chapter is organized as follows. Section 3.2 provides a motivational example of how our approach may be used in practice. Section 3.3 describes our approach in detail. Section 3.4 presents our case studies. Section 3.5 discusses the results of our case studies and some of the design decisions for our approach. Section 3.6 outlines the threats to validity. Finally, Section 3.7 concludes the chapter.

3.2 Motivational Example

Jack, a performance analyst, is responsible for continuous performance testing of a large-scale telecommunications system. Given the continuously evolving field workloads, Jack often needs to update his tests to ensure that the test workloads represent, as much as possible, the field workloads. Jack monitors the field workloads using performance counters (e.g., response time and memory usage). When one or more of these counters deviates from the specified or expected range (e.g., response time exceeds the maximum response time specified in the service level agreements or memory usage exceeds the average historical memory usage), Jack must investigate the cause of the deviation. He may then need to update his tests.

Jack monitors the system's performance in the field and discovers that the system's memory usage exceeds the average historical memory usage. Pressured by time (given the continuously evolving field workloads) and management (who are keen to boast a high quality system), Jack needs to quickly update his performance tests to replicate this issue in his test environment. Jack can then determine why

the system is using more memory than expected. Although the performance counters have indicated that the field workloads have changed (leading to increased memory usage), the only artifacts that Jack can use to understand *how* the field workloads have changed, and hence how his tests should be updated, are execution logs. These logs describe the system's behaviour, in terms of important execution events (e.g., starting, queuing or completing a job), during the test and in the field.

Jack tries to compare the execution logs from the field and the test by looking at how often important events (e.g., receiving a request) occur in the field compared to his test. However, terabytes of execution logs are collected and some events occur millions of times. Further, simply comparing how often each event occurs does not provide the detail that Jack needs to fully understand the differences between the field and test workloads. For example, simply comparing how often each event occurs ignores the use case that generated the events (i.e., the context).

To overcome these challenges, Jack needs an automated, scalable approach to determine whether his tests are representative of the field and, if not, determine how his tests differ so that they can be updated. We present such an approach in the next section.

Using this approach, Jack is shown groups of users whose behaviour best explains the differences between his test workloads and the field. In addition, Jack is also shown key execution events that best explain the differences between each of these groups of users. Jack then discovers a group of users who are using the high-definition group chat feature (i.e., a memory-intensive feature) more strenuously than in the past. Finally, Jack is able to update his test to better represent the users' changing feature preferences and hence, the system's behaviour in the field.

3.3 Approach

This section outlines our approach for validating performance tests by automatically deriving workload signatures from execution logs then comparing the signatures from a test against the signatures from the field. Figure 3.1 provides an overview of our approach. First, we group execution events from the test logs and field logs into workload signatures that describe the workloads. Second, we cluster the workload signatures into groups where a similar set of execution events have occurred. Finally, we analyze the clusters to identify the execution events that correspond to meaningful differences between the performance test and the field. We describe each phase in detail and demonstrate our approach with a working example of a hypothetical chat application.

3.3.1 Execution Logs

The second column of Table 3.1 and Table 3.2 presents the execution logs from our working example. These execution logs contain both static information (e.g., `starts a chat`) and dynamic information (e.g., `Alice` and `Bob`) that changes with each occurrence of an event. Table 3.1 and Table 3.2 present the execution logs from the field and the test respectively. The test has been configured with a simple use case (from 00:01 to 00:06) that is continuously repeated.

3.3.2 Data Preparation

Execution logs are difficult to analyze because they are unstructured. Therefore, we abstract the execution logs to execution events to enable automated statistical

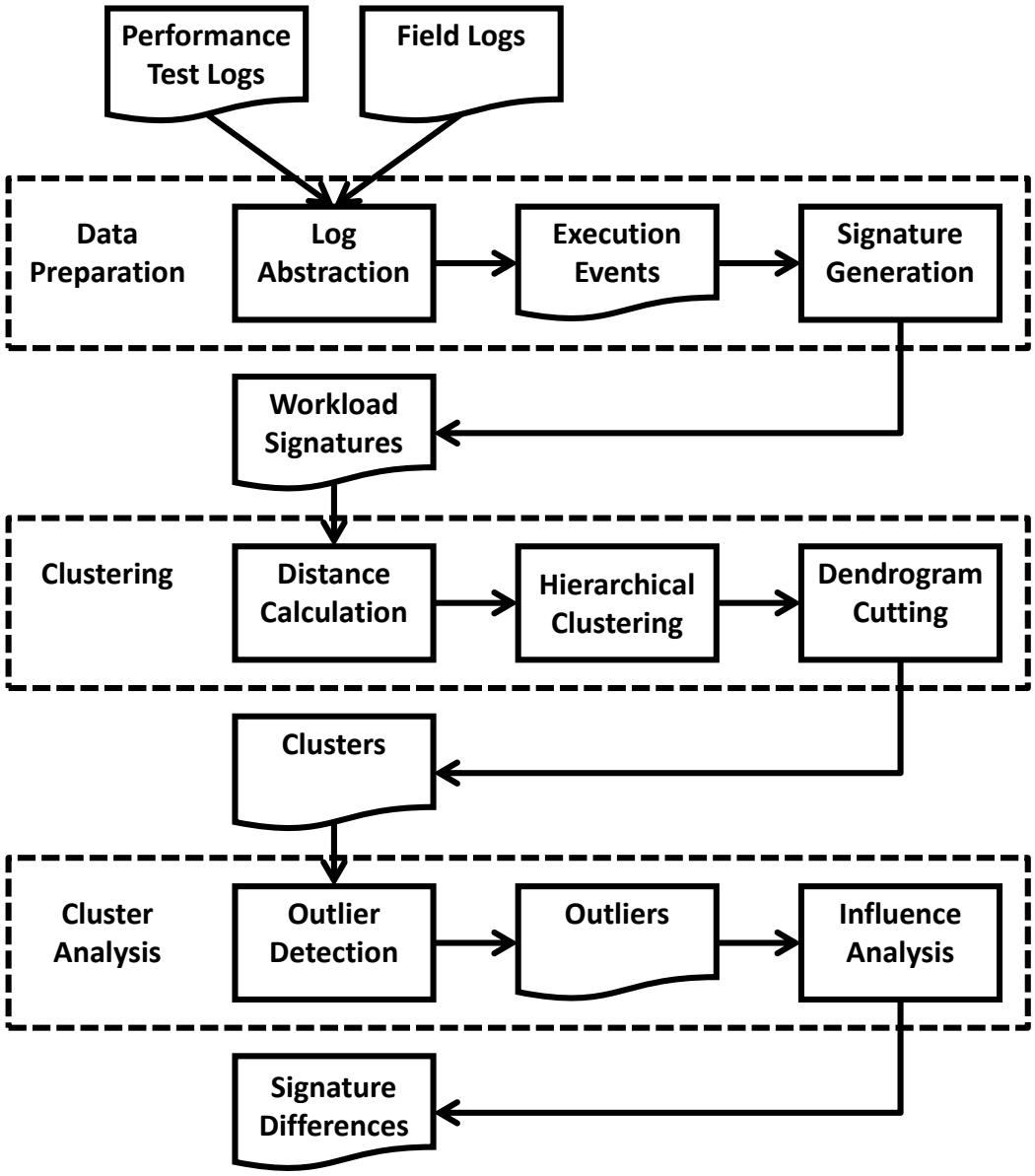

Figure 3.1: An Overview of Our Approach.

Table 3.1: Abstracting Execution Logs to Execution Events: Execution Logs from the Field

Time	User	Log Line	Execution Event	Execution Event ID
00:01	Alice	starts a chat with Bob	starts a chat with ___	1
00:01	Alice	says "hi, are you busy?" to Bob	says ___ to ___	2
00:03	Bob	says "yes" to Alice	says ___ to ___	2
00:05	Carl	starts a chat with Dan	starts a chat with ___	1
00:05	Carl	says "do you have files?" to Dan	says ___ to ___	2
00:08	Dan	Initiate file transfer to Carl	Initiate file transfer to ___	3
00:09	Dan	Initiate file transfer to Carl	Initiate file transfer to ___	3
00:12	Dan	says "got it?" to Carl	says ___ to ___	2
00:14	Carl	says "thanks" to Dan	says ___ to ___	2
00:14	Carl	ends the chat with Dan	ends the chat with ___	4
00:18	Alice	says "ok, bye" to Bob	says ___ to ___	2
00:18	Alice	ends the chat with Bob	ends the chat with ___	4

Table 3.2: Abstracting Execution Logs to Execution Events: Execution Logs from a Performance Test

Time	User	Log Line	Execution Event	Execution Event ID
00:01	USER1	starts a chat with USER2	starts a chat with ___	1
00:02	USER1	says "MSG1" to USER2	says ___ to ___	2
00:03	USER2	says "MSG2" to USER1	says ___ to ___	2
00:04	USER1	says "MSG3" to USER2	says ___ to ___	2
00:06	USER1	ends the chat with USER2	ends the chat with ___	4
00:07	USER3	starts a chat with USER4	starts a chat with ___	1
00:08	USER3	says "MSG1" to USER4	says ___ to ___	2
00:09	USER4	ays "MSG2" to USER3	says ___ to ___	2
00:10	USER3	says "MSG3" to USER4	says ___ to ___	2
00:12	USER3	ends the chat with USER4	ends the chat with ___	4
00:13	USER5	starts a chat with USER6	starts a chat with ___	1
00:14	USER5	says "MSG1" to USER6	says ___ to ___	2
00:15	USER6	says "MSG2" to USER5	says ___ to ___	2
00:16	USER5	says "MSG3" to USER6	says ___ to ___	2
00:18	USER5	ends the chat with USER6	ends the chat with ___	4

analysis. We then enrich the execution events by leveraging the time stamps and worker IDs in the execution logs). Finally, we generate workload signatures that represent the behaviour of the system's users.

3.3.2.1 Log Abstraction

We abstract the execution logs using the approach proposed by Jiang et al. (2008a). In addition, many execution logs and their corresponding execution events have been manually reviewed by multiple, independent system experts to verify the correctness of the abstraction. These experts have several years of experience working with these logs including extensive experience manually abstracting the logs using regular expression matching tools (e.g., Perl). Therefore, they have deep knowledge of how the execution logs should be abstracted (i.e., they can manually verify that static information is not removed from the execution logs while the dynamic information is removed from the execution logs).

Table 3.1 and Table 3.2 present the execution events and execution event IDs (a unique ID automatically assigned to each unique execution event) for the execution logs from the field and from the test in our working example. These tables demonstrate the input (i.e., the log lines) and the output (i.e., the execution events) of the log abstraction process. For example, the `starts a chat with Bob` and `starts a chat with Dan` log lines are both abstracted to the `starts a chat with ___` execution event.

3.3.2.2 Signature Generation

We enrich the execution events by leveraging time stamps and worker IDs in the execution logs to create workload signatures. Workload signatures describe the users' behaviour in terms of feature usage expressed by the execution events. In our approach, a workload signature represents either 1) the behaviour of one of the system's users, or 2) the aggregated behaviour of all of the system's users at one point in time. We use the term "user" to describe any type of end user, whether a human or software agent. For example, the end users of a system such as Amazon.com are both human and software agents (e.g., "shopping bots" that search multiple websites for the best prices). Workload signatures are represented as points in an n-dimensional space (where n is the number of unique execution events).

Workload signatures representing individual users are generated for each user because workloads are driven by the behaviour of the system's users. We also found cases when an execution event only causes errors when over-stressed by an individual user (i.e., one user executing the event 1,000 times has a different impact on the system's behaviour than 100 users each executing the event 10 times) (Syer et al., 2014). Therefore, it is important to identify users whose behaviour is seen in the field, but not during the test.

Workload signatures representing individual users are generated in two steps. First, we identify all of the unique worker IDs that appear in the execution logs. Workers represent a logical "unit of work" where a workload is the sum of one or more units of work. In systems primarily used by human end users (e.g., e-commerce and telecommunications system), worker IDs may include user names, email addresses or device IDs. In systems primarily used for processing large

amounts of data (e.g., distributed data processing frameworks such as Hadoop), worker IDs may include job IDs or thread IDs. The second column of Table 3.3 presents all of the unique worker IDs identified from the execution logs of our working example. Second, we generate a signature for each worker ID by counting the number of times that each type of execution event is attributable to each worker ID. For example, from Table 3.1, we see that `Alice` starts one chat, sends two messages and ends one chat. Table 3.3 shows the signatures generated for each user using the events in Table 3.1 and Table 3.2.

Table 3.3: Workload Signatures Representing Individual Users

	Worker ID	**Execution Event ID**			
		1 `start chat`	2 `send message`	3 `transfer file`	4 `end chat`
Field Users	`Alice`	1	2	0	1
	`Bob`	0	1	0	0
	`Carl`	1	2	0	1
	`Dan`	0	1	2	0
Test Users	`USER1`	1	2	0	1
	`USER2`	0	1	0	0
	`USER3`	1	2	0	1
	`USER4`	0	1	0	0
	`USER5`	1	2	0	1
	`USER6`	0	1	0	0

Workload signatures representing the aggregated users are generated for short periods of time (e.g., 1 minute) to represent the traditional notion of a "workload" (i.e., the total number and mix of incoming requests to the system). The system's resource usage is highly dependent on these workloads. Unlike the workload signatures representing individual users, the workload signatures representing aggregated users capture the "burstiness" (i.e., the changes in the number of request

per seconds) of the workload. Therefore, it is important to identify whether the aggregated user behaviour that is seen in the field is also seen during the test.

Workload signatures representing the aggregated users are generated by grouping the execution logs into time intervals (i.e., grouping the execution logs that occur between two points in time). Grouping is a two step process. First, we specify the length of the time interval. We have previously found that time intervals of 90 seconds to 150 seconds perform well when generating workload signatures that represent the aggregated user behaviour (Syer et al., 2013). However, these time intervals may vary between systems. System experts should determine the optimal time interval (i.e., a time interval that provides the necessary detail without an unnecessary overhead) for their systems. Alternatively, system experts may specify multiple time intervals and generate overlapping signatures (e.g., generating signatures representing the aggregated user behaviour in 1, 3 and 5 minute time intervals). Second, we generate a signature for each time interval by counting the number of times that each type of execution event occurs in that time interval. For example, from Table 3.1, we see that one chat is started and three messages are sent between time `00:01` and `00:06`. Table 3.4 shows the signatures generated for each six second time interval using the events in Table 3.1 and Table 3.2. From Table 3.4, we see that all three signatures generated from the test are identical. This is to be expected because the test was configured with a simple use case (from 00:01 to 00:06) that is continuously repeated.

Our approach considers the individual user signatures and the aggregate user signatures separately. Therefore, the Clustering and Cluster Analysis phases are

Table 3.4: Workload Signatures Representing The Aggregated Users

	Time	Execution Event ID			
		1 start chat	2 send message	3 transfer file	4 end chat
Field Times	00:01-00:06	2	3	0	0
	00:07-00:12	0	1	2	0
	00:13-00:18	0	2	0	2
Test Times	00:01-00:06	1	3	0	1
	00:07-00:12	1	3	0	1
	00:13-00:18	1	3	0	1

applied once to the individual user signatures and once to the aggregate user signatures. For brevity, we demonstrate the remainder of our approach using only the individual user signatures in Table 3.3.

3.3.3 Clustering

The second phase of our approach is to cluster the workload signatures into groups where a similar set of events have occurred. We can then identify groups of similar, but not necessary identical, workload signatures.

The clustering phase in our approach consists of three steps. First, we calculate the dissimilarity (i.e., distance) between every pair of workload signatures. Second, we use a hierarchical clustering procedure to cluster the workload signatures into groups where a similar set of events have occurred. Third, we convert the hierarchical clustering into k partitional clusters (i.e., where each workload signature is a member in only one cluster).

3.3.3.1 Distance Calculation

Each workload signature is represented by one point in an n-dimensional space (where n is the number of unique execution events). Clustering procedures rely on identifying points that are "close" in this n-dimensional space. Therefore, we must specify how distance is measured in this space. A larger distance between two points implies a greater dissimilarity between the workload signatures that these points represent. We calculate the distance between every pair of workload signatures to produce a distance matrix.

We use the Pearson distance, a transform of the Pearson correlation (Fulekar, 2008), as opposed to the many other distance measures (Cha, 2007; Frades and Matthiesen, 2009; Fulekar, 2008), as the Pearson distance often produces a clustering that is a closer match to the manually assigned clusters (Huang, 2008; Sandhya and Govardhan, 2012). We find that the Pearson distance performs well when clustering workload signatures (i.e., using the Pearson distance to cluster workload signatures results in our approach having a higher precision than if we had used a different distance measure). See Section 3.5.3 and Syer et al. (2014) for a detailed analysis comparing the precision of our approach using different distance measures.

We first calculate the Pearson correlation (ρ) between two workload signatures using Equation 3.1. This measure ranges from -1 to $+1$, where a value of 1 indicates that the two workload signatures are identical, a value of 0 indicates that there is no relationship between the signatures and a value of -1 indicates an inverse relationship between the signatures (i.e., as the occurrence of specific execution events increase in one workload signature, they decrease in the other).

$$\rho = \frac{n \sum_i^n x_i \times y_i - \sum_i^n x_i \times \sum_i^n y_i}{\sqrt{(n \sum_i^n x_i^2 - (\sum_i^n x_i)^2) \times (n \sum_i^n y_i^2 - (\sum_i^n y_i)^2)}} \tag{3.1}$$

where x and y are two workload signatures and n is the number of execution events.

We then transform the Pearson correlation (ρ) to the Pearson distance (d_ρ) using Equation 3.2.

$$d_\rho = \begin{cases} 1 - \rho & \text{for } \rho \geq 0 \\ |\rho| & \text{for } \rho < 0 \end{cases} \tag{3.2}$$

Table 3.5 presents the distance matrix produced by calculating the Pearson distance between every pair of workload signatures in our working example.

Table 3.5: Distance Matrix

	Alice	Bob	Carl	Dan	USER1	USER2	USER3	USER4	USER5	USER6
Alice	0	0.184	0	0.426	0	0.184	0	0.184	0	0.184
Bob	0.184	0	0.184	0.826	0.184	0	0.184	0	0.184	0
Carl	0	0.184	0	0.426	0	0.184	0	0.184	0	0.184
Dan	0.426	0.826	0.426	0	0.426	0.826	0.426	0.826	0.426	0.826
USER1	0	0.184	0	0.426	0	0.184	0	0.184	0	0.184
USER2	0.184	0	0.184	0.826	0.184	0	0.184	0	0.184	0
USER3	0	0.184	0	0.426	0	0.184	0	0.184	0	0.184
USER4	0.184	0	0.184	0.826	0.184	0	0.184	0	0.184	0
USER5	0	0.184	0	0.426	0	0.184	0	0.184	0	0.184
USER6	0.184	0	0.184	0.826	0.184	0	0.184	0	0.184	0

3.3.3.2 Hierarchical Clustering

We use an agglomerative, hierarchical clustering procedure (Tan et al., 2005) to cluster the workload signatures using the distance matrix calculated in the previous step. The clustering procedure starts with each signature in its own cluster and proceeds to find and merge the closest pair of clusters (using the distance matrix), until only one cluster (containing everything) is left. One advantage of hierarchical clustering is that we do not need to specify the number of clusters prior to performing the clustering. Further, performance analysts can change the number of clusters (e.g., to produce a larger number of more cohesive clusters) without having to rerun the clustering phase.

Hierarchical clustering updates the distance matrix based on a specified linkage criterion. We use the average linkage, as opposed to the many other linkage criteria (Frades and Matthiesen, 2009; Tan et al., 2005), as the average linkage is the de facto standard (Frades and Matthiesen, 2009; Tan et al., 2005). The average linkage criterion is also the most appropriate when little information about the expected clustering (e.g., the relative size of the expected clusters) is available. We find that the average linkage criterion performs well when clustering workload signatures (i.e., using the average linkage criterion to cluster workload signatures results in our approach having a higher precision than if we had used a different linkage criterion). See Section 3.5.3 and Syer et al. (2014) for a detailed analysis comparing the precision of our approach using different linkage criteria.

When two clusters are merged, the average linkage criterion updates the distance matrix in two steps. First, the merged clusters are removed from the distance

matrix. Second, a new cluster (containing the merged clusters) is added to the distance matrix by calculating the distance between the new cluster and all existing clusters. The distance between two clusters is the average distance (as calculated by the Pearson distance) between the workload signatures of the first cluster and the workload signatures of the second cluster (Frades and Matthiesen, 2009; Tan et al., 2005).

We calculate the distance between two clusters ($d_{x,y}$) using Equation 3.3.

$$d_{x,y} = \frac{1}{n_x \times n_y} \times \sum_{i}^{n_x} \sum_{j}^{n_y} d_\rho(x_i, y_j) \tag{3.3}$$

where $d_{x,y}$ is the distance between cluster x and cluster y, n_x is the number of workload signatures in cluster x, n_y is the number of workload signatures in cluster y and $d_\rho(x_i, y_j)$ is the Pearson distance between workload signature i in cluster x and workload signature j in cluster y.

Figure 3.2 shows the dendrogram produced by hierarchically clustering the workload signatures from our working example.

3.3.3.3 Dendrogram Cutting

The result of a hierarchical clustering procedure is a hierarchy of clusters. This hierarchy is typically visualized using hierarchical cluster dendrograms. Figure 3.2 is an example of a hierarchical cluster dendrogram. Such dendrograms are binary tree-like diagrams that show each stage of the clustering procedure as nested clusters (Tan et al., 2005).

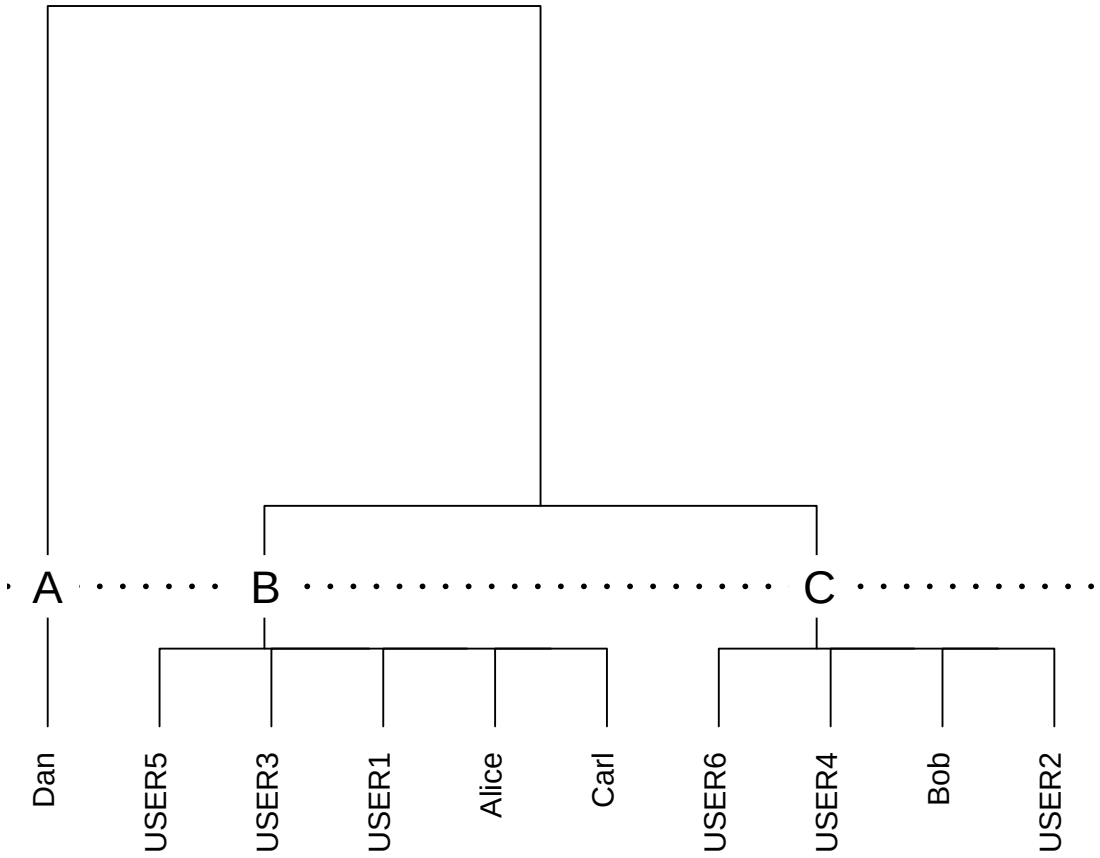

Figure 3.2: Sample Dendrogram. The dotted horizontal line indicates where the dendrogram was cut into three clusters (i.e., Cluster A, B and C).

To complete the clustering procedure, the dendrogram must be cut at some height. This height represents the maximum amount of intra-cluster dissimilarity that will be accepted within a cluster before that cluster is further divided. Cutting the dendrogram results in a clustering where each workload signature is assigned to only one cluster. Such a cutting of the dendrogram is done either by 1) manual (visual) inspection or 2) statistical tests (referred to as stopping rules).

Although a visual inspection of the dendrogram is flexible and fast, it is subject to human bias and may not be reliable. Therefore, we use a stopping rule to

determine where to cut the dendrogram. We use the Calinski-Harabasz stopping rule (Calinski and Harabasz, 1974), as opposed to the many other stopping rules (Calinski and Harabasz, 1974; Duda and Hart, 1973; Milligan and Cooper, 1985; Mojena, 1977; Rousseeuw, 1987), as the Calinski-Harabasz stopping rule most often cuts the dendrogram into the correct number of clusters (Milligan and Cooper, 1985). We find that the Calinski-Harabasz stopping rule performs well when cutting dendrograms produced by clustering workload signatures (i.e., using the Calinski-Harabasz stopping rule to cut the dendrograms produced by clustering workload signatures results in our approach having a higher precision than if we had used a different stopping rule). See Section 3.5.3 and Syer et al. (2014) for a detailed analysis comparing the precision of our approach using different stopping rules.

The Calinski-Harabasz stopping rule is a pseudo-*F-statistic*, which is a ratio reflecting within-cluster similarity and between-cluster dissimilarity. The optimal clustering will have high within-cluster similarity (i.e., the workload signatures within a cluster are similar) and a high between-cluster dissimilarity (i.e., the workload signatures from two different clusters are dissimilar).

The dotted horizontal line in Figure 3.2 shows where the Calinski-Harabasz stopping rule cut the hierarchical cluster dendrogram from our working example into three clusters (i.e., the dotted horizontal line intersects with solid vertical lines at three points in the dendrogram). Cluster A contains one user (Dan), cluster B contains four users (Alice, Carl, USER1 and USER3) and cluster C contains three users (Bob, USER2 and USER4).

3.3.4 Cluster Analysis

The third phase in our approach is to identify the execution events that correspond to the differences between the workload signatures from the performance test and the field. As execution logs may contain billions of events describing the behaviour of millions of users, this phase will only identify the most important workload signature differences. Therefore, our approach helps system experts to update their performance tests by identifying the most meaningful differences between their performance tests and the field. Such "filtering" provides performance analysts with a concrete list of events to investigate.

The cluster analysis phase of our approach consists of two steps. First, we detect outlying clusters. Outlying clusters contain workload signatures that are not well represented in the test (i.e., workload signatures that occur in the field significantly more than in the test). Second, we identify key execution events of the outlying clusters. We refer to these execution events as "signature differences". Knowledge of these signature differences may lead performance analysts to update their performance tests. "Event A occurs 10% less often in the test relative to the field" is an example of a signature difference that may lead performance analysts to update a test such that Event A occurs more frequently.

3.3.4.1 Outlying Cluster Detection

Clusters contain workload signatures from the performance test and/or the field. When clustering workload signatures from a field-representative performance test and the field, we would expect that each cluster would have the same proportion of workload signatures from the field compared to workload signatures from the

test. Clusters with a high proportion of workload signatures from the test relative to the field would then be considered "outlying" clusters. These outlying clusters contain workload signatures that represent behaviour that is seen in the field, but not during the test.

We identify outlying clusters using a *one-sample upper-tailed z-test for a population proportion*. These tests are used to determine whether the observed sample proportion is significantly larger than the hypothesized population proportion. The difference between the observed sample proportion and the hypothesized population proportion is captured by a *Z Score* (Sokal and Rohlf, 2011). Higher *z-scores* indicate an increased probability that the observed sample proportion is greater than the hypothesized population proportion (i.e., that the cluster contains a greater proportion of workload signatures from the field). Hence, as the *Z Score* of a particular cluster increases, the probability that the cluster is an outlying cluster also increases. *One-sample z-tests for a proportion* have successfully been used to identify outliers in software engineering data using these hypotheses (Jiang et al., 2008c; Kremenek and Engler, 2003; Syer et al., 2014).

We construct the following hypotheses to be tested by a *one-sample upper-tailed z-test*. Our *null hypothesis* assumes that the proportion of workload signatures from the field in a cluster is less than 90%. Our *alternate hypothesis* assumes that this proportion is greater than 90%.

Equation 3.4 presents how the *Z Score* of a particular cluster is calculated.

$$p = \frac{n_x}{n_x + n_y} \tag{3.4}$$

$$\sigma = \sqrt{\frac{p_0 \times (1 - p_0)}{n_x + n_y}} \tag{3.5}$$

$$z = \frac{p - p_0}{\sigma} \tag{3.6}$$

where n_x is the number of workload signatures from the field in the cluster, n_y is the number of workload signatures from the test in the cluster, p is the proportion of workload signatures from the field in the cluster, σ is the standard error of the sampling distribution of p and p_0 is the hypothesized population proportion (i.e., 90%, the null hypothesis).

We then use the *Z Score* to calculate a *p-value* to determine whether the sample population proportion is significantly greater than the hypothesized proportion population. This *p-value* accounts for differences in the total number of workload signatures from the test compared to the field as well as variability in the proportion of workload signatures from the field across the clusters.

Equation 3.7 presents how the *p-value* of a particular cluster is calculated.

$$Z(x, \mu, \sigma) = \frac{1}{\sigma \times \sqrt{2 * \pi}} \times e^{\frac{-(x-\mu)^2}{2 \times \sigma^2}} \tag{3.7}$$

$$p = P(Z > z) \tag{3.8}$$

where μ is the average proportion of workload signatures in a cluster, σ is the standard deviation of the proportion of workload signatures in a cluster, $Z(x, \mu, \sigma)$ is the normal distribution given μ and σ and p is the *p-value* of the test.

Table 3.6 presents the size (i.e., the number of workload signatures in the cluster), breakdown (i.e., the number of workload signatures from the performance test and the field), *Z Score* and *p-value* for each cluster in our working example (i.e., each of the clusters that were identified when the Calinski-Harabasz stopping rule was used to cut the dendrogram in Figure 3.2).

Table 3.6: Identifying Outlying Clusters

		# Signatures from:			
Cluster	**Size**	**Field**	**Test**	***Z Score***	***p-value***
A	1	1	0	0.333	0.68
B	5	2	3	-3.737	1.00
C	4	1	3	-4.333	1.00

From Table 3.6, we find that clusters with a greater proportion of workload signatures from the field have a larger *Z Score* and smaller *p-value*. For example, the proportion of workload signatures from the field in Cluster A is 100% (i.e., $1/1$) and the corresponding *Z Score* is 0.333, whereas the proportion of workload signatures from the field in Cluster B is 40% (i.e., $2/5$) and the corresponding *Z Score* is -3.737.

From Table 3.6, we also find that the proportion of workload signatures from the field in any one cluster is not significantly more than 90% (i.e., no *p-values* are less than 0.05). Therefore, no clusters are identified as outliers . However, outliers are extremely difficult to detect in such small data sets. Therefore, for the purposes of this working example, we will assume that Cluster A has been identified as an outlier because its *Z Score* (0.333) is much larger than the *z-scores* of Cluster B (-3.737) or Cluster C (-4.333).

3.3.4.2 Signature Difference Detection

We identify the differences between workload signatures in outlying clusters and the average ("normal") workload signature using statistical measures (i.e., *unpaired two-sample two-tailed Welch's unequal variances t-tests* (Student, 1908; Welch, 1997) and *Cohen's d* effect size (Cohen, 1988)). This analysis quantifies the importance of each execution event in differentiating a cluster. Knowledge of these events may lead performance analysts to update their tests.

First, we determine the execution events that differ significantly between the workload signatures in the outlying clusters and the average workload signature. For example, execution events that occur 10 times more often in the workload signatures of an outlying cluster compared to the average workload signature should likely be flagged for further analysis by a system expert.

We perform an *unpaired two-sample two-tailed Welch's unequal variances t-test* to determine which execution events differ significantly between the workload signatures in an outlying cluster and the average workload signature. These tests are used to determine whether the difference between two population means is statistically significant (Student, 1908; Welch, 1997). The difference between the two population means is captured by a *t-statistic*. Larger absolute *t-statistics* (i.e., the absolute value of the *t-statistic*) indicate an increased probability that the two population means differ (i.e., that the number of times an execution event occurs in the workload signatures of an outlying cluster compared to the average workload signature differs). Hence, as the absolute value of the *t-statistic* of a particular execution event and outlying cluster increases, the probability that the number of times

an execution event occurs in the workload signatures of an outlying cluster compared to the average workload signature also increases. *T-tests* are one of the most frequently performed statistical tests (Elliott, 2006).

We construct the following hypotheses to be tested by an *unpaired two-sample two-tailed Welch's unequal variances t-test*. Our *null hypothesis* assumes that an execution event occurs the same number of times in the workload signatures of an outlying cluster compared to the average workload signature. Conversely, our *alternate hypothesis* assumes that the execution event does not occur the same number of times in an outlying cluster compared to the average workload signature.

Equation 3.9 presents how the *t-statistic* for a particular execution event and a particular outlying cluster is calculated.

$$\sigma = \sqrt{\frac{(n_x - 1) \times \sigma_x^2 + (n_y - 1) \times \sigma_y^2}{n_x + n_y + 2}} \tag{3.9}$$

$$t = \frac{\mu_x - \mu_y}{\sqrt{\frac{\sigma_x^2}{n_x} + \frac{\sigma_x^2}{n_y}}} \tag{3.10}$$

where n_x is the number of workload signatures in the outlying cluster, n_y is the total number of workload signatures that are not in the outlying cluster, μ_x is the average number of times the execution event occurs in the workload signatures in the outlying cluster, μ_y is the average number of times the execution event occurs in all the workload signatures that are not in the outlying cluster, σ_x is the variance of the number of times the execution event occurs in the workload signatures in the outlying cluster, σ_y is the variance of the number of times the execution event occurs in all of the workload signatures that are not in the outlying cluster, σ is the pooled standard deviation of the number of times the execution event occurs in the

workload signatures in the outlying cluster and the number of times the execution event occurs in all the workload signatures and t is the *t-statistic*.

We then use the *t-statistic* to calculate a *p-value* to test whether the difference between the number of times an execution event occurs in the workload signatures of an outlying cluster compared to the average workload signature is statistically significant.

Equation 3.11 presents how the *p-value* for a particular execution event and a particular outlying cluster is calculated.

$$v = \frac{\frac{\sigma_x^2}{n_x} + \frac{\sigma_x^2}{n_y}}{\frac{(\frac{\sigma_x^2}{n_x})^2}{n_x - 1} + \frac{(\frac{\sigma_y^2}{n_y})^2}{n_y - 1}} \tag{3.11}$$

$$T = \frac{\Gamma(\frac{v+1}{2})}{\sqrt{v \times \pi} \times \Gamma(\frac{v}{2})} \times (1 + \frac{x^2}{v})^{-\frac{v+1}{2}} \tag{3.12}$$

$$p = 2 \times P(T < t) \tag{3.13}$$

where v is the degree of freedom, Γ is the gamma function, T is the *t-distribution* and p is the *p-value* of the test.

Table 3.7 shows the *t-statistic* and the associated *p-value* for each execution event in the outlying cluster (i.e., Cluster A).

Table 3.7: Identifying Influential Execution Events

Execution Event ID	***p-value***	***Cohen's d***	***Cohen's d Interpretation***
1	0.39	0.95	Medium
2	0.39	0.95	Medium
3	0.02	2.85	Large
4	0.39	0.95	Medium

From Table 3.7, we find that Execution Event ID 3 (i.e., `initiate file transfer`) differs significantly between Cluster A and the average workload signatures (i.e., $p < 0.01$). From the workload signatures in Table 3.3, we see that Execution Event ID 3 occurs twice in Dan's workload signature (i.e., the only workload signature in Cluster A), but never in the other workload signatures.

Second, we determine the most important execution events that differ between the workload signatures in the outlying clusters and the average workload signature. For example, if execution events "A" and "B" occur 2 and 10 times more often in the workload signatures of an outlying cluster compared to the average workload signature, then execution event "B" should be flagged for further analysis by a system expert rather than execution event "A."

We calculate the *Cohen's d* effect size to determine the most important execution events that differ between the workload signatures in the outlying clusters and the average workload signature. *Cohen's d* effect size measures the difference between two population means (Cohen, 1988). Larger *Cohen's d* effect sizes indicate a greater difference between the two population means, regardless of statistical significance. Hence, as the *Cohen's d* effect size of a particular execution event and outlying cluster increases, the difference between the number of times an execution event occurs in the workload signatures of an outlying cluster compared to the average workload signature also increases.

Equation 3.14 presents how *Cohen's d* is calculated for a particular execution event and a particular outlying cluster.

$$\sigma = \sqrt{\frac{(n_x - 1) \times \sigma_x^2 + (n_y - 1) \times \sigma_y^2}{n_x + n_y + 2}} \tag{3.14}$$

$$d = \frac{\mu_x - \mu_y}{\sigma} \tag{3.15}$$

where n_x is the number of workload signatures in the outlying cluster, n_y is the total number of workload signatures that are not in the outlying cluster, μ_x is the average number of times the event occurs in the workload signatures in the outlying cluster, μ_y is the average number of times the event occurs in all the workload signatures that are not in the outlying cluster, σ_x is the variance in the number of times the event occurs in the workload signatures in the outlying cluster, σ_y is the variance in the number of times the event occurs in all of the workload signatures that are not in the outlying cluster, σ is the pooled standard deviation of the number of times the event occurs in the workload signatures in the outlying cluster and the number of times the event occurs in all the workload signatures and d is *Cohen's d*. *Cohen's d* effect size is traditionally interpreted as follows:

$$\begin{cases} \text{trivial} & \text{for } d < 0.2 \\ \text{small} & \text{for } 0.2 < d \leq 0.5 \\ \text{medium} & \text{for } 0.5 < d \leq 0.8 \\ \text{large} & \text{for } d > 0.8 \end{cases} \tag{3.16}$$

However, such an interpretation was originally proposed for the social sciences. Kampenes et al. (2007) performed a systematic review of 103 software engineering papers and empirically established the following interpretation of effect sizes in software engineering research.

$$\begin{cases} \text{trivial} & \text{for } d < 0.17 \\ \text{small} & \text{for } 0.17 < d \leq 0.6 \\ \text{medium} & \text{for } 0.6 < d \leq 1.4 \\ \text{large} & \text{for } d > 1.4 \end{cases} \tag{3.17}$$

From Table 3.7, we find that Execution Event ID 3 (i.e., `initiate file transfer`) has a large (i.e., $d > 1.4$) effect size indicating that the difference in Execution Event ID 3 between the workload signatures in Cluster A and the average workload signature is large.

Finally, we identify the influential events as any execution event where the *t-test p-value* indicates a statistically significant difference (i.e., $p < 0.01$) between the workload signatures in the outlying cluster and the average workload signature and the *Cohen's d* indicates a large difference (i.e., $d > 1.4$) between the workload signatures in the outlying cluster and the average workload signature. Table 3.7 shows the *Cohen's d* and its interpretation for each execution event in the outlying cluster (i.e., Cluster A).

From Table 3.7, we find that the difference in Execution Event ID 3 (i.e., `initiate file transfer`) between the workload signatures in Cluster A and the average workload signature is statistically significant (i.e., $p < 0.01$) and large (i.e., $d > 1.4$). Therefore, our approach identifies one workload signature (i.e., the workload signature representing the user `Dan`) as a key difference between the performance test and the field of our working example. In particular, the `initiate file transfer` event is not well represented in the test (in fact it does not occur at all). Performance analysts should then adjust the workload intensity of the file transfer functionality in the test.

In our simple working example, performance analysts could have examined how many times each execution event had occurred and identified events that occur much more frequently in the test compared to the field. However, manual analysis is not feasible in practice. For example, our enterprise case studies contain hundreds of different types of execution events and millions of log lines. Further, some execution events have a different impact on the system's behaviour based on the manner in which the event is executed. For example, our second enterprise case study identifies execution events that only causes errors when over-stressed by an individual user (i.e., one user executing the event 1,000 times has a different impact on the system's behaviour than 100 users each executing the event 10 times). Therefore, in practice performance analysts cannot simply examine event occurrence frequencies.

3.4 Case Studies

This section outlines the setup and results of our case studies. First, we present two case studies using Hadoop applications. We then discuss the results of three case studies using an enterprise system. Table 3.8 outlines the systems and data sets used in our case studies.

Our case studies aim to determine whether our approach can detect workload signature differences due to feature, intensity and issue differences between a performance test and the field. Our case studies include systems whose users are either human (Enterprise System) or software (Hadoop) agents.

We compare our results with the current state-of-the-practice. Currently, performance analysts validate performance tests by comparing the number of times each execution event has occurred during the test compared to the field and investigating any differences (Alspaugh et al., 2014; Cataldo et al., 2013; Hassan et al., 2008; Rosa, 2016). Therefore, we rank the events based on the difference in occurrence between the test and the field. We then investigate the events with the largest differences. In practice, performance analysts do not know how many of these events should be investigated. Therefore, we examine the same number of events as our approach identifies such that the effort required by performance analysts to manually analyze the events flagged by either 1) our approach or 2) the state of the practice is approximately equal. For example, if our approach flags 10 execution events, we examine the top 10 events ranked by the state-of-the-practice. We then compare the precision of our approach to the state-of-the-pratice. We define precision as the percentage of execution events that our approach identified as meaningful differences between the system's behaviour in the field and during

Table 3.8: Case Study Subject Systems.

	Hadoop		**Enterprise System**		
Application domain	Data processing		Telecom		
License	Open-source		Enterprise		
Performance Test Data					
# Log Lines	3,862	6,851	169,627	9,295,418	6,788,510	2,341,174
Notes	Performance test driven by the Hadoop WordCount application.	Performance test driven by the Hadoop WordCount application.	Performance test driven by the Hadoop Exotic Songs application.	Use-case performance test driven by a load generator.	Performance test driven by a replay script.	Field-representative performance test driven by a replay script.
Field Data						
# Log Lines	6,120	45,262	173,235	6,788,510	7,383,738	2,517,558
Notes	Machine failure in the field.	Java heap space exception in the field.	Performance degradation in the field.	No errors in the field.	Crash in the field.	No errors in the field.
Case Study						
Case study name	Machine failure in the field.	Java heap space error in the field.	LZO Compression enabled in the field.	Comparing use-case performance tests to the field.	Comparing replay performance tests to the field.	Comparing field-representative replay performance tests to the field.
Type of Differences	Issue difference	Issue difference	Feature difference	Intensity and feature differences	Intensity difference	No difference
Results						
Influential Events	12	3	2	28	5	0
Precision	91.7%	66.7%	100%	92.9%	100%	*100%
Precision (State-of-the-Practice)	58.3%	66.7%	100%	42.9%	0%	*100%
Precision (Statistical Comparison)	0%	0%	100%	14.9%	0%	0%

*no execution events were flagged because the field and the test do not differ.

the test that multiple, independent system experts confirmed are meaningful differences between the system's behaviour in the field and during the test. These experts have several years of experience using these logs to detect and diagnose performance problems in these systems. Therefore, they have deep knowledge of how the execution logs are used to understand the system's behaviour in practice.

We also compare our results to the results of a basic statistical comparison of the execution logs from a performance test and the field. We use the same statistical measures that are outlined in Section 3.3.4.2 (i.e., *t-tests* and *Cohen's d*) to statistically compare the number of times each execution event has occurred during the test compared to the field. This statistical comparison is identical to our approach when one workload signature representing the aggregated user behaviour is generated from the test and another from the field (i.e., our approach without clustering). We flag all events with a statistically significant (i.e., $p < 0.01$) and large (i.e., $d > 1.4$) difference between the test and the field. This comparison demonstrates the value added by our approach compared to a simple statistical comparison of the execution logs.

3.4.1 Hadoop Case Study

3.4.1.1 The Hadoop Platform

Our first case study system are two applications that are built on the Hadoop platform. Hadoop is an open-source distributed data processing platform that implements MapReduce (Dean and Ghemawat, 2008; Hadoop, 2014).

MapReduce is a distributed data processing framework that allows large amounts of data to be processed in parallel by the nodes of a distributed cluster of machines

(Dean and Ghemawat, 2008). The MapReduce framework consists of two steps: a Map step, where the input data is divided amongst the nodes of the cluster, and a Reduce step, where the results from each of the nodes is collected and combined.

Operationally, a Hadoop application may contain one or more MapReduce steps (each step is a "Job"). Jobs are further broken down into "tasks," where each task is either a Map task or a Reduce task. Finally, each task may be executed more than once to support fault tolerance within Hadoop (each execution is an "attempt").

3.4.1.2 The WordCount Application

The first Hadoop application used in this case study is the WordCount application (MapReduce Tutorial, 2014). The WordCount application is a standard example of a Hadoop application that is used to demonstrate the Hadoop platform and the MapReduce framework. The WordCount application reads one or more text files (a corpus) and counts the number of times each unique word occurs in the corpus.

Machine Failure in the Field

We monitored the performance of the Hadoop WordCount application during a performance test. The performance test workload consisted of 3.69 GB of text files (i.e., the WordCount application counts the number of times each unique word occurs in these text files). The cluster contains five machines, each with dual Intel Xeon E5540 (2.53GHz) quad-core CPUs, 12GB memory, a Gigabit network adaptor and SATA hard drives. While this cluster is small by industry standards (Chen et al., 2012), recent research has shown that almost all failures can be reproduced on three machines (Yuan et al., 2014).

We then monitored the performance of the Hadoop WordCount application in the field and found that the performance was much less than expected based on our performance tests. We found that the throughput (completed attempts/sec) was much lower than the throughput achieved during testing and that the average network I/O (bytes/sec transfered between the nodes of the cluster) was considerably lower than the average historical network I/O. Therefore, we compare the execution logs from the field and the test to determine whether our tests accurately represent the current conditions in the field.

We apply our approach to the execution logs collected from the WordCount application in the field and during the test. We generate a workload signature for each attempt because these attempts are the "users" of the Hadoop platform. These workload signatures represent the individual user behaviour discussed in Section 3.3.2.2. We also generate workload signatures for each 1 minute, 3 minute and 5 minute time interval. These workload signatures represent the aggregated user behaviour discussed in Section 3.3.2.2.

Our approach identifies 12 workload signature differences (i.e., execution events that best describe the differences between the field and the test) for analysis by system experts. We only report a selection of these execution events here for brevity.

```
INFO org.apache.hadoop.hdfs.DFSClient:
    Abandoning block blk_id
```

```
INFO org.apache.hadoop.hdfs.DFSClient:
    Exception in createBlockOutputStream java.io.IOException:
    Bad connect ack with firstBadLink ip_address

WARN org.apache.hadoop.hdfs.DFSClient:
    Could not get block locations.  Source file - Aborting...

INFO org.apache.hadoop.mapred.TaskRunner:
    Runnning cleanup for the task
```

These execution events indicate that the WordCount application 1) cannot retrieve data from the Hadoop File System (HFS), 2) has a "bad" connection with the node at `ip_address` and 3) cannot reconnect to the datanode (datanodes store data in the HFS) at `ip_address`. The remaining execution events are warning messages associated with this error. Made aware of this issue, performance analysts could update their performance tests to test how the system responds to machine failures and propose redundancy in the field.

The last execution event is a clean-up event (e.g., removing temporary output directories after the job completes)(OutputCommitter, 2016). This execution event occurs more frequently in the field compared to the test because a clean-up is always run after an attempt fails (MapReduce Tutorial, 2014). However, system experts do not believe that this is a meaningful difference between the system's behaviour in the field and during the test. Hence, we have correctly identified 11 events out of the 12 flagged events. The precision of our approach is 91.7%.

We also use the state-of-the-practice approach (outlined in Section 3.4) to identify the execution events with the largest occurrence frequency difference between the field and the test. We examine the top 12 execution events ranked by largest difference in occurrence in the field compared to the test. We find that 7 of these events describe important differences between the field and the test (all of these events were found by our approach) However, 5 of these events do not describe important differences between the field and the test (e.g., the clean-up or a start up event such as initializing JVM metrics (Metrics 2.0, 2016)). Therefore, the precision of the state-of-the-practice is 58.3% (i.e., 7/12).

We also use a statistical comparison of the execution logs (outlined in Section 3.4) to identify the execution events that differ between the field and the test. However, no events were flagged using this method. Therefore, a statistical comparison of the execution logs from a test and the field does not provide performance analysts with any insight into the differences between the field and the test.

Java Heap Space Error in the Field

We monitored the performance of the Hadoop WordCount application during a performance test. The performance test workload consisted of 15GB of text files.

We then monitored the Hadoop WordCount application in the field and found that the throughput (completed attempts/sec) was much lower than the throughput achieved during testing. We also found that the ratio of completed to failed attempts was much lower (i.e., more failed attempts relative to completed attempts) in the field compared to our performance test. Therefore, we compared the execution logs from the test and the field to determine whether our tests accurately represent the current conditions in the field.

Our approach identifies the following 3 workload signature differences:

```
FATAL org.apache.hadoop.mapred.Child:  Error running child:
    java.lang.OutOfMemoryError:  Java heap space
    at org.apache.hadoop.io.Text.setCapacity(Text.java:240)
    at org.apache.hadoop.io.Text.append(Text.java:216)
    at org.apache.hadoop.util.LineReader.readLine
     (LineReader.java:159)
    at org.apache.hadoop.mapred.LineRecordReader.next
     (LineRecordReader.java:133)
    at org.apache.hadoop.mapred.LineRecordReader.next
     (LineRecordReader.java:38)
    at org.apache.hadoop.mapred.MapTask$TrackedRecordReader.moveToNext
    (MapTask.java:236)
    at org.apache.hadoop.mapred.MapTask$TrackedRecordReader.next
     (MapTask.java:216)
    at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:48)
    at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:436)
    at org.apache.hadoop.mapred.MapTask.run(MapTask.java:372)
    at org.apache.hadoop.mapred.Child$4.run(Child.java:255)
    at java.security.AccessController.doPrivileged(Native Method)
    at javax.security.auth.Subject.doAs(Subject.java:415)
    at org.apache.hadoop.security.UserGroupInformation.doAs
     (UserGroupInformation.java:1121)
    at org.apache.hadoop.mapred.Child.main(Child.java:249)
```

```
 INFO org.apache.hadoop.mapred.Task:
    Aborting job with runstate:  FAILED


INFO org.apache.hadoop.mapred.Task:
    Cleaning up job
```

These execution events indicate that the WordCount application 1) suffers a `java.lang.OutOfMemoryError` and 2) the `java.lang.OutOfMemoryError` causes attempts to fail. When performance analysts consult with the official Hadoop documentation, they find that input files are split using line-feeds or carriage-returns (TextInputFormat, 2016). Further, when performance analysts examine the input files that the Hadoop WordCount application fails to process, they find that these files lack line-feeds or carriage-returns due to a conversion error between DOS and UNIX. Made aware of this issue, performance analysts could configure a maximum line size using RecordReader (RecordReader, 2016) to prevent this error in the field.

As before, the last execution event is a clean-up event that system experts do not believe is a meaningful difference between the system's behaviour in the field and during the test. Hence, we have correctly identified 2 events out of the 3 flagged events. Therefore, the precision of our approach is 66.7%.

We also use the state-of-the-practice approach to identify execution events with the largest occurrence frequency difference between the field and the test. We find that the state-of-the-practice flags the same events as our approach. Therefore, the precision of the state-of-the-practice 66.7% (i.e., 2/3).

We also use a statistical comparison to identify the execution events that differ between the field and the test. A statistical comparison of the execution events flags 19 events. These events describe the lack of successful processing of all input files in the field compared to the test. For example, the `INFO org.apache.hadoop.mapred.Task: attempt_id is done. And is in the process of committing` event occurs much more frequently in the test compared to the field. Therefore, the precision of the statistical comparison is 0% because these events do not describe the most important differences between the field and the test (i.e., the events related to the `OutOfMemoryError` event).

3.4.1.3 The Exotic Songs Application

The second Hadoop application used in this case study is the Exotic Songs application (Kawa, 2012). The Exotic Songs application was developed to leverage the Million Songs data set (Million Song Dataset, 2012). The Million Songs data set contains metadata for one million different songs (the data set is 236GB and does not include the actual songs). The data set was developed 1) to encourage research on scalable algorithms and 2) to provide a benchmark data set for evaluating algorithms (Million Song Dataset, 2015). The Exotic Songs application analyzes the Million Song data set to find "exotic" (i.e., popular songs produced by artists that live far away from other artists) songs.

Compression Enabled in the Field

We monitored the performance of the Hadoop Exotic Songs application during a performance test. The performance test workload consisted of the full Millions Songs data set. We followed the following Microsoft TechNet blog to deploy the

underlying Hadoop cluster (Klose, 2014). The cluster contains 1) one DNS server, 2) one master node and 3) ten worker nodes.

We are grateful to Microsoft for 1) providing us access to such a large-scale deployment and 2) working closely with us to setup and troubleshoot our deployment.

We then monitored the Hadoop Exotic Songs application in the field and found that the throughput (completed attempts/sec) was much lower than the throughput achieved during testing. We also found that the CPU usage was much higher in the field compared to our performance test. Therefore, we compare the execution logs from the test and the field to determine whether our tests accurately represent the current conditions in the field.

Our approach identifies the following two workload signature differences:

```
INFO com.hadoop.compression.lzo.GPLNativeCodeLoader:
    Loaded native gpl library

INFO com.hadoop.compression.lzo.LzoCodec:
    Successfully loaded & initialized native-lzo library
```

These execution events indicate that the Exotic Songs application is loading the Hadoop LZO compression libraries in the field. LZO is a fast and lossless data compression algorithm that is widely used in the field. The Hadoop LZO compression libraries support 1) splitting LZO files for distributed processing and 2) (de)compressing streaming data (input and output streams) (Hadoop-LZO, 2011). Made aware of this issue, performance analysts could configure compression during

performance testing to better understand the performance of their system. Hence, we have correctly identified 2 events out of the 2 flagged events. Therefore, the precision of our approach is 100%.

We also use 1) the state-of-the-practice approach and 2) a statistical comparison to identify execution events with the largest occurrence frequency difference between the field and the test. We find that these approaches both flag the same events as our approach. Therefore, the precision of these approaches is 100% (i.e., 2/2).

3.4.2 Enterprise System Case Study

Although our Hadoop case study was promising, we perform three case studies on an enterprise system to examine the scalability of our approach. We note that these data sets are much larger than our Hadoop data set (see Table 3.8).

3.4.2.1 The Enterprise System

Our second system is a large-scale enterprise software system in the telecommunications domain. For confidentiality reasons, we cannot disclose the specific details of the system's architecture, however the system is responsible for simultaneously processing millions of client requests and has very high performance requirements.

Performance analysts perform continuous performance testing to ensure that the system continuously meets its performance requirements. Therefore, analysts must continuously ensure that the performance tests accurately represent the current conditions in the field.

3.4.2.2 Comparing Use-Case Performance Tests to the Field

Our first enterprise case study describes how our approach was used to validate a use-case performance test (i.e., a performance test driven by a workload generator) by comparing the system behaviour during the test and in the field. A workload generator was configured to simulate the individual behaviour of thousands of users by concurrently sending requests to the system based on preset use-cases. The system had recently added several new clients. To ensure that the existing use-cases accurately represent the workloads driven by these new clients, we use our approach to compare a test to the field.

We use our approach to generate workload signatures for each user in the field and the test. We also generate workload signatures for each 1 minute, 3 minute and 5 minute interval. We then compare the workload signatures from the field to the test. Our approach identifies 28 execution events. These results were then given to multiple, independent system experts who confirmed:

1. 24 events are under-stressed in the test relative to the field. In general, these events relate to variations in the intensity (i.e., changes in the number of events per second) of events in the field compared to the relatively steady-state of the test.

2. 2 events are over-stressed in the test relative to the field.

3. 2 events are artifacts of the difference in configuration between the test and field environments (i.e., these events correspond to communication between the system and an external system that only occurs in the field) and are not important differences between the test and the field.

In summary, our approach correctly identifies 26 execution events (92.9% precision) that correspond to important differences between the system's behaviour during the test and in the field. Such results can be used to improve the tests in the future (i.e., by tuning the use-cases and the workload generator to more accurately represent the field conditions).

In contrast, the state-of-the-practice approach has a precision of only 42.9% and a statistical comparison flags 201 events with a precision of only 14.9%.

3.4.2.3 Comparing Replay Performance Tests to the Field

Our second enterprise case study describes how our approach was used to validate a performance replay test (i.e., a performance test driven by a replay script) by comparing the system's behaviour during the replay test to the system's behaviour in the field.

Replay scripts record the behaviour of real users in the field then play back the recorded behaviour during a replay test, where heavy instrumentation of the system is feasible. In theory, replay scripts can be used to perfectly replicate the conditions in the field during a replay test (Krishnamurthy et al., 2006). However, replay scripts require complex software to concurrently simulate the millions of users and billions of requests captured in the field. Therefore, replay scripts do not scale well and use-case performance tests that are driven by workload generators are still the norm (Meira et al., 2012).

Performance analysts monitoring the system's behaviour in the field observed a spike in memory usage followed by a system crash. We use our approach to understand the cause of this crash, and why this problem was not discovered during

testing. Our approach identifies 5 influential execution events that differ between the workload signatures of the replay test and the field.

These results were given to performance analysts who confirmed that these 5 events are under-stressed in the replay test relative to the field. In particular, these events cause errors when over-stressed by an individual user (i.e., one user executing the event 1,000 times has a different impact on the system's behaviour than 100 users each executing the event 10 times). This type of behaviour cannot be identified from the occurrence frequencies or aggregate event counts.

In summary, our approach correctly identifies 5 influential execution events that correspond to differences between the system's behaviour during the replay test and in the field. Using this information, performance analysts update their replay tests. They then see the same behaviour during testing as in the field. Therefore, our results provide performance analysts with a concrete recommendation to help diagnose the cause of this crash.

In contrast, the state-of-the-practice approach has a precision of 0% and a statistical comparison flags 5 events with a precision of 0%.

3.4.2.4 Comparing Field-Representative Performance Tests and the Field

Our third enterprise case study describes how our approach can validate a field-representative performance replay test (i.e., a performance test driven by a replay script) by comparing the system behaviour across a performance test and the field. This test is known to be field-representative because it was successfully used to replicate a performance issue (i.e., a memory leak) in the field.

We use our approach to validate whether this test is truly representative of the field. Our approach identifies that no execution events differ between the workload signatures of the replay test and the field. Therefore, our results provide performance analysts with confidence in this test.

As our approach did not identify any execution events that differ between the workload signatures of the replay test and the field, we cannot compare our approach against the state-of-the-practice (i.e., the state-of-the-practice would examine the top 0 events). However, a statistical comparison of the execution events flags 2 events. These 2 events are artifacts of the test (i.e., these events correspond to functionality used to setup the tests) and are not important differences between the test and the field. Therefore, the precision of a statistical comparison is 0%.

Our approach flags events with an average precision of 92%, outperforming the state-of-the-practice approach and the statistical comparison approach.

3.5 Discussion

3.5.1 Comparison to Other Approaches

Our case studies show that our approach performs well (average precision of 90%) when detecting differences between the execution logs describing the test and field workloads. In particular, our approach outperforms 1) the state-of-the-practice (average precision 53.6%) and 2) a basic statistical comparison of the execution logs from a test and the field (average precision of 3.0%).

One reason for this outperformance may be that our approach breaks the workloads into workload signatures that represent two complementary components of the workload (i.e., the individual and aggregated user behaviour). Our approach then clusters the workload signatures and detects outlying clusters. Finally, our approach uses *unpaired two-sample two-tailed t-tests* to detect workload signature differences and *Cohen's d* to to filter unimportant differences. However, we used the same statistical tests to compare the workloads without breaking them down into workload signatures. The precision of our approach (90%) is considerably greater than these statistical tests alone (3%).

3.5.2 Formulations of the Workload Signature

Our approach also uses two complimentary formulations of the workload signatures (i.e., workload signatures representing 1) the individual user behaviour and 2) the aggregated user behaviour). These two formulations are able to detect different types of workload differences.

Workload signatures that represent the aggregated user behaviour are able to identify a set of execution events that occur together in the field, but not in the test. In our two Hadoop case studies, all of the execution events that were flagged were identified as signature differences between the workload signatures that represent the aggregated user behaviour. (i.e., the workload signatures representing the individual user behaviour were not able to detect workload differences between the field and the test). This is not surprising because the cause of the workload differences (i.e., failure of the Hadoop HDFS datanode in the field and no line-feeds or carriage-returns in any of the input files in the field) affect all users (i.e., the

attempts). Similarly, in our first enterprise case study, all of the execution events that were flagged were identified as signature differences between the workload signatures that represent the aggregated user behaviour. This is also not surprising because most of these differences relate to variations in the intensity (i.e., changes in the number of events per second) of events in the field in comparison to the relatively steady-state of the test.

Workload signatures that represent the individual user behaviour are able to identify users whose behaviour is seen in the field, but not in the test. In our second enterprise case study, all of the execution events that were flagged were identified as signature differences between the workload signatures that represent the individual user behaviour. This is not surprising because these differences relate to an event that causes errors when over-stressed by an individual user.

3.5.3 Sensitivity Analysis

The clustering phase of our approach relies on three different statistical measures: 1) a distance measure (to determine the distance between each workload signature), 2) a linkage criterion (to determine which clusters should be merged during the hierarchical clustering procedure) and 3) a stopping rule (to determine the number of clusters by cutting the hierarchical cluster dendrogram). We verify that these measures perform well when clustering workload signatures. Therefore, we determine the distance measure, linkage criterion and stopping rule that give our approach the highest precision using our Hadoop case study data (similar results hold for our other case studies). This analysis also serves to analyze the sensitivity of our results to changes in these measures.

3.5.3.1 Determining the Distance Measure

The agglomerative hierarchical clustering procedure begins with each performance signature in its own cluster and proceeds to identify and merge clusters that are "close." The "closeness" of two clusters is measured by some distance measure. The best known distance measure will result in a clustering that is closest to the manually assigned clusters.

We determine the distance measure by comparing the results obtained by our approach (i.e., the execution events that we flag) when different distance measures are used. Table 3.9 presents how the number of flagged events and the precision (the percentage of correctly flagged events) is impacted by several common distance measures (Cha, 2007; Frades and Matthiesen, 2009; Fulekar, 2008). From Table 3.9, we find that the Pearson distance produces results with higher precision than any other distance measure. Therefore, we use the Pearson distance to cluster workload signatures.

Table 3.9: Determining the Distance Measure

Distance Measure	**#Events**	**Precision**
Pearson distance	12	91.7%
Cosine distance	6	33.3%
Euclidean distance	29	72.4%
Jaccard distance	26	76.9%
Kullback-Leibler Divergence	33	72.7%

3.5.3.2 Determining the Linkage Criteria

The hierarchical clustering procedure takes a distance matrix and produces a dendrogram (i.e., a hierarchy of clusters). The abstraction from a distance matrix to a dendrogram results is some loss of information (i.e., the distance matrix contains the distance between each pair of workload signatures, whereas the dendrogram presents the distance between each cluster). The best known linkage criterion will enable the hierarchical clustering procedure to produce a dendrogram with minimal information loss.

We determine the linkage criterion by comparing the results obtained by our approach (i.e., the execution events that we flag) when different distance measures are used. Similar to our analysis of the distance measure, Table 3.10 presents how the number of flagged events and the precision is impacted by several common linkage criteria (Frades and Matthiesen, 2009; Tan et al., 2005). From Table 3.10 we find that the average linkage criterion produces results with higher precision than any other linkage criteria. Therefore, we use the average linkage criterion to cluster workload signatures.

Table 3.10: Determining the Linkage Criteria

Distance Measure	**#Events**	**Precision**
Average	12	91.7%
Single	24	79.2%
Ward	14	85.7%
Complete	0	NA

3.5.3.3 Determining the Stopping Rule

To complete the clustering procedure, dendrograms must be cut at some height so that each workload signature is assigned to only one cluster. Too few clusters will not allow outliers to emerge (i.e., they will remain nested in larger clusters) while too many clusters will lead to over-fitting and many false positives.

We determine the stopping rule by comparing the results obtained by our approach (i.e., the execution events that we flag) when different stopping rules are used. Similar to our analysis of the distance measure, Table 3.11 presents how the number of flagged events and the precision is impacted by several common linkage criteria (Milligan and Cooper, 1985).

Table 3.11: Determining the Stopping Rule

Distance Measure	**#Events**	**Precision**
Calinski-Harabasz	12	91.7%
Duda and Hart	0	NA
C-Index	31	71.0%
Gamma	29	69.0%
Beale	2	50%
Cubic Clustering Criterion	0	NA
Point-Biserial	13	92.3%
G(+)	29	69.0%
Davies and Bouldin	27	74.1%
Stepsize	0	NA

From Table 3.11, we find that the Calinski-Harabasz and Point-Biserial stopping rules have the greatest precision. We select the Calinski-Harabasz stopping rule because it is has a slightly higher precision than the Point-Biserial stopping rule across our other case studies. The Calinski-Harabasz stopping rule is also widely accepted as the best known stopping rule and used as the benchmark stopping rule.

3.6 Threats to Validity

This section outlines our threats to validity.

3.6.1 Threats to Construct Validity

3.6.1.1 The Sequencing of Events

Our approach does not explicitly consider the sequencing of events (although some of this information is represented in the workload signatures that represent the aggregate user behaviour). Therefore, our approach may not be able to detect differences in the sequencing of events during the test compared to the field. However, event sequencing requires reliable information regarding the timing of events within the system and timing information may not be reliable in large-scale distributed systems (Lamport, 1978). Fortunately, Lamport time stamps (Lamport, 1978) or vector clocks (Fidge, 1988; Mattern, 1988) may correct any inconsistencies in the machine's clocks.

3.6.1.2 Statistical Tests

Our approach relies on two statistical tests. First, we use a *one-sample upper-tailed z-test for a population proportion* to identify outlying clusters. We then use an *unpaired two-sample two-tailed Welch's unequal variances t-test* (along with *Cohen's d* effect size) to identify workload signature differences. These tests were able to identify meaningful differences between the system's behaviour during a performance test and the system's behaviour in the field in our case studies. However, these tests have assumptions that should be met for their proper application. Failure to satisfy these

assumptions may affect the precision of our approach and undermine the statistical robustness of our approach.

The *one-sample upper-tailed z-test for a population proportion* and the *unpaired two-sample two-tailed Welch's unequal variances t-test* assume that:

1. The sampling is independent and random – this assumption is satisfied because our approach considers 1) all of the execution logs to generate workload signatures, 2) all of the clusters to identify outlying clusters and 3) all of the unique execution events to identify signature differences (i.e., selecting one execution log, cluster or unique execution event has no influence on the selection of another execution log, cluster or unique execution event). Further, we assume that the test logs are independently and randomly sampled from the test (i.e., the test can replicated). We also assume that the field logs are independently and randomly sampled from the field (see Section 3.6.2.1).

2. The population is normally distributed – this assumption is satisfied by the central limit theorem (Elliott, 2006).

The *Welch's unequal variances t-test*, unlike the *Student's t-test*, does not assume that the sample sizes or the population variances are equal.

In addition to the underlying test assumptions, our use of statistical tests poses a second threat to validity. These tests may fail to correctly identify the most meaningful differences between a performance test and the field. For example, our thresholds (i.e., $p < 0.01$ and $d > 1.4$) may be too lenient (raising our recall, but lowering our precision) or too stringent (raising our precision, but lowering our recall). Future work should evaluate the impact of these thresholds on the precision and relative recall of our approach.

3.6.1.3 Evaluation

We evaluated our approach by determining the precision with which our approach flags execution events that differ between the workload signatures of a performance test and the field (i.e., the percentage of flagged events that are relevant to understanding the difference between the system's behaviour during a performance test and the system's behaviour in the field). While multiple, independent system experts have verified these results, we do not have a gold standard data set. Further, complete system knowledge is required to exhaustively enumerate every difference between a particular test and the field. Therefore, we cannot calculate the recall of our approach (i.e., the percentage of events that are relevant to understanding the difference between the system's behaviour during a performance test and the system's behaviour in the field that are flagged by our approach). However, our approach is intended to help performance analysts identify differences between a test and the field by flagging execution events for further analysis (i.e., to provide performance analysts with a starting point). Therefore, our goal is to maximize precision so that analysts have confidence in our approach. In our experience working with industry experts, performance analysts agree with this view (Hassan and Flora, 2007). Additionally, we were able to identify at least one execution event that differed between the test and the field in all of our case studies (except our enterprise case study where no differences were expected). Hence we were able to evaluate the precision of our approach in our case studies.

3.6.2 Threats to Internal Validity

3.6.2.1 Field-Representative Field Logs

Our approach determines whether a performance test is field-representative by comparing execution logs from the test and the field. The performance test may be designed to replicate the expected field workloads rather than the extremes of the field workloads. However, execution logs describing the field workload only covers a specific period of time (e.g., a few hours or a few days). Field workloads may have rare short-term workload anomalies that performance analysts may not want to replicate during performance testing. For example, during a 2013 movie premiere, Twitter's workload spiked to a record-setting 143,199 tweets per second compared to an average of 5,700 tweets per second (Krikorian, 2013). Such short-term workload anomalies may lead performance analysts to compare their performance test against a field workload that is not representative of the expected field workloads. Field workloads may also have significant variability throughout the day, week, month and year (Eldin et al., 2014; Poggi et al., 2010, 2014). For example, e-commerce websites experience significant seasonality (e.g., an increase in traffic around the holiday shopping season).

This threat can be mitigated by performance analysts. Performance analysts should use execution logs that describe the field workload over a period of operation that they want to replicate during performance testing. Performance analysts may also use execution logs that describe the field workload over longer periods of time (i.e., where short-term workload anomalies are less influential) because our

approach has been designed to scale. However, our approach can be used to compare performance test workloads to the field despite short-term workload anomalies. Such comparisons may help performance analysts understand and replicate field issues (e.g., our Hadoop case studies and our second enterprise case study).

3.6.2.2 Execution Log Quality/Coverage

Our approach generates workload signatures by characterizing a user's behaviour in terms of feature usage expressed by the execution events. However, it is possible that there are no execution logs to indicate when certain features are executed. Therefore, our approach is incapable of identifying these features in the event that their usage differs between a test and the field. However, this is true for all execution log based analysis, including manual analysis.

This threat may be mitigated by using automated instrumentation tools that would negate the need for developers to manually insert output statements into the source code. However, we leave this to future work as automated instrumentation imposes a heavy overhead on the system and is often unfeasible in the field (Bernat and Miller, 2011; Laurenzano et al., 2015; Maplesden et al., 2015; Uh et al., 2006). Further, Shang et al. report that execution logs are a rich source of information that capture developers' intuition and knowledge about a system's behaviour (Shang et al., 2011). Hence, automated instrumentation tools may not provide as deep an insight into the system's behaviour as execution logs.

3.6.2.3 Defining Users for Signature Generation

In our experience, large-scale software systems are typically driven by human agents. However, users may be difficult to define in systems that are driven by software agents (e.g., web services (Greenwood et al., 2007)) or when users are allowed to have multiple IDs. Defining the users of a particular system is a task for the system experts. However, such a determination only needs to be made the first time our approach is used, afterwards this may be definition is reused. Further, the users of a particular system are typically defined when the system is being designed and developed because users, or "actors," are essential to Unified Modelling Language (UML) use case diagrams (Object Management Group, 2016). The UML 2.0 specification states: *[u]se cases are a means for specifying required usages of a system. Typically, they are used to capture the requirements of a system, that is, what a system is supposed to do. The key concepts associated with use cases are* actors*,* use cases*, and the* subject*. The subject is the system under consideration to which the use cases apply. The users and any other systems that may interact with the subject are represented as actors. Actors always model entities that are outside the system. The required behavior of the subject is specified by one or more use cases, which are defined according to the needs of actors* (Object Management Group, 2016). The UML 2.0 specification further states: *[a]n Actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is* external *to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). Actors may represent roles played by human users, external hardware, or other subjects* (Object Management Group, 2016). Therefore, users can be defined for every large-scale software system.

3.6.2.4 Defining the Aggregated Users for Signature Generation

Our approach generates workload signatures that represent the aggregated users are generated by grouping the execution logs into time intervals (i.e., grouping the execution logs that occur between two points in time). In our case studies, we used multiple time intervals (i.e., 1, 3 and 5 minute time intervals) to generate these workload signatures. However, these time intervals may not produce the optimal results (e.g., using a 90 second time interval may have resulted in higher precision). We have mitigated this threat by using multiple time intervals that are known to generate accurate workload signatures for the particular systems in our case studies (Syer et al., 2013). However, it is possible that these time intervals are specific to the systems in our case studies.

3.6.3 Threats to External Validity

3.6.3.1 Generalizing Our Results

The studied software systems represent a small subset of the total number of large-scale software systems. Therefore, it is unclear how our results will generalize to additional software systems, particularly systems from other domains (e.g., e-commerce). However, our approach does not assume any particular architectural details. Hence, there is no barrier to our approach being applied to other systems. Further, we evaluated our approach on two different systems: 1) an open-source distributed data processing system and 2) an enterprise telecommunications system that is widely used in practice.

The clustering phase of our approach relies on three different statistical measures: 1) a distance measure (to determine the distance between each workload signature), 2) a linkage criterion (to determine which clusters should be merged during the hierarchical clustering procedure) and 3) a stopping rule (to determine the number of clusters by cutting the hierarchical cluster dendrogram). We evaluated several possible distance measures, linkage criterion and stopping rules by determining which ones gave our approach the highest precision. While the same set of statistical measures performed well across our five case studies, these measures may not generalize to other software systems. However, the main contribution of our work is the overall approach to comparing performance tests to the field, rather than determining a universal distance measure, linkage criteria and stopping rule for clustering workload signatures.

Our approach may not perform well on small data sets (where we cannot generate many workload signatures) or data sets where one set of execution logs (either the test or field logs) is much larger than the other. However, the statistical measures that we have chosen are invariant to scale. Further, we evaluated our approach on small data sets. In our first Hadoop WordCount case study, the logs from the performance test only have 3,862 execution events (millions or billions of events are expected in large Hadoop deployments (Chen et al., 2012)). We also evaluated our approach on a data set where one set of execution logs is much larger, on a relative and/or absolute basis, than the other. In our second Hadoop WordCount case study, the field logs are 6.6 times larger than the test logs and in our first enterprise case study, the field logs contain 2.5 million more execution events than the test logs.

3.7 Conclusions

In the previous chapter, we surveyed the literature on 1) characterizing a system's behaviour and 2) comparing a system's behaviour during performance testing to its historical behaviour. We found that prior work does not fully leverage the execution logs nor does it enrich such logs with additional sources of valuable information. Therefore, this chapter presented our first approach for enriching the execution events by leveraging *both* the static components of the execution logs *and* the time stamps and worker IDs in the execution logs. Our approach validates a performance test by deriving workload signatures from performance tests and the field using execution events that have been enriched with the time stamps and worker IDs in the execution logs. We then use statistical techniques to identify differences between the workload signatures from the field and the event signatures from the performance test. Such differences may be used by performance analysts to update their tests to more accurately represent the field workloads.

We performed six case studies on two systems: one open-source system and one enterprise system. Our case studies explored how our approach can be used to identify feature differences, intensity differences and issue differences between performance tests and the field. Performance analysts and system experts have confirmed that our approach provides valuable insights that help to validate their tests and to support the continuous performance testing process.

In the next chapter, we present an approach for enriching the execution events by leveraging *both* the static components of the execution logs *and* the dynamic information in the execution logs.

CHAPTER 4

Enriching the Execution Events by Leveraging the Dynamic Information in Execution Logs

Performance issues are the primary cause of failures in today's large-scale software systems. Therefore, performance analysts devote significant time and resources to certification testing (i.e., performance testing using a field-representative workload) their systems to ensure that they meet their release certification benchmark (i.e., that they comply with the service level agreements). Yet diagnosing performance issues remains a major challenge because these systems are highly configurable and constantly evolving. Current approaches to diagnosing performance issues require considerable manual effort and a high degree of system-specific expertise to review gigabytes execution logs to understand the system's behaviour.

In this chapter, we propose an approach to enrich the execution events by leveraging the information in the execution logs to compare the system's behaviour in the field to its behaviour during certification testing. Such a comparison should help performance analysts to understand the cause of any performance issues that are observed in the field. We perform four case studies on two large systems: 1) one open-source system and 2) one enterprise system. Our approach identifies all of the important differences between the system's behaviour in the field compared to its behaviour during certification testing.

4.1 Introduction

The increasing importance of large-scale software systems (e.g., Amazon.com and Google's GMail) poses new challenges for the software performance engineering field (Software Engineering Institute, 2006). These systems are highly configurable (e.g., Hadoop contains 875 different configuration options (Hadoop, 2016)) and constantly evolving (e.g., eBay changes over 100,000 lines of code every week (Hofer, 2011)). Failures in these systems are often associated with performance issues, rather than with feature bugs (Compuware, 2006; Dean and Barroso, 2013; Simic and Conklin, 2012; Weyuker and Vokolos, 2000). Therefore, certifying that these systems comply with their service level agreements (e.g., throughput, response time or mean time to failure requirements) has become essential to ensuring the problem-free operation of these systems (Burger and Reussner, 2011; Hassan and Zhang, 2006; Nivas and Csallner, 2011).

Performance analysts certify that their systems comply with their service level agreements with a "certification test." A certification test is a performance test using

a field-representative workload. Therefore, performance analysts can monitor how the system behaves during certification testing to ensure that the system is able to comply with its service level agreements under a realistic workload. The results of the certification test then form a baseline of the system's behaviour. Performance analysts can use this baseline to diagnose performance issues observed in the field.

Current approaches to diagnosing performance issues require considerable manual effort and a high degree of system-specific expertise to review gigabytes execution logs to understand the system's behaviour. However, documentation describing the expected system behaviour is rarely up-to-date (Ernst et al., 2015; Holvitie et al., 2014; Parnas, 1994). Fortunately, execution logs, which record notable events at runtime, are readily available in most large-scale systems to support remote issue resolution and legal compliance. Further, these logs contain developer and operator knowledge (i.e., they are manually inserted by developers) whereas instrumentation tends to view the system as a black-box (Shang et al., 2011, 2014). Hence, execution logs are often the most sensible data available to describe and monitor a system's behaviour. However, existing approaches to analyzing execution logs are limited by 1) the overhead of the approach in practice (Stearley, 2004; Vaarandi, 2003; Xu et al., 2009a), 2) the lack of system knowledge by performance analysts (Damásio et al., 2002; Jiang et al., 2008a,b), 3) the lack of consistent log formats (Rabkin et al., 2010), 4) the large variation in the number and frequency of execution logs (Nagappan and Vouk, 2010; Vaarandi, 2003) and 5) the inaccessibility and difficulty of parsing source code (Xu et al., 2009a,b, 2010). These approaches parse the execution logs to execution events by identifying and removing the dynamic information in the execution logs. For example, `Receiving file: 1MB of 7MB`

and `Receiving file: 1MB of 9GB` are both parsed to `Receiving file: ___ of ___`. Therefore, performance analysts cannot distinguish between `Receiving file: 1MB of 7MB` and `Receiving file: 1MB of 9GB` after the execution logs have been parsed. However, these two events likely have very different impacts on the system's behaviour (i.e., receiving a 9GB file is likely to take more time and resources than receiving a 7MB file).

In this chapter, we propose an automated approach to diagnose performance issues by comparing the system's behaviour in the field to its behaviour during certification testing. Our approach enriches the execution events by leveraging the dynamic information in the execution logs. We derive event signatures from execution logs, then uses statistical techniques to identify differences between the event signatures from the field and the event signatures from the certification test. Such differences should help performance analysts to understand the cause of any performance issues observed in the field.

This chapter makes four contributions:

1. We present an approach to enrich the execution events by leveraging the dynamic information in the execution logs.

2. We demonstrate a novel approach for parsing execution logs into their static and dynamic components.

3. We demonstrate an automated approach to identify performance deviations from certification tests by comparing the system's behaviour in the field to its behaviour during a certification test. Our approach can identify important execution events that best explain the differences between a system's behaviour in the field to its behaviour during a certification test.

4. Through four case studies, we show that our approach is scalable and can help performance analysts to understand the cause of any performance issues observed in the field.

4.1.1 Organization of the Chapter

This chapter is organized as follows: Section 4.2 provides a motivational example of how our approach may be used in practice. Section 4.3 describes our approach in detail. Section 4.4 presents our case studies. Section 4.5 outlines the threats to validity. Finally, Section 4.6 concludes the chapter.

4.2 Motivational Example

Peter, an operator of a large-scale distributed file system, observes a performance issue in his system when it is deployed in the field. The system's throughput (i.e., the number of successfully completed file transfers per second) in the field is much lower than the expected throughput from the system's certification test.

Pressured by time (given the competitive marketplace), management (who are keen to boast a high quality system) and the complexity of analyzing the large volume of unstructured data collected from his system, Peter is introduced to an automated approach that can diagnose performance issues observed in the field by comparing the system's behaviour in the field compared to its behaviour during certification testing. This approach enriches the execution events by leveraging the dynamic information in the execution logs, derives event signatures from the enriched execution events then compares the signatures from the field against the

signatures from the certification test to identify execution events that explain the performance issue that Peter observed in the field. Thus, this approach provides Peter with a concrete starting point (i.e., specific execution events) for his analysis.

Using this approach, Peter discovers that his system is breaking files into larger "chunks" in the field compared to during the performance test than expected. Files are broken into chunks to improve the system's ability to tolerate and recover from file transfer failures (i.e., the system only needs to resend the current file chunk rather than the entire file when file transfer failures occur). File transfer failures occur more often when transferring larger chunks because these transfers take more time. Hence, transferring larger chunks decreases the number of successfully completed file transfers per second. Therefore, Peter was able to identify the cause of the system's performance issue.

4.3 Approach

This section outlines our approach to enrich the execution events by leveraging the dynamic information in the execution logs and diagnose performance issues observed in the field. Figure 4.1 provides an overview of our approach. First, we identify the static and dynamic components of the execution logs. Second, we enrich the execution events by leveraging the dynamic information in the execution logs to create event signatures that describe the occurrences of each execution event. Finally, we analyze each event signature to identify the execution events that correspond to meaningful differences between the certification test and the field. We will describe each phase in detail and demonstrate our approach with a working example of a hypothetical web scraping application.

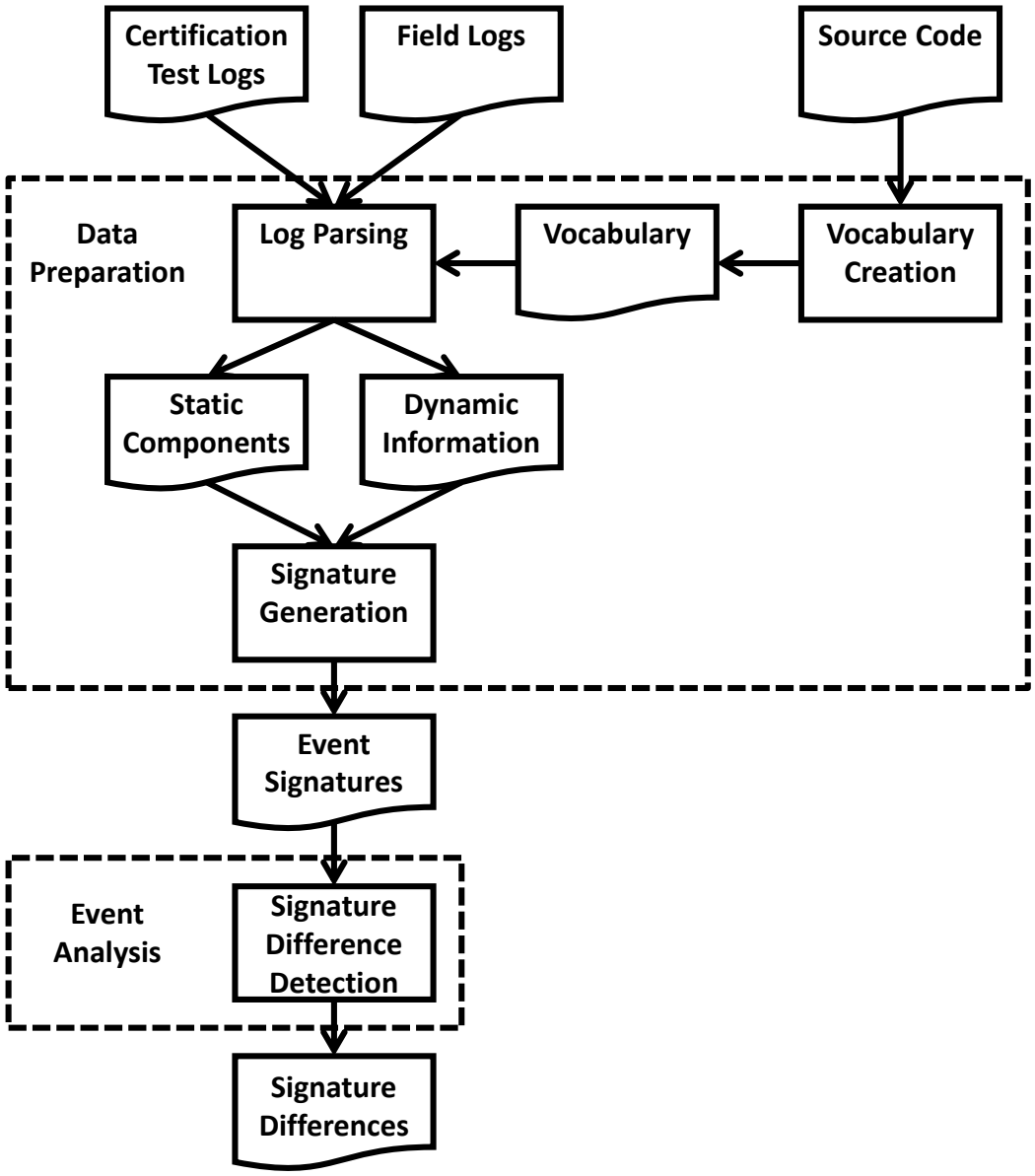

Figure 4.1: An Overview of Our Approach.

4.3.1 Execution Logs

The second column of Table 4.1 and Table 4.2 presents the execution logs from our working example. These execution logs contain both static information (e.g., `Connecting to` and `URL contained`) and dynamic information (e.g., `www.2-tokens.com` and `2`) that changes with each occurrence of an event. Table 4.1 and Table 4.2 present the execution logs from the field and the certification test respectively.

4.3.2 Source Code

Listing 4.1 partially shows the source code from our example.

4.3.3 Data Preparation

Execution logs are difficult to analyze because they are unstructured. Therefore, we parse the execution logs to execution events to enable automated statistical analysis. We then generate event signatures that represent the occurrences of each execution event.

4.3.3.1 Log Parsing

We parse the execution logs to abstract the execution logs to execution events while retaining the dynamic information in the execution logs. However, existing approaches to parse execution logs are limited by 1) the overhead of the approach in practice (Stearley, 2004; Vaarandi, 2003; Xu et al., 2009a), 2) the lack of system knowledge by performance analysts (Damásio et al., 2002; Jiang et al., 2008a,b),

Table 4.1: Parsing Execution Logs to Execution Events: Execution Logs from the Field

Time	Log Line	Static Components	Dynamic Information	Execution Event ID
00:01	`Connecting to www.73-tokens.com`	`Connecting to ___`	`www.73-tokens.com`	1
00:02	`URL contained 73 tokens`	`URL contained ___ tokens`	`73`	2
00:03	`Disconnecting from www.73-tokens.com`	`Disconnecting from ___`	`www.73-tokens.com`	3
00:03	`Connecting to www.87-tokens.com`	`Connecting to ___`	`www.87-tokens.com`	1
00:04	`URL contained 87 tokens`	`URL contained ___ tokens`	`87`	2
00:04	`Disconnecting from www.87-tokens.com`	`Disconnecting from ___`	`www.87-tokens.com`	3
00:06	`Connecting to www.99-tokens.com`	`Connecting to ___`	`www.99-tokens.com`	1
00:12	`URL contained 99 tokens`	`URL contained ___ tokens`	`99`	2
00:13	`Disconnecting from www.99-tokens.com`	`Disconnecting from ___`	`www.99-tokens.com`	3

Table 4.2: Parsing Execution Logs to Execution Events: Execution Logs from a Certification Test

Time	Log Line	Static Components	Dynamic Information	Execution Event ID
00:01	`Connecting to www.2-tokens.com`	`Connecting to ___`	`www.2-tokens.com`	1
00:02	`URL contained 2 tokens`	`URL contained ___ tokens`	`2`	2
00:02	`Disconnecting from www.2-tokens.com`	`Disconnecting from ___`	`www.2-tokens.com`	3
00:03	`Connecting to www.4-tokens.com`	`Connecting to ___`	`www.4-tokens.com`	1
00:04	`URL contained 4 tokens`	`URL contained ___ tokens`	`4`	2
00:04	`Disconnecting from www.4-tokens.com`	`Disconnecting from ___`	`www.4-tokens.com`	3
00:06	`Connecting to www.3-tokens.com`	`Connecting to ___`	`www.3-tokens.com`	1
00:07	`URL contained 3 tokens`	`URL contained ___ tokens`	`3`	2
00:07	`Disconnecting from www.3-tokens.com`	`Disconnecting from ___`	`www.3-tokens.com`	3
00:09	`Connecting to www.2-tokens.com`	`Connecting to ___`	`www.2-tokens.com`	1
00:10	`URL contained 2 tokens`	`URL contained ___ tokens`	`2`	2
00:11	`Disconnecting from www.2-tokens.com`	`Disconnecting from ___`	`www.2-tokens.com`	3

Listing 4.1: Sample Source Code.

```
1  import java.io.InputStreamReader;
2  import java.net.URL;
3  import java.net.URLConnection;
4  import java.util.Scanner;
5
6  Web Scraping Application
7  public class WebScraper {
8        WebScraper logger
9    static Logger log = Logger.getLogger(WebScraper.class);
10
11   Scrape the URL
12   public static void scrape(String webpage) {
13     try {
14       // Open a connection to the URL
15       log.info(``Connecting to ''.concat(webpage));
16       URL url = new URL(webpage);
17       URLConnection connection = url.openConnection();
18       Scanner in = new Scanner(new InputStreamReader(connection.
             getInputStream()));
19
20       // Scrape the URL token by token
21       int count = 0;
22       while (in.hasNext()) {
23         count++;
24       }
25       log.info(``URL contained ''.concat(count).concat(`` tokens''));
26
27       // Close the connection to the URL
28       log.info(``Disconnecting from ''.concat(webpage));
29       in.close();
30     } catch(Exception e) {
31       log.warning(``Error connecting to ''.concat(webpage));
32     }
33   }
34 }
```

3) the lack of consistent log formats (Rabkin et al., 2010), 4) the large variation in the number and frequency of execution logs (Nagappan and Vouk, 2010; Vaarandi, 2003) and 5) the inaccessibility and difficulty of parsing source code (Xu et al., 2009a,b, 2010). Further, existing approaches to parse execution logs discard all dynamic information. Yet, the dynamic information contained in the execution logs is often critical to understanding the system's behaviour. For example, `Processing item 1 / 1` and `Processing item 1 / 1,000` are both be parsed to `Processing item ___/___`. Therefore, performance analysts cannot distinguish between `Processing item 1 / 1` and `Processing item 1 / 1,000` after the execution logs have been parsed. However, these two execution events likely have very different impacts on the system's memory usage (i.e., 1 item in the queue likely consumes less memory than 1,000 items). Therefore, we propose a new approach to parse execution logs to execution events using information retrieval techniques.

Our approach to parsing execution logs is a two-step process.

The first step in our approach to parsing execution logs is to create a vocabulary. We tokenize the source code and add each alphanumeric token (converted to lower case) to a vocabulary (i.e., a "bag-of-words"). Our intuition is that tokens that appear in the source code (e.g., a request type or status message) can be used to identify the static components of each log line. However, while building a vocabulary does require access to the source code, we do not need to parse the source code.

Table 4.3 shows the vocabulary built from Listing 4.1. For example, `import` and `java.io.InputStreamReader` from line 1 of Listing 4.1 are both added to the vocabulary.

Table 4.3: Vocabulary Built from Listing 4.1.

1	import
2	java.io.inputstreamreader
3	java.net.url
4	java.net.urlconnection
5	java.util.scanner
6	public
7	class
8	webscraper
9	static
10	logger
11	log
12	logger.getlogger
13	webscraper.class
14	void
15	scrape
16	string
17	webpage
18	try
19	log.info
20	connecting
21	to
22	concat
23	url
24	new
25	urlconnection
26	connection
27	url.openconnection
28	scanner
29	in
30	inputstreamreader
31	connection.getinputstream
32	int
33	count
34	while
35	in.hasnext
36	contained
37	tokens
38	disconnecting
39	from
40	in.close
41	catch
42	exception
43	e
44	log.warning
45	error

The second step in our approach to parsing execution logs is to use the vocabulary to identify the static components and dynamic information in the execution logs. We tokenize each log line and check whether each token appears in the vocabulary. Tokens that appear in the vocabulary (i.e., the static components of the log line) remain in the log line while tokens that do not appear in the vocabulary (i.e., the dynamic components of the log line) are remove from the log line and replaced with ‘‘___’’ to indicate the presence dynamic information. In contrast to traditional log parsing approaches, the dynamic information is written to a separate file rather than being discarded. Therefore the result of this step is two files: 1) one file with the static components from each log line and 2) one file with the dynamic information from each log line.

Table 4.1 and Table 4.2 present the execution events and execution event IDs (a unique ID automatically assigned to each unique execution event) for the execution logs from the field and from the certification test from our working example. These tables demonstrate the input (i.e., the log lines) and the output (i.e., the static components and the dynamic information from each log line) of the log parsing process. For example, the `Connecting to www.3-tokens.com` and `Connecting to www.2-tokens.com` log lines are both parsed to the `Connecting to ___` execution event.

4.3.3.2 Signature Generation

We enrich the execution events by leveraging the dynamic information in the execution logs to create event signatures. Event signatures describe the occurrences of

each execution event in terms of the dynamic information contained in the execution events. Therefore, the signature for a particular event is the distribution of the event's dynamic information. An event signature is generated for each execution event.

Table 4.4 presents the event signatures from the field and Table 4.5 presents the event signatures from the certification test. From these tables, we can see the dynamic information contained in the execution events. For example, from Table 4.2 we find that when the execution event `URL contained ___ tokens` occurs in the field, the dynamic information is either `2`, `3` or `99`. Therefore, the signature of the `URL contained ___ tokens` event is the `(2, 3, 99)` distribution.

4.3.4 Event Analysis

The final phase of our approach is to identify the execution events whose dynamic information differs between the event signatures from the field and the event signatures from the certification test. As execution logs may contain billions of execution events, this phase will only identify the most important event signature differences. Therefore, our approach helps system experts to diagnose performance issues observed in the field by identifying the most meaningful differences between their system's behaviour in the field compared to its behaviour during certification testing. Such "filtering" provides performance analysts with a concrete list of events to investigate.

Table 4.4: Event Signatures from the Field

Execution Event ID	Static Components	Dynamic Information
1	Connecting to ___	www.73-tokens.com www.87-tokens.com www.99-tokens.com
2	URL contained ___ tokens	73 87 99
3	Disconnecting from ___	www.73-tokens.com www.87-tokens.com www.99-tokens.com

Table 4.5: Event Signatures from a Certification Test

Execution Event ID	Static Components	Dynamic Information
1	Connecting to ___	www.2-tokens.com www.4-tokens.com www.3-tokens.com www.2-tokens.com
2	URL contained ___ tokens	2 4 3 2
3	Disconnecting from ___	www.2-tokens.com www.4-tokens.com www.3-tokens.com www.2-tokens.com

4.3.4.1 Signature Difference Detection

We identify differences between event signatures from the field and event signatures from the certification test using statistical measures (i.e., *unpaired two-sample two-tailed Welch's unequal variances t-tests* (Student, 1908; Welch, 1997) and *Cohen's d* effect size (Cohen, 1988, 1992) for dynamic information that is continuous and *Fisher's exact test* (Fisher, 1924) and *Cohen's w* effect size (Cohen, 1988, 1992) for dynamic information that is categorical). This analysis quantifies the importance of each execution event in differentiating the system's behaviour in the field to its behaviour during certification testing. Knowledge of these events may help performance analysts to diagnose performance issues observed in the field.

Algorithm 1 shows the pseudocode for detecting whether an event signature from the field differs from the correspond event signature from the certification test. For example, when comparing the event signature for Execution Event ID 2 from the field to the certification test, we see from Table 4.4 and Table 4.5 that the field signature is `(73, 87, 99)` and the test signature is `(2, 4, 3, 2)`.

First, we determine whether the event signature contains 1) categorical data, 2) continuous data or 3) "other" data that is not of interest. Categorical data consists of two or more numeric (e.g., error codes) or textual (e.g., request statuses) levels. At least 80% of the levels have occurred five or more times (Cochran, 1954). For example, `(200, 200, 404, 200, 404, 200, 200, 404, 404, 404)` is a distribution of HTTP request statuses and would be classified as categorical because there are two levels (i.e., `200` and `404`) and 100% of the levels have occurred five or more times. Continuous data consists of numeric data (e.g., file sizes) that is not classified as categorical. Data that is neither 1) categorical nor 2) continuous is classified

```
/* Determine the type of data                                        */
if 80% of the categories in the data have occurred five or more times then
  | type = categorical
end
else if the data is numeric then
  | type = continuous
end
else
  | return false
end
/* Determine the statistical test to use                             */
if categorical then
  | p value = unpaired two-sample two-tailed Welch's unequal variances test
end
else
  | p value = Fisher-Freeman-Halton exact test
end
/* Determine the effect size to use                                  */
if categorical then
  | effect size = Cohen'd d effect size
end
effect size = Cohen'd w effect size
/* Determine if the difference between the field and test
   signatures is statistically significant and large                 */
if p value < 0.01 and effect size > 1.4 then
  | return true
end
else
  | return false
end
```

Algorithm 1: Signature Difference Detection Pseudocode.

as "other" data that is not of interest. Such data includes, but is not limited to, user names, IP addresses, URLs and message contents.

Table 4.6 shows the type for each event signature from our working example. `URL contained ___ tokens` is classified as continuous data and `Connecting to ___` and `Disconnecting from ___` are both classified as other data.

Table 4.6: Event Signature Differences

Execution Event ID	**Static Components**	**Type**	**Significance** *p-value*	**Effect Size** *Cohen's d or w*
1	`Connecting to ___`	Other		
2	`URL contained ___ tokens`	Continuous	0.00779	10.115
3	`Disconnecting from ___`	Other		

Second, we identify the execution events whose signatures differ significantly between the field and the certification test (i.e., whether the difference between the dynamic information in the field compared to the dynamic information in the certification test is statistically significant).

We compare event signatures of continuous data using an *unpaired two-sample two-tailed Welch's unequal variances t-test* to determine whether the difference between the dynamic information in the field compared to the dynamic information in the certification test is statistically significant.

We construct the following hypotheses to be tested by an *unpaired two-sample two-tailed Welch's unequal variances t-test*. Our *null hypothesis* assumes that the dynamic information in the field is the same as the dynamic information in the certification test. Conversely, our *alternate hypothesis* assumes that the dynamic information in the field differs from the dynamic information in the certification test.

Table 4.6 shows the *p-value* for each event signature from our working example.

From Table 4.6, we find that Execution Event ID 2 (i.e., `URL contained ___ tokens`) differs significantly between the field and the certification test (i.e., $p < 0.01$). From the event signatures in Table 4.4 and Table 4.5, we see that dynamic information in Execution Event ID 2 is much larger in the field (`(73, 87, 99)`) compared to the certification test (`(2, 4, 3, 2)`).

We compare event signatures of categorical data using a *Fisher-Freeman-Halton exact test* to determine whether the difference between the dynamic information in the field compared to the dynamic information in the certification test is statistically significant. These tests are used to determine whether the difference between two distributions of categorical data is statistically significant (Fisher, 1924; Freeman and Halton, 1951). For example, whether the proportion of `200`, `400` and `404` HTTP request statuses in the field differs from the proportion of `200`, `400` and `404` HTTP request statuses in the certification test. The *Fisher-Freeman-Halton exact test* is mathematically complex and computationally intensive. Therefore, we refer the reader to Freeman and Halton (1951) for further details.

We construct the following hypotheses to be tested by a *Fisher-Freeman-Halton exact test*. Our *null hypothesis* assumes that the dynamic information in the field is the same as the dynamic information in the certification test. For example, if 67% of HTTP request statuses in the field are `200` and 33% of HTTP request statuses in the field are `404`, then 67% of HTTP request statuses in the certification test are `200` and 33% of HTTP request statuses in the certification test are `404`. Conversely, our *alternate hypothesis* assumes that the dynamic information in the field differs from the dynamic information in the certification test.

Third, we determine whether the difference between the dynamic information in the field and the dynamic information in the certification test is large regardless of statistical significance (i.e., small differences may be statistically significant if the execution event occurs many times in the field and the certification test).

We compare event signatures of continuous data using *Cohen's d* effect size to determine whether the difference between the dynamic information in the field compared to the dynamic information in the certification test is large.

From Table 4.6, we find that Execution Event ID 2 (i.e., `URL contained ___ tokens`) has a large (i.e., $d > 1.4$) effect size indicating that the difference in the dynamic information in Execution Event ID 2 is much larger in the field (`(73, 87, 99)`) compared to the certification test (`(2, 4, 3, 2)`).

We compare event signatures of categorical data using *Cohen's w* effect size to determine whether the difference between the dynamic information in the field compared to the dynamic information in the certification test is large. *Cohen's w* effect size measures the difference between two distributions of categorical data (Cohen, 1992). *Cohen's w* is similar to *Cohen's d*. However, *Cohen's d* measures the difference between two distributions of continuous data whereas *Cohen's w* measures the difference between two distributions of categorical data. Larger *Cohen's w* effect sizes indicate a greater difference between two distributions of categorical data, regardless of statistical significance. Hence, as the *Cohen's w* effect size of a particular event signature increases, the difference between the dynamic information in the field compared to the dynamic information in the certification test also increases.

Equation 4.1 presents how *Cohen's w* effect size for a particular event signature is calculated.

$$w = \sqrt{\sum_{i=1}^{k} \left(\frac{P_{1i} - P_{0i}}{P_{0i}} \right)^2} \tag{4.1}$$

where k is the number of levels in the event signature, P_{1i} is the proportion of events in the field with level i, P_{0i} is the proportion of events in the certification test with level i and w is *Cohen's w* effect size. *Cohen's w* effect size is traditionally interpreted as follows:

$$\begin{cases} \text{trivial} & \text{for } w < 0.1 \\ \text{small} & \text{for } 0.1 < w \leq 0.3 \\ \text{medium} & \text{for } 0.3 < w \leq 0.5 \\ \text{large} & \text{for } w > 0.5 \end{cases} \tag{4.2}$$

However, such an interpretation was originally proposed for the social sciences. Therefore, we use the interpretation of effect sizes proposed by Kampenes et al. (2007) and presented in Equation 3.17.

Finally, we identify the influential events as any execution event where the *p-value* indicates a statistically significant difference (i.e., $p < 0.01$) between the event signatures from the field and the event signature from the certification test and the

Cohen's d or *Cohen's w* indicates a large difference (i.e., $d > 1.4$) between the event signatures from the field and the event signature from the certification test From Table 4.6, we find that the difference between the Execution Event ID 2 (i.e., `URL contained ___ tokens`) signature from the field compared to the Execution Event ID 2 signature from the certification test is statistically significant (i.e., $p < 0.01$) and large (i.e., $d > 1.4$). Therefore, our approach identifies one execution event that best explains the difference in the system's behaviour in the field compared to its behaviour during certification testing. Performance analysts then realize that the cause of the performance issue is due to attempts to scrape large websites.

4.4 Case Studies

This section outlines the setup and results of our case studies. First, we present two case studies using a Hadoop application. We then discuss the results of two case studies using an enterprise system. Table 5.9 outlines the systems and data sets used in our case studies.

4.4.1 The Hadoop Platform

Our first case study system is an application that is built on the Hadoop platform. Hadoop is an open-source distributed data processing platform that implements MapReduce (Dean and Ghemawat, 2008; Hadoop, 2014).

Table 4.7: Case Study Subject Systems.

	Hadoop	**Enterprise System**
Application Domain	Data processing	Telecommunications
License	Open-source	Enterprise
Certification Test Data		
# Log Lines	223,545	2,867,532	2,867,532
Field Data		
# Log Lines	45,262	223,572	182,298,912	2,685,760
Notes	The system experienced decreased throughput in the field	The system experienced no performance issues in the field	The system's memory usage was much higher in the field than expected	The system experienced a memory leak in the field and during certification testing
Case Study		
Case Study Name	Decreased Throughput Issue in the Field	Decreased Throughput Issue Fixed in the Field	Enterprise Memory Issue in the Field	Enterprise Memory Leak Fix in the Field and Certification Test
Results		
Influential Events	2	0	1	0
Precision	100%	*100%	100%	*100%
*no execution events were flagged because the field and the test do not differ.		

4.4.1.1 The WordCount Application

The Hadoop application used in this case study is the WordCount application (MapReduce Tutorial, 2014).

The WordCount application is deployed on a cluster consisting of five machines (each with dual Intel Xeon E5540 (2.53GHz) quad-core CPUs, 12GB memory, a Gigabit network adaptor and SATA hard drives). While this cluster is small by industry standards (Chen et al., 2012), recent research has shown that almost all failures can be reproduced with a few as three machines (Yuan et al., 2014).

4.4.1.2 Decreased Throughput in the Field

We then monitored the Hadoop WordCount application in the field and found that the throughput (completed attempts/sec) was much lower than the throughput achieved during certification testing. We also found that the ratio of completed to failed attempts was much lower (i.e., more failed attempts relative to completed attempts) in the field compared to our certifications test. Therefore, we compare the system's behaviour in the field to its behaviour during certification to diagnose this performance issue.

We apply our approach to the execution logs that are collected from the system in the field and during certification testing. Our approach identifies the following execution events as most likely to be responsible for the difference between the system's behaviour in the field compared to its behaviour during certification testing:

```
INFO org.apache.hadoop.mapred.Merger:
    Merging ___ sorted segments
```

```
INFO org.apache.hadoop.mapred.Merger:
    Merging ___ intermediate segments out of a total of ___
```

The Hadoop Wordcount application reads a text file and splits the file into lines using line-feeds or carriage-returns (TextInputFormat, 2016). A Map task is then created for each line to 1) split the line into tokens separated by white spaces and 2) output a key-value pair of <token, 1> (MapReduce Tutorial, 2014; StringTokenizer, 2016). The output from the Map tasks are written to an in-memory buffer (i.e., the MapOutputBuffer) (Grishchenko, 2014). When the buffer is filled more than a certain amount (i.e., 80% by default), the contents of the buffer (i.e., the output from the Map tasks) is sorted and written to the disk (i.e., a segment). Finally, when all of the Map tasks are complete, all of the segments are read from the disk and merged into one file.

Our approach informs performance analysts that the number of `sorted segments` and `intermediate segments` is much larger in the field compared to the certification test. When performance analysts check the buffer size in the field, they find that the buffer size has been decreased in the field. Hadoop's memory is split between the WordCount application and the buffer. Therefore, decreasing the buffer size allows the application to use more memory if necessary. Operators in the field may change how much memory is allocated to the application compared to the buffer depending on the requirements for the applications using the Hadoop cluster.

Made aware of this issue, performance analysts could configure the Hadoop to use a larger buffer size to prevent this issue in the field.

4.4.1.3 Decreased Throughput Issue Fixed in the Field

To fix the decreased throughput issue, performance analysts increase the size of the buffer in the field. Performance analysts then deploy the fix to the field. To verify that the decreased throughput issue was fixed, we apply our approach to the execution logs that are collected from the system in the field and during certification testing. Our approach identifies that there are no meaningful differences between the system's behaviour in the field compared to its behaviour during certification testing. Therefore, our results provide performance analysts with confidence in their fix.

4.4.2 Enterprise System Case Study

We perform additional case studies on an enterprise system to examine the scalability of our approach. We note that the data sets from our enterprise case studies are considerably larger than the data sets from our Hadoop case studies (see Table 5.9).

4.4.2.1 The Enterprise System

The enterprise system used in our case studies is a large-scale enterprise software system in the telecommunications domain. For confidentiality reasons, we cannot disclose the specific details of the system's architecture. However, the system is responsible for simultaneously processing millions of client requests and has very high performance requirements.

Performance analysts regularly perform certification tests prior to releasing new versions of the system to ensure that the system meets its service level agreements.

4.4.2.2 Enterprise Memory Issue in the Field

Our first enterprise case study describes how our approach was used to diagnose a rapid and sustained increase in the system's memory usage (i.e., a memory spike). The system's operators had observed a memory spike in the field when the system was made available to a number of new users.

We use our approach to diagnose the spike in the system's memory usage. Our approach identifies one execution event (out of 1,203 unique execution events) that best explain the difference between the system's behaviour in the field compared to its behaviour during certification testing. Our results indicate that the cause of the memory spike is rapidly queuing a large number of work items. Multiple, independent system experts have confirmed that rapidly queuing these work items causes an increase in the system's memory usage until these work items are completed.

4.4.2.3 Enterprise Memory Leak Fix in the Field and Certification Test

Our second enterprise case study describes how our approach was used to validate a memory leak fix. The system's operators had observed a memory leak in the field and developers had implemented a fix to resolve the memory leak.

We use our approach to validate whether the developers' fix actually resolves the memory leak. Our approach identifies that no execution events (out of 401 unique execution events) differ between the system's behaviour in the field compared to its behaviour during certification testing (i.e., the system's behaviour in the field does not differ from its behaviour during certification testing). Therefore, our results indicate that the developers' fix did not resolve the memory leak. Multiple, independent system experts have confirmed our results.

4.5 Threats to Validity

This section outlines our threats to validity.

4.5.1 Threats to Construct Validity

4.5.1.1 Categorical and Continuous Data

Our approach relies on classifying whether the dynamic information in the execution logs is 1) categorical 2, continuous or 3) "other" (not of interest) data. Such classification informs the statistical measures that we use to compare the dynamic information from the field to the certification test. However, our approach may misclassify some dynamic information. For example, numeric errors codes may be classified as continuous data or categorical data with many categories may be classified as other data. However, we manually verified how the dynamic information was classified in our case studies.

4.5.1.2 Statistical Tests

Our approach relies on *unpaired two-sample two-tailed Welch's unequal variances t-test* (along with a *Cohen's d* effect size) to identify event signature differences when the dynamic information is continuous. Failure to satisfy the assumptions of this statistical test may affect the precision of our approach and undermine the statistical robustness of our approach (see Chapter 3 for a detailed discussion of this threat to validity).

4.5.1.3 Evaluation

We evaluated our approach by determining the precision with which our approach flags execution events that differ between the event signatures of the field and a certification test. While performance analysts have verified these results, we do not have a gold standard data set (see Chapter 3 for a detailed discussion of this threat to validity).

4.5.2 Threats to Internal Validity

4.5.2.1 Field-Representative Certification Tests

Our approach diagnoses performance issues in the field by comparing the system's behaviour in the field to its behaviour during certification testing. Therefore, our approach assumes that the certification test is representative of the field (certification tests are, by definition, representative of the field). However, the system's behaviour in the field is based on the behaviour of thousands or millions of users interacting with the system (i.e., the field workload). The field workload continuously evolves as the user base changes, as features are activated or disabled and as user feature preferences change. Such evolving field workloads may lead to certification tests that are not representative of the field (Bertolotti and Calzarossa, 2001; Voas, 2000). However, this threat is addressed by our approach to Comparing Workloads (see Chapter 3).

4.5.2.2 Execution Log Quality/Coverage

Our approach assumes that the cause of performance differences are manifested within the execution logs. However, it is possible that there are no execution logs to indicate when certain functionality is exercised (see Chapter 3 for a detailed discussion of this threat to validity).

4.5.2.3 Causes of Performance Differences

Our approach identifies important execution events that best explain the differences between the system's behaviour in the field compared to its behaviour during a certification test. However, our approach is not able to determine whether these differences are caused by a misconfiguration or a performance-degrading change to the system's source code. Therefore, performance analysts may allocate resources to examining the system's configuration when the true cause of a performance difference is in the system's source code. This threat may be mitigated by using our approach to compare a system's behaviour before a change (e.g., a release or a reconfiguration) to the system's behaviour after the change. Using our approach to compare a system's behaviour before a change to the system's behaviour after the change would allow performance analysts to control for the cause of the differences by allowing the configuration *or* the source code to change, *but not both*. However, misconfigurations and performance-degrading changes are both of interest to performance analysts.

4.5.3 Threats to External Validity

4.5.3.1 Access to Source Code

Our approach to parsing execution logs requires a vocabulary created from the system's source code. Yet, the developers of closed source systems may be unwilling to distribute the source code of their systems. However, while building a vocabulary does require access to the source code, we do not need to parse the source code.

4.5.3.2 Generalizing Our Results

The studied software systems represent a small subset of the total number of large-scale software systems. Therefore, it is unclear how our results will generalize to additional software systems (see Chapter 3 for a detailed discussion of this threat to validity).

4.6 Conclusions

In the previous chapter, we presented an approach for enriching the execution events by leveraging *both* the static components of the execution logs *and* the time stamps and worker IDs in the execution logs. Therefore, in this chapter, presented an approach for enriching the execution events by leveraging the dynamic information in the execution logs. Our approach diagnoses performance issues by deriving event signatures from certification tests and the field using execution events that have been enriched by leveraging *both* the static components of the execution logs *and* the dynamic information in the execution logs. We then use statistical techniques to identify differences between the event signatures from the field and the

event signatures from a certification test. Such differences should help performance analysts to understand the cause of any performance issues observed in the field.

We performed four case studies on two systems: 1) one open-source system and one enterprise system. Our case studies explored how our approach can be used to identify performance deviations from certification tests caused by performance issues in the field. Performance analysts and system experts have confirmed that our approach provides valuable insights that help to diagnose performance issues observed in the field.

In the next chapter, we present an approach for further enriching the execution events by combining the execution events with the performance counters.

CHAPTER 5

Enriching the Execution Events by Combining Execution Logs and Performance Counters

The rise of large-scale software systems poses many new challenges for the software performance engineering field. Failures in these systems are often associated with performance issues, rather than with feature bugs. Therefore, certifying that these systems meet their release certification benchmark (i.e., that they comply with the service level agreements) is essential to ensuring their problem-free operation. Deviations from these benchmarks can be identified by comparing the system's performance during performance testing to its performance during certification testing (i.e., performance testing using a field-representative workload). However, the current state of performance testing requires considerable manual review of the

execution logs (to understand system behaviour) and performance counters (to understand resource usage) and a high degree of system-specific expertise.

In this chapter, we propose an automated approach to enrich the execution events by combining the execution logs and performance counters to compare the system's performance during performance testing to its performance during certification testing. We perform three case studies on two large systems (one open-source system and one enterprise system) to determine whether our approach can diagnose performance deviations from the release certification benchmark. Our approach helps performance analysts understand the cause of performance regressions from release certification benchmarks with a precision of 90.3% compared to only 32.9% for the state-of-the-practice.

5.1 Introduction

The highly distributed and configurable nature of today's large-scale software systems (e.g., Amazon.com and Google's GMail) poses many new challenges for the software performance engineering field (Software Engineering Institute, 2006). Failures in these systems are often associated with performance issues, rather than with feature bugs (Compuware, 2006; Dean and Barroso, 2013; Simic and Conklin, 2012; Weyuker and Vokolos, 2000). Such performance issues have led to several high-profile failures with significant financial and reputational repercussions.

Therefore, certifying that these systems comply with their service level agreements has become essential to ensuring the problem-free operation of these systems (Burger and Reussner, 2011; Hassan and Zhang, 2006; Nivas and Csallner, 2011).

Such certification is performed by "certification testing." A certification test is performed by performance testing the system with a field-representative workload. Therefore, certification testing allows performance analysts to ensure that the system is able to comply with its service level agreements under a realistic workload. The results of the certification test then form a baseline of the system's performance when the system complies with its service level agreements. The system's performance during future performance tests (e.g., performance testing prior to releasing a new version) is then compared to its performance during certification testing to determine whether the system's performance has changed.

The current state of certification testing requires considerable manual effort and a high degree of system-specific expertise from the performance analysts. Performance analysts *detect* performance deviations from the release certification benchmark by comparing the system's performance during performance testing to its performance during certification testing using performance counters. However, the system's performance is based on thousands or millions of users simultaneously interacting with many of the system's features. Therefore, performance analysts must *diagnose* which feature or use case is the cause of the performance deviations. However, comparing performance counters can only determine whether the performance of the *system*, not the performance of a particular *feature* or *use case*, differs between the performance test and the certification test. Performance improvements in one feature may hide performance regressions in another feature. Further, documentation describing the system is rarely up-to-date (Ernst et al., 2015; Holvitie et al., 2014; Parnas, 1994), research into diagnosing performance deviations is limited (Jiang and Hassan, 2000) and instrumenting the system imposes a heavy

overhead and is often infeasible in practice (Bernat and Miller, 2011; Laurenzano et al., 2015; Maplesden et al., 2015; Uh et al., 2006). Fortunately, execution logs and performance counters are readily available in most large-scale systems to support remote monitoring, issue resolution and legal compliance (Frey, 2015; The Sarbanes-Oxley Act, 2006). Further, these logs contain developer knowledge (i.e., they are manually inserted into the system by developers) (Shang et al., 2011). Hence, execution logs (to understand system behaviour) and performance counters (to understand resource usage) are the best source of data available to describe the system's behaviour.

In this chapter, we propose an automated approach to identify performance deviations from the release certification benchmark by comparing the system's performance during performance testing to its performance during certification testing. Our approach enriches the execution events by combining the execution logs with the performance counters that are collected during the system's execution. We derive performance signatures from the execution logs and performance counters. We then use statistical techniques to identify differences between the performance signatures of the performance test and the performance signatures of certification test. Such differences should help performance analysts to understand the cause of performance deviations, if any, from the release certification benchmark.

This chapter makes three contributions:

1. We present an approach to enrich the execution events by combining the execution logs with the performance counters.

2. We demonstrate an automated approach to identify performance deviations by comparing the system's performance during performance testing to its performance during certification testing. Our approach can identify important execution events that best explain the differences between a system's performance during performance testing compared to its performance during certification testing.

3. Through three case studies, we show that our approach is scalable and can help performance analysts to understand performance deviations from the release certification benchmark. We also show how our approach outperforms the state-of-the-practice in all three case studies.

5.1.1 Organization of the Chapter

This chapter is organized as follows: Section 5.2 provides a motivational example of how our approach may be used in practice. Section 5.3 describes our approach in detail. Section 5.4 presents our case studies. Section 5.5 outlines the threats to validity. Finally, Section 5.6 concludes the chapter.

5.2 Motivational Example

Ian is responsible for certification testing a new large-scale system to verify that the system's passes its release certification benchmark. Two months later, a new version of the system is released. Ian performance tests the new version and finds that CPU usage is higher than expected.

Pressured by time (given the competitive marketplace), management (who are keen to boast a high quality system) and the complexity of analyzing performance tests, Ian is introduced to an automated approach that can identify performance deviations from the release certification benchmark by comparing the system's performance during performance testing to its performance during certification testing. This approach enriches the execution events by combining the execution logs with the performance counters, derives performance signatures from the enriched execution events then compares the signatures from the performance test against the signatures from the certification test to identify execution events that explain the performance deviations from the release certification benchmark. Thus, this approach provides Ian with a concrete starting point (i.e., specific execution events) for his analysis.

Using this approach, Ian discovers that I/O events use more CPU during the performance test than expected. Ian then realizes that compression is enabled in the new version to reduce the system's disk usage. Therefore, Ian was able to identify the cause of the system's performance issue.

5.3 Approach

This section outlines our approach to enrich the execution events and compare the system's performance during performance testing to its performance during certification testing. Figure 5.1 provides an overview of our approach. We describe each step in detail below and demonstrate our approach with a working example of a real-time chat system.

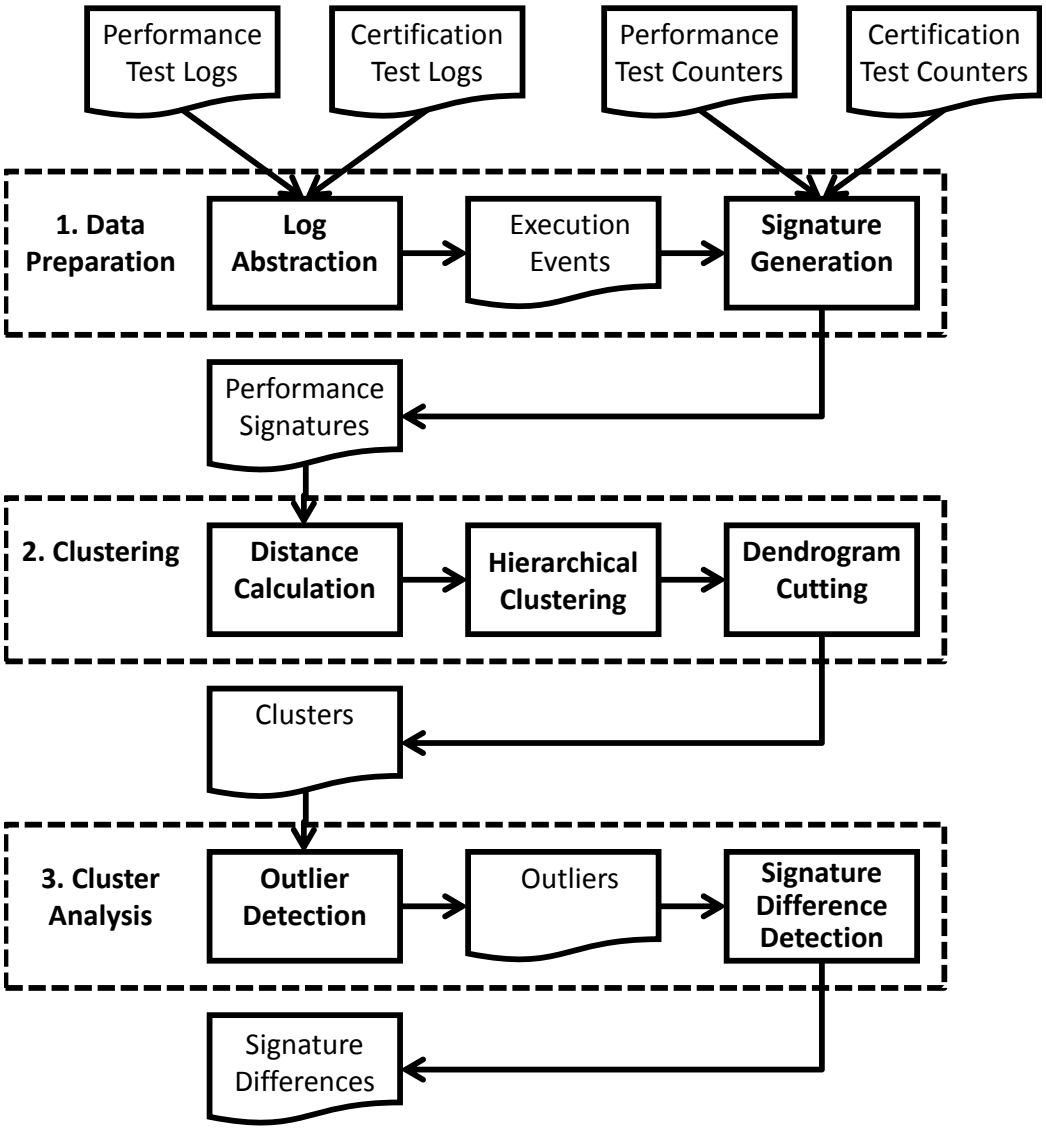

Figure 5.1: An Overview of Our Approach.

The first step in our approach is to abstract the execution logs into execution events and to enrich the execution events by combining the execution logs and performance counters. We then create performance signatures from the enriched execution events. Each performance signature represents a period of time where we can measure the number of execution events that have occurred (i.e., the log activity) and the change in resource usage (i.e., the resource usage delta).

The second step is to cluster the performance signatures into groups where a similar set of events have occurred. We expect that whenever a similar set of events occurs the resource usage delta should also be similar. Differences in the resource usage delta are considered performance deviations.

The third step is to identify clusters where the performance signatures from the performance test and the certification test have different resource usage deltas. These clusters represent periods of time where a similar set of events have occurred, but the change in resource usage differs between the performance test and the certification test. We then identify the events that best differentiate these clusters. Knowledge of these events should help performance analysts understand the cause of performance deviations.

5.3.1 Input Data

Our approach uses two data sources to generate performance signatures: 1) execution logs and 2) performance counters.

5.3.1.1 Execution Logs

The second column of Table 5.1 and Table 5.2 presents the execution logs from the performance test and the certification test of our working example respectively. These execution logs contain both static information (e.g., `starts a chat`) and dynamic information (e.g., `Alice` and `Bob`) that changes with each occurrence of an event. The certification test has been configured with a simple use case (from 00:01 to 00:11) that is continuously repeated.

5.3.1.2 Performance Counters

Performance counters describe system resource usage (e.g., CPU usage, memory usage, network I/O and disk I/O), performance (e.g., response time and throughput) and reliability (e.g., mean time-to-failure). Performance analysts must identify and collect the set of counters that are relevant to their system. Each of these counters may then be analyzed independently. Performance counters are sampled at periodic intervals (e.g., 30 seconds) by resource monitoring tools (e.g., PerfMon (PerfMon, 2016) and Pidstat (SYSSTAT, 2016)).

Table 5.3 and Table 5.4 presents the performance counters (i.e., cumulative network I/O) for our working example.

5.3.2 Data Preparation

The first step in our approach is to prepare the execution logs and performance counters for automated analysis. Data preparation is a two-step process. First, we remove implementation and instance-specific details from the execution logs in order to generate a set of execution events. Second, we generate performance

Table 5.1: Abstracting Execution Logs to Execution Events: Execution Logs from a Performance Test

Time	Log Line	Execution Event	Execution Event ID
00:01	Alice starts a chat with Bob	starts a chat with ___	1
00:02	Alice says "hi, are you busy?" to Bob	says ___ to ___	2
00:05	Carl starts a chat with Dan	starts a chat with ___	1
00:06	Carl says "do you have files?" to Dan	says ___ to ___	2
00:10	Dan says "yes" to Carl	says ___ to ___	2
00:11	Dan Initiate file transfer to Carl	Initiate file transfer to ___	3
00:13	Dan Initiate file transfer to Carl	Initiate file transfer to ___	3
00:15	Dan says "got it?" to Carl	says ___ to ___	2
00:18	Carl says "thanks" to Dan	says ___ to ___	2
00:19	Carl ends the chat with Dan	ends the chat with ___	4
00:21	Bob says "yes" to Alice	says ___ to ___	2
00:22	Alice says "ok, bye" to Bob	says ___ to ___	2
00:22	Alice ends the chat with Bob	ends the chat with ___	4

Table 5.2: Abstracting Execution Logs to Execution Events: Execution Logs from a Certification Test

Time	Log Line	Execution Event	Execution Event ID
00:01	USER1 starts a chat with USER2	starts a chat with ___	1
00:03	USER1 says "MSG1" to USER2	says ___ to ___	2
00:05	USER2 says "MSG2" to USER1	says ___ to ___	2
00:07	USER1 Initiate file transfer to USER2	Initiate file transfer to ___	3
00:09	USER1 says "MSG3" to USER2	says ___ to ___	2
00:11	USER1 ends the chat with USER2	ends the chat with ___	4
00:13	USER3 starts a chat with USER4	starts a chat with ___	1
00:15	USER3 says "MSG1" to USER4	says ___ to ___	2
00:17	USER4 says "MSG2" to USER3	says ___ to ___	2
00:19	USER3 Initiate file transfer to USER4	Initiate file transfer to ___	3
00:21	USER3 says "MSG3" to USER4	says ___ to ___	2
00:23	USER3 ends the chat with USER4	ends the chat with ___	4

Table 5.3: Performance Counters from the Certification Test

Time	Cumulative Network I/O (bytes)
00:00	0
00:04	4
00:08	7
00:12	59
00:16	113
00:20	116
00:20	121

Table 5.4: Performance Counters from a Performance Test

Time	Cumulative Network I/O (bytes)
00:00	0
00:04	4
00:08	18
00:12	21
00:16	23
00:20	35
00:24	40

signatures by counting the number of execution events and the change in resource usage over each sampling interval.

5.3.2.1 Log Abstraction

We abstract the execution logs using the approach proposed by Jiang et al. (2008a). In addition, many execution logs and their corresponding execution events have been manually reviewed by multiple, independent system experts to verify the correctness of the abstraction.

Table 5.1 and Table 5.2 present the execution events and execution event IDs (a unique ID automatically assigned to each unique execution event) for the execution logs from the performance test and the certification test respectively. These tables demonstrate the input (i.e., the log lines) and the output (i.e., the execution events) of the log abstraction process. For example, the `starts a chat with Bob` and `starts a chat with Dan` log lines are both abstracted to the `starts a chat with ___` execution event.

5.3.2.2 Signature Generation

We enrich the execution events by combining the execution logs and the performance counters to create performance signatures. Performance signatures characterize the impact on resource usage of the aggregate user behaviour in terms of feature usage expressed by the execution events during some small period of time.

We combine performance counters with the execution events using time stamps. When a log line is generated or a performance counter is sampled, the log line or performance counter is written to a log/counter file along with the date and time of generation/sampling.

Although performance counters and execution logs both contain time stamps, combining these two is a major challenge. This is because performance counters are sampled at periodic intervals, whereas execution logs are generated continuously. Therefore, we must discretize the execution logs such that they co-occur with the performance counters.

Discretization also allows us to account for the delayed impact of some functionality on the performance counters. For example, there may be a slight delay

between when a log line is generated and when the associated functionality is executed. Discretization also helps to reduce the overhead imposed on the system during performance testing because the performance counter sampling frequency can be reduced.

During the period of time between two successive samples of the performance counters (i.e., a "time-slice"), zero or more log lines may be generated by events occurring within the system. For example, a log line may be generated when a new work item is started, queued or completed or an error is encountered during the time-slice. The execution events are then discretized by creating a performance signature for each time-slice. This signature is created by counting the number of times that each type of execution event occurred during the time-slice and by calculating the change in resource usage between the start and end of the time-slice. Therefore, each performance signature has two components: 1) a log activity component, which is a count of each execution event that has occurred during the time-slice and 2) a resource usage delta over the time-slice. Execution logs and performance counters are collected from the same machine, averting the need to synchronize the data collection (Lamport, 1978).

Our signature generation technique is agnostic to the content and format of the performance counters and execution logs. We do not rely on transaction/thread/job IDs and we do not assume any tags other than a time stamp.

Table 5.5 shows the performance signatures from our working example.

We repeat this procedure for each performance counter (i.e., a set of performance signatures is generated for each performance counter).

Table 5.5: Performance Signatures

		Log Activity (Execution Event ID)				Network I/O
		1	2	3	4	Delta
Performance Test Signatures	00:04	1	1	0	0	3
	00:08	1	1	0	0	4
	00:12	0	1	1	0	52
	00:16	0	1	1	0	54
	00:20	0	1	0	1	3
	00:24	0	2	0	1	5
Certification Test Signatures	00:04	1	1	0	0	4
	00:08	0	1	1	0	14
	00:12	0	1	0	1	3
	00:16	1	1	0	0	2
	00:20	0	1	1	0	12
	00:24	0	1	0	1	5

5.3.3 Clustering

The second phase of our approach is to cluster the performance signatures into groups where a similar set of events have occurred. The clustering phase in our approach consists of three steps. First, we calculate the dissimilarity (i.e., distance) between the log activity component of every pair of performance signatures. Second, we use a hierarchical clustering procedure to cluster the performance signatures into groups where a similar set of events have occurred. Third, we convert the hierarchical clustering into k partitional clusters (i.e., where each performance signature is a member in only one cluster). We cluster performance signatures using the same approach that we used to cluster workload signatures (see Chapter 3 for a detailed discussion of each step in the clustering phase).

5.3.3.1 Distance Calculation

We calculate the distance between the log activity component of every pair of performance signatures using the Pearson distance (d_ρ). This is because we expect that similar log activity should lead to similar resource usage deltas (i.e., whenever a particular set of events occurs in the system, the impact on resource usage should be similar).

Table 5.6 presents the distance matrix produced by calculating the Pearson distance between every pair of performance signatures in our working example.

5.3.3.2 Hierarchical Clustering

We use an agglomerative, hierarchical clustering procedure (with the average linkage criteria) to cluster the performance signatures using the distance matrix calculated in the previous step.

Figure 5.2 shows the dendrogram produced by hierarchically clustering the performance signatures from our working example.

5.3.3.3 Dendrogram Cutting

We use the Calinski-Harabasz stopping rule (Calinski and Harabasz, 1974) to determine where to cut the dendrogram.

The dotted horizontal line in Figure 5.2 shows where the Calinski-Harabasz stopping rule cuts the hierarchical cluster dendrogram from our working example into three clusters (i.e., the dotted horizontal line intersects with solid vertical lines at three points in the dendrogram).

Table 5.6: Distance Matrix

		Performance Test Signatures					Certification Test Signatures					
		00:04	00:08	00:12	00:16	00:20	00:24	00:04	00:08	00:12	00:16	00:20	00:24
Performance Test Signatures	00:04	0	0	1	1	1	0.698	0	1	1	0	1	1
	00:08	0	0	1	1	1	0.698	0	1	1	0	1	1
	00:12	1	1	0	0	1	0.698	1	0	1	1	0	1
	00:16	1	1	0	0	1	0.698	1	0	1	1	0	1
	00:20	1	1	1	1	0	0.095	1	1	0	1	1	0
	00:24	0.698	0.698	0.698	0.698	0.095	0	0.698	0.698	0.095	0.698	0.698	0.095
Certification Test Signatures	00:04	0	0	1	1	1	0.698	0	1	1	0	1	1
	00:08	1	1	0	0	1	0.698	1	0	1	1	0	1
	00:12	1	1	1	1	0	0.095	1	1	0	1	1	0
	00:16	0	0	1	1	1	0.698	0	1	1	0	1	1
	00:20	1	1	0	0	1	0.698	1	0	1	1	0	1
	00:24	1	1	1	1	0	0.095	1	1	0	1	1	0

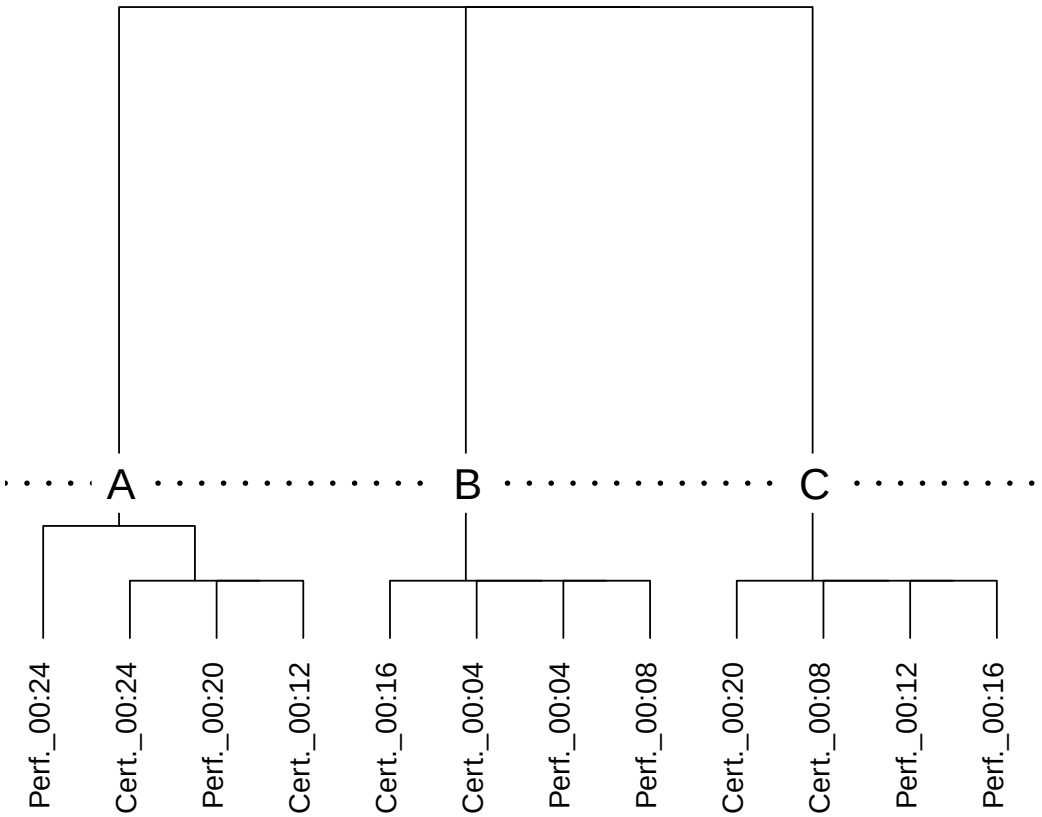

Figure 5.2: Sample Dendrogram. The dotted horizontal line indicates where the dendrogram was cut into three clusters (i.e., Cluster A, B and C).

5.3.4 Cluster Analysis

The third phase in our approach is to identify the execution events that correspond to the differences between the performance signatures from the performance test and the performance signatures from the certification test. The cluster analysis phase of our approach consists of two steps. First, outlying clusters are detected. Outlying clusters contain performance signatures where similar events occurred in the performance test and the certification test, but the impact of resource usage differs between the performance test and the certification test. Second, the key

execution events of the outlying clusters are identified. We refer to these execution events as "signature differences". These signature differences should help performance analysts identify performance deviations from the release certification benchmark.

We analyze performance signature clusters using an approach similar to the one that we used to analyze workload signature clusters in Chapter 3. However, we detect outlying clusters of performance signatures by determining whether the resource usage deltas of the performance signatures from the performance test are "too low" or "too high" compared to performance signatures from the certification test whereas we detected outlying clusters of workload signatures by determining whether the proportion of workload signatures from the field was "too high." We then detect signature differences between clusters of performance signatures using the same approach that we used to detect differences between clusters of workload signatures (see Chapter 3 for a detailed discussion of the signature difference detection step of the cluster analysis phase).

5.3.4.1 Outlying Cluster Detection

Clusters contain performance signatures from the performance test and the certification test. Outlying clusters contain performance signatures where similar events occurred in the performance test and the certification test, but the impact of resource usage differs between the performance test and the certification test.

We identify outlying clusters using statistical measures (i.e., *unpaired two-sample two-tailed Welch's unequal variances t-tests* (Student, 1908; Welch, 1997) and *Cohen's d* effect size (Cohen, 1988)). This analysis quantifies whether a similar set

execution events leads to different resource usage in the performance test compared to the certification test. Knowledge of these events may lead performance analysts to update their tests (e.g., reconfiguring the system).

First, we determine whether the difference between the resource usage deltas of the performance signatures from the performance test and the certification test are statistically significant (i.e., we test whether the impact on resource usage of a particular set of events difference between the performance test and the certification test). We perform this test for each cluster.

We perform an *unpaired two-sample two-tailed Welch's unequal variances t-test* to determine whether the resource usage deltas differ significantly between the performance signatures from the performance test and the performance signatures from the certification test.

We construct the following hypotheses to be tested by an *unpaired two-sample two-tailed Welch's unequal variances t-test*. Our *null hypothesis* assumes that resource usage deltas do not differ between the performance signatures from the performance test and the performance signatures from the certification test. Conversely, our *alternate hypothesis* assumes that resource usage deltas differ between the performance signatures from the performance test and the performance signatures from the certification test.

Table 5.7 shows the *t-test p-value* for each cluster in our working example.

Second, we determine the size of the difference between the resource usage deltas of the performance signatures from the performance test and the performance signatures from the certification test. We also perform this test for each cluster.

Table 5.7: Identifying Outlying Clusters

Cluster	***p-value***	***Cohen's d***	***Cohen's d Interpretation***
A	1	0	Trivial
B	0.712	0.447	Small
C	0.00125	28.284	Large

We calculate the *Cohen's d* effect size to determine the size of the difference between the resource usage deltas of the performance signatures from the performance test and the performance signatures from the certification test.

Table 5.7 shows the *Cohen's d* for each cluster in our working example.

Finally, we identify the outlying clusters as any cluster with a *t-test p-value* less than 0.01 and a *Cohen's d* greater than 1.4 (i.e., a large effect).

From Table 5.7, we identify cluster C as an outlying cluster (i.e., the *t-test p-value* is less than 0.01 and the *Cohen's d* is greater than 1.4). Cluster C contains two performance signatures from the certification test (i.e., Cert._00:12 and Cert._00:16) and two performance signatures from the test (i.e., Perf._00:08 and Perf._00:20). From Table 5.5, we find that the average network I/O delta for the two performance signatures from the performance test is 52 bytes, whereas the average network I/O delta for the two certification test signatures is only 13 bytes.

5.3.4.2 Signature Difference Detection

We identify the differences between performance signatures in outlying clusters and the average ("normal") performance signature using statistical measures (i.e., *unpaired two-sample two-tailed Welch's unequal variances t-tests* (Student, 1908; Welch,

1997) and *Cohen's d* effect size (Cohen, 1988)). This analysis quantifies the importance of each execution event in differentiating a cluster. Knowledge of these events may lead performance analysts to update their tests (e.g., reconfiguring the system).

First, we determine the execution events that differ significantly between the performance signatures in the outlying clusters and the average performance signature. For example, execution events that occur 10 times more often in the performance signatures of an outlying cluster compared to the average performance signature should likely be flagged for further analysis by a system expert.

We perform an *unpaired two-sample two-tailed Welch's unequal variances t-test* to determine which execution events differ significantly between the performance signatures in an outlying cluster and the average performance signature.

We construct the following hypotheses to be tested by an *unpaired two-sample two-tailed Welch's unequal variances t-test*. Our *null hypothesis* assumes that an execution event occurs the same number of times in the performance signatures of an outlying cluster compared to the average performance signature. Conversely, our *alternate hypothesis* assumes that the execution event does not occur the same number of times in the performance signatures of an outlying cluster compared to the average performance signature. We are interested in execution events that occur both 1) more often and 2) less often in the performance signatures of an outlying cluster compared to the average performance signature.

Table 5.8 shows the *t-test p-value* for each execution event in the outlying cluster (i.e., Cluster A).

Second, we determine the most important execution events that differ between the performance signatures in the outlying clusters and the average performance

Table 5.8: Performance Signatures Differences

Execution Event ID	***p-value***	***Cohen's d***	***Cohen's d Interpretation***
1	0.0388	0.764	Medium
2	0.3388	0.326	Small
3	0.000660	1.528	Large
4	0.0388	0.764	Medium

signature. For example, if execution events "A" and "B" occur 2 and 10 times more often in the performance signatures of an outlying cluster compared to the average performance signature, then execution event "B" should be flagged for further analysis by a system expert rather than execution event "A."

We calculate the *Cohen's d* effect size to determine the most important execution events that differ between the performance signatures in the outlying clusters and the average performance signature.

Table 5.8 shows the *Cohen's d* for each execution event in the outlying cluster (i.e., Cluster C).

Finally, we identify the influential events as any execution event with a *t-test p-value* less than 0.01 and a *Cohen's d* greater than 1.4.

From Table 5.8, we find that the difference in Execution Event ID 3 between the performance signatures in Cluster C and the average performance signature is statistically significant (i.e., $p < 0.01$) and large (i.e., $d > 1.4$). Therefore, our approach identifies one execution event (i.e., initiating a file transfer) that best explains the difference in the system's performance during a performance test compared to its performance during a certification test. When performance analysts examine the configuration parameters that impact network I/O and file transfers, they

discover that compression is enabled during certification testing, but not during performance testing. Performance analysts should then update their performance tests to more accurately represent the certification test deployment configurations (i.e., field-representative deployment configurations) by enabling compression during performance testing.

5.4 Case Studies

This section outlines our case studies. First, we present a feasibility study using a Hadoop application. Our feasibility study demonstrates our approach on a relatively small Hadoop application with a single known performance deviation from the certification test. Recent research has shown that most field failures can be reproduced on clusters with as few as three nodes (Yuan et al., 2014). Finally, we discuss the results of two enterprise case studies. Table 5.9 outlines the systems and data sets that are used in our case studies.

5.4.1 State-of-the-Practice

We compare our approach to the state-of-the-practice. Currently, performance analysts use control charts to identify when performance counters exceed the control limits (i.e., more than two standard above or below the historical average) (Nguyen et al., 2011). Performance analysts then compare the number of times each execution event occurs during times when the counters exceed the control limits (i.e., non-conforming times) compared to when the counters are within the control limits (i.e., conforming times) (Alspaugh et al., 2014; Cataldo et al., 2013; Hassan et al.,

Table 5.9: Case Study Subject Systems.

	Hadoop	**Enterprise System**
Application Domain	Data processing	Telecom
License	Open-source	Enterprise
Certification Test Data		
# Log Lines	54,905	2,199,727	3,433,875
Duration	6.17 minutes	1,142 minutes	479 minutes
Performance Test Data		
# Log Lines	54,918	1,964,463	620,824
Notes	The system's CPU and memory usage was greater than the historical memory usage	The system's memory usage continually increased until the system was restarted	The system's response time was much longer than expected when under heavy load
Case Study		
Case Study Name	Hadoop compression misconfiguration	Enterprise error handler performance defect	Enterprise database misconfiguration
Results		
Influential Events	1	18	16
Precision	100%	83.3%	87.5%
Precision (State-of-the-Practice)	0%	61.1%	37.5%

2008; Rosa, 2016). Therefore, we rank the events based on the difference in occurrence during non-conforming times compared to conforming times and investigate the events with the largest differences. In practice, performance analysts do not know how many of these events should be investigated. Therefore, we examine the same number of events as our approach identifies such that the effort required to manually analyze the events identified by either our approach or the state-of-the-practice is approximately equal. For example, if our approach identifies 10 events, we examine the top 10 events ranked by the state-of-the-practice. We then compare the precision of our approach to the state-of-the-practice. We define precision as the percentage of events that our approach identifies as meaningful differences between the system's performance during the performance test compared to the certification test that were confirmed by multiple, independent system experts.

5.4.2 Hadoop Compression Misconfiguration

5.4.2.1 The Hadoop Platform

Our first case study system is an application that is built on the Hadoop platform. Hadoop is an open-source distributed data processing platform that implements MapReduce (Dean and Ghemawat, 2008; Hadoop, 2014).

5.4.2.2 The WordCount Application

The Hadoop application used in this case study is the WordCount application (MapReduce Tutorial, 2014).

5.4.2.3 Hadoop Compression Misconfiguration

The Certification Test

We conduct a certification test of the WordCount application on a cluster consisting of five machines (each with dual Intel Xeon E5540 (2.53GHz) quad-core CPUs, 12GB memory, a Gigabit network adapter and SATA hard drives). We use the default configuration of the Hadoop platform. During the certification test, we collect the default Hadoop platform logs (JobConf, 2016). We also collect the performance counters using the Hadoop Metrics 2.0 API (Metrics 2.0, 2016).

The Performance Test

We conduct a performance test of the WordCount application while collecting the default Hadoop platform logs (JobConf, 2016) and the performance counters provided by the Hadoop Metrics 2.0 API (Metrics 2.0, 2016). However, we enable compression during this test. Compression is not enabled by default, however many operators enable compression to reduce the I/O workload on the system. We find that CPU usage and memory usage occasionally "spike" (i.e., rapidly increases then decreases). Such spikes occur while files are being compressed.

Results

We apply our approach to the execution logs and performance counters that are collected from the system during the certification test and the performance test. Our approach identifies the following execution event as most likely to be responsible for the difference between the system's performance during performance testing compared to its performance during certification testing:

```
INFO org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter:
    Saved output of task attempt_id to file
```

This execution event clearly relates to I/O. Therefore, our approach correctly identified 1 event (100% precision) that corresponds to the difference between the system's performance during the performance test compared to the certification test. Hence, our feasibility study demonstrated that our approach can correctly identify performance deviations from the release certification benchmark in small systems.

In contract, the state-of-the-practice flags the following execution event:

```
DEBUG org.apache.hadoop.mapred.TaskTracker:
    Got heartbeatResponse from JobTracker with responseId:
    ___ and ___ actions
```

Therefore, the state-of-the-practice had a precision of 0% because the one event that was flagged by the state-of-the-practice does not correspond to the true cause of the difference between system's performance during the performance test compared to the certification test.

5.4.3 Enterprise System Case Study

Although our Hadoop case study was promising, we perform two additional case studies on an enterprise system to examine the scalability of our approach. We note that these data sets are much larger than our Hadoop data set (see Table 5.9).

5.4.3.1 The Enterprise System

Our second case study system is a large-scale enterprise software system in the telecommunications domain. For confidentiality reasons, we cannot disclose the specific details of the system's architecture. However, the system is responsible for simultaneously processing millions of client requests and has very high performance requirements.

Performance analysts perform continuous performance testing to ensure that the system continuously meets its performance requirements. Therefore, analysts must continuously ensure that the system's performance during performance testing is representative of the system's performance during certification testing.

5.4.3.2 Enterprise Error Handler Misconfiguration

Our first enterprise case study describes how our approach was used to identify performance deviations from the release certification benchmark. Operators had recently updated the system to the latest version. The version contained a latent performance defect in a particular error handler. The operators then found that the system's memory usage continuously increased until the system was restarted (or crashed when the system's memory was exhausted).

Our approach identified 18 executions events (out of 344 unique events) that differ between the performance signatures of the performance test and the certification test. These differences were given to multiple independent system experts who confirmed that 15 of these events correspond to one of the system's features and an error handler associated with this feature. The system experts also confirmed that a performance defect in the error handler was the true cause of the continual

increase in memory usage. The remaining 3 events correspond to general utilities that are used by many features (including the error handler).

Our approach correctly identified 15 execution events (83.3% precision) that correspond to differences between the system's performance during the performance test and the certification test. Such results clearly indicate that the cause of the difference between the system's performance during the performance test compared to the certification test was a performance defect in the error handler.

In contrast, the state-of-the-practice had a precision of only 61.1% and did not identify any of the events that correspond the feature's error handler.

5.4.3.3 Enterprise Database Misconfiguration

Our second enterprise case study also describes how our approach was used to identify performance deviations from the release certification benchmark. Operators had made several changes to the system's configuration, including several changes to the configuration of the database. The operators then found that the system's average CPU usage and memory usage is greater than the average historical memory usage and that the database occasionally deadlocks.

Our approach identified 16 executions events (out of 841 unique events) that differ between the performance signatures of the performance test and the certification test. These differences were given to multiple independent system experts who confirmed that 14 of these events correspond to 1) one database-intensive feature and 2) an error message regarding the database deadlock. The system experts also confirmed that the misconfigured database was the true cause of the increase in CPU usage and memory usage as well as the database deadlocks. The remaining

2 events correspond to the authentication of connections (including connections to the system's database).

Our approach correctly identified 14 execution events (87.5% precision) that correspond to differences between the system's performance during the performance test and the certification test. Such results clearly indicate that the cause of the difference between the system's performance during the performance test compared to the certification test was a misconfigured database.

In contrast, the state-of-the-practice had a precision of only 37.5% and did not identify the event that corresponds to the database deadlock error message.

Our approach flags events with an average precision of 90%, outperforming the state-of-the-practice.

5.5 Threats to Validity

This section outlines our threats to validity.

5.5.1 Threats to Construct Validity

5.5.1.1 Timing of Events

Our approach is based on enriching the execution event by combining the execution logs and the performance counters. This is done using the date and time (i.e., time stamp) from each log line and performance counter sample. However, large-scale software systems are often distributed over several physical or virtual machines, therefore the timing of events may not be reliable (i.e., when the clocks of the

machines are not consistent) (Lamport, 1978). Fortunately, Lamport time stamps (Lamport, 1978) or vector clocks (Fidge, 1988; Mattern, 1988) may correct any inconsistencies in the machine's clocks. However, the performance counters and execution logs used in our case studies were generated from the same machine. Therefore, there are no issues regarding the timing of events. System experts also agree that this timing information is correct.

The timing of events may also be impacted if the time stamps in the performance counters and execution logs do not reliably reflect when the counters/logs were sampled/generated. However, we have found that the time stamps reflect the times that the counters/logs were sampled/generated, as opposed to the time the counters/logs were written to a file. Therefore, the time stamps of the performance counters and execution logs reliably reflect the true order of the events.

Our approach should only be used when the performance counters and execution logs are reliably collected.

5.5.1.2 Statistical Tests

Our approach relies on *unpaired two-sample two-tailed Welch's unequal variances t-test* (along with a *Cohen's d* effect size) to 1) identify outlying clusters and 2) to identify performance signature differences. Failure to satisfy the assumptions of this statistical test may affect the precision of our approach and undermine the statistical robustness of our approach (see Chapter 3 for a detailed discussion of this threat to validity).

5.5.1.3 Evaluation

We have evaluated our approach by determining the precision with which our approach flags execution events. While performance analysts have verified these results, we do not have a gold standard data set (see Chapter 3 for a detailed discussion of this threat to validity).

5.5.2 Threats to Internal Validity

5.5.2.1 Selecting Performance Counters

Our approach requires performance counters. However, there are hundreds of different performance counters, many of which are redundant (Malik et al., 2010). For example, memory usage may be measured by a number of different counters including, 1) allocated virtual memory, 2) allocated private (non-shared) memory and 3) the size of the working set (amount of memory used since the last sample). Performance analysts should sample all of the counters that may be relevant. Once the test is complete, performance analysts can than detect whether performance differences are seen in any of the counters and use our approach to diagnose these differences. However, performance analysts may require system-specific expertise to select an appropriate set of performance counters.

5.5.2.2 Selecting A Sampling Interval for the Performance Counters

Our approach requires performance counters. However, performance analysts must specify how often the performance counters are sampled (i.e., the sampling interval). We have previously found that sampling intervals between 90 seconds and

150 seconds perform well when comparing performance signatures. However, performance analysts may need to tune this parameter based on the duration of their performance tests and the sampling overhead on their system.

5.5.2.3 Execution Log Quality/Coverage

Our approach assumes that the cause of performance differences are manifested within the execution logs. However, it is possible that there are no execution logs to indicate when certain functionality is exercised (see Chapter 3 for a detailed discussion of this threat to validity).

5.5.2.4 Causes of Performance Differences

Our approach identifies important execution events that best explain the differences between the system's performance during a performance test and its performance during a certification test. However, our approach is not able to determine whether a performance difference is caused by a misconfiguration or a performance-degrading change to the system's source code (see Chapter 3 for a detailed discussion of this threat to validity).

5.5.3 Threats to External Validity

5.5.3.1 Generalizing Our Results

The studied software systems represent a small subset of the total number of large-scale software systems. Therefore, it is unclear how our results will generalize to additional software systems (see Chapter 3 for a detailed discussion of this threat to validity).

5.6 Conclusions

In the previous chapter, we presented an approach for enriching the execution events by leveraging *both* the static components of the execution logs *and* the dynamic information in the execution logs. Therefore, in this chapter, we presented an approach for enriching the execution events by combining the execution logs and the performance counters. Our approach identifies performance deviations from the release certification benchmark by deriving performance signatures from performance tests and certification tests using execution events that have been enriched with information from the performance counters. We then use statistical techniques to identify differences between the performance signatures of the performance test and the performance signatures of certification test. Such differences should help performance analysts to understand the cause of performance deviations, if any, from certification tests.

We performed three case studies on two systems: one open-source system and one enterprise system. Our case studies explored how our approach can be used to identify performance deviations from the release certification benchmark caused by performance defects and misconfigurations. Performance analysts and system experts have confirmed that our approach provides valuable insights that help to identify performance deviations from the release certification benchmark and to support the continuous performance testing process.

CHAPTER 6

Conclusions

Performance testing is the primary software performance engineering approach used to prevent failures in today's large-scale software systems (Woodside et al., 2007). The primary goal of performance testing is to understand how a system behaves during performance testing and how such behaviour differs from its historical behaviour (i.e., the system's behaviour during a previous performance test or the system's behaviour in the field). However, the system's behaviour is based on the behaviour of thousands or millions of users and continuously evolves as 1) the user base changes, 2) features are added or removed and 3) user feature preferences change (Bertolotti and Calzarossa, 2001; Voas, 2000). Further, the system's behaviour is described by gigabytes or terabytes of execution logs and performance counters (Chen et al., 2012; Syer et al., 2013, 2014; Xu et al., 2010). Little work

has been done to fully leverage the execution logs or enrich the execution events with additional sources of valuable information (Simic and Conklin, 2012). Therefore, performance analysts increasingly need support to enable performance testing within their cost and time constraints (Barber, 2004).

Therefore, in this thesis, we surveyed the state-of-the-art of performance testing large-scale software systems with a focus on how execution logs are used to 1) characterize a system's behaviour and 2) compare a system's current behaviour to its historical behaviour. We found that prior work tends to view execution logs only as execution events (i.e., prior work does not fully leverage execution logs nor does it enrich such logs with additional sources of valuable information). Motivated by our survey, prior research and experience, we proposed the following research hypothesis.

While there has been much work on characterizing a system's behaviour and comparing the system's current behaviour to its historical behaviour, prior work does not fully leverage the execution logs nor does it enrich such logs with additional sources of valuable information. Systematic approaches to comprehensively characterize a system's behaviour and compare this richer characterization of a system's current behaviour to its historical behaviour are needed to provide performance analysts with a deeper understanding of the behaviour of their system.

Therefore, the main contribution of our thesis is to present novel approaches that 1) *characterize* a system's behaviour using execution events that have been enriched with additional sources of valuable information and 2) *compare* the system's behaviour during performance testing to its historical behaviour to identify

the most significant differences. Our approaches fully leverage the information in the execution logs (i.e., *both* the static components of the execution logs *and* the dynamic information in the execution logs) and enrich the execution events with additional sources of valuable information (i.e., performance counters) to provide performance analysts with a more complete understanding of their system's behaviour. Through case studies on large-scale software systems, including open-source and enterprise systems, we showed that our approaches are scalable and can help performance analysts to understand the differences between their system's current behaviour and its historical behaviour. Knowledge of such differences should help performance analysts to 1) detect and diagnose performance regressions and 2) improve their performance tests to be more representative of the field. We also showed that by enriching the execution events our approaches outperform the state-of-the-practice.

This thesis opens up several promising avenues for future work.

First, our approaches compare the system's behaviour during performance testing to its historical behaviour to identify the most significant differences. As we demonstrated in our case studies, our approaches each identify different types of differences between the system's behaviour during performance testing to its historical behaviour. The approach that we presented in Chapter 3 determines whether there are any feature differences (i.e., differences in the exercised features), intensity differences (i.e., differences in how often each feature is exercised) and issue differences (i.e., new errors appearing in the field) between the system's behaviour during performance testing and its historical behaviour. The approach that we presented in Chapter 4 determines whether there are any feature usage difference (i.e.,

differences in how a particular feature is used) between the system's behaviour during performance testing and its historical behaviour. Finally, the approach that we presented in Chapter 5 determines whether there are any resource usage differences (e.g., CPU and memory usage) between the system's behaviour during performance testing and its historical behaviour. However, our approaches are currently used independently and in parallel (i.e., all three approaches are used simultaneously to compare the system's behaviour during performance testing to its historical behaviour). Therefore, future work should investigate the relationship between our approaches.

Second, execution logs contain developer and operator knowledge (i.e., they are manually inserted by developers). However, it is possible that there are no execution logs to indicate when certain functionality is exercised. Our approaches cannot provide any insight into these features nor can we enrich logs that do not exist. Therefore, future work should determine whether event traces generated by automated instrumentation tools (e.g., JProfiler (JProfiler, 2016) or the JVM Tool Interface (JVM Tool Interface, 2013)) can be enriched using the approaches presented in this thesis. Event traces may also be simpler to enrich with additional sources of valuable information (e.g., event traces and performance counters may be easier to combine than execution logs and performance counters).

Third, our approaches fully leverage the information in the execution logs and enrich the execution events with additional sources of valuable information (i.e., performance counters). However, other sources of information may also be valuable to performance analysts. For example, enriching the execution logs with valuable information from crash reports and unit test results could provide insight into

defect-prone events (e.g., whether the execution events occur preceding crashes or failed unit tests). Therefore, execution events could be enriched with additional sources of valuable information.

Finally, execution logs could be enriched with additional sources of valuable information to address new challenges. For example, execution events could be enriched with information from the development history to determine why a particular execution event is logged (e.g., whether the execution event is logged because it helps developers to diagnose defects).

Bibliography

Cristina L. Abad, Nathan Roberts, Yi Lu, and Roy H. Campbell. A Storage-Centric Analysis of MapReduce Workloads: File Popularity, Temporal Locality and Arrival Patterns. In *Proceedings of the International Symposium on Workload Characterization*, pages 100–109, Nov 2012.

Sara Alspaugh, Beidi Chen, Jessica Lin, Archana Ganapathi, Marti A. Hearst, and Randy Katz. Analyzing Log Analysis: An Empirical Study of User Log Mining. In *Proceedings of the Conference on Large Installation System Administration*, pages 53–68, Nov 2014.

Cristiana Amza, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick

Rajamani, Emmanuel Cecchet, and Julie Marguerite. Specification and Implementation of Dynamic Web Site Benchmarks. In *Proceedings of the International Workshop on Workload Characterization*, pages 3–13, Nov 2002.

Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating Realistic Workloads for Network Intrusion Detection Systems. In *Proceedings of the International Workshop on Software and Performance*, pages 207–215, Jan 2004.

Paul Ausick. NASDAQ Gets Off Cheap in Facebook IPO SNAFU. `http://finance.yahoo.com/news/nasdaq-gets-off-cheap-facebook-174557126.html`, 2012. Last Accessed: 26-Jun-2016.

Alberto Avritzer and Brian Larson. Load Testing Software Using Deterministic State Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 82–88, Jul 1993.

Alberto Avritzer and Elaine J. Weyuker. Generating Test Suites for Software Load Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 44–57, Aug 1994.

Alberto Avritzer and Elaine J. Weyuker. The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. *Transactions on Software Engineering*, 21(9):705–716, Sep 1995.

Alberto Avritzer, Joe Kondek, Danielle Liu, and Elaine J. Weyuker. Software Performance Testing Based on Workload Characterization. In *Proceedings of the International Workshop on Software and Performance*, pages 17–24, Jul 2002.

Giovann Ballocca, Roberto Politi, V. Russo, and Giancarlo Ruffo. Benchmarking a Site with Realistic Workload. In *Proceedings of the International Workshop on Workload Characterization*, pages 3–13, Nov 2002.

Scott Barber. Creating Effective Load Models for Performance Testing with Incomplete Empirical Data. In *Proceedings of the International Workshop on Web Site Evolution*, pages 51–59, Sep 2004.

Marcelo De Barros, Jing Shiau, Chen Shang, Kenton Gidewall, Hui Shi, and Joe Forsmann. Web Services Wind Tunnel: On Performance Testing Large-Scale Stateful Web Services. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 612–617, Jun 2007.

Julie Bataille. Operational Progress Report. `http://www.hhs.gov/digitalstrategy/blog/2013/12/operational-progress-report.html`, 2013. Last Accessed: 11-Nov-2015.

Michela Becchi, Mark Franklin, and Patrick Crowley. A Workload for Evaluating Deep Packet. Inspection Architectures. In *Proceedings of the International Symposium on Workload Characterization*, pages 79–89, Sep 2008.

David Benoit. Nasdaq's Blow-by-Blow on What Happened to Facebook. `http://blogs.wsj.com/deals/2012/05/21/nasdaqs-blow-by-blow-on-what-happened-to-facebook/`, 2012. Last Accessed: 26-Jun-2016.

Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the Workshop on Program Analysis for Software Tools*, pages 9–16, Sep 2011.

Laura Bertolotti and Maria Carla Calzarossa. Models of Mail Server Workloads. *Performance Evaluation*, 46(2-3):65–76, Oct 2001.

David Breitgand, Maayan Goldstein, Ealan Henis, and Onn Shehory. Performance Management via Adaptive Thresholds with Separate Control of False Positive and False Negative Errors. In *Proceedings of the International Conference on Symposium on Integrated Network Management*, pages 195–202, Jun 2009.

Teodora Sandra Buda. Generation of Test Databases Using Sampling Methods. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 366–369, Jul 2013.

Erik Burger and Ralf Reussner. Performance Certification of Software Components. *Journal of Electronic Notes in Theoretical Computer Science*, 279(2):33–41, Dec 2011.

Yuhong Cai, John Grundy, and John Hosking. Synthesizing Client Load Models for Performance Engineering via Web Crawling. In *Proceedings of the International Conference on Automated Software Engineering*, pages 353–362, Nov 2007.

T. Calinski and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics*, 3(1):1–27, Jan 1974.

Marcelo Cataldo, David Garlan, James Herbsleb, Amber Lynn McConahy, Young-Suk Ahn Park, and Bradley Schmerl. Software Platforms for Smart Building Ecosystems: Understanding the Key Architectural Capabilities and Trade-offs. Technical Report CMU-ISR-13-104, Carnegie Mellon University, Feb 2013.

Sung-Hyuk Cha. Comprehensive survey on distance/similarity measures between probability density functions. *International Journal of Mathematical Models and Methods in Applied Sciences*, 1(4):300–307, Nov 2007.

David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weyuker. A Framework for Testing Database Applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 147–157, Aug 2000.

Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proceedings of the Very Large Data Bases Endowment*, 5(12):1802–1813, Aug 2012.

Jacqui Cheng. Steve Jobs on MobileMe. `http://arstechnica.com/apple/2008/08/steve-jobs-on-mobileme-the-full-e-mail/`, 2008. Last Accessed: 26-Jun-2016.

Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Anomaly? Application Change? or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 452–461, Jun 2008.

Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Automated Anomaly Detection and Performance Modeling of Enterprise Applications. *Transactions on Computer Systems*, 27(3):6:1–6:32, Nov 2009.

Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. Event Logs for the

Analysis of Software Failures: A Rule-Based Approach. *Transactions on Software Engineering*, 39(6):806–821, Jun 2013.

William G. Cochran. Some methods for strengthening the common chi-squared tests. *Biometrics*, 10(4):417–451, Jan 1954.

Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge Academic, second edition, 1988.

Jacob Cohen. A Power Primer. *Psychological Bulletin*, 112(1):155–159, Jul 1992.

Coleman Parkes. The Avoidable Cost of Downtime. `http://www.ca.com/~/media/files/articles/avoidable_cost_of_downtime_part_2_ita.aspx`, 2011. Last Accessed: 26-Jun-2016.

Compuware. Application Performance Management Survey. `http://www.cnetdirectintl.com/direct/compuware/Ovum_APM/APM_Survey_Report.pdf`, 2006. Last Accessed: 25-Oct-2016.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the Symposium on Cloud Computing*, pages 143–154, Jun 2010.

Cristiano Costa, Italo Cunha, and Jussara M. Almeida. GENIUS: a Generator of Interactive User Media Sessions. In *Proceedings of the International Workshop on Workload Characterization*, pages 29–36, Oct 2004.

Domenico Cotroneo, Roberto Pietrantuono, Leonardo Mariani, and Fabrizio Pastore. Investigation of Failure Causes in Workload-driven Reliability Testing. In

Proceedings of the International Workshop on Software Quality Assurance, pages 78–85, Sep 2007.

Carlos V. Damásio, Peter Fröhlich, Wolfgang Nejdl Luis Moniz, Pereira, and Michael Schroeder. Using Extended Logic Programming for Alarm-Correlation in Cellular Phone Networks. *Applied Intelligence*, 17(2):187–202, Jul 2002.

Miyuru Dayarathna and Toyotaro Suzumura. Graph Database Benchmarking on Cloud Environments with XGDBench. *Genetic Programming and Evolvable Machines*, 21(4):509–533, Dec 2014.

Eduardo Cunha de Almeida, Gerson Sunyé, Yves La Traon, and Patrick Valduriez. Testing Peer-to-Peer Systems. *Empirical Software Engineering*, 15(4):346–379, Aug 2010.

Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, Feb 2013.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan 2008.

Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Decoupling Datacenter Studies from Access to Large-scale Applications: A Modeling Approach for Storage Workloads. In *Proceedings of the International Symposium on Workload Characterization*, pages 51–60, Nov 2011.

Christina Delimitrou, Sriram Sankar, Aman Kansal, and Christos Kozyrakis. ECHO: Recreating Network Traffic Maps for Datacenters with Tens of Thousands of

Servers. In *Proceedings of the International Symposium on Workload Characterization*, pages 14–24, Nov 2012.

John A. Dilley. Web Server Workload Characterization. Technical Report HPL-96-160, Hewlett-Packard Laboratories, Dec 1996.

Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. Realistic Load Testing of Web Applications. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 57–68, Mar 2006.

Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons Inc, first edition, 1973.

Subhasri Duttagupta and Manoj Nambiar. Performance Extrapolation for Load Testing Results of Mixture of Applications. In *Proceedings of the European Symposium on Computer Modeling and Simulation*, pages 424–429, Nov 2011.

Ahmed Ali Eldin, Ali Rezaie, Amardeep Mehta, Stanislav Razroev, Sara Sjostedt de Luna, Oleg Seleznjev, Johan Tordsson, and Erik Elmroth. How Will Your Workload Look Like in 6 Years? Analyzing Wikimedia's Workload. In *Proceedings of the International Conference on Cloud Engineering*, Mar 2014.

Alan C. Elliott. *Statistical Analysis Quick Reference Guidebook*. Sage Publications, first edition, 2006.

Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 50–60, Aug 2015.

Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Australian Computer Science Conference*, pages 56–66, Feb 1988.

Ronald Aylmer Fisher. On a distribution yielding the error functions of several well known statistics. *Proceedings of the International Congress of Mathematics*, 85(1): 87–94, Jan 1924.

King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, Kim Martin, and Parminder Flora. Mining Performance Regression Testing Repositories for Automated Performance Analysis. In *Proceedings of the International Conference on Quality Software*, pages 32–41, Jul 2010.

Itziar Frades and Rune Matthiesen. Overview on techniques in cluster analysis. *Bioinformatics Methods In Clinical Research*, 593:81–107, Mar 2009.

G. H. Freeman and J. H. Halton. Note on an exact treatment of contingency, goodness of fit and other problems of significance. *Biometrika*, 38(1-2):141–149, Jun 1951.

Jim Frey. Log Analytics for Network Operations Management. Technical report, Enterprise Management Association, Feb 2015.

Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution Anomaly Detection in Distributed Systems Through Unstructured Log Analysis. In *Proceedings of the International Conference on Data Mining*, pages 149–158, Dec 2009.

Qiang Fu, Jian-Guang Lou, Qingwei Lin, Rui Ding, Dongmei Zhang, and Tao Xie. Contextual Analysis of Program Logs for Understanding System Behaviors. In

Proceedings of the Working Conference on Mining Software Repositories, pages 397–400, May 2013.

Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Proceedings of the International Conference on Software Engineering*, pages 24–33, May 2014.

M. H. Fulekar. *Bioinformatics: Applications in Life and Environmental Sciences*. Springer, first edition, 2008.

Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube Traffic Characterization: A View from the Edge. In *Proceedings of the Conference on Internet Measurement*, pages 15–28, Oct 2007.

Katerina Goseva-Popstojanova, AjayDeep Singh, Sunil Mazimdar, and Fengbin Li. Empirical Characterization of Session-Based Workload and Reliability for Web Servers. *Empirical Software Engineering*, 11(1):868–882, Mar 2006.

Dominic Greenwood, Margaret Lyell, Ashok Mallya, and Hiroki Suguri. The IEEE FIPA approach to integrating software agents and web services. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1412–1418, May 2007.

Alexey Grishchenko. Hadoop MapReduce Comprehensive Description. `https://0x0fff.com/hadoop-mapreduce-comprehensive-description/`, 2014. Last Accessed: 04-Jul-2016.

Hadoop. Hadoop. http://hadoop.apache.org/, 2014. Last Accessed: 26-Jun-2016.

Hadoop. Onboarding. http://hadoop.apache.org/docs/r2.7.2/, 2016. Last Accessed: 26-Jun-2016.

Hadoop-LZO. Hadoop-LZO. https://github.com/twitter/hadoop-lzo, 2011. Last Accessed: 26-Jun-2016.

Robert J. Hall. A Method and Tools for Large Scale Scenarios. *Automated Software Engineering*, 15(2):113–148, Jun 2008.

Chandler Harris. IT Downtime Costs $26.5 Billion In Lost Revenue. http://www.informationweek.com/it-downtime-costs-$265-billion-in-lost-revenue/d/d-id/1097919?, 2011. Last Accessed: 26-Jun-2016.

Ahmed E. Hassan and Parminder Flora. Performance engineering in industry: current practices and adoption challenges. In *Proceedings of the International Workshop on Software and Performance*, pages 209–209, Feb 2007.

Ahmed E. Hassan and Ken Zhang. Using Decision Trees to Predict the Certification Result of a Build. In *Proceedings of the International Conference on Automated Software Engineering*, pages 189–198, Sep 2006.

Ahmed E. Hassan, Daryl J. Martin, Parminder Flora, Paul Mansfield, and Dave Dietz. An Industrial Case Study of Customizing Operational Profiles Using Log Compression. In *Proceedings of the International Conference on Software Engineering*, pages 713–723, May 2008.

Joseph L. Hellerstein, Sheng Ma, and Chang-Shing Perng. Discovering Actionable Patterns in Event Data. *IBM Systems Journal*, 41(1):475–493, Jul 2002.

Petra Hofer. Onboarding. `http://www.ebaytechblog.com/2011/05/04/onboarding/`, 2011. Last Accessed: 26-Jun-2016.

Johannes Holvitie, Ville Leppänen, and Sami Hyrynsalmi. Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey. In *Proceedings of the International Workshop on Managing Technical Debt*, pages 35–42, Sep 2014.

Anna Huang. Similarity measures for text document clustering. In *Proceedings of the New Zealand Computer Science Research Student Conference*, pages 44–56, Apr 2008.

Frank Huebner, Kathleen S. Meier-Hellstern, and Paul Reeser. Performance Testing for IP Services and Systems. In *Performance Engineering, State of the Art and Current Trends*, pages 283–299, Jan 2001.

Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley Computer Publishing, John Wiley & Sons, Inc., first edition, 1991.

Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing Data Analysis Workloads in Data Centers. In *Proceedings of the International Symposium on Workload Characterization*, pages 66–76, Sep 2013.

Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A. Mckee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and Subsetting Big Data Workloads. In

Proceedings of the International Symposium on Workload Characterization, pages 191–201, Oct 2014.

Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In *Proceedings of the International Conference on Automatic Computing*, pages 111–122, Dec 2005.

Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. Multiresolution Abnormal Trace Detection Using Varied-Length n-Grams and Automata. *Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37(1):86–97, Jan 2007.

Zhen Ming Jiang and Ahmed E. Hassan. A Survey on Load Testing of Large-Scale Software Systems. *Transactions on Software Engineering*, 41(11):1091–1118, Nov 2000.

Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. An Automated Approach for Abstracting Execution Logs to Execution Events. *Journal of Software Maintenance and Evolution*, 20(4):249–267, Jul 2008a.

Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Abstracting Execution Logs to Execution Events for Enterprise Applications. In *Proceedings of the International Conference on Quality Software*, pages 181–186, Aug 2008b.

Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated Identification of Load Testing Problems. In *Proceedings of the International Conference on Software Maintenance*, pages 307–316, Oct 2008c.

Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated Performance Analysis of Load Tests. In *Proceedings of the International Conference on Software Maintenance*, pages 125–134, Sep 2009.

JobConf. JobConf. `https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/JobConf.html`, 2016. Last Accessed: 26-Jun-2016.

Robert Johnson. More Details on Today's Outage. `https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919`, 2010. Last Accessed: 26-Jun-2016.

JProfiler. Java Profiler - JProfiler. `https://www.ej-technologies.com/products/jprofiler/overview.html`, 2016. Last Accessed: 27-Jul-2016.

Tom Howell Jr. and Stephen Dinan. Price of fixing, upgrading obamacare website rises to $121 million. `http://www.washingtontimes.com/news/2014/apr/29/obamacare-website-fix-will-cost-feds-121-million/`, 2014. Last Accessed: 26-Jun-2016.

JVM Tool Interface. JVM Tool Interface. `https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`, 2013. Last Accessed: 27-Jul-2016.

Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. A Systematic Review of Effect Size in Software Engineering Experiments. *Information and Software Technology*, 49(11-12):1073–1086, Nov 2007.

Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *Proceedings of the International Conference on Cluster, Cloud and Grid Computing*, pages 94–103, May 2010.

Adam Kawa. Process a Million Songs with Apache Pig. `http://blog.cloudera.com/blog/2012/08/process-a-million-songs-with-apache-pig/`, 2012. Last Accessed: 26-Jun-2016,.

Olivia Klose. Hadoop on Linux on Azure. `https://blogs.technet.microsoft.com/oliviaklose/2014/06/17/hadoop-on-linux-on-azure-1/`, 2014. Last Accessed: 26-Jun-2016.

Ted Kremenek and Dawson Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the International Conference on Static Analysis*, pages 295–315, Jun 2003.

Raffi Krikorian. New Tweets per second record, and how! `https://blog.twitter.com/2013/new-tweets-per-second-record-and-how`, 2013. Last Accessed: 26-Jun-2016.

Diwakar Krishnamurthy, Jerome A. Rolia, and Shikharesh Majumdar. A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems. *Transactions on Software Engineering*, 32(11):868–882, Nov 2006.

Thomas Kunz. High-Level Views of Distributed Executions: Convex Abstract Events. *Automated Software Engineering*, 4(2):179–197, Apr 1997.

Christian Kurz, Carlos Guerrero, and Günter Haring. Extending TPC-W to Allow for

Fine Grained Workload Specification. In *Proceedings of the International Workshop on Software and Performance*, pages 167–174, Jul 2005.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Jul 1978.

Alf Larsson and Abdelwahab Hamou-Lhadj. Mining Telecom System Logs to Facilitate Debugging Tasks. In *Proceedings of the International Working Conference on Software Maintenance*, pages 536–539, Sep 2013.

Michael A. Laurenzano, Joshua Peraza, Laura Carrington, Ananta Tiwari, William A. Ward Jr., and Roy Campbell. Pebil: Binary instrumentation for practical data-intensive program analysis. *Cluster Computing*, 1(18):1–14, Mar 2015.

Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining Invariants from Console Logs for System Problem Detection. In *Proceedings of the USENIX Annual Technical Conference*, pages 24:1–24:14, Jun 2010.

Xingmin Luo, Fan Ping, and Mei-Hwa Chen. Clustering and Tailoring User Session Data for Testing Web Applications. In *Proceedings of the International Conference on Software Testing Verification and Validation*, pages 336–345, Apr 2009.

Haroon Malik, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. Automatic Comparison of Load Tests to Support the Performance Analysis of Large Enterprise Systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 222–231, Mar 2010.

Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems. In *Proceedings*

of the International Conference on Software Engineering, pages 1012–1021, May 2013.

David Maplesden, Ewan Tempero, John Hosking, and John C. Grundy. Performance Analysis for Object-Oriented Software: A Systematic Mapping. *Transactions on Software Engineering*, 41(7):691–710, Jul 2015.

MapReduce Tutorial. MapReduce Tutorial. `http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`, 2014. Last Accessed: 26-Jun-2016.

Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226, Oct 1988.

Jorge Augusto Meira, Eduardo Cunha de Almeida, Yves Le Traon, and Gerson Sunye. Peer-to-peer load testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 642–647, Apr 2012.

Daniel A. Menascé. Workload Characterization. *Internet Computing*, 7(5):89–92, Sep 2003.

Daniel A. Menascé, Virgilio A. F. Almeida, Rodrigo Fonseca, and Marco A. Mendes. A Methodology for Workload Characterization of E-commerce Sites. In *Proceedings of the Conference on Electronic Commerce*, pages 119–128, Nov 1999.

Metrics 2.0. Metrics 2.0. `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/metrics2/package-summary.html`, 2016. Last Accessed: 26-Jun-2016.

Ningfang Mi, Ludmila Cherkasova, Kivanc Ozonat, Julie Symons, and Evgenia Smirni. Analysis of Application Performance and its Change via Representative Application Signatures. In *Proceedings of the Symposium on Network Operations and Management*, pages 216–223, Apr 2008.

Glenn W. Milligan and Martha C. Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50(2):159–179, Jun 1985.

Million Song Dataset. Million Song Dataset. `http://labrosa.ee.columbia.edu/millionsong/`, 2012. Last Accessed: 26-Jun-2016.

Million Song Dataset. Million Song Dataset. `https://aws.amazon.com/datasets/million-song-dataset/`, 2015. Last Accessed: 26-Jun-2016.

R. Mojena. Hierarchical grouping methods and stopping rules: An evaluation. *The Computer Journal*, 20(4):353–363, Nov 1977.

Ricardo Morin, Anil Kumar, and Elena Ilyina. A Multi-Level Comparative Performance Characterization of SPECjbb2005 Versus SPECjbb2000. In *Proceedings of the International Symposium on Workload Characterization*, pages 67–75, Oct 2005.

John Murrell. Firefox Download Stunt Sets Record For Quickest Meltdown. `http://www.siliconbeat.com/2008/06/17/firefox-download-stunt-sets-record-for-quickest-meltdown/`, 2008. Last Accessed: 26-Jun-2016.

Meiyappan Nagappan and Mladen A. Vouk. Abstracting Log Lines to Log Event

Types for Mining Software System Logs. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 114–117, May 2010.

Meiyappan Nagappan, Kesheng Wu, and Mladen A. Vouk. Efficiently Extracting Operational Profiles from Execution Logs Using Suffix Arrays. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 41–50, Nov 2009.

Priya Nagpurkar, William Horn, U. Gopalakrishnan, Niteesh Dubey, Joefon Jann, and Pratap Pattnaik. Workload Characterization of Selected JEE-based Web 2.0 Applications. In *Proceedings of the International Symposium on Workload Characterization*, pages 109–118, Sep 2008.

Thanh H. D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated Verification of Load Tests Using Control Charts. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 282–289, Dec 2011.

Thanh H. D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated Detection of Performance Regressions Using Statistical Process Control Techniques. In *Proceedings of the International Conference on Performance Engineering*, pages 299–310, Apr 2012.

Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. An Industrial Case Study of Automatically Identifying Performance Regression-causes. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 232–241, May 2014.

Tuli Nivas and Christoph Csallner. Managing Performance Testing with Release Certification and Data Correlation. In *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, Sep 2011.

Object Management Group. UML2.0. `http://www.omg.org/spec/UML/2.0/`, 2016. Last Accessed: 23-Aug-2016.

OutputCommitter. OutputCommitter. `http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/OutputCommitter.html`, 2016. Last Accessed: 26-Jun-2016.

David Lorge Parnas. Software Aging. In *Proceedings of the International Conference on Software Engineering*, pages 279–287, May 1994.

Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. Industry Practices and Event Logging: Assessment of a Critical Software Development Process. In *Proceedings of the International Conference on Software Engineering*, pages 169–178, May 2015.

PerfMon. PerfMon. `http://perfmon.sourceforge.net/`, 2016. Last Accessed: 26-Jun-2016.

Nicolas Poggi, David Carrera, Ricard Gavalda, Jordi Torres, and Eduard Ayguade. Characterization of Workload and Resource Consumption for an Online Travel and Booking Site. In *Proceedings of the International Symposium on Workload Characterization*, pages 1–10, Dec 2010.

Nicolas Poggi, David Carrera, Ricard Gavaldà, Eduard Ayguadé, and Jordi Torres. A Methodology for the Evaluation of High Response Time on E-commerce Users and Sales. *Information Systems Frontiers*, 16(5):867–885, Nov 2014.

Ariel Rabkin, Wei Xu, Avani Wildani, Armando Fox, David Patterson, and Randy Katz. A Graphical Representation for Identifier Structure in Logs. In *Proceedings of the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, pages 3:1–3:9, Oct 2010.

RecordReader. RecordReader. `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/RecordReader.html`, 2016. Last Accessed: 26-Jun-2016.

Zujie Ren, Xianghua Xu, Jian Wan, Weisong Shi, and Min Zhou. Workload Characterization on a Production Hadoop Cluster: A Case Study on Taobao. In *Proceedings of the International Symposium on Workload Characterization*, pages 3–13, Nov 2012.

Zujie Ren, Weisong Shi, and Jian Wan. Towards Realistic Benchmarking for Cloud File Systems: Early Experiences. In *Proceedings of the International Symposium on Workload Characterization*, pages 88–98, Oct 2014.

Marcello La Rosa. Process Mining: A Database of Applications. In *Proceedings of the Annual Meeting of the IEEE Task Force on Process Mining*, Sep 2016.

P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20(1):53–65, Nov 1987.

Vikram Saletore, Karthik Krishnan, Vish Viswanathan, and Matt Tolentino. HcBench: Methodology, Development, and Characterization of a Customer Usage Representative Big Data/Hadoop Benchmark. In *Proceedings of the International Symposium on Workload Characterization*, pages 77–86, Sep 2013.

Felix Salfner and Steffen Tschirpke. Error Log Processing for Accurate Failure Prediction. In *Proceedings of the Conference on Analysis of System Logs*, pages 4:1–4:8, Dec 2008.

S. Ratna Sandeep, M. Swapna, Thirumale Niranjan, Sai Susarla, and Siddhartha Nandi. CLUEBOX: A Performance Log Analyzer for Automated Troubleshooting. In *Proceedings of the Conference on Analysis of System Logs*, pages 1:1–1:8, Dec 2008.

Nadella Sandhya and Aliseri Govardhan. Analysis of similarity measures with wordnet based text document clustering. In *Proceedings of the International Conference on Information Systems Design and Intelligent Applications*, pages 703–714, Jan 2012.

Ebada Sarhan, Atif Ghalwash, and Mohamed Khafagy. Specification and Implementation of Dynamic Web Site Benchmark in Telecommunication Area. In *Proceedings of the International Conference on Computers*, pages 863–867, Jul 2008.

Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. In *Proceedings of the Working Conference on Reverse Engineering*, pages 335–344, Oct 2011.

Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. Assisting Developers of Big Data Analytics Applications When Deploying on Hadoop Clouds. In *Proceedings of the International Conference on Software Engineering*, pages 402–411, May 2013.

Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan, and Zhen Ming Jiang. Understanding Log Lines Using Development Knowledge. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 21–30, Sep 2014.

Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):20:1–20:27, Feb 2015.

Bojan Simic and Sean Conklin. Improving the Usability of APM Data: Essential Capabilities and Benefits. `https://assets.extrahop.com/uploads/trac-market-insight-usability-of-apm-data.pdf`, 2012. Last Accessed: 28-Mar-2014.

Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Longman Publishing Co., Inc., first edition, 2002.

Software Engineering Institute. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon University, 2006.

Robert R. Sokal and F. James Rohlf. *Biometry: the principles and practice of statistics in biological research*. W. H. Freeman, fourth edition, 2011.

Jon Stearley. Towards Informatic Analysis of Syslogs. In *Proceedings of the International Conference on Cluster Computing*, pages 309–318, Sep 2004.

Robert Stets, Kourosh Gharachorloo, and Luiz Andre Barroso. A Detailed Comparison of Two Transaction Processing Workloads. In *Proceedings of the International Workshop on Workload Characterization*, pages 37–48, Nov 2002.

Christopher Stewart, Matthew Leventi, and Kai Shen. Empirical Examination of a Collaborative Web Application. In *Proceedings of the International Symposium on Workload Characterization*, pages 90–96, Sep 2008.

StringTokenizer. StringTokenizer. `https://docs.oracle.com/javase/7/docs/api/java/util/StringTokenizer.html`, 2016. Last Accessed: 26-Jun-2016.

Student. The probable error of a mean. *Biometrika*, 6(1):1–25, Mar 1908.

Mark D. Syer, Bram Adams, and Ahmed E. Hassan. Industrial case study on supporting the comprehension of system behaviour. In *Proceedings of the International Conference on Program Comprehension*, pages 215–216, Jun 2011a.

Mark D. Syer, Bram Adams, and Ahmed E. Hassan. Identifying Performance Deviations in Thread Pools. In *Proceedings of the International Conference on Software Maintenance*, pages 83–92, Sep 2011b.

Mark D. Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues. In *Proceedings of the International Conference on Software Maintenance*, pages 110–119, Sep 2013.

Mark D. Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Continuous Validation of Load Test Suites. In *Proceedings of the International Conference on Performance Engineering*, pages 232–241, Mar 2014.

SYSSTAT. SYSSTAT. `http://sebastien.godard.pagesperso-orange.fr/`, 2016. Last Accessed: 26-Jun-2016.

Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. SALSA: Analyzing Logs As State Machines. In *Proceedings of the International Conference on Analysis of System Logs*, pages 6:1–6:8, Dec 2008.

Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Cluster Analysis: Basic Concepts and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., first edition, 2005.

TextInputFormat. TextInputFormat. `http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/TextInputFormat.html`, 2016. Last Accessed: 26-Jun-2016.

The Sarbanes-Oxley Act. The Sarbanes-Oxley Act 2002. `http://soxlaw.com/`, 2006. Last Accessed: 26-Jun-2016.

Jeff Tian, Sunita Rudraraju, and Zhao Li. Evaluating Web Software Reliability Based on Workload and Failure Data Extracted from Server Logs. *Transactions on Software Engineering*, 30(11):754–769, Nov 2004.

Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 56–64, Oct 2006.

Risto Vaarandi. A Data Clustering Algorithm for Mining Patterns from Event Logs. In *Proceedings of the Workshop on IP Operations Management*, pages 119–126, Oct 2003.

Jeffrey Voas. Will the Real Operational Profile Please Stand Up? *IEEE Software*, 17 (2):87–89, Mar 2000.

Bernard Lewis Welch. The generalization of "student's" problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, Jan 1997.

Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Transactions on Software Engineering*, 26(12):1147–1156, Dec 2000.

Alex Williams. Amazon web services outage caused by memory leak and failure in monitoring alarm. `https://techcrunch.com/2012/10/27/amazon-web-services-outage-caused-by-memory-leak-and-failure-in-monitoring-alarm/`, 2012. Last Accessed: 26-Jun-2016.

Murray Woodside, Greg Franks, and Dorina C. Petriu. The Future of Software Performance Engineering. In *Proceedings of the Future of Software Engineering*, pages 171–187, May 2007.

Huafeng Xi, Jianfeng Zhan, Zhen Jia, Xuehai Hong, Lei Wang, Lixin Zhang, Ninghui Sun, and Gang Lu. Characterization of Real Workloads of Web Search Engines. In

Proceedings of the International Symposium on Workload Characterization, pages 15–25, Nov 2011.

Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting Large-scale System Problems by Mining Console Logs. In *Proceedings of the Symposium on Operating Systems Principles*, pages 117–132, Oct 2009a.

Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Online System Problem Detection by Mining Patterns of Console Logs. In *Proceedings of the International Conference on Data Mining*, pages 588–597, Dec 2009b.

Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Experience Mining Google's Production Console Logs. In *Proceedings of the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, pages 5:1–5:9, Oct 2010.

Philip S. Yu, Ming-Syan Chen, Hans-Ulrich Heiss, and Sukho Lee. On Workload Characterization of Relational Database Environments. *Transactions on Software Engineering*, 18(4):347–355, Apr 1992.

Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing Logging Practices in Open-source Software. In *Proceedings of the International Conference on Software Engineering*, pages 102–112, Jun 2012.

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive

systems. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 249–265, Oct 2014.

Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security Versus Performance Bugs: A Case Study on Firefox. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 93–102, May 2011.

Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A Large Scale Empirical Study on User-Centric Performance Analysis. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 410–419, Apr 2012a.

Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A Qualitative Study on Performance Bugs. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 199–208, May 2012b.

Jian Zhang and Shing-Chi Cheung. Automated Test Case Generation for the Stress Testing of Multimedia Systems. *Software: Practices and Experiences*, 32:1411–1435, Dec 2002.

Steve Zhang, Ira Cohen, Julie Symons, and Armando Fox. Ensembles of Models for Automated Diagnosis of System Performance Problems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 644–653, Jul 2005.

Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Benchmarking approach for designing a mapreduce performance model. In *Proceedings of the International Conference on Performance Engineering*, pages 253–258, Apr 2013.

Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, and Zhiteng Huang. COSBench: A Benchmark Tool for Cloud Object Storage Services. In *Proceedings of the International Conference on Performance Engineering*, pages 998–999, Jun 2012.

Qing Zheng, Haopeng Chen, Yaguang Wang, Jian Zhang, and Jiangang Duan. COSBench: Cloud Object Storage Benchmark. In *Proceedings of the International Conference on Performance Engineering*, pages 199–210, Apr 2013.

APPENDIX A

Literature Survey Criteria

In this appendix, we briefly explain the criteria that we used to conduct the literature survey we presented in Chapter 2. We surveyed the following venues for relevant papers:

- Transactions on Software Engineering (TSE, 1975-2015).
- International Conference on Software Engineering (ICSE, 1976-2014).
- Conference on Software Maintenance (CSM, 1988-1993) and its successors the International Conference on Software Maintenance (ICSM, 1994-2013) and the International Conference on Software Maintenance and Evolution (ICSME, 2014).
- International Symposium on Software Testing and Analysis (ISSTA 1993-2014).

- Automated Software Engineering Journal (ASEJ, 1994-2015).
- Empirical Software Engineering (EMSE, 1996-2015).
- International Conference on Automated Software Engineering (ASE, 1997-2014).
- International Workshop on Software and Performance (WOSP, 1998-2008) and its successor the International Conference on Performance Engineering (ICPE, 2010-2014).
- Workload Characterization: Methodology and Case Studies (WWC, 1999) and its successors the International Workshop on Workload Characterization (WWC, 2001-2004) and the International Workload Characterization Symposium (IISWC, 2005-2014).
- Asia-Pacific Conference on Quality Software (APAQS, 2000-2001) and its successor the International Conference on Quality Software (QSIC, 2003-2014).
- International Workshop on Mining Software Repositories (MSR, 2004-2014).
- International Conference on Software Testing, Verification, and Validation (ICST, 2008-2014).
- Workshop on the Analysis of System Logs (WASL, 2008-2009) and its successors the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML, 2010) and the Workshop on Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML, 2011-2012).

We also surveyed the following venues that were co-located with the above venues:

- International Workshop on Software Test Evaluation (STEV, 2007) – STEV was co-located with QSIC in 2007.
- Workshop on Data Quality for Software Engineering (DQ4SE, 2009) – DQ4SE was co-located with QSIC in 2009.
- International Workshop on Load Testing of Large Software (LT, 2012) – LT was co-located with ICST in 2012.
- The Symposium on Engineering Test Harnesses (TSE-TH, 2013) –TSE-TH was co-located with QSIC in 2013.
- International Workshop on Large-Scale Testing (LT, 2014) – LT was co-located with ICPE in 2014.

In total, 10,987 abstracts from the above venues were surveyed. We then identified 54 papers as relevant for our literature survey. We consider a paper relevant for our literature survey if it presents a new or enhanced approach to 1) workload characterization or 2) workload comparison of large-scale software systems. Papers that present a new or enhanced approach to workload characterization include keywords such as “benchmarking,” “operational profiling,” “workload characterization,” “workload modelling,” “workload profiling” and “workload tracing.” These papers must focus on large-scale software systems. Papers that present a new or enhanced approach to workload comparison include keywords such as “anomaly detection,” “benchmark comparison,” “load test analysis,” “performance baseline,”

"performance comparison," "performance regression," "performance test analysis," "reliability test analysis," "stress test analysis" and "workload comparison." These papers must focus on large-scale software systems and compare at least two workloads (i.e., the system's current behaviour to its historical behaviour). We also consider a paper relevant for our literature survey if it presents an empirical study on execution logs or presents a new or enhanced approach to abstracting execution logs. These papers include keywords such as "execution logs," "log abstraction," "log parsing" and "log processing." Finally, we read the abstracts for each of the references of the 54 papers and identified an additional 34 papers. We restrict our literature survey to 1) papers published in the venues above and 2) papers referenced by papers published in the venues above that meet our inclusion criteria. for practical reasons. These papers are summarized in Appendix B.

APPENDIX B

Brief Summaries of the Surveyed Papers

In this appendix, we briefly describe each paper that was included in the literature survey that we presented in Chapter 2.

B.1 Workload Characterization

Characterizing a system's workload is crucial to understanding how a system's users are interacting with the system and how the system behaves as a result (e.g., understanding use cases and feature preferences). The most common application of workload characterization is to generate a benchmark workload for performance testing (e.g., characterizing the field workload to create a performance test workload that is representative of the field). Therefore, we present relevant literature

on 1) benchmarks and 2) workload characterization.

B.1.1 Benchmarks

Abad et al. (2012) characterize a six month workload trace from two Hadoop clusters (one with 4,146 nodes and one with 1,958 nodes) at Yahoo. They characterized file popularity, temporal locality and arrival patterns of the workloads

de Almeida et al. (2010) propose a framework to performance peer-to-peer systems that allows performance analysts to control each node. The tool allows performance analysts to simulate peers joining and leaving the system.

Amza et al. (2002) create benchmarks for 1) e-commerce and 2) bulletin board web sites with dynamic content.

Antonatos et al. (2004) propose a tool that allows performance analysts to manually create and execute a benchmark workload against network intrusion detection systems.

Becchi et al. (2008) create a benchmark for regular expression matching systems.

Bertolotti and Calzarossa (2001) create a benchmark for email server systems.

Buda (2013) proposes a tool that automatically populates a database with data sampled from the field.

Chays et al. (2000) propose a tool that automatically populates a database with data that conforms to pre-specified constraints (e.g., table relationships).

Chen et al. (2012) characterize the MapReduce workload at Facebook and Cloudera.

Cooper et al. (2010) create benchmarks for cloud transaction processing systems.

Costa et al. (2004) propose an approach to create hierarchical models of user behaviour based on workload characterization. They also propose GENIUS, a tool to create a workload based on these models of user behaviour and additional workload parameters (e.g., workload duration and intensity).

Dayarathna and Suzumura (2014) extend the work by Cooper et al. (2010) to create a benchmark workload for graph databases.

Gill et al. (2007) characterize a three month workload trace of YouTube usage on a corporate network.

Jia et al. (2013) propose BCBench, a benchmark workload for data centres created by characterizing eleven different data analysis workloads.

Kavulya et al. (2010) characterize a ten month workload trace from the M45 Hadoop cluster at Yahoo. They characterized resource utilization patterns, job patterns and sources of failures in the workload.

Morin et al. (2005) perform an empirical comparison of a system's performance under two different server-size Java benchmarks. They find that the system's performance differs considerably between the two benchmarks.

Nagpurkar et al. (2008) characterize workloads for two Web 2.0 web sites: 1) Lotus Connections and 2) Java Pet Store 2.0.

Poggi et al. (2010) characterize the workload for an online travel and booking web site.

Ren et al. (2012) characterize a two week workload trace from a 2,000 node production cloud file store and processing platform (Hadoop) at Taobao.

Ren et al. (2014) propose a benchmark for cloud file systems by surveying a two week I/O workload trace from a 2,500 node production cloud file store and processing platform (Hadoop).

Saletore et al. (2013) propose a Hadoop benchmark based on typical customer usage.

Sarhan et al. (2008) create a benchmark for telecommunication web sites with dynamic content.

Stets et al. (2002) perform an empirical comparison of a system's performance under two different benchmarks. They find that the system's performance differs considerably between the two benchmarks.

Stewart et al. (2008) create a benchmark for collaborative web sites (e.g., social networks and wiki-based web sites).

Xi et al. (2011) characterize the workload of three web search systems by search query patterns. They find that search query patterns do not follow well-defined distributions and that synthetic workloads are not representative of the field.

Yu et al. (1992) characterize the workload for the DB2 database component of an accounting system using the Relational Database Workload AnalyzeR (REDWAR) developed by IBM.

Zheng et al. (2012, 2013) propose Cloud Object Storage Benchmark, a tool that allows performance analysts to manually create and execute a benchmark workload against cloud storage systems. The tool can also be used to help performance analysts 1) identify performance bottlenecks and 2) configure their system.

B.1.2 Workload Characterization

Avritzer and Larson (1993); Avritzer and Weyuker (1994, 1995) propose an approach to workload characterization by profiling the system and creating a Markov model of the system from the data collected. These models are then used to generate a workload for performance testing.

Avritzer et al. (2002) motivate the need to design performance testing workload by characterizing field workloads. They also present a case study where workload characterization helped to improve performance tests and uncover a performance bottleneck.

Ballocca et al. (2002) propose a tool that automatically creates a benchmark workload against e-commerce systems by mining user behaviour from execution logs.

Cai et al. (2007) propose MaramaMTE+, a tool to model user behaviour using a stochastic form chart. Their tool 1) builds a model of the web site's structure using WebSphinx, 2) creates a stochastic form chart of the user's potential behaviour (augmented with domain knowledge) and 3) generates a workload from the stochastic form chart susing JMeter.

Barros et al. (2007) propose an approach to workload characterization by creating Markov models of user behaviour from execution logs. Their approach resulted in performance tests that were more field representative and identified more functional and performance issues. They also discuss workload characterization challenges such as data integrity and sanitization.

Delimitrou et al. (2011) propose an approach to characterize data centre workloads using hierarchical representations of state-diagrams. They also propose an

approach to generate I/O workloads using these models.

Delimitrou et al. (2012) propose ECHO, a tool to characterize network traffic workloads using hierarchical Markov Chain models. Their tool can also produce a workload for performance testing based on these models.

Dilley (1996) proposes an approach to characterize web site workloads by measuring the distribution of request type, response type and network traffic (e.g., request rise and response time).

Draheim et al. (2006) propose an approach to workload characterization by creating Stochastic form-oriented models of user behaviour from dynamic information (e.g., execution logs).

Goseva-Popstojanova et al. (2006) propose an approach to characterize a user's behaviour when interacting with web sites. Their approach characterizes a user's behaviour using high-level measures (e.g., session length, number of requests per session and number of sessions per day/hour).

Hall (2008) discusses the challenges of constructing workloads out of smaller scenarios (e.g., scenarios that simulate the behaviour of a single user). He also proposes an approach to large-scale testing based on specifying large-scale scenarios (e.g., scenarios that simulate the behaviour of groups of users).

Hassan et al. (2008) propose an approach to identify repeated event sequences in the execution logs using log compression. Their approach 1) breaks the execution logs into equal-sized periods, 2) compresses each period and 3) measures the compression ratio of the period. Periods with a high compression ratio tend to contain a large number of repeated event sequences.

Jia et al. (2014) propose an approach to 1) identify the most important performance counters for characterizing big data workloads and 2) identify redundant workloads using principle component analysis (PCA).

Krishnamurthy et al. (2006) propose a tool that automatically creates a benchmark workload against e-commerce systems by mining user behaviour from execution logs.

Kurz et al. (2005) propose an extension to the TPC-W (an e-commerce benchmarking tool) to re-create the benchmark workload based on the field workload.

Luo et al. (2009) propose an approach to select a representative sample of use cases (i.e., sequences of URLs) from the field the 1) execute every feature at least once and 2) execute every defect-prone path at least once. Their approach 1) characterizes use cases from the field, 2) clusters the use cases based on the URLs in the sequence and 3) selects one use case from each cluster.

Menascé et al. (1999) propose an approach to workload characterization by profiling the system and creating a Customer Behavior Model Graph (CBMG) model (a state-based model) of the system from the execution logs.

Menascé (2003) propose an approach to workload characterization by examining the workload from four different perspectives: 1) the business perspective, 2) the session perspective, 3) the function perspective and 4) the request perspective. Performance analysts should examine the workload from each of these perspectives.

Nagappan et al. (2009) propose a scalable approach to identify execution sequences. Their approach 1) constructs a suffix array of all possible execution sequences and 2) determines the frequency of each unique sequence.

Tian et al. (2004) propose new measures for characterizing a web site's workload based on information contained in the execution logs. These counters are then used to evaluate a site's reliability.

Zhang and Cheung (2002) propose an approach to workload characterization by profiling the system and creating a Petri net of the system from the data collected. These models are then used to generate a workload for performance testing.

B.2 Performance Comparison

Comparing a system's performance to its historical performance is crucial to understanding whether a system's performance is changing and, if so, how the system's performance is changing. Execution logs and performance counters are the most common data source used to compare a systems performance. However, execution logs are not typically designed for automated analysis and must be abstracted to execution events to enable such analysis. Therefore, we present relevant literature on 1) execution logs, 2) execution log abstraction, 3) execution log analysis and 4) performance counter analysis.

B.2.1 Execution Logs

Cinque et al. (2013) performed an empirical study on the limitations of current logging practices and proposed a rule-based approach to make execution logs more effective in analyzing failures. Their approach automatically adds logging to a system's source code to help log failures with little performance overhead.

Fu et al. (2014) performed an empirical study of logging practices and found that logs are used to record 1) important execution points and 2) unexpected situations. The authors also build a classifier to help developers determine where to insert logging statements.

Shang et al. (2014) performed an empirical study of logging practices in open-source software systems and found that 1) operators often ask about the meaning, cause, context, solution and impact of execution logs and 2) development knowledge can be used to address many of the questions asked by operators.

Yuan et al. (2012) performed an empirical study of logging practices in open-source software systems and found that logging is 1) pervasive, 2) helpful in diagnosing defects and 3) improved over time. The authors also build a classifier to help developers determine the proper logging level (i.e., verbosity).

B.2.2 Execution Log Abstraction

Many of the approaches to log abstraction are presented alongside approaches to execution log analysis. Therefore, some of the papers discussed in this section also appear in Section B.2.3.

Jiang et al. (2008a,b) propose an approach to abstract execution logs to execution events using 1) a small set of pre-specified rules, 2) order-insensitive token-based clustering to identify event categories (i.e., groups of similar events) and 3) order-sensitive token-based clustering to identify event types (i.e., specific events). Their approach outperforms the state-of-the-practice with high precision (90%) and recall (98.4%).

Salfner and Tschirpke (2008) propose an approach to preprocess execution logs such that they can be used to build a hidden semi-Markov model. Their approach 1) assigns an error ID to each error message and 2) clusters the error messages into sequences with high accuracy (0.66 F-measure).

Vaarandi (2003) propose Simple Logfile Clustering Tool (SLCT), a tool to abstract execution logs to execution events using frequent itemset mining. Their tool 1) tokenizes each log line, 2) summarizes the positional frequency of each token (i.e., the frequency of finding a particular token at a particular position), 3) clusters tokens that occur together and 4) identifies clusters that occur more frequently than a pre-defined threshold.

Nagappan and Vouk (2010) improve the SLCT tool by using the relative positional frequency, rather than the absolute positional frequency, to cluster tokens that occur together. Their improvement improves SLCT's ability to abstract log lines that 1) do not occur frequently or 1) frequently occur with the same dynamic information.

Xu et al. (2009a,b, 2010) propose an approach to abstract execution logs to execution events by parsing the source code. Their approach 1) identifies logging statements in the source code, 2) identifies the static (i.e., constant strings) and dynamic (i.e., variables) components of each logging statement and 3) creates a message template of each logging statement. Execution logs are then abstracted by applying the message template to recognize and remove dynamic information from the execution logs.

B.2.3 Execution Log Analysis

Jiang et al. (2005, 2007) mine execution logs from a baseline to create a finite-state model of the system. Their approach uses this model to flag anomalies in new tests.

Cotroneo et al. (2007) mine execution logs from a baseline to create a finite-state model of the system. Their approach uses this model to compare failing jobs against successful ones.

Fu et al. (2009) mine execution logs from a baseline to create a finite-state model of the system. Their approach uses this model to flag anomalies in new tests.

Fu et al. (2013) propose an approach to help performance analysts understand a system's runtime behaviour by mining execution logs. Their approach 1) performs a formal concept analysis (FCA) to mine execution sequences and 2) uses decision trees to model the branch conditions that cause one sequence to follow another.

Hellerstein et al. (2002) propose an approach to identify three common properties of event sequences (i.e., sequence bursts, periodic sequences and dependent sequences) by mining execution logs.

Jiang et al. (2008c) mine execution logs to determine the dominant (expected) behaviour of the system and flag anomalies from the dominant behaviour. Their approach is able to flag $<$0.01% of the execution log lines for closer analysis by system experts.

Jiang et al. (2009) flag performance issues in specific usage scenarios by comparing the distribution of response times for the scenario against a baseline derived from previous tests. Their approach reports scenarios that have performance problems with high precision (77% precision).

Larsson and Hamou-Lhadj (2013) mine frequent itemsets from the execution logs of a baseline to determine the expected behaviour of the system. Performance analysts manually prune the itemsets using domain knowledge. The remaining itemsets are then used to flag anomalies in the field.

Lou et al. (2010) mines execution logs to learn a set of linear invariants that describe the system's behaviour. Their approach uses these invariants to flag anomalies in new tests.

Shang et al. (2013) flag deviations in execution sequences mined from the execution logs of a test deployment and a field deployment of a large-scale system. Their approach reports deviations with a comparable precision to traditional keyword search approaches (23% precision), but reduces the number of false positives by 94%.

Tan et al. (2008) propose SALSA, a tool to flag performance issues in specific usage scenarios of distributed systems by comparing the distribution of event durations for the scenario against a baseline derived from other nodes. Their approach reports scenarios that have performance problems with high true positives (70%-80%) and low false positives (5%-10%).

Xu et al. (2009a,b, 2010) flag deviations in execution sequences mined from the execution logs collected from the field. Their approach 1) parses the source code to identify log abstraction rules, 2) abstracts the execution logs to execution events, 3) groups sequences of events using event IDs (e.g., request ID or session ID) and 4) uses principle component analysis (PCA) to identify anomalous sequences. The results are presented to performance analysts using decision trees. Their approach has been extensively evaluated on several execution logs collected from several

different large-scale systems.

B.2.4 Performance Counter Analysis

Breitgand et al. (2009) propose an approach to detect performance regressions by detecting when the systems performance counters exceed a threshold that is dynamically calculated based on 1) the current workload and 2) historical performance data.

Cherkasova et al. (2008, 2009); Mi et al. (2008) propose an approach to detect performance regressions by 1) modelling the system's CPU usage based on its (transaction based) workload and 2) creating a performance signature that represents the system's performance. Their approach detects anomalies in the system's CPU usage and flags the transactions that are most likely to be responsible for the performance regression.

Duttagupta and Nambiar (2011) propose an approach to predict a system's expected performance in the field by by extrapolating the results of a "small" performance test. Their predicted performance may be used to detect performance or scalability issues in the field.

Foo et al. (2010) use association rule mining to extract correlations between the performance counters collected during a performance test. Their approach compares the correlations from one performance test to a baseline to identify performance deviations with high precision (89%) and moderate recall (62%).

Huebner et al. (2001) propose using a "no-worse-than-the-last-release" threshold to determine whether a system passes performance testing.

Malik et al. (2010) have used principle component analysis (PCA) to generate performance signatures for each component of a system. The authors assess the pair-wise correlations between the performance signatures of a performance test and a baseline to identify performance deviations with high accuracy (79% accuracy).

Malik et al. (2013) use one supervised approach and three unsupervised approaches to 1) identify the most important performance counters and 2) identify the performance deviations. Their approach reduces the number of performance counters by 89% and identifies performance deviations with high precision (95%).

Nguyen et al. (2011) use control charts to identify performance tests with performance regressions and detect which component is the cause of the regression. Their approach 1) automatically determines if a test passes or fails and 2) identifies the subsystem where performance regression originated.

Nguyen et al. (2012) use control charts to identify performance tests with performance regressions. Their approach automatically determines if a test passes or fails with high precision (75%) and recall (100%).

Nguyen et al. (2014) use machine learning to diagnose performance regressions by mining repositories of baseline tests. Their approach is able to diagnose performance regressions with high accuracy (80%).

Sandeep et al. (2008) propose CLUEBOX, a tool to 1) identify the most important performance counters, 2) classify workloads based on when they were last observed and 3) identify performance regressions. Their approach uses 1) principle feature analysis (PFA) to reduce the number of performance counters, 2) random forest (RF) machine learning to identify similar workloads and 3) ranking based on

a performance counter's percentage difference from a baseline to identify performance regressions.

Syer et al. (2011b) propose an approach to identify performance deviations in thread pools by hierarchically clustering covariance matrices of the performance counters. Their approach is able to identify performance deviations (e.g., memory leaks) with high precision (100%) and recall (77%).

Zhang et al. (2005) propose an approach to identify violations of the system's service level agreements (SLA) under changing workloads using ensemble models. Their approach is able to detect violations of the system's SLAs with high accuracy (95%).