Log Engineering: Towards Systematic Log Mining to Support the Development of Ultra-large Scale Software Systems

by

Weiyi Shang

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
May 2014

Abstract

Much of the research in software engineering focuses on understanding the dynamic nature of software systems. Such research typically uses automated instrumentation or profiling techniques on the code. In this thesis, we examine logs as another source of dynamic information. Such information is generated from statements inserted into the code during development to draw the attention of system operators and developers to important run-time events. Such statements reflect the rich experience of system experts. The rich content of logs has led to a new market for log management applications that assist in storing, querying and analyzing logs. Moreover, recent research has demonstrated the importance of logs in understanding and improving software systems. However, developers often treat logs as textual data. We believe that logs have much more potential in assisting developers. Therefore, in this thesis, we propose Log Engineering to systematically leverage logs in order to support the development of ultra-large scale systems.

To motivate this thesis, we first conduct a literature review on the state-of-the-art of software log mining. We find that logging statements and logs from the development environment are rarely leveraged by prior research. Further, current practices of software log mining tend to be ad hoc and do not scale well.

To better understand the current practice of leveraging logs, we study the challenge of

understanding logs and study the evolution of logs. We find that knowledge derived from development repositories, such as issue reports, can assist in understanding logs. We also find that logs co-evolve with the code, and that changes to logs are often made without considering the needs of *Log Processing Apps* that surround the software system. These findings highlight the need for better documentation and tracking approaches for logs.

We then propose log mining approaches to assist the development of systems. We first find that logging characteristics provide strong indicators of defect-prone source code files. Hence, code quality improvement efforts should focus on the code with large amounts of logging statements or their churn. Finally, we present a log mining approach to assist in verifying the deployment of Big Data Analytics applications.

Declaration

Author's Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. 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.

Acknowledgments

I would like to thank my supervisor Dr. Ahmed E. Hassan for all his guidance and support throughout this journey. Ahmed, I could not have imagined having a better advisor and mentor for my Ph.D. study. For me, you have been a tremendous advisor, mentor and a great friend. I would like to thank you for encouraging my research and for allowing me to grow as a researcher. Your advice on both research as well as on my career have been priceless.

A special thank you to my supervisory and examination committee members, Dr. Patrick Martin, Dr. Mohammad Zulkernine, Dr. Thomas R. Dean, and Dr. Mark Harman, for their continued critique, support and guidance. Many thanks to my examiners for their fruitful feedback on my work.

I am very lucky to work and collaborate with some of the brightest researchers during my Ph.D. I would like to thank all of my lab mates and collaborators, Tse-Hsun Chen, Thanh Nguyen, Mark Syer, Nicolas Bettenburg, Shane McIntosh, Dr. Zhen Mining Jiang, Dr. Meiyappan Nagappan, Dr. Emad Shihab, Dr. Yasutaka Kamei, Dr. Haroon Malik, Dr. Stephen Thomas, Dr. Hadi Hemmati, Dr. Bram Adams, and Dr. Michael Godfrey for the many fruitful discussions and collaborations. I learned so much from you all.

I would like to thank BlackBerry and the members of the BlackBerry Performance Engineering Team. I could not evaluate the impact and practical value of my thesis without

the industrial environment and thoughtful feedback generally provided by the BlackBerry team.

Through this entire journey I received so much love, guidance and support from many dear friends. I would like to thank them all for making my Ph.D. journey an enjoyable one.

A special thanks to my family. Words cannot express how grateful I am for all of the sacrifices that you have made on my behalf. Your support for me was what sustained me thus far. At the end I would like express appreciation to my beloved wife Bingyang who spent sleepless nights besides me and is always my support. I dedicate this thesis to my family.

Dedication

To my family.

Related Publications

The following is a list of our publications that are on the topic of this thesis:

- **Weiyi Shang**, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. 2011. An Exploratory Study of the Evolution of Communicated Information about the Execution of Large Software Systems. In Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE 2011). IEEE Computer Society, Washington, DC, USA, 335-344. This work is described in Chapter 4.

  - **This paper received the Best Paper Award for WCRE 2011.**

- **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. Journal of Software: Evolution and Process (JSEP). Volume 26, Issue 1, pages 326, January 2014. This work is described in Chapter 4.

- **Weiyi Shang**. Bridging the divide between software developers and operators using logs. In Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012). IEEE Press, Piscataway, NJ, USA, 1583-1586. This work is described in Chapter 1.

- **Weiyi Shang**, Meiyappan Nagappan, Ahmed E. Hassan. Studying the Relationship between Logging Characteristics and the Code Quality of Platform Software. Empirical Software Engineering: An International Journal (EMSE). In Press. 24 pages. This work is described in Chapter 5.

- **Weiyi Shang**, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. 2013. Assisting Developers of Big Data Analytics Applications when Deploying on Hadoop Clouds. In Proceedings of the 2013 International Conference on Software Engineering (ICSE 2013). IEEE Press, Piscataway, NJ, USA, 402-411. This work is described in Chapter 6.

  - **This paper received the ACM SIGSOFT Distinguished Paper Award for ICSE 2013.**

- **Weiyi Shang**, Meiyappan Nagappan, Ahmed E. Hassan and Zhen Ming Jiang. Understanding Log Lines Using Development Knowledge. Submitted to the 30th International Conference on Software Maintenance and Evolution (ICSME 2014). This work is described in Chapter 3.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Automated software instrumentation techniques are commonly used to study the run-time behaviour of software system [CZD+09]. However, such techniques often impose high overhead, especially for real-world production workloads. To make matters worse, software profiling and instrumentation are commonly performed by non-system experts after the system has been built, based on limited domain and system knowledge. Therefore, extensive instrumentation often leads to an enormous volume of data that is impractical to meaningfully interpret.

In practice, system operators and developers typically rely on the software system's logs to understand the high-level field behaviour of large systems and to diagnose and repair bugs. Such logs consist of the major system activities (e.g., events) and their associated contexts (e.g., a time stamp).

Rather than instrumenting software system in a blind manner, developers typically communicate some information that is considered to be particularly important through logs. The rich yet unstructured nature of logs has created a new market for log management applications (e.g., *Splunk* [BGSZ10], *XpoLog* [xpo] and *Logstash* [logb]) that assist in storing,

querying and analyzing logs. We collectively call these applications, *Log Processing Apps*). Moreover, recent software engineering research has demonstrated the importance of logs in understanding and improving software systems. For example, software operators leverage the rich information in logs to generate workload information for capacity planning of large scale systems [HMF+08; NWV09], to monitor system health [BGSZ10], to detect abnormal system behaviours [JHHF08b], or to flag performance degradations [JHHF09].

Logs are not only used for the convenience of developers and operators, but they are often needed to comply with legal regulations. For example, the Sarbanes-Oxley Act of 2002 [soa] stipulates that the execution of telecommunication and financial applications must be logged.

Although logs are widely used in practice, and their importance has been well-identified in prior research [GWS06; YPZ12], logs have not yet been fully leveraged by practitioners.

All too often, practitioners treat logs as textual data. A typical *Log Processing App* searches through logs using keywords (e.g., "error" and "fail") in order to identify system run-time anomalies. We, on the other hand, believe that logs have much more potential in assisting practitioners. Therefore, we propose *Log Engineering*:

Leveraging logs through systematic and scalable approaches in order to support the development of software systems.

In this thesis, we first present a literature review on the state-of-the-art of software log mining. We then study the challenges associated with understanding and evolving logging statements. Finally, we propose systematic log engineering approaches to assist in prioritizing code review efforts, and in deploying Big Data Analytic applications.

This chapter consists of the following parts: Section 1.1 presents our research hypothesis. Section 1.2 gives an overview of the thesis. Section 1.3 briefly discusses the contributions of this thesis. Section 1.4 presents the organization of the thesis.

1.1 Research Hypothesis

Prior research and our industrial experience lead to the following research hypothesis.

> *Logs are a valuable yet rarely explored source of knowledge about a software system and its operation. There is little research regarding the understanding and evolution of logs*
> *Systematic and scalable log mining approaches are needed to support various software development activities (e.g., code quality improvement, large scale testing and deployment of ultra-large scale applications).*

The goal of this thesis is to empirically demonstrate the importance of logs and to propose approaches to better leverage logs for supporting software development activities. The thesis is divided into two parts along this goal, the first part studies the current practices of leveraging logs through: 1) an empirical study of the challenges in understanding logging statements and 2) an empirical study of the evolution of logs; the second part proposes two approaches that leverage logs to support : 1) code quality improvement efforts and 2) large scale testing and deployment of Big Data Analytic application.

1.2 Thesis Overview

We now give a brief overview of the work presented in this thesis.

1.2.1 Chapter 2: Literature review

Program Analysis is the set of analysis techniques that either analyze software systems without executing them (static analysis) [NNH99] or analyze their run-time data (dynamic analysis) [Bel99]. Static analysis has the ability to automatically highlight possible errors and vulnerabilities in code [AHM+08]. Dynamic analysis has the potential to provide an accurate picture of a software system because it exposes the actual behaviour of a system [CZD+09]. Various types of run-time data, such as logs and execution traces, may be collected during the execution of software systems and analyzed using various techniques.

Software Log Mining (SLM) focuses on mining the code snippets (i.e., logging statements) that generate logs for static analysis and mining the generated logs as the source of run-time data for dynamic analysis. This review chapter will focus on the topic of SLM. We characterize and compare the surveyed literature along the following dimensions: 1) log collection, 2) log transformation techniques, 3) log analysis techniques and 4) log mining research goals. From the literature review, we find that much of the software log mining literature leverages logs using ad hoc approaches and most of the software log mining research focuses on assisting system operators, however, very little work focuses on assisting software development activities.

1.2.2 Chapter 3: What are the challenges in understanding logging statements?

In Chapter 3, we propose using knowledge derived from development repositories such as code commits and issue reports, to assist in understanding log lines. We conduct a case study on three open source systems (*Hadoop*, *Cassandra* and *Zookeeper*). Reading through the mailing lists of the subject systems, we identify five types of information about log lines that are often sought by practitioners. Based on examining 300 randomly sampled logging statements from the source code of the subject systems, we find that four of the five

types of information are typically missing. However, development knowledge, especially issue reports, contains such missing information. We also find that development knowledge can be used to resolve 24 out of 45 real-world inquiries about logs. Based on our study, we propose an initial approach to automatically extract the required information, from development repositories.

1.2.3 Chapter 4: How do logging statements evolve?

In Chapter 4, we perform a case study on two large open source systems and one industrial software system. We explore the evolution of logs by mining the execution logs and the logging statements in the code from these systems. Our study illustrates the need for better traceability between logs and the *Log Processing Apps* that analyze the logs. In particular, we find that the logging statements change at a high rate across versions, which could lead to fragile *Log Processing Apps*. We found that up to 70% of these changes can be avoided and the impact of 15% to 80% of these changes can be controlled through the use of robust analysis techniques by *Log Processing Apps*. We also found that *Log Processing Apps* that track implementation-level logging statement (e.g., performance analysis) and the *Log Processing Apps* that monitor error message logging statements (e.g., system health monitoring) are more fragile than *Log Processing Apps* that track domain-level logging statements (e.g., workload modeling), since the latter logging statements tend to be more stable and long-lived.

1.2.4 Chapter 5: Prioritizing code review and testing efforts using logs and their churn

In Chapter 5, we propose systematic log mining techniques to assist developers in prioritizing code review and testing efforts. We study the relationship between the characteristics of logs, such as log density (i.e., the number of logging statements per lines of code) and

log churn (i.e., the number of changes to logging statements), and code quality, especially for large platform software. We perform a case study on four releases of *Hadoop* and *JBoss*. Our findings show that files with logging statements tend to have higher post-release defect densities than those without logging statements. Inspired by prior studies on code quality, we defined log-related product metrics, such as the number of logging statements in a file, and log-related process metrics such as the number of changed logging statements. We find that the correlations between our log-related metrics and post-release defects are as strong as their correlations with traditional process metrics, such as the number of pre-release defects, which is known to be one the metrics with the strongest correlation with post-release defects. We also find that log-related metrics can complement traditional product and process metrics resulting in up to 40% improvement in explanatory power when modeling defect proneness. Our results show that logging characteristics provide strong indicators of defect-prone source code files. However, we note that removing logging statements is not the answer to better code quality. Instead, our results show that developers often relay their concerns about a piece of code through logging statements. Hence, code quality improvement efforts (e.g., testing and inspection) should focus more on the source code files with large number of logging statements or with large amounts of log churn.

1.2.5 Chapter 6: Verifying the deployment of Big Data Analytic applications using logs

As a first step in assisting developers of Big Data Analytic (BDA) Apps for cloud deployments, we propose a lightweight approach for uncovering differences between pseudo and large scale cloud deployments. Our approach makes use of the readily-available yet rarely leveraged execution logs from the underlying platform of BDA Apps. Our approach abstracts the execution logs, recovers the execution sequences, and compares the sequences between the pseudo and cloud deployments. Through a case study on three representative

Hadoop-based BDA Apps, we show that our approach can rapidly direct the attention of BDA App developers to the major differences between the two deployments. Knowledge of such differences is essential in verifying that BDA Apps perform well in the cloud. Using injected deployment faults, we show that our approach significantly reduces the deployment verification effort ands provides very few false positives when identifying deployment issues.

1.3 Thesis Contributions

In this thesis, we demonstrate that logs are a valuable yet rarely explored source of knowledge about the development of software systems. We study the current practices of leveraging logs and propose approaches for leveraging logs in a systematic fashion to support the development of software systems.

In particular, our contributions are as follows:

1. We demonstrate that many challenges of understanding log lines can be supported through mining development repositories.

2. We show that additional maintenance resources should be allocated to maintain *Log Processing Apps*, especially when major changes are introduced into the software systems. Logging statements continually evolve, therefore traceability techniques are needed to establish and track the dependencies between logs and the *Log Processing Apps*.

3. We show that there is a relationship between logging characteristics and software defects.

4. We propose approaches that leverage logs to verify the deployment of Big Data Analytic applications.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents a literature review of the state-of-the-art research on software log mining. Motivated by this literature review, Chapters 3 to 6 study the current practices for leveraging logs to support the development of software systems. In particular, we study and leverage logs in two parts:

- **Part 1:** Studying the challenges associated with understanding and evolving logging statements
  - *Chapter 3* What are the challenges in understanding logging statements?
  - *Chapter 4* How do logging statements evolve?
- **Part 2:** Log engineering approaches to support software development activities
  - *Chapter 5* Prioritizing code review and testing efforts using logs and their churn.
  - *Chapter 6* Verifying the deployment of Big Data Analytic applications using logs.

Part I

Literature Review

CHAPTER 2

Literature Review of Software Log Mining Research

Software Log Mining (SLM) focuses on mining logging statements for static analysis and mining logs as the source of run-time data for dynamic analysis. There exists prior research that leverages logs to support system operation and software development activities. In this chapter, we conduct a literature review on software log mining research. We survey and compare the literature along the following dimensions: 1) log collection, 2) log transformation techniques, 3) log analysis techniques and 4) log mining research goals. From the literature review, we find that much of the software log mining literature leverages logs using ad hoc approaches and focuses on assisting system operators, however, very little work focuses on assisting in software development activities.

2.1 Introduction

Program Analysis is the set of analysis techniques that analyze software systems either without executing them (static analysis) [NNH99] or by analyzing their run-time data (dynamic analysis) [Bel99]. Static analysis has the ability to automatically highlight possible errors and vulnerabilities in source code [AHM+08]. Dynamic analysis has the potential to provide an accurate picture of a software system because it exposes the system's actual behaviour of a system [CZD+09]. Various types of run-time data, such as logs and execution traces, are collected during the execution of software systems and analyzed using various techniques.

Software Log Mining (SLM) focuses on either mining the logging statements that generate logs or the log lines themselves generated at run-time. For example, a logging statement *LOG.info("Reporting fetch failure for " + mapId + " to jobtracker.");* would generate a log line *Reporting fetch failure for attempt_200910281903_0028_m_001076_0 to jobtracker*. Logs consist of a large number of log lines generated during the execution of a software system. Our research and this literature review will focus on the topic of SLM.

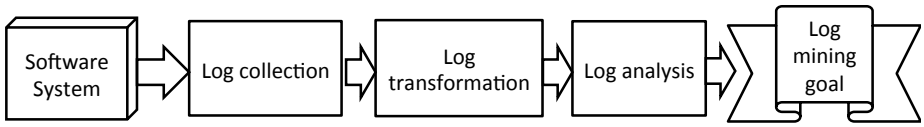


Figure 2.1: Overview of Software Log Mining

More precisely, SLM can be defined as the mining of logging statements that are embedded in the source code and the mining of data generated from such statement during run-time. Large numbers of logging statements and logs are readily available from both system-level testing and production settings. SLM helps developers in identifying bugs and helps operators in ensuring the failure-free operation of the software.

It is important to note that, other sources of dynamic data are also commonly used to study the run-time behaviour of software systems. For example, a large amount of prior research leverages execution traces from automated profiling to verify the correctness of a software system [RS11], to understand its behaviours [KKS06] and to study its evolution [GDG06]. The main difference between logs and execution traces is that the execution traces are typically generated from extensive software profiling and automated instrumentation techniques, while software logs are generated from logging statements already embedded by domain experts (e.g., developers).

Figure 2.1 shows an overview of the SLM process. First, logs are collected from the source code or during the execution of a software system. For example, an operator of a cluster of UNIX nodes may collect logs from all the nodes in the cluster. Then, the collected logs are often transformed into certain data structures, such as vectors and graphs. The operator of the UNIX cluster may then group the logs into sets by machine name or process name. Finally, data analysis techniques are performed on the transformed logs to uncover valuable knowledge. In this step, the operator from the example may count the error-related keyword from each set of logs.

We describe the three steps of SLM in the following paragraphs using more concrete

examples from prior research. For example, Jiang *et al.* [JHHF08b] perform SLM to detect system anomalies. They first collect application logs during load tests. They transformed the collected logs into abstracted event pairs. For example, if log lines about "user purchase" are always before log lines about "updating shopping cart", the transformation technique would generate a event pair "user purchase ->updating shopping cart". In the analysis step, they leverage a statistical approach named *K-stat* to identify system anomalies.

In this chapter, we perform a literature review to study the state-of-the-art of software log mining. We select research papers that are related to the topic of software log mining (see Appendix A.1). We then classify and compare the selected papers along the following dimensions:

- **Log collection**: we present the types and sources of the logs collected in prior SLM research.
- **Log transformation techniques**: we present the techniques used in log transformation and the types of output data.
- **Log analysis techniques**: we present the techniques used to analyze the transformed log data.
- **Log mining goals**: we present the different goals of prior SLM research.

The rest of this chapter is organized as follows: Sections 2.2 to 2.5 present the four dimensions of SLM research. For each dimension, we present our taxonomy and findings. Section 2.6 concludes the chapter.

2.2 Log Collection

Prior SLM work leverages various types of logs from software systems. We categorize log collection along two dimensions: types and sources. Types describe the types of software

system that generates the logs. Sources describe the executing environment where the logs are generated. Although logs are inserted in the code to achieve different purposes, such as debugging and operation, prior research typically does not differentiate between the purposes of logs.

Types of logs:

There are two main types of logs collected in prior studies.

- **Platform logs:** A platform typically acts as a container of applications running on top of it. Platform logs can be leveraged for all the different applications running on top of the platform. *Hadoop* is an example of such software platforms, which support applications running in a distributed environment [Whi09]. The large number of logs generated by such platforms typically do not contain information about the applications running on the platform, but rather platform itself. For example, Tan *et al.* [TPK+08; TPK+09; TKGN10; BKT+10] leverage platform logs from *Hadoop* to provide general visualization and anomaly detection for the applications running on top of the *Hadoop* platform. Other examples of platforms include operating systems (e.g., *UNIX*) and database systems (e.g., *PostgreSQL*).

- **Application logs:** Large software systems, e.g., telecommunication systems, also generate logs at run-time. Several open source applications that generate logs at run-time have been widely used in previous work, such as *RUBBoS* [rub] and *Dell DVD Store* [ds2]. For example, Jiang *et al.* [JHHF08b; JHHF09] leverage logs collected from the *Dell DVD Store* application during load testing in order to identify functional and performance problems.

Sources of logs:

There are two sources of logs studied in prior research.

- **Logs from the field:** Researcher have studied logs collected from field deployments.

For example, Kavulia *et al.* [KTGN10] collected logs from the execution of a *Hadoop* cluster, which had been running in the field for 10 months.

- **Logs during development:** Researchers have also studied logs collected from testing environments. Such logs may be generated during debugging or large system testing. For example, Jiang *et al.* [JHHF08b; JHHF09] collect logs during the execution of load tests.

> *Finding 1. Little research focuses on the logging statements that reside in the source code.*

We find limited prior research leveraging logging statements in the source code [YPZ12; YZP+11; YMX+10; XHF+09b; XHF+09a]. The majority of research collects only run-time logs, i.e., application logs and platform logs, without considering the valuable knowledge in the logging statements that reside in the source code.

> *Finding 2. Little research focuses on logs generated during the development of systems.*

We find that prior research typically collects logs after systems are deployed in the field. There are large numbers of logs generated while a system is being developed and tested. However, few researchers leverage such pre-deployment logs [JHHF08a; JHHF08b; JHHF09].

2.3 Log Transformation

Prior SLM work mainly transforms logs into three types of data structures:

- **Abstracted logs:** Techniques have been proposed to transform logs or logging statements into events. Logs typically do not follow strict formats, but instead contain significant unstructured data. For example, log lines may contain a time stamp and

a free form message, making it difficult to extract any structured information from them. In addition to being unstructured, log lines contain static and dynamic information. The static information is specific to each particular event while the dynamic information describes the context of each execution of a particular event. For example, in a log line "time=1, Trying to launch, TaskID=01A", the text "Trying to launch" is the static information that describes the event and the text "time=1" and "TaskID=01A" are dynamic information which describe the context of the execution. Jiang *et al.* [JHHF08a] propose an approach to identify the static and dynamic parts in logs by leveraging data mining and clone detection techniques. They group log lines by the number of tokens and identify the static parts by looking for matching tokens in the log line groups.

- **Vectors or sets:** Researchers also transform logs into vector-like or set-like data structures. Such vectors or sets represent a series of events, which bring more context information to every single log line. Domain and system knowledge, such as the meaning of particular parameters in logs, is often required to build such vectors or sets. Examples of the transformed vectors or sets include: pairs of logs [JHHF08b], sequences of logs [JHHF09], suffix arrays [NWV09] and time series [BGSZ10].

- **Graphs:** Logs can also be transformed into graphs, where typically abstracted log events are nodes in the graph and edges are added between two consecutive log events in a vector. Therefore, graph-based algorithms are leveraged to analyze logs. For example, Nagappan *et al.* [NR11] propose to transform logs into Directed Cyclic Graphs. They create a tool to transform the logs into graphs and apply existing graph-based algorithms to analyze the logs. Prior research has transformed logs into state-based graphs (like state machines [TPK+08]), to analyze the control and data flow in distributed systems.

- **Matrixes:** Researchers often extract data as a matrix from logs. Each log line will be

transformed into a vector, where each element in the vector is a corresponding value printed in the log line. The entire logs are transformed into a matrix [LFWL10]. Logs are also transformed into parallel coordinates matrix [Tri08] for better visualization.

Finding 3. Prior research primarily uses ad hoc log transformation techniques.

While logs may be transformed into four common data structures, there is no standardized data structure shared across different research efforts. For example, log abstraction is one of the most common practices in log transformation. However, prior research efforts tend to use different abstraction techniques, rather than reusing existing techniques. Standardized log abstraction techniques [JHHF08a; NV10], have seen limited adoption by other researchers.

2.4 Log Analysis

Prior research analyzes the transformed logs using a number of different techniques:

- **Simple calculation:** Simple calculations include solving vectors [LFY+10], calculating probability [JAS+10], calculating response time [JHHF09] and simple filtering [ST08]

- **Directed-graph based algorithms:** Directed-graph based algorithms are available to analyze all logs that are transformed into graphs [NR11]. For example, Nagappan [NR11] *et al.* identify the most visited nodes and edges in directed graphs generated from logs, to recover operational profiles of systems.

- **Static analysis:** Static analysis, including the recovery of call and data flow graphs, may help to understand and improve the logging statements [YZP+11].

- **Model checking:** Model checking techniques are often used to automatically check whether the logs fulfil a given specification [BBS+11]. Beschastnikh *et al.* [BBS+11]

propose an approach to infer model from logs. They use the model to identify anomalies in the system.

- **Visualizations:** Visualizations are used to assist developers and operators in manually examining the logs [DPH10; TPK+09].

- **Statistical methods:** Various statistical methods are widely used to uncover knowledge from logs. Examples of such statistical methods are principal component analysis [XHF+09b] and distribution estimation [XHF+09a] and k-stat [JHHF08b].

- **Data mining techniques:** Data mining techniques are often used to discover patterns in logs. For example, prior research performs invariance mining [BBS+11], co-occurrence analysis [LFWL10] and Bayesian decision theory [LFWL10] in order to recover system run-time models. Linear regression is used to recover the workload of a system [KTGN10].

- **Machine learning techniques:** Machine learning techniques, such as various classification [FLL+13] and prediction techniques [SSN+08], are used to uncover knowledge (like rules) from logs. Researchers then use such knowledge to identify problems [FFH08; ST08].

- **Other analysis techniques:** Various other techniques are used to analyze logs. For example, compression techniques [HMF+08] and computing longest common prefix [NWV09] are used on logs to recover the workload of a system.

> *Finding 4. Prior log mining research does not address the scalability challenges.*

We find that most of the prior research efforts suffer from poor scalability. Prior research leverages ever more complex analysis techniques on logs, such as model checking and data mining techniques. However, many of these techniques typically do not scale well to support large numbers of logs. Although there is some research that leverages existing distributed

computing platforms to address scalability issues [SAH10], most log mining research does not address the scalability challenge when being applied to logs.

2.5 Log Mining Research Goals

1. **Log mining platforms**: The rich yet unstructured nature of logs has created a new market for log mining platforms. Such platforms typically provide features to store large numbers of logs, and support querying and analyzing the logs. For example, Logstash [logb] is an open source log mining platform, which stores and parses logs for further search and analysis.

2. **Log improvement:** Prior research efforts aim to achieve better logging. For example, Yuan *et al.* [YZP+11] propose a tool named Log Enhancer, which automatically adds more context to log lines.

3. **Log mining for system operation**:

   - **Anomaly detection**: Detecting anomalies in ultra-large scale systems is challenging, especially when the system is deployed on hundreds or thousands of nodes. Logs are one of the only sources of data to detect system anomalies. However, not all log lines contain keywords such as "error" or "failure" to indicate an anomaly. More sophisticated techniques are designed to detect system anomalies by analyzing logs. For example, Xu *et al.* [XHF+09b; XHF+09a] distill the printed parameters and system state from the logs and extract a matrix with each vector indicating the values of the parameters and the system state. They perform Principal Component Analysis to identify the usual and unusual correlations among features to detect system anomalies. Their approach has been applied on Google's production logs.

- **System monitoring**: Logs are collected during system execution to monitor the run-time behaviour of a system. For example, Chukwa [RK10] leverage distributed data processing platforms such as *Hadoop* [Whi09] to collect logs in real-time from distributed nodes and to process the logs to monitor the health of the system.
- **Workload recovery and capacity planning**: Logs are often leveraged to recover the workload of a system. Such recovered workloads can assist in the efficient allocation of resources. For example, Kavulia *et al.* [KTGN10] leverage 10 months of logs in order to recover the workload of a *Hadoop* cluster in a cloud-computing setting. The recovered workload assists system administrators in learning which usage aspects impact the performance of their cluster and drive the cost of leasing the cloud-computing resources.

4. **Log mining for software engineering**:

- **Program comprehension**: Understanding the run-time behaviour of large scale systems is challenging. Logs are leveraged by prior research to understand the run-time behaviour of a system. For example, Beschastnikh *et al.* [BBE+11; BBS+11; BABE11] designed a tool named Synoptic, which infers concise and accurate system run-time models from logs.
- **Software testing**: Logs are one of the only sources of information about a system during large scale testing, such as performance testing and load testing. For example, Jiang *et al.* [JHHF09] leverage logs to identify performance degradations in large scale system load tests.
- **Empirical studies**: Researchers perform empirical studies on the characteristic and efficiency of logs. For example, through an empirical study, Yuan *et al.* [YPH+12] find that more than half of the failures could not easily be diagnosed using existing logs.

> *Finding 5. There exists limited SLM research to support software development activities.*

One of the key findings highlighted by this survey is that there exists little SLM research to support software development activities. We find that a majority of prior research focuses on assisting system operation; while logs are not fully leveraged for assisting in software development activities. Software developers often treat logs as a bi-product of the system instead of a valuable source of knowledge. Existing research leverages logs during the maintenance phase, such as program comprehension and regression testing. Logs are rarely used during the early stages of development, such as requirements engineering, design and coding.

2.6 Summary

In this chapter, we presented a literature review on the state-of-the-art software log mining research.

From our literature review, we find that logging statements and logs generated during development are rarely leveraged by prior research. Further, current practices of software log mining tend to be ad hoc and fail to scale. There is limited research on leveraging logs to support software development activities.

Motivated by the findings of our literature review, we first study the current practices of leveraging logs (Chapter 3 and 4). We then propose systematic approaches to leverage logs to support software development activities (Chapter 5 and 6).

Part II

Studying the Challenges Associated with Understanding and Evolving Logging Statements

In our literature review (Chapter 2), we found that there exists limited research that focuses on studying the logging statements that reside in the source code. Therefore, in this part of the thesis, we studying the challenges associated with understanding and evolving such logging statements.

- Chapter 3 studies the challenges associated with understanding log lines generated by logging statements in source code. We propose the use of knowledge derived from development repositories, like code commits and issue reports, to assist in understanding log lines. We identify five types of information about log lines that are often sought by practitioners from a case study on three open source systems (*Hadoop*, *Cassandra* and *Zookeeper*). We find that four of these identified types of information are typically missing. However, development knowledge, especially issue reports, contain such missing information. We also find that development knowledge can be used to resolve 24 out of 45 real-world inquiries about log lines. Based on our study, we propose an initial approach to automatically extract the required information from development repositories.

- Chapter 4 studies the challenges associated with evolving logging statements. A case study on two large open source and one industrial software system illustrates the need for better traceability between logging statements and the *Log Processing Apps* that analyze the logs generated from these statements. In particular, we find that the logging statements change at a high rate across versions, which could lead to fragile *Log Processing Apps*. We found that up to 70% of these changes could have been avoided and the impact of 15% to 80% of the changes can be controlled through the use of robust analysis techniques by *Log Processing Apps*. We also found that *Log Processing Apps* that track implementation-level logging statement (e.g., performance analysis) and the *Log Processing Apps* that monitor error message logging statements (e.g., system health monitoring) are more fragile than *Log Processing Apps* that track

domain-level logging statements (e.g., workload modeling), since the latter logging statements tend to be more stable and long-lived.

CHAPTER 3

What are the Challenges in Understanding Logging Statements?

Developers use logs to communicate important run-time information about software systems. The rich nature of logs has created a new market for log management applications (e.g., *Splunk*, *XpoLog* and *Logstash*) that assist in storing, querying and analyzing logs. Moreover, recent research has demonstrated the importance of logs in understanding and improving software systems. However, all too often practitioners (i.e., operators and developers) are often left without any support to understand the meaning and impact of specific log lines that are generated by logging statements in source code.

In this chapter, we propose using knowledge derived from development repositories such as code commits and issue reports, to assist in understanding log lines. We conduct a case study on three open source systems (*Hadoop*, *Cassandra* and *Zookeeper*). Examining the mailing lists of the subject systems, we identify five types of information about log lines that are often sought by practitioners. By examining 300 randomly sampled logging statements from the source code of the subject systems, we find that four of the five identified types of information are typically missing. However, development knowledge, especially issue reports, contain such missing information. We also find that development knowledge can be used to resolve 24 out of 45 real-world inquiries about log lines. Based on our study, we propose an initial approach to extract the required information automatically from development repositories.

3.1 Introduction

Logs are an important medium to communicate information about the operation of a software system. They report the major system activities (e.g., events) and their associated contexts (e.g., a time stamp) to assist developers and operators in understanding the high-level system behaviours. Moreover, logs capture developer expertise because they are inserted in specific code spots that are considered to be particularly important by developers or of

great interest by operators.

The rich yet unstructured nature of logs has created a new market for log management applications (e.g., *Splunk* [BGSZ10], *XpoLog* [xpo] and *Logstash* [logb]) that assist in storing, querying and analyzing logs. Moreover, recent research has demonstrated the importance of logs in understanding and improving software systems. As we discussed in Chapter 2, logs are used to support system operation and development. For example, software operators leverage the rich information in logs to generate workload information for capacity planning of large scale systems [HMF+08; NWV09], to monitor system health [BGSZ10], to detect abnormal system behaviours [JHHF08b], or to flag performance degradations [JHHF09].

However operators and even developers are often faced with many challenges when trying to understand the meaning of field logs or to answer questions about specific log lines [SJA+13; SJA+11; YZP+11]. In contrast to the extensive and rich research in program comprehension [vMV95], there exists no research in understanding and comprehending logs despite the importance of logs.

Various types of development knowledge, such as development history [HH04; vM03], design rationale and source code concerns [BMS03; RM02] and email discussions [BLR10; DR12] are widely used in program comprehension tasks (such as understanding the software architecture [HH04]). In this chapter, we propose to leverage such development knowledge to help understand log lines. We define the development knowledge of log lines as knowledge that is not directly presented in the log lines, but hidden in the development history of the code surrounding the logging statements that generated the log lines. To better understand the development knowledge and the usefulness of such knowledge in understanding logs, we perform a case study on three open source systems: *Hadoop*, *Cassandra* and *Zookeeper*. Information about these systems are shown in Table 3.1. A preliminary study confirms five types of information (meaning, cause, context, solution and impact) are often requested about log lines. Therefore, we study the log lines and their

Table 3.1: Overview of the subject systems

System	Description	K LOC	Length of history	# logging statements
Hadoop	Distributed platform	580	8 years	5,641
Cassandra	Distributed database	118	4 years	1,080
Zookeeper	Distributed coordination service	78	5 years	1,163

development knowledge along the following four research questions:

RQ1: What types of information are missing in logging statements?

We find that around 80% of the logging statements provide the meaning of the log lines, while the cause, context, solution and impact of the log lines are typically not provided by the logging statements.

RQ2: Can development knowledge provide information about logging statements?

We find that development knowledge can assist in providing more information about logging statements. In particular, issue reports provide the most missing information.

RQ3: Can development knowledge resolve real-world inquiries?

We find that development knowledge can resolve 24 out of 45 real-world inquiries. Using development knowledge to resolve real-world inquiries out-performs results provided by a web search and is comparable to human replies on the mailing lists.

RQ4: Can experts assist in resolving inquiries of logging statements?

We find that logging statement experts (i.e., developers who edit the source code surrounding a logging statement that generates a log line) are often the ones who answer inquiries about logging statements. The answers from experts are short, precise and affirmative. Development knowledge can assist in identifying logging statement experts to resolve logging statement inquiries.

Our results demonstrate the value of leveraging software development knowledge when

understanding log lines. However, collecting such development knowledge for log lines is time consuming. Therefore, we propose an approach to automatically link development knowledge with log lines, similar to the sticky note approach proposed by Hassan and Holt [HH04]. Our automated approach can successfully provide the development knowledge to answer real-world inquiries about 45 different log lines in the studied software systems.

The rest of this chapter is organized as follows: Section 3.2 presents the related work. Section 3.3 presents the results of our preliminary study. Sections 3.4 to 3.7 address the four research questions in our study. Section 3.8 proposes an approach to automatically collect and link development knowledge for log lines. Section 3.9 discusses the threats to validity of our study. Finally, Section 3.10 concludes the chapter.

3.2 Related Work

Documenting Code: There exists several prior studies that focus on providing documentation to assist in understanding of source code. Sridhara *et al.* [SPVS11] describe a technique that automatically generates comments for Java methods. Their technique focuses on describing the roles of parameters in a method. Padioleau *et al.* [PTZ09] performed a qualitative study on the source code comments of three large, open source, operating systems and found that code comments can complement software documentation and record the thoughts of developers during the development in an informal manner. Ibrahim *et al.* [IBAH12] studied the relationship between code comments and code quality. They find that a code change in which a function and its comment are co-updated inconsistently (i.e., they are not co-updated when they have been frequently co-updated in the past, or vice versa), is a risky change. Buse *et al.* [BW08; BW10] have a number of studies on automatically providing documentation. One of their studies automatically generates documentation for exceptions in Java programs [BW08]. This approach is based on an inter-procedural,

context-sensitive analysis, which statistically analyzes the possible causes of the exception. Instead of providing documentation for exceptions, our work exploits development knowledge, such as the code commits and issue reports, to assist in log line understanding.

Improving Log lines: There also exists prior studies that aim to automatically improve the information in log lines by providing the context and solution.

1. Providing context. Some logging libraries, such as Log4j [loga], can automatically output the name of the class, in which the log is generated. Such information is useful to understand the context of the logs. Yuan *et al.* [YZP+11] propose an approach to automatically enhance logging statements by printing the values of the accessible variables. This approach assists in adding more context to the log lines. Shang *et al.* [Sha12] design an approach to automatically provide context information for log lines to assist users in deploying applications in a cloud environment. Beschastnikh *et al.* [BBS+11] build system models from logs to infer the state of a system when an error occurs.

2. Providing solution. Ding *et al.* [DFL+12] design a framework to correlate logs, system issues and corresponding simple solutions. They store such information in a database to assist in providing solutions when similar logs appear. However, the solutions suggested are very general in nature – for example the solution of "rebooting an application" would be associated with several log lines without a clear rationale of the need for such a solution.

Instead of improving the documentation of code or improving the information in the logs lines, our work focuses on improving the documentation for the logging statements that are associated with log lines. Such log lines are widely used in practice, yet typically have poor documentation.

3.3 Preliminary Study

Understanding log lines is critical for practitioners. However, we first need to understand the type of information that is often sought about logs lines by the practitioners. Hence,

Table 3.2: Types of information that operators asked about log lines in the email inquiries.

System	# Inquiries	Meaning	Cause	Context	Solution	Impact	N/A
Hadoop	10	2	8	0	5	0	0
Cassandra	2	0	1	1	1	1	0
Zookeeper	3	0	2	0	0	0	1
Total	15	2	11	1	6	1	1

we first perform an exploratory study that examines several real-world inquiries about logs lines.

We choose three subject software systems that generate large amounts of logs during execution. Table 3.1 shows an overview of our subject systems. The systems are of different sizes, ages and application domains. However, they are all "systems software" (i.e., no user interfaces) – a choice that was made to ensure that these systems make heavy use of logging. We manually read through all the email threads in the user mailing lists of the subject systems – a process that took over 100 man-hours. Since all three subject systems are large-scale server systems, the users of the three subject systems are operators of the systems. We found inquiries for 15 different log lines. For example, in the *Cassandra* user mailing list, an inquiry was made when a particular logged event happens, whether a particular logged event affects other components and how to resolve the logged event.

From the 15 inquiries from the mailing list threads, we identify the different types of information that are sought about a log line by following the coding approach widely used by previous studies [Sea99]. We repeat the process until we cannot find any new types of inquired information. We assign the "N/A" tag to an inquiry if the users only include the log lines in the email without asking for any particular information. Our study finds that there are five types of information that are often sought about log lines. Table 3.2 gives an overview of our results. We describe each type of inquired information below.

1. **Meaning**: A description of the meaning of a log line is often sought. Although developers print textual information in the log lines to indicate meaning, we found that it might be challenging to understand the textual information. For example, an inquiry

for a log line in *Hadoop* asked, "What exactly does this message mean?"

2. **Cause**: A clarification of the cause of a log line is commonly sought. Two-thirds of the studied email inquiries asked about the cause of log lines. For example, an inquiry for *Zookeeper* asked, "Does anybody know why this happened?"

3. **Context**: The context of a log line, e.g. the time of the log line being printed, is sought in some cases. For example, an inquiry for *Cassandra* asked, "when does this occur?"

4. **Solution**: Often, the inquiries seek a solution for avoiding a particular log line when the log line indicates an error. Log lines typically do not contain this information. For example, a log line inquiry for *Hadoop* said "It will be great if some one can point to the direction how to solve this."

5. **Impact**: In some cases, the inquiry might seek the impact of a log line, e.g., whether this line implies performance degradations. For example an inquiry for *Cassandra* asked, "Is it affecting my data?"

We do note that a few industry guideline documents for creating log lines do mention some types of information that we discovered above. For example, the design documents from the SANS consensus project for information systems [san] suggests that log lines should include the subject and object of the event, time of the event, tools that performed the event and the error status of the event. However, other types of information, such as the solution and impact, are not suggested. Moreover, developers may forget to provide some information even though it is required [BvDDT07].

Additionally in our manual inspection, we noticed the following facts:

1. **Inquiries about log lines may not be answered or may take a long period of time to get resolved.**

   Although 12 of the 15 inquiries had replies, we found that 3 of the inquiries were not answered in the mailing list. The maximum time for the first reply is over 105 hours.

2. **Not all email replies are helpful.**

   Some email replies are informative, some replies are brief, some replies lack certainty and some replies are not useful.

3. **Manual analysis of development knowledge is used to answer questions in some cases.**

   In particular, to find out the cause of a *Hadoop* log line, the inquirer manually browsed the source code of *Hadoop* and found the method that generates the log line. He was not familiar with the Java code, but after he posted the code snippets online, an expert replied and resolved his issue.

4. **Log line inquiries are done through various online sources.**

   We were surprised with the small number of inquiries about log lines on the mailing lists of the studied open source projects. In prior industrial collaborations we noted that log lines played a key role in the interactions between customers and developers (e.g., [HMF+08]). Hence we randomly sampled 300 logging statements from the three subject systems and searched for the text in the logging statements using *Google*. If the text in the logging statement is ambiguous, we added the name of the subject system after the logging statement. For example, the text "Child Error" in a *Hadoop* log statement may be ambiguous. We then search for "Child Error Hadoop" as opposed to just "Child Error". We then examined the first 10 results from Google for each logging statement query.

We found that 32, 23 and 18 logging statements (in the set of 300) from *Hadoop*, *Zookeeper* and *Cassandra*, respectively, are discussed or inquired about through other mediums other than the project mailing list. Online issue reports (e.g., the Apache JIRA web

interface[1]) and *Stack Overflow*[2] are the two sources where the logs lines and logging statements are discussed or inquired about most often.

We also manually browsed the top 100 most frequently viewed questions on *Stack Overflow* for the tags "Hadoop" and "Cassandra" and we found 7 and 2 questions that inquire about log lines, respectively. For the tag "Zookeeper", there were only 15 questions in total with one being a log line inquiry. In short, practitioners often inquire about log lines through various mediums – potentially making these inquiries more difficult to archive and retrieve. We believe that development knowledge often contains the answers for such inquiries.

3.4 RQ1: What Types of Information are Missing in Log Lines?

Motivation

Through our preliminary study, we noted that there exists five types of information that are often sought for log lines. In this research question, we investigate whether developers include such information in the logs lines.

Approach

We select a random sample of 100 logging statements from all the logging statements throughout the lifetime of each of the subject systems. We manually read each logging statement and examine whether the text in the logging statement would provide the five types of information, i.e., meaning, cause, context, solution and impact. We manually examine all the code changes to these sampled logging statements throughout their lifetime, in order to determine whether such code changes led to changes in the types of information stored in the text of the logging statement.

[1] http://issues.apache.org/jira last verified January 2014.
[2] http://stackoverflow.com last verified January 2014.

Table 3.3: Number of logging statements (among 100 logging statements) that contain each type of inquired information.

	meaning	cause	context	solution	impact
Hadoop	79	2	29	1	9
Cassandra	79	2	24	0	10
Zookeeper	81	7	33	3	16

Table 3.4: Number and percentage of logging statement changes that change each type of information.

	meaning	cause	context	solution	impact
Hadoop	9 (23%)	0	9 (23%)	0	1 (3%)
Cassandra	5 (6%)	0	28 (33%)	0	1 (1%)
Zookeeper	10 (17%)	0	7 (12%)	0	3 (5%)

Results

What types of information do logging statements provide?

Around 80% of logging statements provide the meaning of the log lines, while the cause, context, solution and impact of the log lines are typically not provided by the logging statement. Table 3.3 shows the number of logging statements that contain each type of information. On average only 3.7%, 32%, 1.3% and 11.7% of the logging statements contain information about the cause, context, solution and impact of the log line, respectively. We find that most log inquiries are concerned with the cause and solution of the log line (shown in Table 3.2), therefore logging statements would barely provide answers to the most common inquiries.

Some logging statements do not provide any of the inquired information types. We find that 14%, 14% and 6% of the log lines from *Hadoop*, *Cassandra* and *Zookeeper*, respectively, do not contain any of the five types of information. Such logging statements may only print a word "error", or an exception stack trace, without providing any additional information.

What types of information in logging statements are changed?

We find that most changes to logging statements do not change information about cause, solution and impact of the logging statements. As shown in Table 3.4, the cause

and solution of the logging statements are not changed during the history of the logging statements and only 1% to 5% of the logging statement changes provide additional information about the impact of the logging statements. On average, 14% and 23% of the logging statement changes modify the meaning and context of the logging statements, which are the top two types of information that exist in logging statements.

> *Around 80% of the logging statements provide the meaning of the logging statements, while the cause, context, solution and impact of the logging statements are typically not included.*

3.5 RQ2: Can Development Knowledge Provide Information about Log Lines?

Motivation

From the results presented in Sections 3.3 and 3.4, we find that logging statements often miss information that are frequently sought after in real-world inquiries about log lines generated by such logging statements. Approaches are needed to assist in resolving the most common types of inquiries about log lines. In this research question, we explore whether development knowledge can assist in resolving such inquiries.

Approach

We identify five sources of development knowledge for each of the 300 logging statements analyzed in Section 3.4. We broadly categorize these five sources of development knowledge into two categories:

1. ***Snapshot knowledge:***

   Snapshot knowledge includes information from the most up-to-date snapshot of the source code associated with a logging statement that generates the log line.

(a) **Source code:** The source code of the method that contains the logging statement may provide information about the log line. For example, the *if* statement that triggers the logging statement can help in explaining the cause for the appearance of a log line.

(b) **Code comments:** Sometimes the source code is not self-explanatory. In these cases, the code comment may be used to explain the source code and the associated logging statements.

(c) **Call graph:** Often, the log lines only describe what happened instead of why the event happened or under what circumstances the event happened. When the reason or the context of a log line is sought out, the answer may be in the methods that trigger the method containing the logging statement. For example, a log line may say that there is an I/O issue and the call graph may describe that the issue happens while the copying of a file over the network.

2. ***Historical knowledge:***

Historical knowledge consists of the information that is generated during the development of logging statements that generates log lines.

(a) **Code commits:** A code commit contains the changes to the code and other corresponding information, such as the check-in comment describing the change and the developer who made the change. For example, the check-in comment for a change that adds or modifies a logging statement (or its surrounding or triggering code, calculated via the call graph) may provide information about the meaning of the log line. For example, the changes to the condition for outputting a log line may uncover the cause for outputting the log line.

(b) **Issue reports:** The source code changes are often due to issues (such as new feature requests and bugs) in the system. These issues are tracked in issue tracking

systems, such JIRA. The report for an issue consists of its description, its resolution and related developer discussions. An issue that is related to a logging statement may be helpful in explaining the rationale of the log line. For example, a logging statement in *Cassandra* is added to the source code for resolving issue "Cassandra-957".[3] The description of the issue report explains the meaning, the context and the cause of the log line.

We manually examine these five sources of development knowledge for each of the 300 randomly sampled logging statements from the subject systems. We examine whether their associated development knowledge can provide the 5 different types of information that are often sought after (i.e., meaning, cause, impact, context and solution (see Section 3.3)). We then check whether these five sources of development knowledge can provide the particular types of information that were missing in these logging statements.

[3]https://issues.apache.org/browse/CASSANDRA-957

Table 3.5: Percentage of the logging statements for which development knowledge can complement by providing the missing information (grouped by each type of information).

	meaning	cause	context	solution	impact
Hadoop	90% (19/21)	58% (57/98)	97% (69/71)	13% (13/99)	44% (40/91)
Cassandra	100% (21/21)	49% (48/98)	86% (65/76)	6% (6/100)	40% (36/90)
Zookeeper	89% (17/19)	51% (47/93)	79% (53/67)	18% (17/97)	39% (33/84)
Average	93% (57/61)	53% (152/289)	87% (187/214)	12% (36/296)	41% (109/265)

Table 3.6: Number of logging statements where each source of development knowledge can provide a particular type of missing information. The largest numbers in each type of missing information are shown in bold font. We study 100 logging statements for each subject system.

Hadoop					
	meaning	cause	context	solution	impact
source code	**83**	12	23	1	5
comment	55	11	37	1	15
call graph	2	25	66	0	3
commit	42	26	61	4	11
issue report	49	**37**	**69**	**12**	**27**
Cassandra					
	meaning	cause	context	solution	impact
source code	**59**	4	14	0	11
comment	47	19	26	1	**18**
call graph	0	11	39	0	1
commit	43	17	45	1	5
issue report	47	**24**	**51**	**5**	15
Zookeeper					
	meaning	cause	context	solution	impact
source code	**59**	9	12	0	13
comment	41	18	30	4	17
call graph	2	6	40	0	0
commit	37	22	45	3	5
issue report	51	**34**	**56**	**17**	**25**

Results

We find that development knowledge can complement logging statements by providing missing information. As shown in Table 3.5, development knowledge can provide most of the missing meaning and context (on average 93% and 87%, respectively), around half of the missing cause and impact (on average 53% and 41% respectively) and only an average of 12% of the missing solution. These percentages are calculated using the number of logging statements with that type of information missing (derived from Table 3.3). We think the reason for such results is that, the meaning and context of the logging statements are the intuitively easiest types of information to retrieve; the cause and impact of

the logging statements are more difficult; while solving a logged error is the most difficult.

In particular, we find that issue reports are the source of development knowledge that can resolve the most log line inquiries (see Table 3.6). Issue reports provide the four most-missing types of information about logging statements (cause, context, solution and impact). Although source code is the source of the most meaning of logging statements, our results from RQ1 (see Section 3.4) show that around 80% of the logging statements already contain the meaning already and Table 3.6 shows that issue reports are the second most valuable source of meaning of the logging statements.

Discussion

We discuss each type of information that is requested from development knowledge.

Meaning. The meaning of a log line is inquired about because the text in the log line is not descriptive enough. As we can see from Table 3.6, source code is the best source of the meaning of a log line. For example, a log line from *Hadoop* prints "-files", which does not have a clear meaning. From the source code, we can find out that the log line corresponds to temporary files defined by a user from command line. Issue reports also provide useful information about the meaning of log lines. For example, the *Hadoop* issue report "HADOOP-182"[4] is associated with the log line "log tracker". In the discussion for the issue report, the meaning of the log line was clearly presented as: "When a Task Tracker is lost (by not sending a heartbeat for 10 minutes), the JobTracker marks the tasks that were active on that node as failed". Therefore, we can say that the "lost" message in the log line means that a heartbeat from the tracker was not received.

Cause. 11 out of 15 inquiries in Section 3.3 were about the cause of a log line. Our results in Table 3.6 show that both sources of historical development knowledge, i.e. commit messages and issue reports, can help in explaining the reason of some log lines. Issue reports are the source of development knowledge of the cause of log lines in most cases. There are typically two scenarios when an issue report would provide such useful information:

[4]https://issues.apache.org/jira/browse/HADOOP-182

1. A logging statement is added to the source code as part of a feature or improvement. For example, to resolve *Hadoop* issue "Hadoop-1171"[5] (where the issue was to enable multiple retries of reading the data), a log line "fetch failure" was added.

2. A logging statement is explained in the discussion of the issue report when a bug in the source code is identified. For example, in the discussion about *Hadoop* issue "HADOOP-1093"[6], the reason for log line "NameSystem.completeFile: failed to complete " in *Hadoop* was explained as a race condition between a client and data server of *Hadoop*.

The commit message is another source of development knowledge of the cause of a log line. For example, a log line "DIR * FSDirectory.mkdirs: failed to create directory" was added in revision 412474 of *Hadoop* with a commit message "Fix DFS mkdirs() to not warn when directories already exist", which clearly indicates the cause.

Impact. Source code can provide the information about which components contain the event causing the log line. For example, from the source code of log line "initialization failed", we find that the logging statement is embedded in the constructor method of class "FSNamesystem". Therefore, we have the information that the initialization failure is in the name index component of the file-system. The source code can also provide information about other components that may be impacted by the event causing the log line. Although some logging libraries, such as Log4j, can provide the information about the component that generates the logs, information about other indirectly impacted components cannot be gathered using the logging library.

Context. Context is important in understanding log lines. Call graphs are one type of development knowledge that provides context for log lines. For example, the call graph of the log line "Column family ID mismatch" of *Cassandra* indicates that the log line is in the update period of the system. Most logging libraries provide a time stamp for each log line.

[5]https://issues.apache.org/jira/browse/HADOOP-1171
[6]https://issues.apache.org/jira/browse/HADOOP-1093

However, such time stamps are not as useful as domain-specific context. For example in MapReduce, the time stamp of a log line is not as useful as knowing if the log line is printed in the map period or reduce period.

Solution. If a log line indicates an error, one would want a solve to the error. However, since many complex reasons could potentially cause one error, log lines typically do not contain information on how to solve the error. It is difficult to provide a solution to a logged error. Prior research by Thakkar *et al.* [TJH+08] proposes a technique to find similar customer engagement reports to solve field errors. Similarly, issue reports can provide some solutions for logged errors. We find cases where developers discuss the log-related issues in the issue tracking system. For example, the log line "Severe unrecoverable error, exiting" from *Zookeeper* is related to issue "ZOOKEEPER-1277"[7], where a developer describes the way to resolve this issue by upgrading to a new version.

> *Sources of development knowledge can assist in providing more information about logging statements. In particular, issue reports are the richest source of information to answer inquiries into logging statements, relative to other sources of development knowledge.*

3.6 RQ3: Can Development Knowledge Resolve Real-world Inquiries?

Motivation

In the previous sections, we found that development knowledge can complement information currently available in logging statements. In this research question, we examine whether we can use development knowledge to answer real-world inquiries about log lines.

[7] https://issues.apache.org/jira/browse/ZOOKEEPER-1277

Approach

We examine the real-world inquiries that we identified in the user mailing lists of the subject systems and from the web searches in Section 3.3. For the 15 log lines inquired about in the user mailing list and 73 search results, we focus on 14 log lines from the user mailing list and 31 web search results where we can clearly identify the particular inquired types of information. Although some log lines are mentioned in the email, issue reports and other locations on the web, we cannot easily identify the inquired types of information.

For each of the 45 inquiries (i.e., mailing list question or web search results), we identify the types of inquired information because each inquiry may contain inquiries to multiple types of information. For example, an inquiry may ask about both the meaning and cause of the log line. We then examine whether the various sources of development knowledge can resolve the inquiries.

Results

Development knowledge can be used to resolve real-world inquiries. Development knowledge provides answers to 9 out of 14 inquiries from the user mailing list. Table 3.7 shows that development knowledge can resolve 14 out of 21 specific items of inquired information from the 14 real-world inquiries from the user mailing list. Issue reports are the richest source of development knowledge in resolving real-world inquiries. Issue reports can provide answers to all 9 mailing list inquiries that were resolved using development knowledge. In addition, 12 out of 14 resolved inquired specific items from the mailing list are resolved using issue reports. Development knowledge provides answers to 15 out of 31 inquiries from the web search results and resolves 16 out of 39 items of requested specific information. Each source of development knowledge performs similarly in resolving web search inquiries. Source code, code comment, call graph, commit and issue report can respectively resolve 4, 6, 3, 5 and 6 items of requested specific information from the web search results.

Table 3.7: Results of using development knowledge to resolve the 14 real-world mailing list inquiries. Each table cell indicates the source of development knowledge that resolves the inquiries. A table cell with "not answered" indicates that the inquiry is not answered by development knowledge. A blank cell indicates that the corresponding information was not inquired about in the mailing list. The first inquiry of Zookeeper did not request any specific information in the email, hence it is excluded from this table.

Hadoop	meaning	cause	context	solution	impact
1	call graph issue report				
2		commit issue report			
3		not answered			
4		issue report		issue report	
5		not answered			
6	commit issue report	commit issue report			
7		not answered		not answered	
8				not answered	
9		not answered			
10		source code issue report		issue report	
Cassandra	**meaning**	**cause**	**context**	**solution**	**impact**
1		code comment issue report	issue report issue report		
2				not answered	source code
Zookeeper	**meaning**	**cause**	**context**	**solution**	**impact**
1		issue report			
2		issue report			

Discussion

We discuss alternative approaches, such as web searching and reading the mailing list, to resolve real-world inquiries in this subsection.

Using web search engine to resolve the real-world inquiries

We first compare the use of development knowledge and the use of a web search engine

to resolve the real-world inquiries. One may consider using a web search engine, such as *Google*, to resolve log line inquiries. We use *Google* to search for the real-world inquired log lines and check whether the first 10 results from the web search engine can answer the inquiries. If the log lines are ambiguous, we add the name of the subject system after the log line.

We focus on the 14 real-world inquired log lines from the user mailing list since the other 31 real-world log lines are also from *Google* web search. For the 14 mailing list inquiries, we find four types of relevant search results from *Google*: the online link to the mailing list, the online link to the development knowledge (e.g., the Apache JIRA web interface), open source community websites (e.g., *Stack Overflow*) and personal websites. We find three inquired log lines where open source community websites (e.g., *Stack Overflow*) can provide useful information and two inquired log lines where personal websites can provide useful information. From such results, we consider that the development knowledge (9 out of 14 inquired log lines) outperforms the results from a web search engine (5 out of 14 inquired log lines).

Using mailing list to resolve the real-world inquiries

We compare the use of development knowledge and the answers from mailing list to resolve the real-world inquiries. We find that the development knowledge is comparable to the answers in the mailing list. In the mailing list, 10 inquiries are resolved after email discussion, while 9 inquiries are resolved by development knowledge. However, we notice that the answers from mailing list are clearer and more precise; while the development knowledge contains more content and needs interpretation to resolve the inquiries.

Development knowledge can help in resolving 9 out of 14 real-world inquiries from the user mailing list and 15 out of 31 real-world inquiries from other sources on the web. Development knowledge outperforms web search and is comparable to browsing the mailing list when trying to resolve log line inquiries.

3.7 RQ4: Can Experts Assist in Resolving Inquiries of Log Lines?

Motivation

Logging statements are embedded in the source code by developers. Intuitively, the developers who added or updated the log lines may be the most suitable individuals to address any log line inquiries. From RQ3, we find that the replies in the user mailing lists of the subject systems often resolve the inquiries (see Section 3.6). We verify whether experts of the log lines, such as the developers who added the logging statements, do provide such information more often than non-experts of the log lines. If the experts provide extra information about log lines, we can use development knowledge and leverage existing techniques [MH02] to identify the experts of the log lines to rapidly re-route log line inquiries to the most appropriate developer (i.e., the owner of the log line).

Approach

Previous research by Mockus *et al.* [MH02] proposed using the code change information to identify the experts of code units, such as modules and methods. Based on this technique, we define an expert for a log line as *"a developer who has committed changes to the method/function that contains the logging statement, which generates the log line."* For example, log line "Lost tracker" of *Hadoop* is in a method called *"lostTaskTracker"* and all 10 developers who have committed changes to the method "lostTaskTracker" are considered experts for that log line.

Table 3.8: Resolved and un-resolved email inquiries.

System	total	resolved		not-resolved		
		by expert	by non-expert	replied by expert	only replied by non-expert	not replied
Hadoop	10	5	1	0	1	3
Cassandra	2	0	2	0	0	0
Zookeeper	3	3	0	0	0	0

We read the email threads of all 15 inquired log lines and assign them to 1 of 5 categories: resolved by expert, resolved by non-experts, not-resolved but replied by expert, not-resolved but only replied by non-expert and not replied. We also calculate the 'time to first reply' for experts and non-experts to measure their response time.

Results

Who resolved the inquiries in the email threads?

We find that an expert is crucial in providing answers to log line inquiries. An overview of the results is presented in Table 3.8. 8 of the 11 resolved inquiries had replies by log experts. The one inquiry from *Hadoop* that was tagged as 'resolved by non-expert' was actually resolved by the person who made the inquiry. The two inquiries resolved by non-experts from *Cassandra* were resolved by other developers of the project. We find that experts have considerable knowledge about the log line itself, as well as the rest of the project. This knowledge assists them in resolving inquiries of log lines. For example, an expert pointed out that an inquired log line from *Zookeeper* was due to a fixed bug. It is hard for a non-expert to provide such information.

It is interesting to observe that all inquiries that are answered by experts are answered through short and affirmative answers. For example, the email that inquired about "Lost tracker" was replied to by one expert and one non-expert. The non-expert asked about the context of the log line, while the expert directly gave the answer and also pointed out that there were bugs related to this log line. Another example is the email that inquired about "fetch failure", which was discussed by 4 different non-experts, and still had no affirmative

Table 3.9: 'Time to first reply' of log line inquiries.

	Expert			Non-expert		
	Median	Min	Max	Median	Min	Max
Hadoop	3 h	3 min	96 h	1 h	8 min	2 h
Cassandra	-	-	-	7 min	77 h	105 h
Zookeeper	5 h	1 h	9h	-	-	-

answer at the end of the discussion. We also note that the experts for all the inquired log lines are still active members of the development team of the three subject systems. Therefore, the log line inquiries that had no replies were not due to the absence of experts from the development team, but most probably due to the experts not being aware of such inquiries.

How long does the first reply to a log line inquiry take?

Experts are important in providing useful information about log lines. The median times for the first reply to a log line inquiry from experts in *Hadoop* and *Zookeeper* are 3 and 5 hours respectively, while the experts from *Cassandra* never replied (see Table 3.9). The 'time to first reply' from experts is sometimes slower than non-experts. From browsing the email replies, we find that reason for slower reply is that experts often replies with a definitive answer, while non-experts often ask for additional information. Therefore, although experts may reply later than non-experts, they often provide more useful information than the non-experts.

Our findings provide evidence that experts are important in providing useful information about log lines. Therefore, finding the expert of a log line may be the most effective way to understand the log line. However, a person with a log line inquiry may not be able to direct their inquiries to experts because the list of log line experts is not easily accessible. Moreover, experts may be too busy to check the mailing list. Automated ways to push inquiries to experts may be of great value for understanding log lines.

Discussion

As we stated in the previous subsection, experts can provide useful information about

a log line. However, we do not know how many experts are there for each log line, how many of these experts are active on the mailing lists and how many orphaned log lines exist without any experts. The results for these questions are below.

How many experts are there for the inquired log lines?

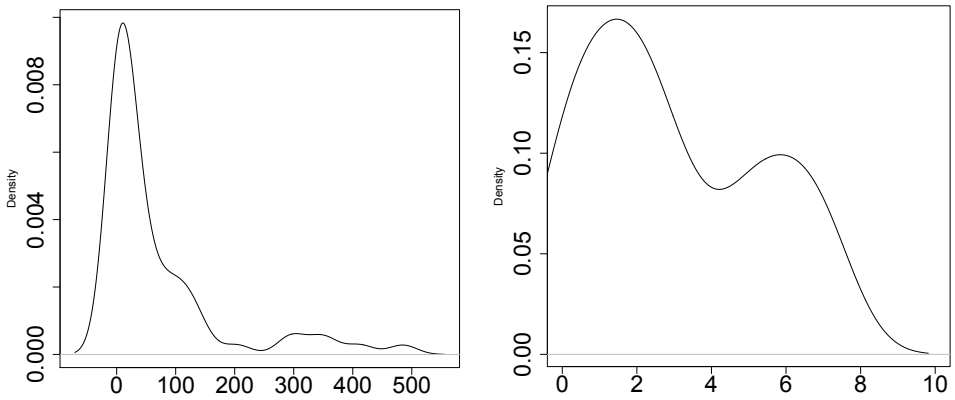
We manually identify the experts of the 300 sampled logging statements in the 3 subject systems. We not only identify the developer who commits the revisions, but we also read the commit comments. If a commit comment says that the revision is contributed on behalf of another developer, we consider the developer who coded the revision as the expert instead of the developer who committed the revision.

We find that, on average, there are 4.6, 3.1 and 2.8 experts for each log line in *Hadoop*, *Cassandra* and *Zookeeper*, respectively. We also find that some log lines from *Hadoop* have over 30 experts because the log lines are embedded inside large methods, while some log lines only have 1 expert. Therefore, it is easier to find an expert for the log lines in large methods because there are more experts to answer the inquiries. Inquiries about log lines with only 1 expert may take longer to answer because the expert may not be available or not currently working as part of the team.

How active are experts on the mailing list?

We examine all experts of the 15 inquired log lines in the mailing list. For each expert, we count the number of emails sent and the number of active years in the mailing list. Figure 3.1(a) shows the distribution of the number of emails sent by the experts of the 10 inquired log lines for *Hadoop*. We can identify three types of experts: 1) heavy email contributors, who might send hundreds of emails over the years and are typically experts of the entire project, 2) medium email contributors, who are typically experts of subsets of the projects and 3) light email contributors who are not typically considered in the "core" development team.

From the number of active years shown in Figure 3.1(b) for the log-line experts in *Hadoop*, we observe that some experts are active in the mailing list throughout the entire



(a) number of emails (b) number of years

Figure 3.1: Density plot of the number of emails and active years of the experts of the inquired log lines from Hadoop.

history of the project (which is 8 years), while other ones are only active for a short period of time.

We also find that experts who resolve log line inquiries tend to be heavy email contributors with long active years. This finding also indicates the value of automatically pushing inquiries to experts since not all experts might be active on the mailing list and are often likely to miss inquiries, that they could easily resolve.

How many orphan log lines are there with no experts around?

We define orphan log lines as log lines whose experts are no longer committing code changes to the project since the last release prior to the inquiry. We examine experts of the 15 log lines inquired in the mailing lists and we find that none of the log lines are orphaned. Such a result indicates that searching for the experts of log lines may be one of the best ways to resolve inquiries about log lines because it is unlikely that the inquired log line is an orphan.

However, as we see in Tables 3.8 and 3.9, there are cases where there are no replies or the replies take as long as 4 days. The person who has posed the inquiry may need information much faster, especially when the log line is associated with an error. Hence, it

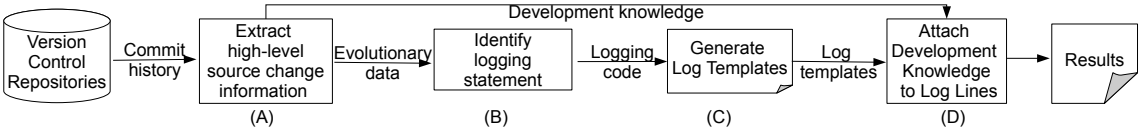


Figure 3.2: Overview of our approach to associate development knowledge to the corresponding log lines

would be beneficial if one could either directly contact the log line expert or automatically get the same information provided by the experts elsewhere and without having to wait for a long period of time.

> *The responses from experts of logs can assist in log line understanding. However, log line experts may not be easily identifiable or available.*

3.8 Automatically Providing Development Knowledge for Log Lines

In the previous sections, we found that development knowledge is a good source of the various types of inquired information about log lines. However, collecting such knowledge is a cumbersome, manual and error-prone process. In this section, we propose an automated approach to associate development knowledge with the corresponding log lines. Figure 3.2 shows a general overview of our approach.

3.8.1 Approach

Step 1 - Extracting high-level source code change information: Similar to C-REX [Has05], J-REX [SJAH09; SAH11; Sha10] is used to extract historical information from Java software systems. We use J-REX to extract high-level source change information from the SVN repositories. J-REX extracts source code snapshots for each Java file revision from the SVN repository. Along with the extraction, J-REX also keeps track of all commit messages and

issue report IDs in the commit messages for all revisions. For each snapshot of a source code file, J-REX builds an abstract syntax tree for the file using the Eclipse JDT parser.

Step 2 - Identifying logging statements: Software projects typically leverage logging libraries to generate logs. Typically, logging source code contains method invocations that call the logging library. For example, in *Hadoop*, a method invocation by an object of class "*Logger*" is considered as a logging statement in the source code. Knowing the logging library of the subject system, we analyze the output of J-REX to identify the logging source code fragments and the corresponding historical changes to these log statements and the source code in the method containing the logging statement.

Step 3 - Generating log templates: Typically a logging statement contains both static parts and dynamic parts. The static parts are the fixed strings and the dynamic parts are the variables whose values are determined at run-time. For example, in a logging statement *LOG.info("Retrying connect to server: Already tried" + n+ "time(s)"*, the variable n is the dynamic part, while *"Retrying connect to server: Already tried"* and *"time(s)"* are the static parts. We identify the static and dynamic parts in each logging statement and create a regular expression for each logging statement [XHF+09b]. The regular expression generated for the above example is *Retrying connect to server: Already tried.*time\(s\)*.

Step 4 - Attaching development knowledge to log lines: Having all the regular expressions for the templates of the logging statements, we are able to match log lines in the logs to the logging statements. For each of the inquired log lines, we find all the log templates that match it.

The output of our automated approach maps, for each log line template, the information from snapshots and historical development knowledge at the method-level (i.e., the method that contains the logging statements). For example, if method A contains a logging statement, the output of our automated approach is the source code of method A, the code comments inside method A or next to the declaration of method A, methods one level before and after method A in the call graph, all the code commits that changes method A and

all the issue reports mentioned in the comments of these commits.

3.8.2 An example

Our automated approach can successfully provide the development knowledge to assist in understanding the 15 inquired log lines in the user mailing lists, 31 inquired log lines from the web and the 300 sampled logging statements. In this subsection, we use the log line "fetch failure" that is inquired in the *Hadoop* mailing list as an example to describe our approach.

Log line. Our approach scans all the log templates in *Hadoop* and finds that the log line partially matches with a log template *(.)* Reporting fetch failure for (.)* to jobtracker (.)**, which is generated by a logging statement *LOG.info ("Reporting fetch failure for " + mapId + " to jobtracker.");*.

Source code. By tracking this logging statement, we find that the logging statement is in the method "checkAndInformJobTracker" in file "ShuffleScheduler.java".

Comment. We find that the code comment right before the method saying *Notify the JobTracker after every read error, if 'reportReadErrorImmediately' is true or after every 'maxFetchFailuresBeforeReporting' failures*.

Call graph. From the call graph of this method, we find this method is called by method "copyFailed" in class "ShuffleScheduler".

Commit. We generate all commits that change this method. One code commit (revision 889496) has message "MAPREDUCE-1171. Allow shuffle retries and read-error reporting to be configurable. Contributed by Amareshwari Sriramadasu.".

Issue report. From the commit message, we found an issue report with id "MAPREDUCE-1171" [8] related to this code change. From the issue report, we find the cause of the log line and the context of the log line.

[8] https://issues.apache.org/jira/browse/MAPREDUCE-1171

From the information in the source code, code comments, call graph, code commit and issue report we have information about:

- ***Meaning:*** There is a data reading error (from comment, commit and issue report).
- ***Cause:*** One of the possible reasons is a configuration mistake (from issue report).
- ***Context:*** The event happens during the shuffle period (from source code, commit, issue report and call graph).
- ***Impact:*** The event impacts the jobtracker component (from comment and call graph).
- ***Solution:*** Adding a configuration option may solve the issue (from issue report).

3.9 Threats to Validity

External validity. Our case study only focuses on 300 randomly sampled logging statements, 15 email log lines inquiries and 31 log line inquiries from the web search results from three "systems" projects with years of history and a large user base. The results may not generalize to other systems, for example if developers do not input useful information in the issue reports or SVN repositories, then these repositories would not be good sources of inquired information about log lines. Nevertheless, our key contribution is to demonstrate an approach that can leverage already-available historical information to assist in understanding log lines. Additional case studies on other open source and commercial systems in different domains are needed. Also this work would not help for systems with limited logging statements. However, if such systems produce graphical interface messages (e.g., error pop-up windows) we simply would need to link such calls to the GUI pop-up function instead of the logging statements.

There may be other sources of development knowledge, such as design documents, which maybe useful to understand log lines. Adding additional information could improve

our approach. A broader study, which includes more sources of development knowledge needs to be conducted to identify the potential of each source in resolving the various types of log line inquiries.

Construct validity. The manual examination throughout the study is performed by the author of this thesis. Although we have some experience in using and studying the subject systems in our previous research (e.g., [SJH+13]), our expertise may be imperfect and limited one particular version.

We choose to associate development knowledge to log lines at the method level. The higher the level, the more development knowledge can be attached, but the more overwhelming such attached knowledge may become. If we set the level lower (e.g., logging statement level), we would lose some useful knowledge about log lines. Future work should explore attaching development knowledge at different levels of granularity.

Based on previous research of source code expertise [MH02], we consider developers who commit code that changes the methods containing the logging statements as the experts of the log lines. Such an assumption may not be entirely correct. There could be other approaches to refine the set of experts for log lines. However, even with this naive approach for identifying experts, we are able to show the importance of experts in resolving log line inquiries.

When a log line can match multiple log templates, we face the challenge of identifying the correct log template. A naive example is, a log statement that simply outputs the value of a variable would match every log line. In the three subject systems in our case study, we find that less than 10% of log templates can match with every log line. We manually check these logging statements and find that in most of the cases, such logging statements output the Java exception message. In such a situation, the person posting a log line inquiry would need to identify the source of the log line in the code, i.e., the logging statement which outputs the log line. Once such identification is done, our approach can attach the correct information. Such identification is not complex since such log lines with exception messages

typically contain the stack trace instead of a high-level message – the stack trace can then be used to identify the corresponding source code logging statement. Future work should explore the automated attachment of development knowledge to such stack traces.

3.10 Chapter Summary

Much of the knowledge about log lines (the meaning and purpose of the log lines), lives only in the minds of the developers who embedded the logging statements in the source code. Users and operators rely heavily on logs for various tasks. Due to the lack of communication with developers of large software systems, users and operators often face the challenge of understanding the logs. Such challenges may jeopardize the effectiveness and correctness of log analysis and understanding.

To investigate the challenge of understanding logging statements, we performed a case study on three open source systems, *Hadoop*, *Cassandra* and *Zookeeper*. We manual examine 300 randomly sampled logging statements from the subject systems and we also find 46 real inquiries about log lines in the user mailing lists of the three subject systems and from web search results. From the results of our case study, we find that:

- System operators sometimes find it difficult to understand the log lines. The cause, meaning, impact, context and solution of the log lines are often inquired about in the mailing lists.
- Development knowledge can be leveraged for understanding log lines. In particular, issue reports are useful for log line understanding.

Since collecting development knowledge for log lines requires expertise that the operators may not possess, we also present an automated approach to associate snapshot (like executable source code, code comments and call graph) and historical (like commit messages and issue reports) development knowledge to log lines. In summary, we identify the

importance of leveraging development knowledge for log line understanding. We conclude that the developers should consider the needs of operators and that they should document more information about log lines.

This chapter focuses on challenges associated with understanding logging statements. In the next chapter, we study another aspect of the challenges associated with logging statements – evolving logging statements.

CHAPTER 4

How Do Logging Statements Evolve?

Substantial research focuses on understanding the dynamic nature of software systems in order to improve software maintenance and program comprehension. This research typically makes use of automated instrumentation and profiling techniques without considering domain knowledge. In this chapter, we examine logs as another source of dynamic information that is generated from statements inserted into the codebase during development to draw the system operators' attention to important run-time events.

The availability of logs has sparked the development of an ecosystem of *Log Processing Apps* (*LPAs*) that surround the software system under analysis to monitor and document various run-time events. The dependence of *LPAs* on the timeliness, accuracy, and granularity of the logs means that it is important to understand the nature of logs and how they evolve over time. In Chapter 3, we examined the challenge of understanding logging statements. In this chapter, we focus on another aspect – the evolution of logs, which to our knowledge, has not yet been empirical studied. In a case study on two large open source and one industrial software system, we explore the evolution of logs by mining the execution logs of these systems and the logging statements, which produce such logs in the source code. Our study illustrates the need for better traceability between logging statements and the *LPAs* that analyze the logs produced by these statements. In particular, we find that logging statements change at a high rate across versions, which could lead to fragile *LPAs*. We found that up to 70% of these changes could have been avoided and the impact of 15% to 80% of the changes can be controlled through the use of robust analysis techniques by *LPAs*. We also found that *Log Processing Apps* that track implementation-level logging statement (e.g., performance analysis) and the *Log Processing Apps* that monitor error message logging statements (e.g., system health monitoring) are more fragile than *Log Processing Apps* that track domain-level logging statements (e.g., workload modeling), since the latter logging statements tend to be more stable and long-lived.

4.1 Introduction

Software profiling and automated instrumentation techniques are commonly used to study the run-time behaviour of software systems [CZD+09]. However, such techniques often impose high overhead, especially for real-world workloads. Worse, software profiling and instrumentation are performed after the system has been built, based on limited domain knowledge. Therefore, extensive instrumentation often leads to an enormous volume of results that are impractical to meaningfully interpret.

In practice, system operators and developers typically rely on logs, consisting of the major system activities (e.g., events) and their associated contexts (e.g., a time stamp), to understand the high-level field behaviour of large systems and to diagnose and repair bugs. Rather than generating tracing information in a blind way, developers choose to explicitly communicate some information that is considered to be particularly important for system operation through logs. The purpose and importance of the information in such logs varies based on their purpose. For example, detailed debugging logs are relevant to developers, while operation logs summarizing the key execution steps are more relevant to operators.

The rich nature of logs has created a whole new market of applications that complement large software systems. We collectively call these applications, *Log Processing Apps* (*LPAs*). Such *Apps* are used, for example, to generate workload information for capacity planning of large scale systems [HMF+08; NWV09], to monitor system health [BGSZ10], to detect abnormal system behaviours [JHHF08b], or to flag performance degradations [JHHF09]. As such, these *LPAs* play an important role in the development and management of large software systems, to the extent that major decisions like adding server capacity or changing company's business strategy can depend on the information derived through such *LPAs*.

Log changes often break the functionality of the *LPAs*. Often, *LPAs* are in-house applications that are highly dependent on the logs. Although *LPAs* are typically built on commercial

platforms by IBM [ibm] and Splunk [spl], the actual link between the *LPAs* and the monitored system depends heavily on the meaning and the specific kind and format of the logs in use. Hence, the *Apps* require continuous understanding and maintenance as the content, format or type of logs changes and as the needs change. In Chapter 3, we studied the challenges of understanding logs. However, since little is known about the evolution of logs, it is unclear how much maintenance effort *LPAs* require.

We choose two open source systems and one closed source software system with different sizes and application domains as the subjects for our case study. We choose ten releases of *Hadoop*, five releases of *PostgreSQL*, and nine releases of a closed source large enterprise application, which we will refer to as *EA*. We study logs at the execution level for these three systems. We study logs at the code level for only *Hadoop* and *PostgreSQL* due to the lack of access to the source code of *EA*. We do not study the *Zookeeper* and *Cassandra* systems from Chapter 3 because they do not have long development histories. Our study is the first step in understanding the maintenance of *LPAs* by studying the evolution of the logs (i.e., their input data).

Our study tracks the logs of the execution of a fixed set of major features across the lifetime of the three studied systems, and analyzes the logging statements in source code of the two studied open source systems. Our study allows us to address the following research questions:

RQ1: How much do logs change over time?

> We find that, over time, the amount of run-time logs, i.e., log lines generated from executing a same set of features of these systems, increases 1.5-2.8 times compared to the first-studied release. The logging statements in the code corresponding to these features increases 0.17-2.45 times. We note that as few as 40% of the run-time logs are unchanged across releases and up to 21% of the run-time logs are modified across

releases. The modifications of the logs may be troublesome as they cause the *LPAs* to be more error-prone.

RQ2: What types of modifications happen to logs?

Examining the modification to logs across releases, we identify eight types of modifications. 10-70% of the modifications can be avoided and the impact of 15-80% of them can be minimized through the use of robust analysis techniques by the *LPAs*. The remaining modifications are risky and should be tracked carefully to avoid breaking the *LPAs* associated with the studied systems.

RQ3: What information is conveyed by short-lived logs?

We find that short-lived logs focus more on system errors and warnings. They often contain implementation-level details and system error messages. Based on these findings, more resources should be allocated to maintain *LPAs* that heavily depend on implementation-level information or *LPAs* that monitor system errors.

The findings from this chapter highlight the need for tools and approaches (e.g., traceability techniques) to ease the maintenance of *LPAs*.

The rest of this chapter is organized as follows: Section 4.2 presents an example to motivate our work. Section 4.3 presents the data preparation steps for our case study. Section 4.4 presents our case studies and the answers to our research questions. Section 4.5 discusses the limitations of our study. Section 4.6 discusses prior work. Section 4.7 concludes the chapter.

4.2 A Motivating Example

We use a hypothetical, but realistic, motivating example to illustrate the impact of log changes on the development and maintenance of *LPAs*.

The example considers an online file storage system that enables customers to upload, share and modify files. Initially, there were execution logs used by operators to monitor the performance of the system. The information recorded in the execution logs contained system events, such as “user requests file”, “start to transfer file” and “file delivered”.

Release n

Suppose that operators identified a performance problem in release n-1. In order to diagnose the problem, developer Andy added more information to the context of the execution events in logs, such as the ID of the thread that handles file uploads. Using the added context in the execution logs, the operators identified the root cause of the performance problem and developer Andy resolved it. A simple *LPA* was written to continuously monitor for the re-occurrence of the problem by scanning the logs for the corresponding system events.

Release n+1

The file upload feature was overhauled, during which the developers changed the communicated events and their associated log entries. These context changes to the log files led to failures of the *LPAs*, since they could no longer parse the log lines correctly.

The application also started giving false alarms. After several hours of analysis, the root-cause of the false alarm was identified. The logs had been changed by another developer, Bob, who was not aware that others made use of this information since there is no traceability between the information and the *LPA*. To avoid these problems reoccurring in the future, developer Andy marked the dependence of the *LPA* on this log event in an ad hoc manner through some basic code comments.

From the motivating example, we can observe the following:

- Logs are crucial for understanding and resolving field problems and bugs.
- Logs are continuously changing due to development and field requirements.

Table 4.1: Overview of the studied releases of *Hadoop* (minor releases in italic)

Release	Release Date	K SLOC
0.14.0	20 August, 2007	122
0.15.0	29 October, 2007	137
0.16.0	7 February, 2008	181
0.17.0	20 May, 2008	158
0.18.0	22 August, 2008	174
0.19.0	21 November, 2008	293
0.20.0	22 April, 2009	250
0.20.1	14 September, 2009	258
0.20.2	26 February, 2010	259
0.21.0	23 August, 2010	201

- *LPAs* are highly dependent on the logs.

Unfortunately, today there are no approaches to ensure traceability between source code and *LPAs*, leading *LPAs* to be very fragile, as they have to adapt to continuously changing logs.

4.3 Case Study Setup

To study the evolution of logs, we mine both the actual logs generated dynamically at runtime as well as the logging statements in the source code. Depending on the usage scenario, some of the logging statements will result in actual logs, while other logging statements will not be executed and hence not generated. In this section, we present the studied systems and our approach to recover logs from their execution.

4.3.1 Studied systems

We chose two open source systems and one closed source software system with different sizes and application domains as the subjects for our case study. We chose ten releases of *Hadoop*[1], five releases of *PostgreSQL*[2], and nine releases of a closed source large enterprise

[1]http://hadoop.apache.org/ last verified January 2014.
[2]http://www.postgresql.org/ last verified January 2014.

Table 4.2: Overview of the studied releases of *PostgreSQL*

Release	Release Date	K SLOC
8.2	5 December, 2006	471
8.3	4 February, 2008	533
8.4	1 July, 2009	576
9.0	20 September, 2010	613
9.1	12 September, 2011	654

application, which we will refer to as *EA*.

Hadoop is a large distributed data processing platform that implements the MapReduce [DG08] data processing paradigm. We use releases 0.14.0 to 0.21.0 for our study as shown in Table 4.1. We chose these releases since 0.14.0 is the earliest one that is able to run in our experimental environment and 0.21.0 is the most recent release at the time of our previous study [SJA+11]. Among the studied releases, 0.20.1 and 0.20.2 are minor releases of the 0.20.0 series. Because *Hadoop* is widely used in both academia and industry, various *LPAs* (e.g., *Chukwa* [RK10] and Salsa [TPK+08]) are designed to diagnose system problems as well as monitor the performance of *Hadoop*.

PostgreSQL is an open source database management system written in C. We chose releases 8.2 to 9.1 for our study because they are the releases that are able to run on our experimental environment (*Windows Server*). The overview of the releases is shown in Table 4.2. All the releases used in our study are major releases. Various *LPAs* have been developed for *PostgreSQL*, e.g., *pgFouine* [pgf] analyzes *PostgreSQL* logs to determine whether certain queries need optimization.

The enterprise application (*EA*) in our study is a large scale, communication application that is deployed in thousands of enterprises worldwide and used by millions of users. Due to a Non-Disclosure Agreement, we cannot reveal additional details about the application. We do note that it is considerably larger than *Hadoop* and *PostgreSQL*. Moreover, it has a much larger user base and longer history. We studied nine releases of *EA*. The first seven minor releases are from one major release series and the later two releases are from another

major release. We name the release numbers 1.0 to 1.6 for the first major release and 2.0 to 2.1 for the second major release. There are currently several *LPAs* for the *EA*. These *LPAs* are used for functional verification, performance analysis, capacity planning, system monitoring, and field diagnosis of customer deployments worldwide.

We do not choose to study *Zookeeper* and *Cassandra* from Chapter 3 since they do not have long development history to study the evolution of logs.

4.3.2 Uncovering logs and logging statements

We perform an execution-level and code-level analysis of logs over-time. Studying logs at both execution-level or code-level is important, since execution-level logs contain the information that *LPAs* actually depend on and the code-level analysis would help us understand the execution-level findings.

Execution Level

Our execution-level approach to recover the logs of software systems consists of the following three steps: 1) system deployment, 2) data collection, and 3) log abstraction.

1. System Deployment

For this study, we seek to understand the logs of each system based on exercising the same set of features across several releases of the studied systems. To achieve our goal, we run every version of each systems with the same workload in an experimental environment.

2. Data collection

In this step, we collect the execution logs from these systems. We apply realistic runtime workloads to these systems and collect the logs generated during the execution of the systems.

The *Hadoop* workload consists of two example programs, *wordcount* and *grep*. The *wordcount* program generates the frequency of all the words in the input data and the *grep*

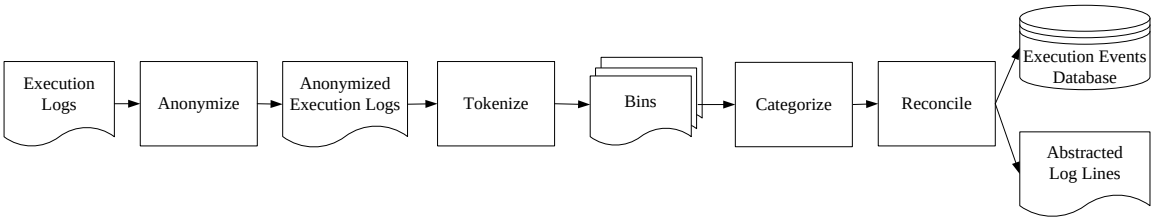


Figure 4.1: Overall framework for log abstraction.

program searches the input data for a string pattern. In our case study, the input data for both *wordcount* and *grep* is a set of data with a total size of 5 GB. The search pattern of the *grep* program in our study is one particular word ("fail").

To collect consistent and comparable data for the *EA*, we choose a subset of features that are available in all versions of the *EA*. We simulate the real-world usage of the *EA* system through a specialized workload generator, which exercises the configured set of features for a given number of times. We perform the same 8-hour standard load test [Bei84] on each of the *EA* releases. A standard load test mimics the real-world usage of the system and ensures that all of the common features are covered during the test.

We deploy the database of the *Dell DVD Store* [ds2] on a *PostgreSQL* database server for each of the releases. We extract database queries from the source code of the *Dell DVD Store* as the typical workload of the *Dell DVD Store* database. We leverage *Apache JMeter* [jme] to execute the queries repetitively for 2 hours.

Note that at the execution level, we use realistic run-time scenarios and workloads. Hence, our experiments cannot guarantee full coverage of all features and hence run-time events. This is the reason why we also need to perform the code-level analysis presented in the next sub-section.

3. Log abstraction

We analyze the generated execution logs. Execution logs (e.g., Table 4.3) typically do not follow a strict format. Instead, they often use inconsistent formats [BvDDT07]. For example, one developer may choose to use "," as a separator, while another developer may

choose to use " \t". The free-form nature of these logs makes it hard to automatically extract information from them. Moreover, log lines typically contain a mixture of static and dynamic information. The static values contain the description of the execution events, while the dynamic values indicate the corresponding *context* of these events.

One must identify the different kinds of system events based on the event instances in the collected execution logs. We use a technique proposed by Jiang *et al.* [JHHF08a] to automatically extract the execution events and their associated context from the logs. Figure 4.1 shows the overall process of log abstraction. As shown in Table 4.4, the descriptions of task types, such as "Trying to launch", are static values, i.e., system events. The time stamps and task IDs are dynamic values (i.e., the context for these events). The log abstraction technique normalizes the dynamic values and uses the static values to create abstracted execution events. We consider the abstracted execution events as representations of the system events.

First, the anonymize step uses heuristics to recognize dynamic values in log lines. For example, "TaskID=01A" will be anonymized to "TaskID=$id". The tokenize step separates the anonymized log lines into different groups (i.e., bins) according to the number of words and estimated parameters in each log line. Afterwards, the categorize step compares log lines within each bin and abstracts them into the corresponding execution events (e.g., "Reduce"). Similar execution events with different anonymized parameters are categorized together. Since the anonymize step uses heuristics to identify dynamic information in a log line, there is a chance that the heuristic might fail to anonymize some dynamic information. The reconcile step identifies such dynamic values by analyzing the difference between the execution events within the same bin. Case studies in previous research [JHHF08a] show that the precision and recall of this log abstraction technique both are high, e.g., over 80% precision and recall.

Table 4.3: Example of execution log lines

#	**Log lines**
1	time=1, Trying to launch, TaskID=01A
2	time=2, Trying to launch, TaskID=077
3	time=3, JVM, TaskID=01A
4	time=4, Reduce, TaskID=01A
5	time=5, JVM, TaskID=077
6	time=6, Reduce, TaskID=01A
7	time=7, Reduce, TaskID=01A
8	time=8, Progress, TaskID=077
9	time=9, Done, TaskID=077
10	time=10, Commit Pending, TaskID=01A
11	time=11, Done, TaskID=01A

Table 4.4: Abstracted execution events

Event	**Event**	#
E_1	time=$t, Trying to launch, TaskID=$id	1,2
E_2	time=$t, JVM, TaskID=$id	3,5
E_3	time=$t, Reduce, TaskID=$id	4,6,7
E_4	time=$t, Progress, TaskID=$id	8
E_5	time=$t, Commit Pending, TaskID=$id	10
E_6	time=$t, Done, TaskID=$id	9,11

Code Level

Our code-level approach consists of two steps: 1) code abstraction and 2) identification of logging statements.

1. Code abstraction

We download the source code of each release of the studied systems. For *Hadoop*, a Java based system, we leverage the *Eclipse JDT parser* [jdt] to create abstract syntax trees for each source code file. For *PostgreSQL*, we use *TXL* [Cor11] to parse code into abstract syntax trees that are stored in XML format.

2. Identification of logging statements

Software projects typically use logging libraries to generate logs. One of the most widely

used logging libraries in Java systems is *Log4j* [loga]. We browse the source code of the studied systems and identify the logging libraries.

Using this knowledge, we analyze the generated abstract syntax trees to identify the logging-related source code fragments and changes. Typically, logging code contains method invocations that call the logging library. For example, for systems using *Log4j*, a method invocation like "*LOG*" is considered to be logging code. Changes that involve such code are considered as log churn.

With the extracted logs from both execution and code level analyses, we study the evolution of such logs. In particular, we measure three aspects: 1) how much do logs change over time, 2) what types of modifications happen to logs and 3) what information is conveyed in short-lived logs. We plan to answer these three question by performing empirical studies on both open source and enterprise systems.

We only study the logs at the code-level for *Hadoop* and *PostgreSQL* due to the lack of access to the source code of *EA*.

4.4 Case Study Results

In this section, we present the findings on our research questions. For each research question, we present our motivation, approach, and results.

4.4.1 RQ1: How much do logs change over time?

Motivation

The evolution of logs impacts the maintenance of *LPAs*. The frequent changing of the logs makes *LPAs* fragile due to the lack of established traceability techniques between *LPAs* and logs. Hence, changes to execution events give an indication of the complexity of designing and maintaining *LPAs*.

Approach

At the execution level, we use the number of unique abstracted execution events (we call the abstracted execution events as "events" for short in the later part of this chapter) as a measurement of the number of logs. We also study the log changes by measuring the percentages of unchanged, added and deleted execution events. Given the current release *n* and the previous release *n-1*, the percentages of unchanged and added execution events are defined by the ratio of the number of unchanged and added events in release *n* over the number of total execution events in the current release (*n*), respectively. The percentage of deleted execution events is defined the same as unchanged and added events, except over the number of total execution events in the previous release (*n-1*). For example, the percentage of unchanged execution events in release *n* ($P_{unchanged\ n}$) is calculated as:

$$P_{unchanged\ n} = \frac{\#\ unchanged\ events_n}{\#\ total\ events_n} \tag{4.1}$$

We identify modified events by manually examining added and deleted events. We use the frequency of execution events in two releases to assist in mapping between modified events with similar wording across releases. Given the large number of events of *EA* (over 1,900 across all releases), we only examine the top 40 most occurring events since they represent more than 90% of the total number of logs. This means that for *EA*, we use a similar equation as (4.1), except that the number of *total execution events* for *EA* is always 40 as we only examine 40 execution events.

At the code level, we use the number of logging statements as a measure of the number of logs. The unchanged, added and deleted logging statements at the code level are studied in a similar way as at the execution level. To assist in understanding the logging statement changes at the code level, we also calculate the code churns (total number of added and deleted lines of code) of each studied release. We leverage *J-REX* [SJAH09; SAH11; Sha10], a high-level evolutionary extractor of source code history similar to C-REX [Has05], to

identify the modification of logging statements during the development of the systems.

Results: The total amount of logs

Execution level:

We first study the number of logs in the history of both systems. Figure 4.2 shows the growth trend of logs in *Hadoop*. The growth of logs is faster than the growth of source code (shown in Table 3.1). At the execution level, we note that the number of logs in the last studied release ($0.21.0$) is 2.8 times the number of the first studied release ($0.14.0$), while the size of the source code has increased by less than 30% (201 KLOC to 259 KLOC). In particular, the number of logs increases significantly in release $0.21.0$ even though the size of the corresponding source code decreases by 20%. We also note that the logs increase more between major releases than between two minor releases. The large increase of the number of logs at the execution-level in major releases indicates that additional maintenance effort might be needed for *LPAs* to continue operating correctly even if the existing *LPAs* do not use logs about new features or the additional logs about the currently analyzed features.

The logs of *PostgreSQL*, shown in Figure 4.3, also shows a growth trend at the execution level. Unlike *Hadoop*, there exists no release in *PostgreSQL* that has significantly larger log growth than other releases. We believe that the reason is that *PostgreSQL* is a mature and stable system with more than 20 years history, while the development of *Hadoop* started in 2005. Therefore, the developers of *PostgreSQL* would be unlikely to add or delete substantial features in the system or significantly change the architecture of the system in one release. The stable nature of the logs in *PostgreSQL* may suggest that its *LPAs* are easier to maintain. The developers of *LPAs* can focus more on adding features to the *LPAs* by analyzing the added logs, and focus less on maintaining features based on old logs.

For the *EA* system, we only study execution level logs due to lack of access to the source code data. Since we study only two major releases, we study the number of logs in each major release (nine releases in total) instead of generalizing a trend over the two studied

releases. We note that the number of logs does not change significantly in the first major release, while the logs do increase significantly in the second major release. The logs of the last studied release (2.1) is 1.5 times the amount of the first studied release (1.0).

Code level:

The trend of logging statement growth is similar to that of log growth at the execution level. As an exception, the number of logging statements in release 0.17.0 of *Hadoop* increases at the execution level but decreases at the code level. Table 4.6 shows that over 48% of the logging statements in release 0.16.0 is removed in release 0.17.0. The release notes [hadb] show that the *HBase* component of *Hadoop* of release 0.16.0 became a separate project right before the release of 0.17.0 of *Hadoop*.

In both *Hadoop* and *PostgreSQL*, less than 10% of the logging statements are observed at the execution level. Operators typically examine the logs that appear in the field when the software system is upgraded to a new release and adapt their *LPAs* accordingly. However, our results indicate that most of the logging statements are not observed during a typical execution of the system. The changes to such logging statements may cause problems in the *LPAs*. Developers of the software system may consider documenting all the logging statements in the system and transferring such knowledge to the operators in the field. On the other hand, it may be good news for *LPA* developers, since most of the changed logging statements may do not show up during execution, therefore the *LPAs* should not break in practice.

Results: The number of changed logs

Execution level:

A closer analysis of the logs across releases shows that for all the studied systems at the execution level, most (over 60%) of the old logs remain the same in new major releases (see Tables 4.7, 4.6 and 4.5). The logs are more stable across minor releases (on average, over 80% remain the same). This is good news for developers of *LPAs*. However, from

Tables 4.7, 4.6 and 4.5, we observe that, on average, around 1% (*EA*), 7.5% (*Hadoop*) and 6.3% (*PostgreSQL*) of the logs changes are log modifications. Such logs may be troublesome for maintainers of *LPAs* since they may need to modify their code to account for such changes.

It is important to note that all these log changes are for a fixed set of executed features, i.e., although the executed features remain the same, the logs clearly do not. We studied the release information for both releases and read through the change logs to better understand the rationale for large log changes. We find that internal implementation changes often have a big impact on the logs. For example, according to Table 4.6, release $0.18.0$ (in bold font) is one of the releases with the highest percentage of log changes. Release $0.18.0$ introduced new Java classes for *Hadoop* jobs (a core functionality of *Hadoop*) to replace the old classes. Release $0.21.0$ officially replaced the old MapReduce implementation named "mapred", with a new implementation named "MapReduce". Table 4.6 shows that release $0.18.0$ and $0.21.0$ have the largest amounts of code churn, which shows evidence that both releases have significant changes in the source code. Similarly, the 1.3 and 2.0 releases (bold in Table 4.5) of *EA* have significant behavioural and architectural changes compared with their previous releases. Similarly, although *PostgreSQL* has a much longer history and mature architecture and design than *Hadoop*, we still observe added, deleted and modified logs across releases. For example, release 8.3 has the largest amount (48.6%) of added logs, which corresponds to the new "autovacuum" feature. It appears that the subject systems communicate a significant amount of implementation-level information, leading their logs to vary considerably due to internal changes.

Code level:

We observed different percentages of unchanged, added, deleted and modified logging statements at the code level compared to the percentages at the execution level. For example, in *Hadoop*, release $0.15.0$, $0.16.0$ and $0.19.0$ have a large number of added logging statements at the code level, but the added logs at the execution level is low. From the

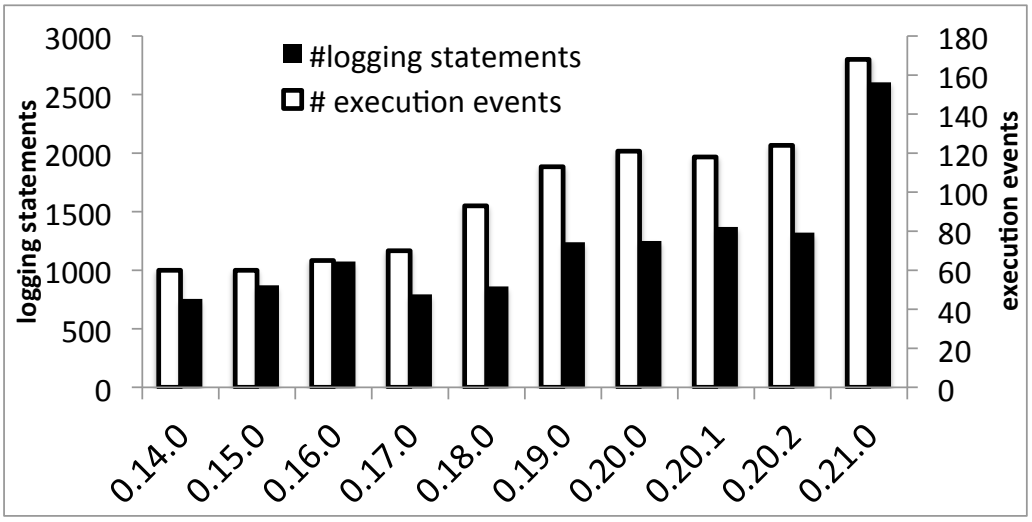


Figure 4.2: Growth trend of logs (execution level and code level) in *Hadoop*.

release notes and development history, we find out that new components with extensive logging statements had been added in the source code of these releases, but these components are not deployed in the release. For example, *HBase* is added into *Hadoop* in release $0.15.0$ but it is not, by default, deployed with *Hadoop*. In release $0.19.0$ of *Hadoop*, a collection of sub-projects is added into the "contrib" folder of *Hadoop*, including *Chukwa* [RK10] and *Hadoop Streaming* [hadc]. Neither of these sub-projects is deployed with *Hadoop* by default. However, we cannot observe the same in *PostgreSQL*. Release 8.3 of *PostgreSQL* has the largest number added logs at both the execution level and the code level. According to the release notes [pos], the large addition of logs is due to the addition of a new feature to support "multiple concurrent autovacuum processes".

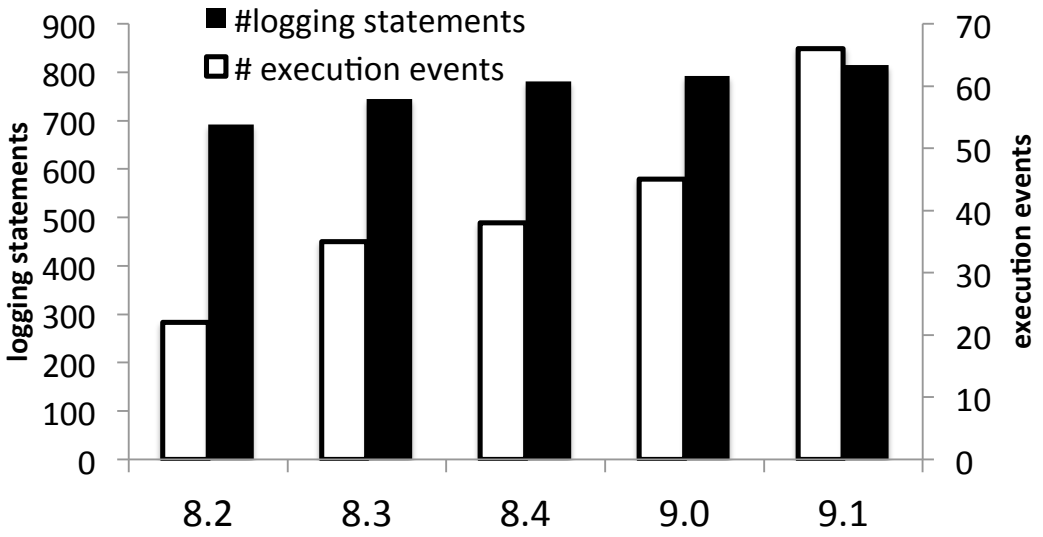


Figure 4.3: Growth trend of logs (execution level and code level) in *PostgreSQL*.

Table 4.5: Percentage of unchanged, added, deleted and modified logs in the history of *EA* (bold font indicates large changes).

	Unchanged	Added	Modified	Deleted
1.1	92.1%	7.9%	0.0%	2.8%
1.2	**64.8%**	**32.9%**	2.3%	**34.7%**
1.3	77.8%	19.7%	2.5%	10.3%
1.4	86.0%	13.4%	0.7%	19.5%
1.5	85.7%	13.5%	0.8%	15.4%
1.6	96.5%	3.2%	0.3%	3.2%
2.0	**61.2%**	**38.0%**	0.7%	**20.8%**
2.1	74.2%	25.6%	0.2%	14.2%

The logs of the studied systems tend to contain implementation-level information that provides LPAs *with detailed knowledge of the internals of the systems. However, such logging style makes* LPAs *fragile, in particular for major releases (e.g., in 0.21.0 of* Hadoop*, only 38% of logs remained unchanged). For minor releases, we still see a large percentage of logs added or modified (e.g., only 64.8% logs remained unchanged in* EA *1.2). The logs at the execution level cover less than 10% of logging statements at the code level, and the logging statements at the code level may change differently compared to the logs at the execution level. Developers may consider explicitly documenting transferring knowledge about logging statements to users of logs.*

Table 4.6: Percentage of unchanged, added, modified and deleted logs (in execution level and code level) in the history of *Hadoop* (bold font indicates large changes).

	Execution level					Code level				
	Total	Unchanged	Added	Modified	Deleted	Total	Unchanged	Added	Modified	Deleted
0.15.0	60	81.7%	10.0%	8.3%	10.0%	871	69.7%	**30.3%**	5.1%	15.5%
0.16.0	65	80.0%	16.9%	3.1%	10.0%	1074	72.3%	**27.7%**	3.5%	11.2%
0.17.0	70	81.4%	10.0%	8.6%	3.1%	794	87.7%	12.3%	0.4%	**48.3%**
0.18.0	93	38.7%	**39.8%**	**21.5%**	**20.0%**	862	78.8%	21.2%	3.8%	12.8%
0.19.0	113	66.4%	29.2%	4.4%	14.0%	1239	65.7%	**34.3%**	0.7%	6.2%
0.20.0	121	71.9%	24.0%	4.1%	18.6%	1250	80.7%	19.3%	2.1%	25.7%
0.20.1	118	91.5%	5.9%	2.5%	8.3%	1370	89.7%	10.3%	0.6%	1.4%
0.20.2	124	95.2%	4.8%	0.0%	0.0%	1321	99.0%	1.0%	0.0%	5.0%
0.21.0	168	38.7%	**46.4%**	**14.9%**	**27.4%**	2605	43.6%	**56.4%**	0.7%	13.6%

Table 4.7: Percentage of unchanged, added, modified and deleted logs (in execution level and code level) in the history of *PostgreSQL* (bold font indicates large changes).

	Execution level					Code level				
	Total	Unchanged	Added	Modified	Deleted	Total	Unchanged	Added	Modified	Deleted
8.3	35	45.7%	**48.6%**	5.7%	18.2%	745	73.7%	**22.6%**	3.8%	4.0%
8.4	38	73.7%	10.5%	15.8%	2.9%	781	85.7%	12.4%	1.9%	2.0%
9	45	82.2%	15.6%	2.2%	0.0%	792	93.3%	5.1%	1.6%	1.7%
9.1	66	66.7%	**31.8%**	1.5%	0.0%	815	86.6%	11.7%	1.7%	1.8%

4.4.2 RQ2: What types of modifications happen to logs?

Motivation

In RQ1 we found that up to 21.5% (in Table 4.6) of communicated events are modified. These modified logs have a crucial impact on *LPAs* because *LPAs* expect certain context information and are likely to fail when operating on events with modified context. In contrast, newly added logs are not likely to impact already developed *LPAs* because those applications are unaware of the new logs and will simply ignore them. In short, changes to the context of previously communicated events are more likely to introduce bugs and failures in *LPAs*. For example, during the history of *Hadoop*, "task" (an important concept of the platform) was renamed to "attempt", leading to failures of monitoring tools and to confusion within the user community about the communicated context [hadb]. Therefore, we wish to understand how communicated contexts change.

Approach

We follow a grounded theory [BJ08] approach to identify modification patterns to the context of a logging statement. We manually study all events at the execution level with a modified context and all the modified logging statements at the code level. We analyze what information is modified and how that information is modified. We repeat this process several times until a number of modification types emerge. We then calculate the distribution of different types of modifications. The percentage of each type of modification is calculated as the ratio of the number of occurrences of a type across all the releases over the total number of modifications across all the releases. For example, the percentage of modified logs of type *p* ($P_{\ modified\ p}$) is calculated as:

$$P_{\ modified\ p} = \frac{\#\ modified\ events_{\ p}}{\#\ total\ modified\ events} \tag{4.2}$$

Table 4.8: Log modification types and examples of the execution level analysis.

Pattern	Definitions	Examples	
		Before	After
Adding context (Recoverable)	Additional context is added into the logs.	ShuffleRamManager memory limit *n* MaxSingleShuffleLimit m	ShuffleRamManager memory limit *n* MaxSingleShuffleLimit *m* mergeThreshold *Q*
Deleting context (Unavoidable)	Context is removed from the logs.	Got *n* map output known output *m*	Got *n* output
Redundant context (Avoidable)	Some redundant information is added or the added information can be inferred without being included in the context.	task is in COMMIT_PENDING	task is in commit_pending, status:COMMIT_PENDING
Rephrasing (Avoidable)	The logs are replaced (partially) by new logs.	Hadoop mapred Reduce task fetch *n* bytes	Hadoop MapReduce task Reduce fetch *n* bytes
Merging (Unavoidable)	Several old logs are merged into one.	MapTask record buffer MapTask data buffer	MapTask buffer
Splitting (Unavoidable)	The old log is split into multiple new ones.	Adding task to tasktracker	Adding Map Task to tasktracker; Adding Reduce Task to tasktracker

Table 4.9: Logging statement modification types discovered from code-level analysis (in addition to the types in Table 4.8).

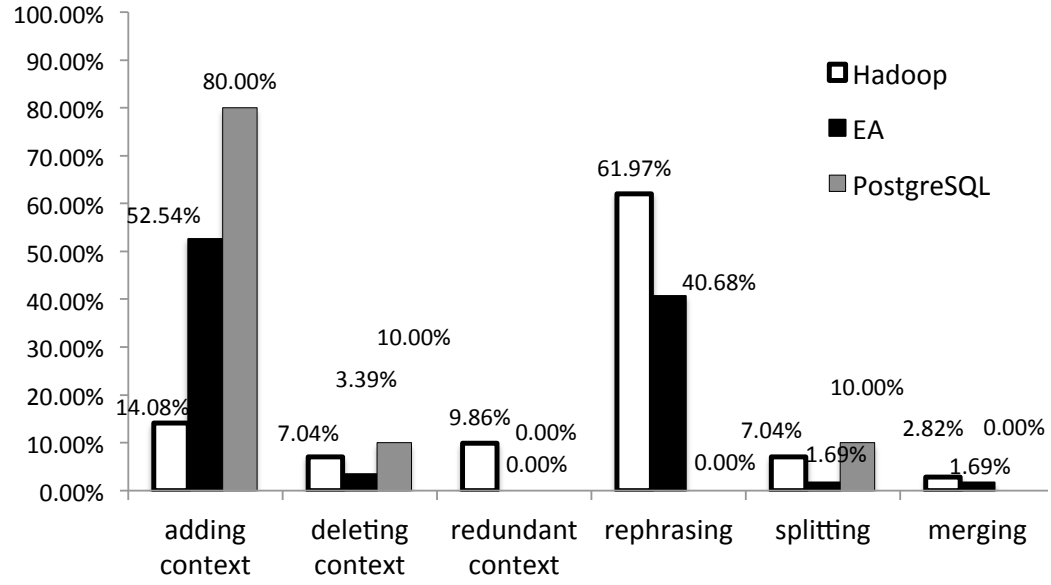
Pattern	Definitions	Examples	
		Before	After
Changing logging level (Recoverable)	The logging level is changed.	$Log.info(\text{"}property\text{'"} + key + \text{"}'is\text{"} + val);$	$Log.debug(\text{"}property\text{'"} + key + \text{"}'is\text{"} + val);$
Changing arguments (Recoverable)	Arguments in the logging statements are changed.	$Log.info(\text{"}$ created trash checkpoint: "+checkpoint);	$Log.info(\text{"}$ created trash checkpoint: "+ checkpoint.touri(). getpath());

Table 4.10: Percentage of avoidable, recoverable and unavoidable log modifications in *Hadoop*, *PostgreSQL* and *EA*.

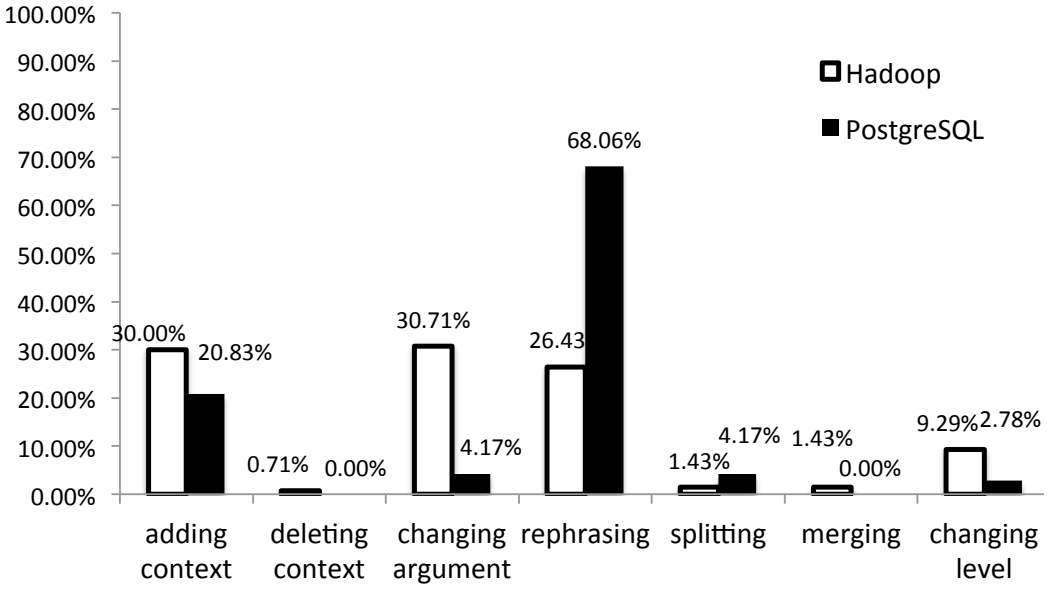
	Execution level			Code level	
	Hadoop	**EA**	**PostgreSQL**	**Hadoop**	**PostgreSQL**
Avoidable	71.83%	40.68%	10.00%	70.00%	68.06%
Recoverable	14.08%	52.54%	80.00%	26.43%	28.78%
Unavoidable	14.09%	6.78%	10.00%	3.57%	4.17%

Table 4.11: Percentages of different types of context modifications in *Hadoop* (execution level).

Release	**Adding context**	**Deleting context**	**Redundant context**	**Rephrasing**	**Merging**	**Splitting**
0.15.0	1.41	0.00	2.82	2.82	0.00	0.00
0.16.0	0.00	0.00	2.82	0.00	0.00	0.00
0.17.0	0.00	0.00	0.00	8.45	0.00	0.00
0.18.0	0.00	0.00	0.00	**28.17**	0.00	0.00
0.19.0	0.00	2.82	0.00	4.23	0.00	0.00
0.20.0	2.82	1.41	1.41	0.00	1.41	0.00
0.20.1	0.00	1.41	0.00	1.41	1.41	0.00
0.20.2	0.00	0.00	0.00	0.00	0.00	0.00
0.21.0	9.86	1.41	1.41	**16.9**	2.82	2.82



(a) Execution level



(b) Code level

Figure 4.4: Distributions of the different types of log modifications across all studied releases.

Table 4.12: Detailed percentages of different types of logging statement modifications in *Hadoop* (code level).

Release	Adding context	Deleting context	Changing argument	Rephrasing	Splitting	Merging	Changing logging level
0.15.0	8.57	0.71	5.71	2.86	0.71	0.00	1.43
0.16.0	9.29	0.00	10.00	3.57	0.00	0.00	5.71
0.17.0	0.00	0.00	1.43	0.00	0.00	0.00	0.71
0.18.0	1.43	0.00	2.86	14.29	0.00	0.00	0.00
0.19.0	2.14	0.00	0.00	2.14	0.00	0.00	0.00
0.20.0	5.00	0.00	7.14	2.14	0.00	0.71	0.00
0.20.1	1.43	0.00	0.71	0.71	0.71	0.71	0.71
0.20.2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.21.0	2.14	0.00	2.86	0.71	0.00	0.00	0.71

Table 4.13: Detailed percentages of different types of logging statement modifications in *PostgreSQL* (code level).

Release	Adding context	Deleting context	Changing argument	Rephrasing	Splitting	Merging	Changing logging level
8.3	8.33	0.00	0.00	**30.56**	0.00	0.00	0.00
8.4	5.56	0.00	0.00	11.11	1.39	0.00	0.00
9.0	5.56	0.00	4.17	9.72	1.39	0.00	2.78
9.1	1.39	0.00	0.00	16.67	1.39	0.00	0.00

Table 4.14: Detailed percentages of different types of context modifications in *PostgreSQL* (execution level).

Release	Adding context	Deleting context	Redundant context	Rephrasing	Merging	Splitting
8.3	10.00	10.00	0.00	0.00	0.00	0.00
8.4	60.00	0.00	0.00	0.00	0.00	0.00
9.0	10.00	0.00	0.00	0.00	0.00	0.00
9.1	0.00	0.00	0.00	10.00	0.00	0.00

Table 4.15: Detailed percentages of different types of context modifications in *EA*.

Release	Adding context	Deleting context	Redundant context	Rephrasing	Merging	Splitting
1.1	0.00	0.00	0.00	0.00	0.00	0.00
1.2	6.78	1.69	0.00	**20.34**	0.00	0.00
1.3	**22.03**	0.00	0.00	10.17	0.00	1.69
1.4	6.78	0.00	0.00	1.69	0.00	0.00
1.5	8.47	1.69	0.00	0.00	0.00	0.00
1.6	1.69	0.00	0.00	1.69	0.00	0.00
2.0	6.78	0.00	0.00	3.39	1.69	0.00
2.1	0.00	0.00	0.00	3.39	0.00	0.00

Results: Log modification types

Table 4.8 tabulates the six types of log modifications identified through our manual examination on the logs at the execution level. The table defines each type and gives a real-world example of it from the studied data. Among all the types, *Rephrasing* and *Redundant context* are avoidable modifications, because neither of them brings any additional information to the logs, and only cause changes in *LPAs*. The *Adding context* modification is typically unavoidable, but a robust log parser should still be able to parse the logs correctly. For example, *Island Grammars* [Moo01] can be leveraged in this case to ignore the added information in logs during the parsing of the log lines. Therefore, *Adding context* is a recoverable modification and has a less negative impact than the avoidable modifications. The other 3 types of modification, i.e., *Merging*, *Splitting* and *Deleting context*, are unavoidable, but the *LPAs* still need to adapt to these modifications. Developers can use a *null* value for the deleted context to make the logs consistent. However, such deleted context may correspond

to removed features and such preferentially deleted context with *null* values may cause the logs to be hard to maintain in the long term. Developers of the system should provide detailed documentation of the unavoidable modifications and inform people who make use of the modified logs. We note that although some *Splitting* modifications look similar to *Adding context*, the two types of modifications are essentially different. *Splitting* is dividing one log event into multiple ones, e.g., splitting the recording of buffer size to different types of buffers, such as data buffer and task buffer; while *Adding context* is providing extra information to the logs, e.g., in addition to recording the buffer size, the free space of the buffer is also recorded.

At the code level, we identify two additional types of logging statement modifications shown in Table 4.9. Both *Changing logging level* and *Changing arguments* are recoverable modifications, since a robust log parser that analyzes the generated logs from these logging statements would likely not be impacted by such logging statement modifications.

Results: Log modifications distribution

Overall:

Figure 4.4 shows the classification distribution of the log modification types at both the execution and the code level across all the studied releases and Table 4.10 shows the percentage of avoidable, recoverable and unavoidable log modifications. We find that the majority of the log modifications are either avoidable or recoverable. Only a small portion of the log modifications are unavoidable. Simply put, developers can improve the maintenance of the *LPAs* by avoiding the avoidable modifications and documenting the unavoidable log modifications.

Execution level:

Tables 4.11, 4.14 and 4.15 show the percentages of context modifications for *Hadoop*, *PostgreSQL* and *EA* at the execution level, broken down per release and pattern. Table 4.11 shows that the two largest numbers (in bold) of context modifications are both instances of

Rephrasing context. They were introduced in release 0.18.0 and 0.21.0 of *Hadoop*. Table 4.15 shows that many *Rephrasing context* instances are introduced in version 0.1.2 of *EA*. As noted in RQ1, all these three releases (0.18.0 and 0.21.0 of *Hadoop* and 1.2 of *EA*) have significant changes to the systems. These results indicate that most of the *Rephrasing context* modifications may have a high correlation to the major changes introduced into the software systems. For example, in release 0.21.0, the old MapReduce library, which is the most essential part of *Hadoop*, was replaced by a whole new implementation. Therefore, the word "mapred" was replaced by the word "MapReduce". As both implementations have the same features, the operator should not need to worry about such implementation changes of the library. However, such *Rephrasing* modifications require updating impacted *LPAs* to ensure their proper operation.

In contrast, even though release 1.3 of the *EA* has many *Adding context* modifications, it does not have a large number of added or deleted logs. This indicates that even though some releases do not introduce major changes into the system, logs may still be modified significantly. A release without major system changes may also impact the *LPAs* significantly.

In *PostgreSQL*, only *Adding context*, *Deleting context* and *Rephrasing context* are observed in the studied releases. 80% of the context modification to the logs of *PostgreSQL* at the execution level is recoverable (*Adding context*). Developers of the *LPAs* of *PostgreSQL* may spend more effort on creating robust log parsers.

We observe that in *Hadoop*, most of the log modifications belong to the *Rephrasing* type while most of the log modifications in *PostgreSQL* are of the *Adding context* type. We believe the reason is that *Hadoop*, as a new project, has more structural changes, which may lead to the log rephrasing. On the other hand, as a project with long history, *PostgreSQL* mainly has features added to it, which may contribute to most of the *Adding context* modifications.

Code level:

Tables 4.12 and 4.13 show the percentage of context modifications for *Hadoop* and *PostgreSQL* at the code level, broken down per release and pattern. For *Hadoop*, the results

of some releases are similar at the code level and the execution level, while the results of some other releases are different at the code level and the execution level. For example, release $0.18.0$ has a large number of *Rephrasing context* modifications at both the code level and the execution level, while release $0.21.0$ of *Hadoop* has large number of *Rephrasing context* modification at the execution level but the number is low at the code level. The reason for high number of *Rephrasing context* modifications at the execution level but low at the code level is because the re-implementation of the MapReduce library of *Hadoop* without removal of the old implementation from the source code. Therefore, at the code level, the new implementation is considered an "added" logging statement, while at the execution level, the new implementation is considered *Rephrasing*, because it replaces the old implementation during the execution.

In *PostgreSQL*, more *Rephrasing context* modifications are observed at the code level. For example, the large percentage of *Rephrasing* at release 8.3 corresponds to rephrasing "can't" to "cannot" in all logging statements. These modifications to the logging statements are not observed in typical execution, thus they are likely to be neglected by developers of *LPAs*. If these logs appear during the execution of the system, the *LPAs* may not be able to process them correctly. Developers of the software may consider spending more effort on tracking these *Rephrasing* modifications using code analysis and informing the users of such logs about any modifications.

*We have identified eight types of log modifications. Two of the types (*Rephrasing *and* Redundant context*) are avoidable, three types (*Adding context, Changing logging level *and* Changing arguments*) are unavoidable but their impact can be controlled through the use of robust parsing techniques. The other three types (*Merging, Splitting *and* Deleting context*) are unavoidable and have a high chance introducing errors. Around 90% of the modifications can be controlled through careful attention by system developers (avoidable context modifications) or careful robust programming of* LPAs *(*Adding Context*).*

4.4.3 RQ3: What information is conveyed in short-lived logs?

Motivation

RQ1 shows that logs are added and deleted in every release. Some logs are added by developers and removed in a short period of time. The *LPAs* depending on such short-lived logs may be extremely fragile. We study the information conveyed in the short-lived logs to understand why such logs exist only within a short period of time. By studying the conveyed information, we can understand the logs at a high level of abstraction instead of considering simple counts of added, removed and modified logs like in the previous two questions.

Approach

We consider the logs that only exist in a single release to be short-lived logs. To understand the purpose of short-lived logs, we extract the logging level using the code analysis of the logging statements to measure the logging levels of short-lived logs. For each logging level, we calculate the percentage of short-lived logging statements.

To further understand the information conveyed by short-lived logs over time, we generate a Latent Dirichlet Allocation (LDA) [BNJ03] model of the topics in short-lived logs.

Each topic in the model is a list of words that has high probability of appearing together in short-lived logs. We put the short-lived logs of each release in a separate file as input documents for LDA. We use *MALLET* [MAL] to generate LDA models with five topics. Each word in the topics has a probability indicating its significance in the corresponding topic. We generated the five words with the highest probability in each topic to determine the information conveyed by logs in the topic. Finally, we examine the words in the five topics and generate a one-sentence summary based on our knowledge about the systems to summarize the information conveyed in short-lived logs.

To compare the different characteristic between short-lived and long-lived logs, we perform the same experiments on long-lived logs. We consider the logs that exist in all studied releases as long-lived.

Results

Logging level:

The results of the code level analysis in Table 4.16 shows that most of the short-lived logging statements in *Hadoop* are at the *info* and *debug* level. Almost 70% of the logging statements at *trace* level and 22% to 25% of the logging statements in *debug* and *error* level only appear in one release, while none of the logging statements at *trace* level exist across all the studied releases.

In *PostgreSQL*, over 75% of the short-lived logging statements are at the *error* level. The logging statements at *error* and *fatal* level account for more than 90% of all the short-lived logging statements. The percentage of short-lived *fatal* level logging statement is much higher than long-lived logging statements. In contrast to *Hadoop*, *PostgreSQL* has more *error* and *fatal* level logging than *info* level logging. Therefore, all *info* logging statements in *PostgreSQL* are short-lived and they only account for 1.43% of the total logging statements.

Topics:

A manual analysis of short-lived logs at the execution level reveals that a small part of

Table 4.16: Logging levels of short-lived and long-lived logs.

Short-lived logs

	Hadoop		PostgreSQL	
	% over total short-lived logs	% short-lived logs in the logging level	% over total short-lived logs	% short-lived logs in the logging level
trace	1.01%	69.70%	-	-
debug	27.36%	25.20%	3.93%	5.21%
info	47.62%	14.70%	1.43%	100.00%
warn	13.96%	14.31%	0.00%	0.00%
error	8.99%	22.17%	76.07%	7.55%
fatal	1.06%	10.86%	16.43%	17.10%
notice	-	-	0.36%	9.09%
log	-	-	1.79%	4.07%

Long-lived logs

	Hadoop		PostgreSQL	
	% over total long-lived logs	% long-lived logs in the logging level	% over total long-lived logs	% long-lived logs in the logging level
trace	0.00%	0.00%	-	-
debug	18.43%	2.48%	5.56%	5.21%
info	54.98%	2.47%	0.00%	0.00%
warn	16.01%	2.39%	0.00%	
error	8.16%	2.93%	82.83%	5.81%
fatal	2.42%	3.62%	3.54%	2.60%
notice	-	-	1.01%	9.09%
log	-	-	7.07%	4.07%

logs corresponds to exceptions and stack traces. We removed such data since it does not represent short-lived logs but primarily rare errors.

Table 4.17 and 4.18 show the topics generated by LDA for *Hadoop*. The words in each topic are sorted by their degree of membership. From the results in Table 4.17, as expected, we find that both short-lived and long-lived topics contain high-level conceptional information such as “job”. However, we also find that the topics in short-lived logs may contain lower-level information, such as the implementation of the system. For example, the word “ipc” in topic #4 means inter-procedural communication between machines. Since the topic is about reading a remote file, the word “ipc” corresponds to an implementation detail of how to read the file. In addition, the information about outputting results and choosing a server in topics #1, #2 and #5 also contain implementation-level information.

Table 4.17: LDA topics of logs in *Hadoop* at the execution level.

Short-lived logs

#	Topic	Summary
1	job output node jobhistory saved	Hadoop saves output to a machine.
2	reducetask jobinprogress choosing server hadoop	Hadoop assigns a reduce task to a machine.
3	mapred map tracker taskinprogress jobtracker	Map task updates its progress.
4	id org local file ipc	Hadoop reads from a local file.
5	task tasktracker attempt outputs tip	Hadoop Attempt saves its output and reports to the task tracker.

Long-lived logs

#	Topic	Summary
1	attempt starting successfully received	Hadoop starts an attempt successfully.
2	sessionid jvm fetcher processname	The jvm metric starts on a process with session id.
3	job initializing jvmmetrics jobtracker	Hadoop initializes a job with jvm metrics recorded.
4	id node tracker tasks	A task with id is on a node.
5	task reduce map completed	Hadoop map or reduce task completed.

Table 4.18: LDA topics of logs in *Hadoop* at the code level.

Short-lived logs

#	Topic	Summary
1	block dir stringifyexception start	Hadoop throws exception when it reads a directory from a data block.
2	job integer finish maps	Hadoop job finishes with a number of Map tasks.
3	tostring getmessage path created	A new path is created in the distributed file system.
4	region tostring regionname regioninfo	HBase region server information is printed.
5	error file closing size	Hadoop fails to close a file.

Long-lived logs

#	Topic	Summary
1	filter tracker defined trackername	Filter a tracker.
2	dst frequency countrecords testpipes	Count records from a server.
3	records addr rename	Rename a record on a node.
4	src patter unknown server	Progress from unknown server.
5	count skipping stringifyexception	Skip records.

At the code level, two of the topics (topic #1 and #5) are about system exceptions or errors. We browsed the short-lived logs of *Hadoop* and found over 15% of them contain low-level details. For the topics in long-lived logs, we find that most of the topics correspond to system events that do not often happen, such as filtering a node and skipping a record. The fact that these features are not in the hot spot of the system might be the reason that these

Table 4.19: Topics of the short-lived logs in *PostgreSQL* at the code level generated by LDA.

#	**Short-lived logs**	#	**Long-lived logs**
1	spi type process tuple	*1*	type block invalid list
2	failed arguments pipe gin	*2*	relationgetrelationname rel fired page
3	cache lookup invalid extract	*3*	failed spi number trigger
4	file number create check	*4*	index relname owner add
5	failed relation join ttdummy	*5*	lookup relation returned manager

logs are not changed during the development of the system. We performed the same study on *EA* at the execution level, with the results similar to the results of *Hadoop*. For example, both long and short-lived logs contains high-level domain knowledge but three topics in short-lived logs contain error messages or implementation details.

We observe that the topics in short-lived logs at the execution level differ to the topics at the code level. The reason is that only a small part of the logs at the code level is executed at run-time. In addition, Table 4.16 shows that almost 70% of the short-lived logging statements of *Hadoop* at the code level are at *trace* level. These logs would not be generated during execution with default configuration.

We do not find short-lived logs of *PostgreSQL* at the execution level. Table 4.19 shows the generated topics by LDA for *PostgreSQL*. Since we do not have experience with the development of *PostgreSQL*, we only show the generated topics without a summary. Compared to the results of *Hadoop*, we can observe error messages, such as “fail” and “invalid”, in both short-lived and long-lived logs of *PostgreSQL* at the code level. Such observation confirms the results of logging level (shown in Table 4.16) that most of the logging statements (both short-lived and long-lived) in *PostgreSQL* are at *error* level.

Some *LPAs* are designed for recovering high-level information about the system, e.g., system workload rather than implementation details. Such *LPAs* would not need the implementation-level information and hence would not be impacted by changes to this kind of logs. However, there are a few *LPAs* that are designed for debugging purposes. Such applications require the implementation-level information and error messages in the short-lived logs,

and would be fragile as their corresponding logs are continuously changing.

> *Short-lived logs contain implementation-level details and error messages to facilitate system development and testing.* LPAs *analyzing implementation-level information and error messages are likely to be more fragile. More maintenance effort is needed for such* LPAs.

4.5 Threats to Validity

This section presents the threats to validity of our study.

4.5.1 External validity

Our study is an exploratory study performed on *Hadoop*, *PostgreSQL* and an enterprise application, *EA*. Even though all the subject systems have years of history and large user bases, more case studies on other software systems in the same domain are needed to see whether our findings can generalize. Similarly, the studied logs are collected from specific workloads, which may not generalize. Future studies should examine in-field execution logs.

4.5.2 Internal validity

Our study includes several manual steps, such as the analysis and classification of log modifications. Our findings may contain subjective biases in such manual steps.

Our study is performed on both major and minor releases of *Hadoop* and *EA*. However, the major and minor releases in the two systems may not contain similar numbers of source code changes. We study *Hadoop* primarily using major releases while we study *EA* primarily using minor releases. The major releases of *Hadoop* may not contain as many significant changes and the minor releases of *EA* may contain large numbers of changes. Therefore,

our findings about major and minor releases may be biased. Studies of more releases of the same systems and more systems would help to counter this bias.

4.5.3 Construct validity

Our execution-level study is mainly based on the abstraction of execution events proposed by Jiang *et al.* [JHHF08a]. This approach, customized to better fit the two subject systems, is shown to have a high precision and recall. However, falsely abstracted log events may still exist, which may potentially bias our results. Other log abstraction techniques might improve the precision and reduce the incorrectly abstracted execution events in our study.

Our code-level study leverages J-REX. The correctness of our code-level study depends on the correctness of J-REX. J-REX has been used in previous research showing good performance and accuracy [SJI+10; SBS+10]. Due to the lack of mature techniques to track the genealogy of logs, our approach cannot identify log modifications automatically. We examined the added and deleted logs and identified the modified logs based on our experience using logs. Such results can be treated by the accuracy of our subjective decision on the modified logs. More mature tracking techniques for log genealogy are needed.

4.6 Related Work

In this section, we give a brief overview of the prior work related to our study.

4.6.1 Non-code based evolution studies

While many prior studies examined the evolution of source code, (e.g., [GJKT97; GT00; LRW+97]), this chapter studies the evolution of software systems from the perspective of non-code artifacts associated with these systems. The non-code artifacts are extensively used in software engineering practice, yet the dependency between such artifacts and their

surrounding ecosystem lacks explicit study. Therefore, understanding the evolution of non-code based software artifacts is important. For example, the evolution of the following non-code artifacts has been studied before:

- **System Documentation:** Software systems evolve throughout their lifetime, as new features are added and existing features are modified due to bug fixes, performance and usability enhancements. Antón *et al.* [AP01] study the evolution of telephony software systems by studying the user documentation of telephony features in the phone books of Atlanta.

- **User Interface:** His *et al.* [HP00] study the evolution of Microsoft Word by looking at changes to its menu structure. Hou *et al.* [HW09] study the evolution of UI features in the Eclipse IDE.

- **Features:** Instead of studying the code directly, some studies have picked specific features and followed their implementation throughout the lifetime of the software system. For example, Kothari *et al.* [KBMS08] propose a technique to evaluate the efficiency of software feature development by studying the evolution of call graphs generated during the execution of these features. Our study is similar to this work, except for using logs instead of call graphs. Greevy *et al.* [GDG06] use program slicing to study the evolution of features.

- **Code Comments:** Comments are a valuable instrument to preserve design decisions and to communicate the intent of the code to programmers and maintainers. Jiang *et al.* [JH06] study the evolution of source code comments and discover that the percentage of functions with header and non-header comments remains consistent throughout the evolution. Fluri *et al.* [FWG07; FWGG09] study the evolution of code comments in 8 software projects.

- **Logs:** To the best of our knowledge, this chapter is the first work that studies the

evolution of logs.

4.6.2 Traceability between Logs and Log Processing Apps

Many software developers consider logs as a final output of their systems. However, for many such systems logs are just the input for a whole range of applications that live in the log-processing ecosystem surrounding these systems.

Our study is the first study to explore how changes in parts of an ecosystem (communicated information, i.e., logs), once released in the field, might impact other parts of the system (*LPAs*). The need for such types of studies was noted by Godfrey and German [GG08], as they recognized that most software systems today are linked with other systems within their ecosystems. For example, in regression tests, the test suites need to be maintained as the functionality changes. Test suites tend to accrue beyond their usefulness because developers are reluctant to remove any tests that some other developers might be depending on.

Lehman's earlier work [LRW+97] recognizes the need for applications to adapt to the changes in their surrounding environment. In this study, we primarily focused on the environmental changes of *LPAs* (i.e., changes to logs). To prove the concept that *LPAs* evolve due to the evolution of logs, we manually examined the items in the issue tracking system (JIRA) of *Chukwa*, a log collector for *Hadoop*. We found 6 items that are caused by the updating of *Hadoop* logs. For example, one of the issues (CHUKWA-132 [3]) corresponds to the failure of log parser when *Hadoop* starts to output logs across multiple lines. Another example is issue CHUKWA-375 [4], which is to update log parsers because of the log changes in *Hadoop*. The issues spread out across releases during the development history of *Chukwa*. Future work, should study the changes in all aspects of the ecosystem, namely the system, the logs, and the *LPAs* that process the logs.

[3] https://issues.apache.org/jira/browse/CHUKWA-132 last verified January 2014.
[4] https://issues.apache.org/jira/browse/CHUKWA-375 last verified January 2014.

Our study and our industrial experience support us in advocating the need for research on tools and techniques to establish and maintain traceability between the logs and the *LPAs*. In addition, systematic techniques of leveraging logs are needed in software engineering activities. However, reducing the maintenance overhead and costs for all apps within the ecosystem of large software systems is essential.

4.7 Chapter Summary

Logs are generated by snippets of code inserted explicitly by domain experts to record valuable information. An ecosystem of *LPAs* analyzes such valuable information to assist in software testing, system monitoring and program comprehension. Yet, these *LPAs* highly depend on logs and are hence impacted by changes to logs. In the previous chapter (Chapter 3), we studied one aspect of the current practices of leveraging logs, i.e., understanding logs. In this chapter, we studied another aspect of the current practices of leveraging logs, i.e., the evolution of logs. We performed an exploratory study on the logs of ten releases of an open source software named *Hadoop*, five releases of another open source software system named *PostgreSQL* and nine releases of a legacy enterprise application.

Our study shows that systems communicate more about their execution through logs as they evolve. During the evolution of software systems, the logs also evolve. Logs change significantly when there are major source code changes (e.g., a new major release), although implementation changes ideally should not have an impact on logs. In addition, we observed eight types of log modifications. Among the log modifications in the studied systems, less than 15% of the modifications are unavoidable and are likely to introduce errors into *LPAs*. We also find that short-lived logs typically contain system implementation-level details and system error messages.

Our results indicate that additional maintenance resources should be allocated to maintain *LPAs*, especially when major changes are introduced into the software systems. Because

of the evolution of logs, traceability techniques are needed to establish and track the dependencies between logs and the *LPAs*.

However, even today, without traceability techniques between logs and *LPAs*, the negative impact can still be minimized by both the system developers (who generate logs) and the developers of *LPAs* (who consume logs). System developers should avoid modifying logs as much as possible. The avoidable log modifications include rephrasing and adding redundant information in the logs. On the other hand, *LPA* developers should write robust log parsers to reduce the negative impact of log changes. In addition, more resources should be allocated to maintain *LPAs* designed for debugging problems (from short-lived logs).

We also find that the observed logs during typical system execution cover less than 10% of the logging statements. The logging statements may evolve differently to the logs at the execution level. The developers of *LPAs* are likely not aware of such difference, leading to unexpected *LPA* bugs and failures.

Due to the differences between log lines at the execution level and the logging statements in the source code, developers may consider providing documentation of the logging statements to the users of the logs, i.e., system operators and developers of the *LPAs*, to support better usage of logs and to avoid potential problems in the *LPAs*.

Chapter 3 and 4 show that logs are a valuable and widely used source of data for software development. Yet, the use of logs in software development are ad hoc and there exists limited support for systematic approaches to leverage logs. In the next two chapters (Chapter 5 and 6), we propose systematic log mining approaches that leverage logs to support software development activities.

Part III

Log Engineering Approaches to Support Software Development Activities

In Chapter 2, we find that current practices of software log mining are often ad hoc and do not scale well. Chapter 3 and 4 show that logs are a valuable and widely used source of data for software development. Yet, the use of logs in software development are ad hoc and there exists limited support for systematic approaches to leverage logs. In this part of the thesis (Chapter 5 and 6), we propose systematic log mining approaches that leverage logs to support software development activities.

A platform software typically acts as a container of applications running on top of it. Platform logs can be leveraged for all the different applications running on top of the platform. *Hadoop* is an example of such software platforms, which support applications running in a distributed environment [Whi09]. *Big Data Analytics Applications (BDA Apps)* are a new category of software applications that analyze large scale data, which is typically too large to fit in memory or even on one hard drive, in order to uncover actionable knowledge using large scale parallel-processing infrastructures [FDCD12]. Such BDA Apps typically run on top of big data analytic platforms, e.g., *Hadoop*. We focus on platform software and BDA Apps in this part of the thesis, since such software produces and depends heavily on logs.

- Chapter 5 proposes an approach to use logging characteristics to assist in prioritizing code review and testing efforts. We empirically study the relationship between logging characteristics and software defects. We define log-related product metrics, such as the number of log lines in a file, and log-related process metrics such as the number of changed log lines. Through a case study on four releases of *Hadoop* and *JBoss*, we find that the correlations between our newly-defined log-related metrics and post-release defects are as strong as their correlations with traditional process metrics, such as the number of pre-release defects, which is known to be strongly correlated with post-release defects. We also find that log-related metrics can complement traditional product and process metrics resulting in up to 40% improvement in

the explanatory power of defect proneness. Our results show that logging characteristics provide strong indicators of defect-prone source code files. However, we note that removing logs is not the answer to better code quality. Instead, our results show that it might be the case that developers often relay their concerns about a piece of code through logs. Hence, code quality improvement efforts (e.g., testing and inspection) should focus more on the source code files with large numbers of logs or with high log churn.

- Chapter 6 proposes a lightweight log mining approach for uncovering differences between pseudo and large scale cloud deployments using the readily-available yet rarely used execution logs from these platforms. Our approach abstracts the execution logs, recovers the execution sequences, and compares the sequences between the pseudo and cloud deployments. Through a case study on three representative Hadoop-based BDA Apps, we show that our approach can rapidly direct the attention of BDA App developers to the major differences between the two deployments. Knowledge of such differences is essential in verifying BDA Apps when analyzing big data in the cloud. Using injected deployment faults, we show that our approach not only significantly reduces the deployment verification effort, but also provides very few false positives when identifying deployment failures.

CHAPTER 5

Prioritizing Code Review and Testing Efforts Using Logs and Their Churn

Platform software plays an important role in the development of large scale applications. Such platforms provide functionality and abstraction on which applications can be rapidly developed and easily deployed. *Hadoop* and *JBoss* are examples of popular open source platform software. Such platform software generate logs to assist operators in monitoring the applications that run on them. These logs capture the doubts, concerns, and needs of developers and operators of platform software. We believe that such logs can be used to better understand code quality. However, logging characteristics and their relation to quality has never been explored. In this chapter, we sought to empirically study this relation through a case study on four releases of *Hadoop* and *JBoss*.

Our findings show that files with logging statements have higher post-release defect densities than those without logging statements in 7 out of 8 studied releases. Inspired by prior studies on code quality, we defined log-related product metrics, such as the number of log lines in a file, and log-related process metrics such as the number of changed log lines. We find that the correlations between our log-related metrics and post-release defects are as strong as their correlations with traditional process metrics, such as the number of pre-release defects, which is known to be strongly correlated with post-release defects. We also find that log-related metrics can complement traditional product and process metrics resulting in an up to 40% improvement in explanatory power of defect proneness.

Our results show that logging characteristics provide strong indicators of defect-prone source code files. However, we note that removing logs is not the answer to better code quality. Instead, our results show that developers often relay their concerns about a piece of code through logs. Hence, code quality improvement efforts (e.g., testing and inspection) should focus more on the source code files with large amounts of logs or with large amounts of log churn.

5.1 Introduction

Large platform software provides an infrastructure for a large number of applications to run on. *Hadoop* and *JBoss* are examples of popular open source platform software. Such software relies heavily on logs to monitor the execution of the applications running on top of them. These logs are generated at run-time by logging statements in the source code. Generating logs during execution plays an essential role in field debugging and support activities. These logs are not only for the convenience of developers and operators, but have already become part of legal requirements. For example, the Sarbanes-Oxley Act of 2002 [soa] stipulates that the execution of telecommunication and financial applications must be logged. Although logs are widely used in practice, and their importance has been well-identified in prior software engineering research [GWS06; YPZ12; SJA+11], logs have not yet been fully leveraged by empirical software engineering researchers to study code quality.

We believe that logs capture developers' concerns and doubts about the code. Developers tend to embed more logging statements to track the run-time behaviour of complex and critical points of code. For example, one developer commented on a bug report (HADOOP-2490[1]) of *Hadoop*, as follows: "*...add some debug output ... so can get more info on why TestScanner2 hangs on cluster startup.*"

Logs also contain rich knowledge from the field. Operators of platform software often need to track information that is relevant from an operational point of view. For example, a user of *Hadoop* submitted a bug report (HADOOP-1034[2]) complaining about the limited amount of logging. In the description of the bug report, the user mentions, "*Only IOException is catched and logged (in warn). Every Throwable should be logged in error*".

To meet the need for run-time information, developers and operators record noteworthy system events, including domain-level and implementation-level events in the logs.

[1]https://issues.apache.org/jira/browse/HADOOP-2490 last verified January 2014.
[2]https://issues.apache.org/jira/browse/HADOOP-1034 last verified January 2014.

In many cases, logs are often used for fixing issues, since logs provide additional diagnostic information. Therefore the inclusion of more logs in a source code file by a developer could be an indicator that this particular piece of source code is more critical or more defect-prone. Hence, there could be a direct link between logging characteristics and code quality. However, except for individual experiences and observations, there are no empirical studies that attempt to understand the relationship between logs and code quality. In Chapter 3 and 4, we studied the current practice of leveraging logs. In this chapter, we propose log mining techniques to assist in prioritizing code review and testing efforts. We seek to study the relationship between the characteristics of logs, such as log density and log churn, and code quality, especially for large platform software. We use post-release defects as a measurement of code quality because it is one of the most important and widely studied aspects of code quality [Shi12]. In order to study this relationship, we perform a case study on four releases of *Hadoop* and four releases of *JBoss*. In particular, we aim to answer the following research questions:

RQ1: Are source code files with logging statements more defect-prone?

We find that source code files with logging statements have higher than average post-release defect densities than those without logging statements in 7 out of 8 studied releases. We also find positive correlations between our log-related metrics and post-release defects. In 7 out of 8 releases, the largest correlations between log-related metrics and post-release defects are at least as large as the correlation between post-release defects and pre-release defects, which prior studies have shown to have the highest correlation to post-release defects. The correlation between the average log churn (number of change log statements in a commit) and post-release defects is the largest among our log-related metrics. Such correlation provides support to our intuition about the tendency of developers to add more logs in the source code files that they feel may be are more defect-prone than others.

RQ2: Can log-related metrics help in explaining post-release defects?

We find that the proposed log-related metrics provide up to a 40% improvement over traditional product and process metrics in explaining post-release defects (i.e., the explanatory power).

This chapter is the first work to establish an empirical link between logs and defects. We observe positive correlations between logging characteristics and post-release defects in all studied releases. Therefore, practitioners should allocate more effort to source code files with more logs or log churn.

However, such positive correlations do not imply that logs are harmful or that they should be removed. For instance, prior research has shown that files with high churn are more defect prone [NBZ06; NB07]. Such studies do not imply that one should not change such files. Instead, this study along with prior studies provides indicators to flag high-risk files that should be carefully examined (tested and/or reviewed) prior to release in order to avoid post-release defects.

The rest of this chapter is organized as follows: Section 5.2 presents a qualitative study to motivate this chapter. Section 5.3 presents the background and related research for this chapter. Section 5.4 presents our new log-related metrics. Section 5.5 presents the design and data preparation steps for our case study. Section 5.6 presents the results of our case study and details the answers to our research questions. Section 5.7 discusses the threats to validity of our study. Finally, Section 5.8 concludes the chapter.

5.2 Motivating Study

In order to better understand how developers make use of logs, we perform a qualitative study. We first collect all commits that have logging statement changes in *Hadoop* release $0.16.0$ to release $0.19.0$ and *JBoss* release 3.0 to release 4.2. We then select a 5% random sample (280 commits for *Hadoop* and 420 commits for *JBoss*) from all of the collected commits with logging statement changes. Once we extract the commit messages from the sample

commits, we follow an iterative process similar to the one from Seaman *et al.* [SSR+08] to identify the reasons that developers change the logging statements in the source code, until we could not find any new reasons. We identify four reasons using this process and their distributions are reported in Table 5.1. These four reasons are described below:

- **Field debugging:** Developers often use logs to diagnose run-time or field defects. For example, the commit message of revision *954705* of *Hadoop* says: "*Region Server should never abort without an informative log message*". Examining the source code, we observe that the Region Server would abort without any logs. In this revision, the developer added logging statements to output the reason for aborting. Developers also change logging statements when they need logs to diagnose pre-release defects. For example, the commit message of revision *636972* of *Hadoop* says: "*Improve the log message; last night I saw an instance of this message: i.e. we asked to sleep 3 seconds but we slept <30 seconds*".

- **Change of a feature:** Developers add and change logs when they change features. For example, in revision *697068* of *Hadoop*, developer added a new "KILLED" status for the job status of *Hadoop* jobs and adapted the logs for the new job status. Changing logs due the change of a feature is the most common reason for log churn.

- **Inaccurate logging level:** Developers sometimes change logging levels because of an inaccurate logging level. For example, developers of *JBoss* changed the logging level at revision *29449* with the commit message "*Resolves (JBAS-1571) Logging of cluster rpc method exceptions at warn level is incorrect.*". The discussion of the issue report "*JBAS-1571*" shows that the developers considered the logged exception as a normal behaviour of the system and the logging level was changed from "warn" to "trace".

- **Logs that are not required:** Developers often think that logs used for debugging are redundant after the defect is fixed and they remove logs after using them for

Table 5.1: Distribution of log churns reasons

	Hadoop	JBoss
Field debugging	32%	16%
Change of feature	59%	75%
Inaccurate logging level	0%	7%
Logs that are not required	9%	2%

debugging. For example, the commit message of revision *612025* of *Hadoop* says: "*Remove chatty debug logging from 2443 patch*".

This motivating study shows that developers change logs for many reasons, such as debugging a feature in the field or when they are confident about a feature. Hence, we believe that there is value in empirically studying the relationship between logging characteristics and code quality.

5.3 Background and Related Work

We now describe prior research that is related to this chapter. We focus on prior work along two dimensions: 1) log analysis and 2) software defect modeling.

5.3.1 Log Analysis

In the research area of computer systems, logs are extensively used to detect system anomalies and performance issues. Xu *et al.* [XHF+09b] created features based on the constant and variable parts of log messages and applied Principal Component Analysis (PCA) to detect abnormal behaviours. Tan *et al.* introduced SALSA, an approach to automatically analyze logs from distributed computing platforms for constructing and detecting anomalies in state-machine views of the execution of a system across machines [TPK+08].

Yuan *et al.* [YZP+11] proposed a tool named *Log Enhancer*, which automatically adds more context to log lines. Beschastnikh *et al.* [BBS+11] designed an automated tool that

infers execution models from logs. The models can be used by developers to verify and diagnose bugs. Jiang *et al.* designed log analysis techniques to assist in identifying functional anomalies and performance degradations during load tests [JHHF08b; JHHF09]. Jiang *et al.* [JHP+09] studied the characteristic of customer problem troubleshooting by using storage system logs. They observed that customer problems with attached logs were resolved sooner than those without logs.

A workshop named "Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques"[3] aims to address the analysis of system logs to assist in managing large software systems.

The existing log analysis research demonstrates the wide use of logs in software development and operation. However, in the aforementioned research, the researchers look at the logs collected at run-time, whereas in our chapter we look at the logging code present in the source code in order to establish an empirical link between logging characteristics and code quality. Therefore, the wide usage of logs in software and the lack of sufficient research motivates our study of the relationship between logging characteristics and code quality in this chapter.

Recent work by Yuan *et al.* [YPZ12] study the logging characteristics in 4 open source systems. They quantify the pervasiveness and the benefit of software logging. They also find that developers modify logs because they often cannot get the correct log message on the first attempt. Our previous research [SJA+11; SJA+13] studies the evolution of logs from both static logging statements and log lines outputted during run-time. We find that logs are often modified by developers without considering the needs of operators. The findings from these previous studies motivates this work. However, previous studies only empirically study the characteristics of logs, but do not establish an empirical link with code quality. This chapter focuses on empirically studying the relationship of such logging characteristics and code quality.

[3] http://sosp2011.gsd.inesc-id.pt/workshops/slaml last verified January 2014.

5.3.2 Software defect modeling

Software engineering researchers have built models to understand the rationale behind software defects. Practitioners use such models to improve their processes and to improve the quality of their software systems. Fenton *et al.* [FN99] provide a critical review of software defect prediction models. They recommend holistic models for software defect prediction, using Bayesian belief networks. Hall *et al.* [HBB+12] recently conduct a systematic review on defect prediction models. They find that the methodology used to build models seems to be influential to predictive performance. Prior research typically builds models using two families of software metrics:

- **Product metrics:** Product metrics are typically calculated from source code.
  - *Traditional product metrics:* Early work by Ohlsson and Alberg [OA96] build defect prediction models using complexity metrics. Nagappan *et al.* [NBZ06] also performed case studies to understand the relationship between source code complexity and software defects. Several studies found that complexity metrics were correlated to software defects although no single set of metrics can explain defects for all projects [NBZ06].
  - *Code dependency metrics:* Prior research investigates the relationship between defects and code dependencies. Zimmermann and Nagappan [ZN08] find code complexity metrics have slightly higher correlation to defects than dependency metrics. However, the dependency metrics perform better than complexity metrics in predicting defects. Nguyen *et al.* [NAH10] replicate the prior study and find that a small subset of dependency metrics have a large impact on post-release failure, while other dependency metrics have a very limited impact.
  - *Topic metrics:* A few recent studies have tried to establish a link between topics and defects. Liu *et al.* [LPF+09] proposed to model class cohesions by latent

topics. They propose a new metric named Maximal Weighted Entropy (MWE) to measure the conceptual aspects of class cohesion. Nguyen *et al.* [NNP11] apply Latent Dirichlet Allocation (LDA) [BNJ03] to the subject systems using K=5 topics, and for each source code entity they multiply the topic memberships by the entity's LOC. They provide evidence that topic-related metrics can assist in explaining defects. Instead of focusing on the cohesiveness of topics in an entity, Chen *et al.* [CTNH12] proposed metrics that focus on the defect-prone topics in a code entity. They find that some topics are much more defect-prone than others and the more defect-prone topics a code entity has, the higher are the chances that it has defects.

- **Process metrics:** Process metrics leverage historical project knowledge. Examples of process metrics are prior defects and prior commits (code churn).

  – *Traditional process metrics:* Several studies have shown that process metrics, such as prior commits and prior defects, better explain software defects than product metrics (i.e., process metrics provide better statistical explanatory power) [MPS08; NB05; NB07; BH10]. Hassan [Has09] used the complexity of source code changes to explain defects. He found that the number of prior defects was a better metric to explain software defects than prior changes. A recent study by Rahman and Devanbu [RD13] analyzed the applicability and efficacy of process and product metrics from several different perspectives. They find that process metrics are generally more useful than product metrics.

  – *Social structure metrics:* Studies haven been conducted to investigate the relationship between social structure and software defects. Wolf *et al.* [WSDN09] carry out a case study to predict build failures by inter-developer communication. Pingzger *et al.* [PNM08] and Meneely *et al.* [MWSO08] use social network metrics to predict software defects. Bacchelli *et al.* [BDL10] investigate the use

of code popularity metrics obtained from email communication among developers for defect prediction. Recent work by Bettenburg *et al.* [BH13; BH10] use a variety of measures of social information to study relationships between these measures and code quality.

– *Ownership metrics:* There is previous defect modeling research focusing on the ownership and developers' expertise of source code. Early work by Mockus and Weiss [MW00] define a metric to measure developers' experience on a particular modification request. They use this metric to predict defects. Bird *et al.* [BGMD10] focus on the low-expertise developers. They find that contributions from low-expertise developers play a big role in the defect prediction model. Rahman *et al.* [RD11] find that stronger ownership by a single author is associated with implicated code. Recent work by Posnett *et al.* [PDDF13] propose using module activity focus and developers' attention focus to measure code ownership and developers' expertise. They find that more focused developers are less likely to introduce defects than less focused developers, and files that receive narrowly focused activity are more likely to contain defects than files that receive widely focused activities.

Due to the limitation of version control systems, most research on defect modeling extract the process metrics on a code-commit level. Mylyn[4] is a tool that can record developers' activity in the IDE. Using Mylyn enables researchers to investigate finer-level process metrics. For example, Zhang *et al.* [ZKZH12] leverage data generated by Mylyn to investigate the effect of file editing patterns on code quality.

Studies show that most product metrics are highly correlated to each other, as well as process metrics [SJI+10]. Among all the product metrics, lines of code has typically been shown to be the best metric to explain post-release defects [HH10]. On the other hand,

[4]http://wiki.eclipse.org/Mylyn last verified January 2014.

prior commits and pre-release defects are the best metrics among process metrics to explain post-release defects [GKMS00]. Prior research rarely considers comments. However, a relatively recent empirical study by Ibrahim *et al.* [IBAH12] studied the relationship between code comments and code quality. They find that a code change in which a function and its comment are co-updated inconsistently (i.e., they are not co-updated when they have been frequently co-updated in the past, or vice versa), is a risky change. Hence they have demostrated an empirical link between commenting characteristics and code quality. Similarly, the goal of this chapter is to investigate and establish an empirical link between logging characteristics and code quality (quantified through post-release defects).

5.4 Log-related Metrics

Prior research has shown that product metrics (like lines of code) and process metrics (like the number of prior commits) are good indicators of code quality. Product metrics are obtained from a single snapshot of the system, which describes the static status of the system. On the other hand, process metrics require past information about the system, capturing the development history of the system. Inspired by prior research, we define log-related metrics that cover both these aspects, namely product and process.

5.4.1 Log-related product metrics

We propose two log-related product metrics, which we explain below.

1. **Log density:** We calculate the number of logging statements in each file. We consider each invocation of a logging library method as a logging statement. For example, with *Log4j* [loga], a "*LOG*" method invocation is considered a logging statement. To factor out the influence of code size, we calculate the log density (LOGD) of a file as:

$$LOGD = \frac{\#\ of\ logging\ statements\ in\ the\ file}{LOC} \tag{5.1}$$

where LOC is the number of total lines of code in a source code file.

2. **Average logging level:** Logging level, such as "INFO" and "DEBUG", are used to filter logs based on their purposes. Intuitively, high-level logs are for system operators and lower-level logs are for development purposes [Gil]. We transform the logging level of each logging statement into a quantitative measurement. We consider all log levels including "TRACE", "DEBUG", "INFO", "WARN", "ERROR" and "FATAL". We consider that the lowest logging level is 1 and the value of each higher logging level increases by 1. For example, the lowest logging level in *Log4j* is "TRACE", therefore we consider the value of the "TRACE" level as 1. One level above "TRACE" is "DEBUG", so the value of a "DEBUG" level logging statement is 2. We calculate the average logging-level (LEVELD) of each source code file as

$$LEVELD = \frac{\sum_{i=1}^{n} logging\ level\ value_i}{n} \tag{5.2}$$

where n is the total number of logging statements in the source code file and $logging\ level\ value_i$ is the logging level value of the i^{th} logging statement in the source code file. The higher-level logs are typically used by operators and lower-level logs are used by developers and testers. Hence the log level acts as an indicator of the users of the logs.

Our intuition behind this metric is that some log levels are better indicators of defects.

5.4.2 Log-related process metrics

We propose two log-related process metrics, which we explain below.

1. **Average number of log lines added or deleted in a commit:** We calculate the average amount of added and deleted logging statements in each file prior to release

(LOGADD and LOGDEL).

$$LOGADD = \frac{\sum_{i=1}^{TPC} \#\ added\ logging\ statements_i}{TPC} \tag{5.3}$$

$$LOGDEL = \frac{\sum_{i=1}^{TPC} \#\ deleted\ logging\ statements_i}{TPC} \tag{5.4}$$

where TPC is the total number of prior commits to a file. Similar to log-related product metrics that were normalized, we normalize the log-related process metrics using TPC. $\#\ added\ logging\ statements\_i$ or $\#\ deleted\ loggingstatements\_i$ is the number of added or deleted logging statements in revision i. The intuition behind this metric is that frequently updating logging statements may be due to extensive debugging or implementation changes, which both may correlate to software defects.

2. **Frequency of defect-fixing code changes with log churn:** We calculate the number of defect-fixing commits in which there was log churn. We calculated this metric (FCOC) as:

$$FCOC = \frac{N(defect\ fixing\ commits \cap log\ churning\ commits)}{TPC} \tag{5.5}$$

where $N(defect\ fixing\ commits \cap log\ changing\ commits)$ is the number of defect-fixing commits in which there was log churn. The intuition behind this metric is that developers may not be 100% confident of their fix. Therefore, they may add some new logs or update old logs. Adding new logging statements, deleting existing logging statements, and adding new information to existing logging statements are all counted as log churn in this metric.

5.5 Case Study Setup

To study the relationship between logging characteristics and code quality, we conduct case studies on two large and widely used open source platform software:

- **Hadoop** [hada] is a large distributed data processing platform that implements the MapReduce [DG08] data-processing paradigm. We use 4 releases (0.16.0 to 0.19.0) of Hadoop in our case study.

- **JBoss Application Server** [jboa] (referred to as "*JBoss*" in the rest of this chapter) is one of the most popular Java EE application servers. We use 4 releases (3.0 to 4.2) of *JBoss* in our case study.

The goal of our study is to examine the relationship between our proposed log-related metrics and post-release defects. Previous studies of software defects [SJI+10; BH10] typically use an *Eclipse* data set provided by Zimmermann *et al.* [ZPZ07]. We do not use this data set because we are interested in platform software, where logging is more prominent. *Eclipse* does not have substantial logging code, therefore, *Eclipse* is not an ideal subject system for our study. *Hadoop* and *JBoss* are two of the largest and most widely used platform software. Both generate large numbers of logs during their execution, and tools have been developed to monitor the status of both systems using their extensive logs [RK10; jbob]. We do not use *EA* and *PostgreSQL* from Chapter 4 because we focus on platform software. To avoid the noise from the logging statements in the unit testing code in both projects, we exclude all the unit testing folders from our analysis. Table 5.2 gives an overview of the subject systems.

Figure 5.1 shows a general overview of our approach. (A) We mine the SVN repository of each subject system using a tool called J-REX [SJAH09; SAH11; Sha10] to produce high-level source code change information. (B) We then identify the log-related source code changes from the output of J-REX. (C) We calculate our proposed log-related metrics and

Table 5.2: Overview of subject systems.

	Release	# changed files	# defects
Hadoop	0.16.0	1,211	98
	0.17.0	1,899	180
	0.18.0	3,084	218
	0.19.0	3,579	175
JBoss	3.0	9,050	1,166
	3.2	25,289	1,108
	4.0	36,473	1,233
	4.2	126,127	3,578

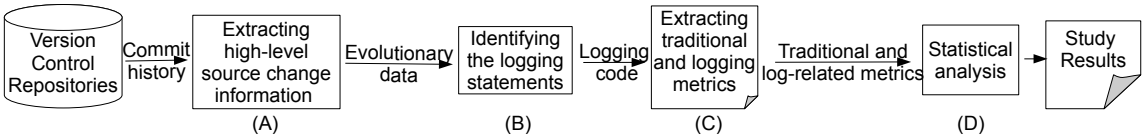


Figure 5.1: Overview of our case study approach.

traditional metrics. (D) Finally, we use statistical tools, such as R [IG96], to perform experiments on the data to answer our research questions. In the rest of this section we describe the first two steps in more detail.

5.5.1 Extracting high-level source change information

Similar to C-REX [Has05], J-REX [SJAH09; SAH11; Sha10] is used to study the evolution of the source code of Java software systems. For each subject system, we use J-REX to extract high-level source change information from their SVN repository.

The approach used by J-REX is broken down into three phases:

1. **Extraction:** J-REX extracts source code snapshots for each Java file revision from the SVN repository.

2. **Parsing:** Using the Eclipse JDT parser, J-REX outputs an abstract syntax tree for each extracted file in XML.

3. **Differencing:** J-REX compares the XML documents of consecutive file revisions to

determine changed code units and generates evolutionary change data. The results are stored in an XML document. There is one XML document for each Java file.

As common practice in the software engineering research community, J-REX uses defect-fixing changes [Has09; CMRH09; NBZ06] in each source code file to approximate the number of defects in them. The approximation is widely adopted because (1) only fixed defects can be mapped to specific source code files, (2) some reported defects are not real, (3) not all defects have been reported, and (4) there are duplicate issue reports.

To determine the defect-fixing changes, J-REX uses a heuristic proposed by Mockus *et al*. [MV00] on all commit messages. For example, a commit message containing the word "fix" is considered a message from a defect-fixing revision. The heuristic can lead to false positives, however, an evaluation of the J-REX heuristics shows that these heuristics identify defect-fixing changes with high accuracy [BKZ11].

5.5.2 Identifying the Logging Statements

Software projects typically leverage logging libraries to generate logs. One of the most widely used logging libraries is *Log4j* [loga]. We manually browse a sample of source code from each project and identify that both subject systems use *Log4j* as their logging library.

Knowing the logging library of each subject system, we analyze the output of J-REX to identify the logging source code fragments and changes. Typically, logging source code contains method invocations that call the logging library. For example, in *Hadoop* and *JBoss*, a method invocation by "*LOG*" with a method name as one of the logging levels is considered a logging statement. We count every such invocation as a logging statement.

5.6 Case Study Results

We now present the results of our case study. For each research question, we discuss the motivation for the question, the approach used to address the question and our experimental

results. For our case study, we study the code quality at the file level.

5.6.1 Preliminary Analysis

We start with a preliminary analysis of the log-related metrics presented in Section 5.5 to illustrate the general properties of the collected metrics.

In the preliminary analysis we calculate seven aggregated metrics for each release of both subject systems: total lines of code, total code churn (total added, deleted and modified lines of code), total number of logging statements, total log churn (total added and deleted logging statements), percentage of source code files with logs, percentage of source code files with pre-release defects, and percentage of source code files with post-release defects. Table 5.3 shows that around 18% to 28% of the source code files contain logging statements. Since less than 30% of the source code files have logs, we calculate the skew and Kurtosis values for our log-related metrics. We observe that our log-related metrics have positive skew (i.e., all the metric values are on the low scale) and large Kurtosis values (i.e., the curve is too tall). To alleviate the bias caused by these high skew and Kurtosis values, we follow a typical approach used in previous research [SJI+10]: to log transform all of the metrics. From this point on, whenever we mention a metric, we actually are referring to its log transformed value.

5.6.2 Results

RQ1. Are source code files with logging statements more defect-prone?

Motivation

Our qualitative study in Section 5.2 shows that software developers often add or modify logs to diagnose and fix software defects. We first explore the data to study whether source code files with logging statements are more likely to be defect-prone.

Table 5.3: Lines of code, code churn, amounts of logging statements, log churns, percentage of files with logging, percentage of files with pre-release defects, and percentage of files with post-release defects over the releases of Hadoop and JBoss.

	Hadoop				JBoss			
	0.16.0	0.17.0	0.18.0	0.19.0	3.0	3.2	4.0	4.2
lines of code	133K	108K	119K	169K	321K	351K	570K	552K
code churn	7k	10k	9k	12k	489k	260k	346k	170K
logging statements	563	881	1,278	1,678	2,438	3,716	5,605	10,379
log churn	601	2,136	2272	1,579	6,233	5,357	5,966	18,614
percentage of files with logging	18%	26%	25%	28%	27%	23%	24%	23%
percentage of files with pre-release defects	16%	26%	27%	42%	50%	34%	27%	31%
percentage of files with post-release defects	34%	27%	46%	29%	45%	34%	33%	26%

Approach

First, we calculate the post-release defect densities of each source code file in each of the studied releases. We compare the average defect density of source code files with and without logging statements. Then, we perform independent two-sample unpaired T-tests to determine whether the average defect-densities for source code files with logs are statistically greater than the average defect-densities for source code files without logs. Finally, we calculate the Spearman correlation between our log-related metrics and post-release defects to determine if our metrics lead to similar prioritization (i.e., similar order) with source code files with more defects having higher metric values.

Results and discussion

We find that source code files with logging statements are more defect-prone. Table 5.4 shows the average post-release defect densities of source code files with and without

Table 5.4: Average defect densities of the source code files with and without logging statements in the studied releases. Largest defect densities are shown in bold. The p-value for significance test is 0.05.

	Hadoop				JBoss			
	0.16.0	0.17.0	0.18.0	0.19.0	3.0	3.2	4.0	4.2
With logging	**0.116**	**0.142**	0.249	**0.140**	**0.172**	**0.145**	**0.101**	**0.092**
Without logging	0.095	0.083	**0.250**	0.124	0.122	0.101	0.075	0.090
Statistically significant	yes	yes	no	no	yes	yes	yes	no
p-value	<0.001	<0.001	0.51	0.16	<0.001	<0.001	<0.001	0.36

logging statements. The results show that in 7 out of the 8 studied releases, source code files with logging statements have higher average post-release defect densities than source code files without logging statements.

We use independent two-sample unpaired one-tailed T-tests to determine whether the average defect density of source code files with logs was statistically greater than those without logs. Our null hypothesis assumes that the average post-release defect densities of source code files with and without logging statements are similar. Our alternate hypothesis was that the average defect density of source code files with logs was statistically greater than those without logs. For 5 of the 7 releases where source code files with logs have higher defect density, the *p*-values are smaller than 0.05 (see Table 5.4). We reject the null hypothesis and conclude that in these 5 releases, the average defect density of source code files with logs are greater than those without logs.

We examine the release $0.18.0$ of *Hadoop*, which is the exception because the average defect densities of source code files with and without logs are similar. We found that there is a structural change in *Hadoop* before release $0.18.0$ and that a large number of defects appear after this release (largest percentage of defect-prone source code files in *Hadoop* as shown in Table 5.3). This might be the reason that in release $0.18.0$ of *Hadoop*, the software source code files with logging statements are not more defect-prone.

Log-related metrics have positive correlation with post-release defects. Table 5.5 presents the Spearman correlations between our log-related metrics and post-release defects. We find that, in 7 out of 8 releases, the largest correlations between log-related metrics and post-release defects are similar (3 releases with a $+/- 0.03$ value) or higher than the correlations between pre-release defects and post-release defects. Since the number of pre-release defects is known to have a positive correlation with post-release defects, this observation supports our intuition of studying the relationship between logging characteristics and code quality.

For the only exception (release 3.0 of *JBoss*), we examine the results more closely and find that the correlation between log-related metrics and post-release defects in this version of *JBoss* is not very different from the other releases of *JBoss*. However, the correlation between pre-release defects and post-release defects in this version of *JBoss* is much higher compared to the other releases of *JBoss*. On further analysis of the data, we find that in release 3.0 of *JBoss*, up to 50% more files (as compared to other releases) had pre-release defects. Therefore, we think that this might be the reason that the correlation between pre-release defect and post-release defects in release 3.0 of *JBoss* is higher than the correlation between post-release defects and our log-related metrics.

Density of logging statements added has higher correlation with post-release defects than density of logging statements deleted. We find that the average logging statements added in a commit has the largest correlation with post-release defects in 5 out of 8 releases, while the correlation between the average deleted logging statements in a commit and post-release defects is much lower than the other log-related metrics (see Table 5.5). We count the number of added and deleted logging statements in source code files with and without defects separately. We find that in *Hadoop*, the total lines of code ratio between defect-prone source code files and non defect-prone source code files is 1.03; while the number of logging statements added in defect-prone source code files (2,309) is around three times the number of logging statements added (736) in non defect-prone source code

Table 5.5: Spearman correlation between log-related metrics and post-release defects. Largest number in each release is shown in bold.

	Hadoop Releases			
	0.16.0	0.17.0	0.18.0	0.19.0
LOGD	0.36	0.26	0.24	**0.24**
LEVELD	0.36	0.25	0.22	0.23
LOGADD	**0.42**	0.27	**0.28**	**0.24**
LOGDEL	0.23	0.09	0.21	0.10
FCOC	0.25	**0.30**	0.18	0.17
PRE	0.15	0.27	0.25	0.12

	JBoss Releases			
	3.0	3.2	4.0	4.2
LOGD	0.26	0.26	0.21	0.13
LEVELD	0.26	0.27	0.22	0.13
LOGADD	0.34	**0.29**	**0.59**	0.19
LOGDEL	0.34	0.18	0.42	0.13
FCOC	0.23	0.20	0.20	0.14
PRE	**0.40**	0.22	0.21	**0.22**

files. This shows that there exists a relation between logs and defect-prone source code files. However, the number of logging statements deleted in defect-prone source code files (268) is only around two times the number of logging statements deleted in non defect-prone source code files (124). Therefore, even though developers delete more log lines in defect-prone source code files, the ratio to non defect-prone source code files is much lower than the ratio to log lines added. Hence, this shows that the developers may delete logs when they feel confident with their source code files. Concrete examples of such logging behaviour have been presented in Section 5.2.

Summary: We find that in 7 out of 8 studied releases, source code files with logging statements have higher average post-release defect densities than those without logging statements. In 5 of these 7 releases, the differences between the average defect density in the source code files with and without logs are statistically significant. The correlations between log-related metrics and post-release defects are similar to the correlations between

post-release defects and pre-release defects (one of the highest correlated metrics to post-release defects). Among the log-related metrics, the average logging statements added in a commit have the highest correlation with post-release defects.

Source code files with logging statements tend to be more defect-prone.

RQ2. Can log-related metrics help in explaining post-release defects?

Motivation

In the previous research question, we showed the correlation between logging characteristics and post-release defects. However, there is a chance that such correlations may be due to other factors, such as lines of code being correlated to both the logging characteristics and post-release defects. To further understand the relationship between logging characteristics and post-release defects, we control for factors that are known to be the best explainers of post-release defects, i.e., lines of code, pre-release defects, and code churn. In particular, we would like to find out whether we can complement the ability of traditional software metrics in explaining post-release defects by using logging characteristics (i.e., our proposed log-related product and process metrics).

Approach

We use logistic regression models to study the explanatory power of our log-related metrics on post-release defects. However, previous studies show that traditional metrics, such as lines of code (LOC), code churn or the total number of prior commits (TPC), and the number of prior defects (PRE), are effective in explaining post-release software defects [NBZ06; GKMS00]. Therefore, we included these metrics as well in the logistic regression models. Note that many other product and process metrics are highly correlated with each other [SJI+10]. To avoid the collinearity between TPC and PRE, we run PCA on TPC and PRE and use the first component as a new metric, which we call TPCPRE:

$$TPCPRE = PCA(TPC, PRE)_{firstcomponent} \tag{5.6}$$

Table 5.6: Spearman correlation between the two log-related product metrics: log density (LOGD) and average logging level (LEVELD), and the three log-related process metrics: average logging statements added in a commit (LOGADD), average logging statements deleted in a commit (LOGDEL), and frequency of defect-fixing code changes with log churn (FCOC).

	Hadoop				JBoss			
	0.16.0	0.17.0	0.18.0	0.19.0	3.0	3.2	4.0	4.2
LOGD and LEVELD	0.74	0.62	0.41	0.64	0.58	0.16	0.19	0.26
LOGADD and FCOC	0.49	0.46	0.69	0.47	0.68	0.58	0.59	0.56
LOGDEL and FCOC	0.25	0.36	0.48	0.18	0.48	0.43	0.43	0.36
LOGADD and LOGDEL	0.56	0.50	0.59	0.55	0.59	0.52	0.54	0.55

Before building the logistic regression models, we study the Spearman correlation between the two log-related product metrics and the three log-related process metrics. From the results in Table 5.6, we find that in some releases, the correlations between the two log-related product metrics and between the three log-related process logging metrics are high.

To address the collinearity as noted in Table 5.6, we derive two new metrics: a log-related product metric (PRODUCT) and a log-related process metric (PROCESS), to capture the product and process aspects of logging respectively. To compute the two new metrics, we ran Principal Component Analysis (PCA) [JW91] once on the log-related product metrics (i.e., log density and average logging level), and once on the log-related process metrics (average logging statements added in a commit and frequency of defect-fixing code changes with log churn) [Har01]. Since the previous section showed that the average deleted logging statements (LOGDEL) has a rather low correlation with post-release defects (see Table 5.5), we decided not to include LOGDEL in the rest of our analysis and models. From each PCA run, we use the first principal component as our new metric.

$$PRODUCT = PCA(LOGD, LEVELD)_{firstcomponent} \tag{5.7}$$

$$PROCESS = PCA(LOGADD, FCOC)_{firstcomponent} \tag{5.8}$$

We used the two combined metrics (PRODUCT and PROCESS) in the rest of the chapter, so that we can build the same models across releases without worrying about the impact of collinearity on our results.

We determine whether the log-related metrics can complement traditional product and process based metrics in providing additional explanatory power. The overview of the models is shown in Figure 5.2. We start with three baseline models that use the best-performing traditional metrics as independent variables.

- **Base(LOC)**: The first base model is built using lines of code as an independent variable to measure the explanatory power of traditional product metrics.
- **Base(TPCPRE)**: The second base model is built using a combination of pre-release defects and prior changes as independent variables to measure the explanatory power of traditional process metrics.
- **Base(LOC+TPCPRE)**: The third base model is built using lines of code and the combination of pre-release defects and prior changes as independent variables to measure the explanatory power of both traditional product and process metrics.

We then build subsequent models in which we add our log-related metrics as independent variables.

- **Base(LOC)+PRODUCT**: We add our log-related product metric (PRODUCT) to the base model of product metrics to examine the improvement in explanatory power due to log-related product metrics.
- **Base(TPCPRE)+PROCESS**: We add our log-related process metric (PROCESS) to the base model of process metrics to examine the improvement in explanatory power due to log-related process metrics.

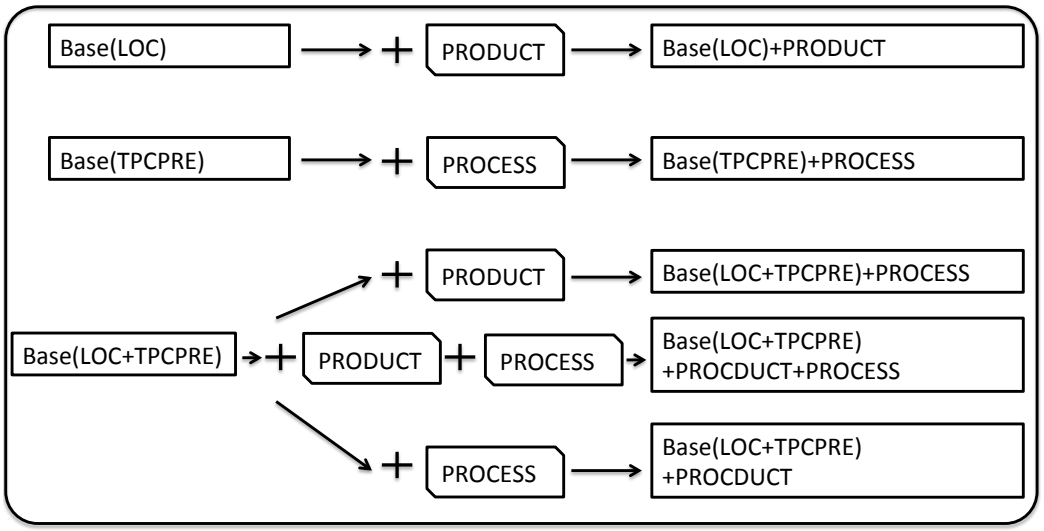


Figure 5.2: Overview of the models built to answer RQ2. The results are shown in Table 5.7, 5.8 and 5.9.

- **Base(LOC+TPCPRE)+PRODUCT**: We add our log-related product metric (PRODUCT) to the base model Base(LOC+TPCPRE) to examine the improvement in explanatory power due to log-related product metrics.

- **Base(LOC+TPCPRE)+PROCESS**: We add our log-related process metrics (PROCESS) to the base model Base(LOC+TPCPRE) to examine the improvement in explanatory power due to log-related process metrics.

- **Base(LOC+TPCPRE)+PRODUCT+PROCESS**: Finally, we add both our log-related product metric (PRODUCT) and our log-related product metric (PROCESS) into the base model Base(LOC+TPCPRE) to examine the improvement in explanatory power due to both log-related metrics.

For each model, we calculate the deviance explained by the models to measure their explanatory power. A higher percentage of deviance explained generally indicates a better model fit and a higher explanatory power for the independent variables.

To understand the relationship between logging characteristics and post-release defects,

we need to understand the effects of the metrics in the model. We follow a similar approach used in prior research [SMK+11; Moc10]. To quantify this effect, we set all of the metrics in the model to their mean value and record the predicted probabilities. Then, to measure the effect of every log metric, we keep all of the metrics at their mean value, except for the metric whose effect we wish to measure. We increase the value of that metric by 10% over the mean value and re-calculate the predicted probability of the dependent variable. We then calculate the change in probability caused by increasing the metric by 10%. The effect of a metric can be positive or negative. A positive effect means that a higher value of the factor increases the likelihood of the dependent variable, whereas a negative effect means that a higher value of the factor decreases the likelihood of the dependent variable. This approach permits us to study metrics that are of different scales, in contrast to using odds ratios analysis, which is commonly used in prior research [SJI+10].

We would like to point out that although logistic regression has been used to build accurate models for defect prediction, our purpose for using the regression model in this chapter is not for predicting post-release defects. Our purpose is to study the explanatory power of log-related metrics and explore its empirical relationship with post-release defects.

Results and discussion:

Log-related metrics complement traditional metrics in explaining post-release defects. Table 5.7 shows the results of using lines of code (LOC) as the base model. We find that the log-related product metric (PRODUCT) provides statistically significant improvement in 7 out of the 8 studied releases. The log-related product metric (PRODUCT) provides up to a 43% improvement in the explanatory power over the Base (LOC) model.

Table 5.8 shows the results of using process metrics (TPCPRE) as the base model. In 5 out of 8 models, the log-related process metric (PROCESS) provides statistically significant ($p < 0.05$) improvement. In particular, release $0.16.0$ of *Hadoop* has the largest improvement (360%) over the base model.

Table 5.9 shows the results of using both product and process metrics in the base models.

Table 5.7: Deviance explained (%) improvement for product software metrics by logistic regression models.

Hadoop Releases				
	0.16.0	0.17.0	0.18.0	0.19.0
Base(LOC)	14.37	12.74	3.23	8.14
Base+PRODUCT	15.85(+10%)*	12.76 (+0%)	4.62(+43%)**	9.7(19%)***

JBoss Releases				
	3.0	3.2	4.0	4.2
Base(LOC)	5.26	5.67	4.49	2.28
Base+PRODUCT	6.25(+19%) ***	6.41(+13%)***	4.93(+10%)***	2.56(+12%)***

*** p<0.001, ** p<0.01, * p<0.05, $\diamond$ p <0.1

In all studied releases, except for release $0.17.0$ of *Hadoop*, at least one log-related metric is statistically significant in enhancing the base model (in bold font). The log-related metrics provide up to a 40% in the explanatory power of the traditional metrics.

Release $0.17.0$ of *Hadoop* is the release where neither product nor process log-related metrics are significant. In that release, we note that the number of source code files with logs increased from 18% to 26% (see Table 5.3). Some logs may be added into defect-free source code files when there is such a large increase in logs. We performed an independent two-sample unpaired T-test to determine whether the average log densities of source code files with post-release defects was statistically different to the average log densities of source code files without post-release defects. The p-value of the test is 0.22. Hence there is no statistical evidence to show that the log densities of defect-prone and defect-free source code files differ in release $0.17.0$ of *Hadoop*. We think this may be the reason that log-released product metrics do not have significant explanation power in *Hadoop* release $0.17.0$.

Log-related metrics have a positive effect on the likelihood of post-release defects. In Table 5.10 we show the effect of the PRODUCT and PROCESS metrics on post-release defects. We measure the effect by increasing the value of a metric by 10% from its mean value, while keeping all other metrics at their mean value in a model. We only discuss the effect of

Table 5.8: Deviance explained (%) improvement for process software metrics by logistic regression models.

	Hadoop Release			
	0.16.0	0.17.0	0.18.0	0.19.0
Base(TPCPRE)	2.49	8.47	4.53	2.44
Base+PROCESS	11.47(+360%)***	8.55 (+1%)	5.09 (+12%) ⋄	3.69(+51%)***

	JBoss Release			
	3.0	3.2	4.0	4.2
Base(TPCPRE)	10.38	3.75	4.56	2.37
Base+PROCESS	10.55(+2%) ⋄	4.71(+26%)***	4.83(+6%)***	2.73(+15%)***

*** p<0.001, ** p<0.01, * p<0.05, ⋄ p <0.1

Table 5.9: Deviance explained (%) improvement using both product and process software metrics by logistic regression models. The values are shown in bold if the model "Base+PRDUCT+PROCESS" has at least one log metric statistically significantly.

	Hadoop Release			
	0.16.0	0.17.0	0.18.0	0.19.0
Base(LOC+TPCPRE)	14.69	13.34	5.3	8.32
Base+PRODUCT	16.56(+13%)**	13.34 (+0%)	6.21 (+17%)*	9.84(+18%)***
Base+PROCESS	19.17(+30%)**	13.4 (+0%)	5.72 (+8%)	8.85(+6%)*
Base+PRODUCT +PROCESS	**20.5(+40%)**	13.42 (+1%)	**6.36 (+20%)**	**9.98(+20%)**

	JBoss Release			
	3.0	3.2	4.0	4.2
Base(LOC+TPCPRE)	12.09	6.46	6.45	3.22
Base+PRODUCT	12.79(+6%)***	6.98 (+8%)***	6.69 (+4%)**	3.34(+4%)*
Base+PROCESS	12.09(+0%)	6.94 (+8%)***	6.51 (+1%)*	3.41(+6%)**
Base+PRODUCT +PROCESS	**12.93(+7%)**	**7.23 (+12%)**	**6.73 (+4%)**	**3.47(+8%)**

*** p<0.001, ** p<0.01, * p<0.05, ⋄ p<0.1

log-related metrics that are statistically significant in model Base(LOC+TPCPRE)+PRODUCT +PROCESS (shown in Table 5.9). The effects of the log-related product metric (PRODUCT) are positive. Since the log-related product metric (PRODUCT) is the combination of log density and average logging level, this result implies that more logging statements and/or a higher logging level leads to a higher probability of post-release defects. Table 5.10 shows

Table 5.10: Effect of log-related metrics on post-release defects. Effect is measured by setting a metric to 110% of its mean value, while the other metrics are kept at their mean values. The bold font indicates that the metric is statistically significant in the Base(LOC+TPCPRE)+PRODUCT+PROCESS model.

	Hadoop Release			
	0.16.0	0.17.0	0.18.0	0.19.0
PRODUCT	**2.2%**	-0.1%	**1.9%**	**3.6%**
PROCESS	**2.5%**	0%	0.3%	0.3%

	JBoss Release			
	3.0	3.2	4.0	4.2
PRODUCT	**1.8%**	**0.8%**	**0.7%**	4.7%
PROCESS	-0.5%	**0.5%**	**0.1%**	**2.5%**

that in all 4 releases where the log-related process metric (PROCESS) is statistically significant, the log-related process metric (PROCESS) has a positive effect on defect-proneness. The result shows that in some cases, developers change logs to monitor components that may be defect-prone. For example, in revision 226841 of *Hadoop*, developers enhanced the logging statement that tracks nodes in the machine cluster to determine the rationale for field failures of nodes in their cluster. Therefore, in some source code files, the more logs added and/or more defect fixes with log churn, the higher the probability that the source code file is defect-prone.

Summary: Log-related metrics complement traditional product and process metrics in explaining post-release defects. In particular, log-related product metrics contribute to an increase in explanatory power in 7 out of 8 studied releases, and log-related process metrics contribute to an increase in explanatory power for 5 out of 8 studied releases. We also find that increases in either log-related product or process metrics increases defect-proneness.

Our results show that there exists a strong relationship between logging characteristics and code quality.

5.7 Threats to Validity

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

External Validity

Our study is performed on *JBoss* and *Hadoop*. Although both subject systems have years of history and large user bases, more case studies on other platform software in other domains are needed to determine whether our findings can be generalized. There are other types of software systems that make use of logs only while the system is under development. The logs are removed when the system is released. Even though such systems do not benefit from the field feedback through logs, logging is still a major tool to diagnose and fix defects. Our findings may generalize to such software systems. However, future studies are needed on the logs of such types of systems.

Internal Validity

Our study is based on the version control repositories of the subject systems. The quality of the data contained in the repositories can impact the internal validity of our study.

Our analysis of the link between logs and defects cannot claim causal effects, as we are investigating correlations, rather than conducting impact studies. The explanative power of log-related metrics on post-release defects does not indicate that logs cause defects. Instead, it indicates the possibility of a relation that should be studied in depth through user studies.

The deviance explained in some of the models may appear low, however this is expected and should not impact the conclusions. One reason for such low deviance is that in a few releases, the percentage of source code files with defects is less than 30% [MGF07; ZNW10]. Moreover, only around 20% to 30% of the source code files contain logging statements. The deviance explained can be increased by adding more variables to the model in RQ2, however we would need to deal with the interaction between the added variables.

Construct Validity

The heuristics to extract logging source code may not be able to extract every logging

statement in the source code. However, because the case study systems use logging libraries to generate logs at runtime, the method in the logging statements are consistent. Hence, this heuristic will capture all the logging statements.

Our software defect data is based on the data produced by J-REX, a software evolution tool that generates high-level evolutionary source code change data. J-REX uses heuristics to identify defect-fixing changes. The results of this chapter are dependent on the accuracy of the results from J-REX. We are confident in the results from J-REX as it implements the algorithm used previously by Hassan *et al.* [Has08] and Mockus *et al.* [MV00]. However, previous research shows that the misclassified defect-fixing commits may introduce negative effects on the performance of prediction techniques on post-release defects [BBA+09]. We select a random sample of 337 and 366 files for *Hadoop* and *JBoss*, respectively. Only 14% and 2% of the files for *Hadoop* and *JBoss* are misclassified, respectively. Both random sample sizes achieve 95% confidence level with a 5% confidence interval [Smi03]. Other approaches can be used to identify defect-fixing commits, such as the data in the issue tracking systems, to perform additional case studies to further understand the relationship between logging characteristics and code quality. J-REX compares the abstract syntax trees between two consecutive code revisions. A modified logging statement is reported by J-REX as an added and a deleted logging statement. Such limitation of J-REX may result in inaccuracy of our metrics. Other techniques to extract the log-related metrics should be explored.

In addition, we find that, on average, there is a logging statement for every 160 and 130 lines of source code for *Hadoop* and *JBoss*, respectively. A previous study by Yuan *et al.* [YZP+11] shows that the ratio between source code and logging code is 30:1. We think the reason for such a discrepancy is that the log density metric (LOGD) defined in this chapter uses the number of logging statements instead of lines of logging code, and the total lines of code instead of source lines of code. We calculated the ratio between total lines of code and number of logging statements for the four subject systems studied in prior

research. We found that the ratios are 155:1, 146:1, 137:1 and 70:1 for *Httpd*, *Openssh*, *PostgreSQL* and *Squid*, respectively. Such ratios are similar to the ratios in our two studied systems. In this chapter, we extract our log-related metrics from an AST. We chose to use the AST so that we can accurately extract every invocation to a logging method. Hence due to this choice, we can only get the number of logging statements and not number of lines of logging code.

The possibility of post-release defects can be correlated to many factors other than just logging characteristics, such as the complexity of code and pre-release defects. To reduce such a possibility, we included 3 control metrics (lines of code, pre-release defects, and prior changes) that are well known to be good predictors of post-release defects in our logistic regression model [NBZ06; MPS08]. However, other factors may also have an impact on post-release defects. Future studies should build more complex models that consider these other factors.

Source code from different components of a system may have different characteristics. Logs may play different roles in components with different levels of importance. Value and importance of code is a crucial topic, yet it has been rarely investigated. However, this chapter introduces a way to use logs as a proxy to investigate the role of different parts of the code.

5.8 Chapter Summary

Logging is one of the most frequently-employed approaches for diagnosing and fixing software defects. Logs are used to capture the concerns and doubts of developers as well as operators' needs for run-time information about the software. In this chapter, we propose an approach to leveraging logs to assist in prioritizing code review and testing efforts. The relationship between logging characteristics and software quality has never been empirically studied before. This chapter is a first attempt (to the best of our knowledge) to build

an empirical link between logging characteristics and software defects. The highlights of our findings are:

- We found that source code files with logging statements have higher post-release defect densities than those without logging statements.
- We found a positive correlation between source code files with log lines added by developers and source code files with post-release defects.
- We found that log-related metrics complement traditional product and process metrics in explaining post-release defects.

Our findings do not advocate the removal of logs that are a critical instrument used by developers to understand and monitor the field quality of their software. Instead, the findings of this chapter suggest that software maintainers should allocate more preventive maintenance effort on source code files with more logs and log churn, because such source code files may be the ones where developers and operators may have more doubts and concerns, and hence are more defect-prone.

CHAPTER 6

Verifying the Deployment of Big Data Analytic Applications Using Logs

Big data analytics is the process of examining large amounts of data (big data) in an effort to uncover hidden patterns or unknown correlations. Big Data Analytics Applications (BDA Apps) are a new type of software applications, which analyze big data using massive parallel processing platforms (e.g., *Hadoop*). Developers of such applications typically develop them using a small sample of data in a pseudo-cloud environment. Afterwards, they deploy the applications in a large scale cloud environment with considerably more processing power and larger input data (reminiscent of the mainframe days). Working with BDA App developers in industry over the past three years, we noticed that the run-time analysis and debugging of such applications in the deployment phase cannot be easily addressed by traditional monitoring and debugging approaches.

In this chapter, as a first step in assisting developers of BDA Apps for cloud deployments, we propose a lightweight approach for uncovering differences between pseudo and large scale cloud deployments. Our approach makes use of the readily-available yet rarely used execution logs from platform software. Our approach abstracts the execution logs, recovers the execution sequences, and compares the sequences between the pseudo and cloud deployments. Through a case study on three representative Hadoop-based BDA Apps, we show that our approach can rapidly direct the attention of BDA App developers to the major differences between the two deployments. Knowledge of such differences is essential in verifying BDA Apps when analyzing big data in the cloud. Using injected deployment faults, we show that our approach not only significantly reduces the deployment verification effort, but also provides very few false positives when identifying deployment failures.

6.1 Introduction

Big-Data Analytics Applications (BDA Apps) are a new category of software applications that analyze large scale data, which is typically too large to fit in memory or even on one hard

drive, in order to uncover actionable knowledge using large scale parallel-processing infrastructures [FDCD12]. Big data can come from sources such as run-time information about traffic, tweets during the Olympic games, stock market updates, usage information of an online game [Win], or the data from any other rapidly growing data-intensive software system. For instance, EBAY[1] has deployed BDA Apps to optimize the search of products by analyzing over 5 PBs of data using more than 4,000 CPU cores [eba].

For over three years, we have worked closely with BDA App developers in industry. We noted and found that developing BDA Apps brings many new challenges compared to traditional programming and testing practices. Among these challenges in the different phases of BDA App development, the deployment phase introduces unique challenges related to verifying and debugging the BDA executions, as BDA App developers want to know if their BDA App will function correctly once deployed. Similar observations were recently noted in an interview of 16 professional BDA App developers at Microsoft [FDCD12].

In practice, the deployment of BDA Apps in the cloud follows these three steps: 1) developers implement and test the BDA App in a small or pseudo cloud (using virtual or physical machines) environment using a small data sample, 2) developers deploy the application on a larger cloud with a considerably larger data set and processing power to test the application in a real-world setting, and 3) developers verify the execution of the application to make sure all the data are processed and that all jobs are successful. The traditional approach for deployment verification is to simply search for known error-keywords related to unusual executions. However, such verification approaches are very ineffective in large cloud deployments. For instance, a common basic approach for identifying deployment problems is searching for "killed" jobs in the generated execution logs (the output of the internal instrumentation) of the underlying platform hosting the deployed application [Whi09]. However, a simple keyword search would lead to false positive results because a platform such as Hadoop may intervene in the execution of a job, kill it and restart it

[1] www.ebay.com last verified January 2014.

elsewhere to achieve better performance, or it may start and kill speculative jobs [Whi09]. Considering the large amount of data and logs, such false positives rapidly overwhelm the developers of BDA Apps.

In this chapter, we propose an approach to verify the run-time execution of BDA Apps after deployment. The approach abstracts the platform's execution logs from both the small (pseudo) and large scale cloud deployments, groups the related abstracted log lines into execution sequences for both deployments, then examines and reports the differences between the two sets of execution sequences. Ideally, these two sets should be identical for a successful deployment. However, due to platform configurations and data size differences, the underlying platform may execute the applications differently. Among the delta sets of execution sequences between these two sets, we filter out sequences that are due to well-known platform-related (in our case study Hadoop) differences. The remaining sets of sequences are potential deployment failures/anomalies that should be reported and carefully examined.

We have implemented our approach as a prototype tool and performed a case study on three representative Hadoop [Whi09] BDA Apps. The choice of Hadoop is due to it being one of the most used platforms for Big-Data Analytics in industry today. However, our general idea of using the underlying platform's logs as a means for BDA App monitoring in the cloud, is easily extensible to other platforms, such as Microsoft Dryad [IBY+07]. The case study results show that our log abstraction and execution sequence clustering not only significantly reduces the number of logs (by between 86% to 97%) that should be verified, but it also provides much higher precision for identifying deployment failures/anomalies compared to a traditional keyword search approach (commonly used in practice today). In addition, practitioners who have used our approach in practice have noted that the reporting of the abstracted execution sequences, rather than raw log lines, provides a summarized context that dramatically improves their efficiency in identifying and investigating failures/anomalies.

The rest of this chapter is organized as follows. We present a motivating example in Section 6.2. We present *Hadoop*, the platform that we studied in Section 6.3. We present our approach to summarize logs into execution log sequences in Section 6.4. We present the setup for our case study in Sections 6.5. We present the results of our case study in Section 6.6. We discuss other features of our approach in Section 6.7 and discuss the limitations of our approach in Section 6.8. We present prior work related to our approach in Section 6.9. Finally, Section 6.10 concludes the chapter.

6.2 A Motivating Example

We now present a hypothetical but realistic motivating example to better explain the challenges in deploying BDA Apps in a cloud environment.

Assume a developer, Ian, developed a BDA App that analyzes the user information from a large scale social network. Ian has thoroughly tested the App on an in-house small-scale cloud environment with a small sample of testing data. Before officially releasing the App, Ian needs to deploy the App in a large scale cloud environment and run the App with real-world large scale data. After the test run of the App in the real cloud setup, Ian needs to verify whether the App behaves as expected or not, in the testing environment.

Ian followed a traditional approach to examine the behaviour of the App in the cloud environment. He leveraged the logs from the underlying platform (e.g., Hadoop) to find whether there are any problematic log lines. After downloading all the logs from the cloud environment, Ian found that the logs are of enormous size because the cloud environment contains thousands of nodes and the processed real-world data is in PB scale, which makes the manual inspection of the logs impossible. Therefore, Ian performed a simple keyword search on the logs. The keywords are based on his own experience of developing BDA Apps. However, the keyword search still returns thousands of problematic log lines. By manually exploring the problematic log lines, Ian found that a large portion of the log lines

do not indicate any problematic executions (i.e., false positives). For example, the run-time scheduler of the underlying platform often kills remote processes and re-starts them locally to achieve better performance. However, such kill operations lead to seemingly problematic logs that are retrieved by his keyword search. Moreover, for each log line, Ian must trace through the log files across multiple nodes to gain some context about the generated log files (and in many instances he discovers that such log lines are expected and are not problematic ones). In short, identifying deployment problems of the BDA App is excessively difficult and time consuming. Moreover, this difficulty increases considerably as the size of the analyzed data grows and the size of the cloud increases.

From the above example, we observe that verifying the deployment of BDA Apps in a cloud environment with large scale data is challenging. Although today, developers primarily use grep [Sor09] to locate possible troublesome instrumentation logs, uncovering the related context of the troublesome logs is still challenging with enormously large data (as noted in recent interviews of BDA App developers [FDCD12]).

In the following sections, we present our approach, which summarizes the large amount of platform logs and presents them in tables where developers can easily note troublesome events and where they are able to easily view such events in the context of their execution (since the table shows summarized execution sequences).

6.3 Large Scale Data Analysis Platforms: Hadoop

Hadoop is one of the most widely used platforms for the development of BDA Apps in practice today. We briefly present the programming model of Hadoop, then present the *Hadoop* logs that we use in our case studies.

6.3.1 The MapReduce programming model

Hadoop is an open source distributed platform [Whi09] that is supported by Yahoo! and is used by Amazon, AOL and a number of other companies. To achieve parallel execution, *Hadoop* implements a programming model named MapReduce. This programming model is implemented by many other cloud platforms as well [IBY+07; MTF11].

MapReduce [DG08] is a distributed divide-and-conquer programming model that consists of two phases: a massively parallel "Map" phase, followed by an aggregating "Reduce" phase. The input data of MapReduce is broken down into a list of key/value pairs. Mappers (processes assigned to the "Map" phase) accept the incoming pairs, process them in parallel and generate intermediate key/value pairs. All intermediate pairs having the same key are then passed to a specific Reducer (process assigned to the "Reduce" phase). Each Reducer performs computations to reduce the data to one single key/value pair. The output of all Reducers is the final result of a MapReduce run.

To illustrate MapReduce, we consider an example MapReduce process that counts the frequency of word lengths in a book. Mappers take each single word from the book and generate a key/value pair of the form "word length/dummy value". For example, a Mapper generates a key/value pair of "5/hello" from the input word "hello". Afterwards, the key/-value pairs with the same key are grouped and sent to Reducers. Each Reducer receives the list of all key/values pairs for a particular word length and hence can simply output the size of this list. If a reducer receives a list with key "5", for example, it will count the number of all the words with length "5". If the size is n, it generates an output pair "5/n" which means there are n words with length "5" in the book.

6.3.2 Components of Hadoop

Hadoop has three types of execution components. Each component has logging enabled in it. Such platform logging tracks the operation of the platform itself (i.e., how the platform

is orchestrating the MapReduce processing). Today, such logging is enabled in all deployed *Hadoop* clusters and it provides a glimpse into the inner working mechanism of the platform itself. Such an inner working mechanism is impacted by any problems in the cloud on which the platform is executing. The three execution components and a brief example of the logs generated by them are as follows:

- **Job.** A *Hadoop* program consists of one or more MapReduce steps running as a pipeline. Each MapReduce step is a Job in Hadoop. A JobTracker is a process initialized by the *Hadoop* platform to track the status of the Jobs. The information tracked by the JobTracker includes the overall information of the Job (e.g., input data size) and the high-level information of the execution of the Job. The high-level information of the Job's execution corresponds to the executions of Map and Reduce. For example, a Job log may say that "the Job is split into 100 Map Tasks" and "Map TaskId=id is finished at time t1".

- **Task.** The execution of a Job is divided into multiple Tasks based on the MapReduce programming model. Therefore, a Task can be either a Map Task that corresponds to the Map in the *MapReduce* programming model, or a Reduce Task. The *Hadoop* platform groups a set of Map or Reduce executions together to create a Task. Therefore, each Task contains more than one execution of Map or Reduce. Similar to the JobTracker, the TaskTracker monitors the execution of a Task. For example, a Task log may say "received commit of Task Id=id".

- **Attempt.** To support fault tolerance, the *Hadoop* platform allows each Task to have multiple trials of execution. Each execution is an Attempt. Typically, only when an Attempt of a Task has failed, another Attempt of the same Task will start. This restart process continues until the Task is successfully completed or the number of failed Attempts is larger than a threshold. However, there are exceptions, such as "speculative execution", which we discuss later in this chapter. The attempt is also monitored

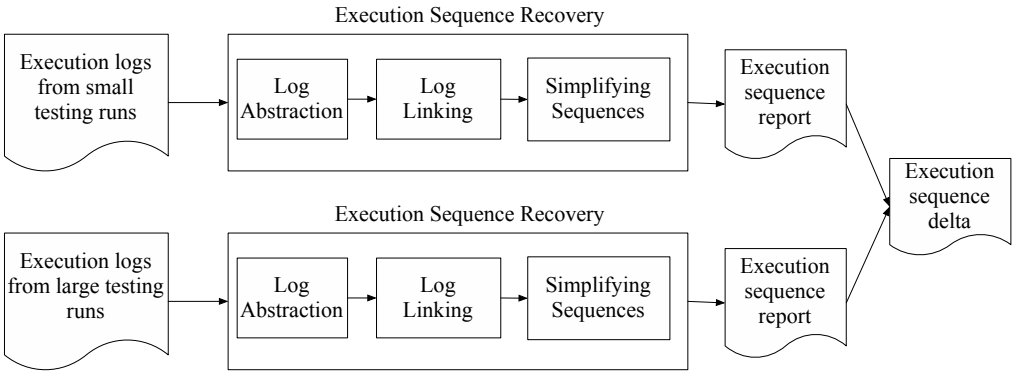


Figure 6.1: Overview of our approach.

by the TaskTracker and the detailed execution information of the Attempt, such as "Reading data for Map task with TaskID=id", is recorded in the Attempt logs.

The Job, Task and Attempt logs form the source of information used by our approach. We use the former kinds of logs instead of application-level logs since such logs provide information about the inner working of the platform itself, and not the application, which is assumed to be correctly implemented for our purposes. In particular, the platform logs provide us with information about any deployment problems.

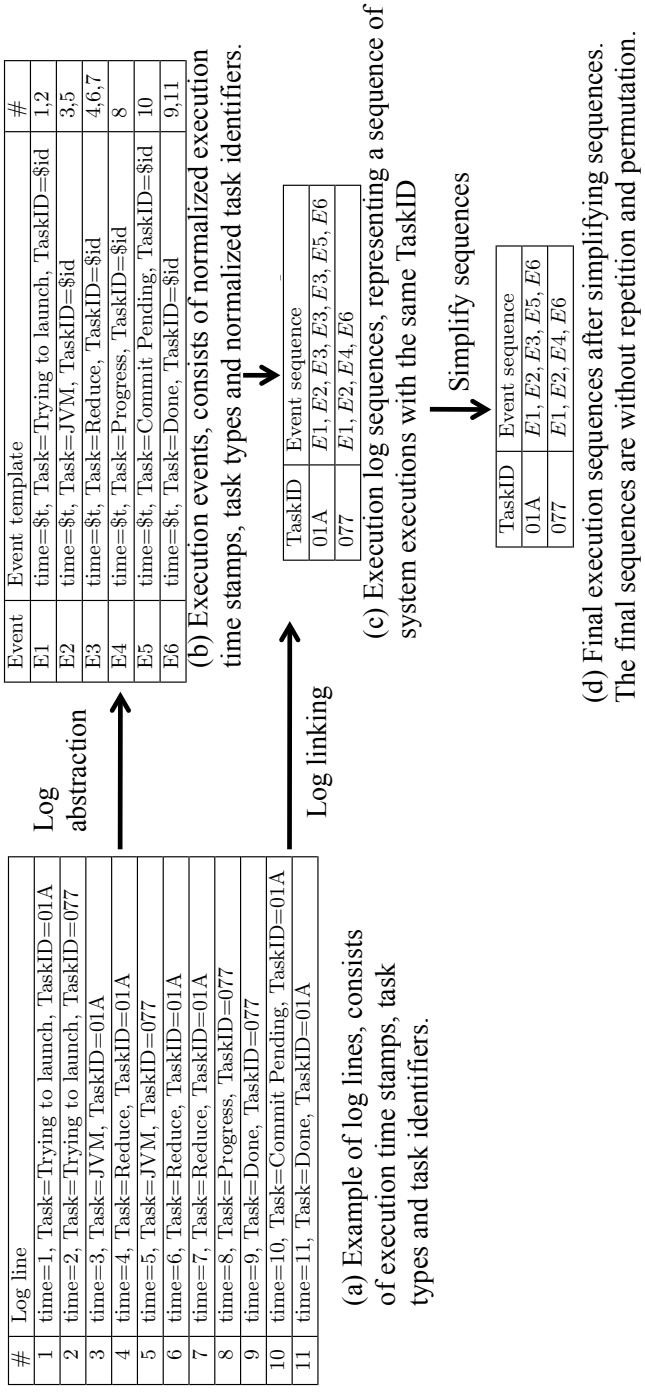
6.4 Approach

The basic idea behind our approach is to cluster the platform logs to improve their comprehensibility, and to help understand and flag differences in the run-time behaviour.

As mentioned before, our approach is based on the analysis of platform logs of BDA Apps. These logs are generated by the statements embedded by the platform developers because they consider the information to be particularly important. Containing rich knowledge, but not fully explored, platform logs typically consist of the major system activities and their associated contexts (e.g., operation ids). Logs are a valuable resource for studying the run-time behaviour of a software system, because they are generated by the internal

instrumentations and are readily available. However, previous research shows that logs are continuously changing and evolving [SJA+11]. Therefore, ad hoc approaches based on keyword search may not always work. Thus we propose an approach that does not rely on particular phrases or the format of the logs. Figure 6.1 shows an overview of our approach.

Our approach compares the run-time behaviour of the underlying platform of BDA Apps in a testing environment with a small testing data sample to the cloud environment with large scale data. To overcome the enormous number of logs generated by a BDA platform and to provide useful context for the developers looking at our results, we recover the execution sequences of the logs.



#	Log line
1	time=1, Task=Trying to launch, TaskID=01A
2	time=2, Task=Trying to launch, TaskID=077
3	time=3, Task=JVM, TaskID=01A
4	time=4, Task=Reduce, TaskID=01A
5	time=5, Task=JVM, TaskID=077
6	time=6, Task=Reduce, TaskID=01A
7	time=7, Task=Reduce, TaskID=01A
8	time=8, Task=Progress, TaskID=077
9	time=9, Task=Done, TaskID=077
10	time=10, Task=Commit Pending, TaskID=01A
11	time=11, Task=Done, TaskID=01A

Event	Event template	#
E1	time=$t, Task=Trying to launch, TaskID=$id	1,2
E2	time=$t, Task=JVM, TaskID=$id	3,5
E3	time=$t, Task=Reduce, TaskID=$id	4,6,7
E4	time=$t, Task=Progress, TaskID=$id	8
E5	time=$t, Task=Commit Pending, TaskID=$id	10
E6	time=$t, Task=Done, TaskID=$id	9,11

TaskID	Event sequence
01A	$E1, E2, E3, E3, E3, E5, E6$
077	$E1, E2, E4, E6$

TaskID	Event sequence
01A	$E1, E2, E3, E5, E6$
077	$E1, E2, E4, E6$

Figure 6.2: An example of our approach for summarizing the run-time behaviour of BDA Apps.

6.4.1 Execution Sequence Recovery

In this step, we recover sequences of the execution logs. The log sequence clustering includes three phases.

Log abstraction

Log files typically do not follow strict formats, but instead contain significant unstructured data. For example, log lines may contain the task type, the execution time stamp and free form text – making it hard to extract any structured information from them. In addition to being in free form, log lines contain static and dynamic information. The static information is specific to each particular event while the dynamic values of the logs describe the event context. We use a technique proposed by Jiang *et al.* [JHHF08a] to abstract logs. This technique is designed to be generalizable as it does not rely on any log formats. Using the technique, we first identify the static and dynamic values of the logs based on a small sample of logs. Then we apply the identified static and dynamic parts to the entire logs to abstract the logs. Figure 6.2 shows an example of a log file with 11 log lines and how we process it. Each log line contains the execution time stamp, the task type, and the task ID. The log lines are abstracted into six different system events, as shown in Figure 6.2-b. The "$id" and "$t" identifiers indicate two dynamic values.

Log linking

This phase uses dynamic values, such as "$id", to link log lines into a sequence. The linking heuristic is based on the dynamic values. In our example, TaskID is used for log linking since TaskID represents some kind of session "ID". Therefore, line 1 and line 3 in the input data in Figure 6.2-a can be linked together since they contain the same TaskID. Similar to log abstraction, we also identify the linkage among a few IDs based on a small sample of data, then apply the linking on full data.

Figure 6.2-c shows the resulting sequences after abstracting the logs and linking them into sequences using the TaskID values. Events E1, E2, E3, E5 and E6 are linked together (note that event E3 has been executed three times) and events E1, E2, E4, E6 are linked together since the same TaskID values are shared, among them.

Simplifying sequences

An example of repetition is a sequence caused by loops. For example, for sequences about reading data from a remote node, there would be repeated events about fetching the data. Without this step, similar log sequences that include different occurrences of the same event are considered different sequences, although they indicate the same system behaviour in essence. These repeated events need to be suppressed to improve the readability of the generated summaries. Therefore, we use regular expression techniques to detect and suppress the repetitions. For the example shown in Figure 6.2, our technique detects the repetition of E3 in the sequence "E1, E2, E3, E3, E3, E5, E6", and reduces this sequence to "E1, E2, E3, E5, E6".

The second step of simplifying sequences is dealing with permutations of sequences. The reason why permutations occur is that the events may execute asynchronously on the distributed computing platforms, although the corresponding sequences result in the same system behaviour. We group the permutations of a sequence together to simplify the sequences. For example, if we recovered two sequences "E1, E2, E3, E4" and "E1, E3, E2, E4", we would group these two sequences together in this step.

After simplifying sequences, we obtain the final log sequences in Figure 6.2-d.

6.4.2 Generating reports

We generate a log sequence report in HTML format. Figure 6.3 shows an example report. The report consists of two parts: an overview of the number of execution log sequences

and a list of sample log lines. To ease the comparison of different reports, each event is represented by the same unique number across the reports. An example sequence from the analyzed logs is shown in each row of the report to provide developers an actual example with a realistic context.

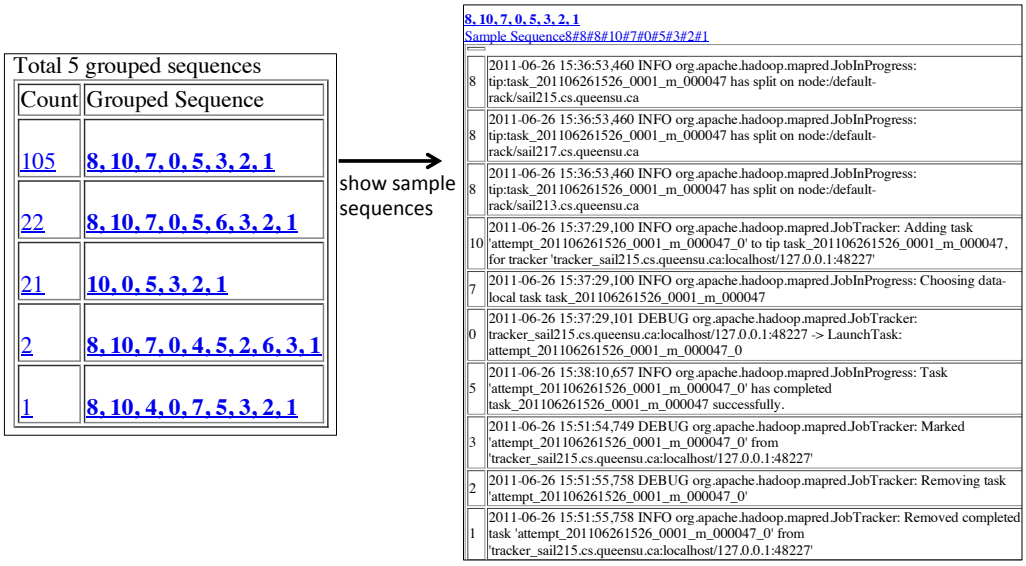


Figure 6.3: An example of our log sequences report.

6.5 Case Study

In this section, we present the design of the case study that we performed to evaluate our approach.

6.5.1 Subject applications

We use three BDA Apps as subjects for our study. Two out of the three applications are chosen to be representative of industrial BDA Apps. In addition, to avoid potential bias from the development of the applications, we chose one application that is developed from scratch and another application that is re-engineered by migrating a set of Perl scripts to the

Table 6.1: Overview of the three subject BDA Apps.

	WordCount	**PageRank**	**JACK**
Source	Hadoop: Official Example	Google: Developed from scratch	RIM: Migrated from Perl
Domain	File processing	Social network	Log analysis
Injected problem	Machine failure	Missing supporting library	Lack of disk space

Table 6.2: Overview of the BDA App's input data size.

	WordCount & JACK	**PageRank**
Large Data	$3.69GB$	$1.08GB$
Small Data	$597MB$	$10.7MB$

Hadoop platform. In addition, to ease the replication of our approach by others, we chose a third application from *Hadoop*'s official example package. The overview of the three BDA Apps is shown in Table 6.1.

- **WordCount.** *WordCount* is an application that is released with *Hadoop* as an example of the MapReduce programming. The *WordCount* application analyzes the input files and counts the number of occurrences of each word in the input files.

- **PageRank.** *PageRank* [PBMW99] is a program used by the Google Internet search engine for rating Web pages. We implemented the *PageRank* algorithm on *Hadoop*.

- **JACK.** *JACK* is an industrial application that uses data mining techniques to identify problems in load tests [JHHF08b]. This tool is used in practice on a daily basis. We migrated *JACK* to the *Hadoop* platform.

Note that none of the three above BDA Apps have their own logs and the logs that our approach uses are the platform (Hadoop) logs generated during the execution of these applications.

6.5.2 The experiment's environment setting

As input data for *WordCount* and *JACK*, we use two groups of execution log files from a large enterprise application. The input data for the *PageRank* application, however, comes from two social-network datasets from the *Stanford Large Network Dataset Collection*[2]. Table 6.2 summarizes the overall size of the input data for the studied applications.

As a proof of concept, we performed our experiments on an in-house small cloud (a computing cluster of 40 cores across five machines). Each machine has Intel Xeon E5540 (2.53GHz) with 8 cores, 12 GB memory, a Gigabit network adaptor and SATA hard drives. The operating system of the machines is Ubuntu 9.10.

6.6 Case Study Results

In this section, we present our research questions, and the results of our case study. For each research question, we present the motivation of the question, our approach to answer the question, and the results.

RQ1: How much effort reduction does our approach provide when verifying the deployment of BDA Apps in the cloud?

Motivation

Developers often use simple text search techniques to identify troublesome events when deploying BDA Apps. For example, keywords such as "kill" and "fail" are often used to find problematic tasks in Hadoop. Due to the decision by the underlying platform (e.g. algorithms that *Hadoop* uses for assigning tasks to machines), a problematic event might be caused by other reasons than an actual deployment failure. Two commonly seen examples of such reasons on the *Hadoop* platform are "Task exceptions" and "Speculative execution":

- **Task exceptions.** When there is an exception during the execution of a *Hadoop* task,

[2]http://snap.stanford.edu/data/ last verified January 2014.

the task will be killed and restarted on another cloud node. Therefore, a keyword search for "kill" on the log lines would flag such *Hadoop* decisions as a sign of failure, even though this is supposed to be transparent to developer.

- **Speculative execution.** The overall performance of a *Hadoop* Job may slow down because of some slow-running tasks. To optimize the performance, *Hadoop* replicates the unfinished tasks on idle machines. When one of the replicated tasks, or the original task, is finished, *Hadoop* commits the results from the task and kills other replicas. This mechanism is similar to the *Backup Task* in Google's MapReduce platform [DG08]. The replication and killing are decided at run-time and are not signs of deployment failures. However, again, a keyword search would flag them as a problem to be verified.

Therefore, a simple text search for such keywords may result in a very large set of irrelevant log lines for manual verification. In this research question, we investigate whether our approach saves any effort in the verification process of cloud deployments.

Approach

To evaluate our approach in terms of effort reduction, we use the number of log lines that must be examined as a basic approximation of the amount of effort. We first use the traditional (most-often-used in practice today (e.g., [FDCD12])) approach of searching for keywords in the raw log lines as a baseline for comparison. The keywords that we use in this experiment are common basic keywords ("kill", "error", "fail", "exception" and "died") that are usually a sign of failure in a log line. We applied this search on all three BDA Apps. We measure the number of log lines with these keywords as the baseline effort.

To apply our approach for deployment verification, we first recover execution sequences from the three BDA Apps, when deployed on a cloud environment. We then compare the two sets of log sequences (small-scale environment and large cloud) and identify the *delta set* (the execution sequences without an exact match). The last step involves searching

Table 6.3: Effort required to verify the cloud deployment using our approach versus the traditional keyword search.

	Using our approach		**Using keyword search**
	# execution sequences	# unique log events	#log line with keyword
WordCount	19	64	467
PageRank	55	83	1,739
JACK	20	67	726

Table 6.4: Repeated execution sequences between running the BDA Apps once, twice and three times.

	once and twice (%)	twice and three times (%)
WordCount	98.4	99.1
PageRank	95.0	95.1
JACK	99.5	99.8

for the same keywords as the traditional approach to measure the number of execution sequences and log events that are required to be examined.

Results

The results from Table 6.3 show that with our approach the number of execution log sequences (and their corresponding number of log events) to verify is 19(64), 55(83), and 20(67) for WordCount, PageRank, and JACK respectively. However, the number of raw log lines to verify after the keyword search, i.e., the traditional approach, is 467, 1739, and 726 for WordCount, PageRank, and JACK respectively. Therefore, our approach provides 86%, 95%, and 97% effort reduction over the traditional approach, ignoring the fact that verifying a log line may require more effort than verifying a log event. Indeed, verifying a log line requires checking the other log lines to get a context of the failure, whereas the log events are already shown in the context (i.e., the execution sequences). Also, notice that our approach does not incur any instrumentation overhead since the platform logs are already available.

Another interesting point is that when the input data grows, several new execution sequences and log lines appear. That is due to the fact that the behaviour is not present with

Table 6.5: Number of log lines generated by running BDA Apps once, twice and three times.

	once	twice	three times
WordCount	78 K	309 K	393 K
PageRank	109 K	217 K	474 K
JACK	237 K	419 K	666 K

the smaller runs. However, when moving to bigger runs, the execution sequences will not increase dramatically. The reason is that most of the runs, at the abstract level, are identical. Therefore, the size of the final sequences to verify will show very minor increases. However, the log lines to be verified using traditional approaches always increase in proportion to the data. Table 6.4 shows the number of repeated execution sequences when running the same BDA App once, twice, and three times. In Table 6.5, we report the number of log lines to be verified using the traditional approach for the same BDA App executions. The large portion of repeated execution sequences in the number of execution sequences versus the rapid growth in the number of log lines emphasizes the effectiveness of our approach in terms of effort reduction during verification of the deployment of a BDA App in the cloud.

Our approach reduces the verification effort by 86% to 97% when verifying the cloud deployment of BDA Apps.

RQ2: How precise and informative is our approach when verifying cloud deployments?

Motivation

As discussed in RQ1, a flagged sequence (using our approach) or a flagged log line (using the traditional approach), may be caused by some other reasons than an actual deployment failure. We consider such flagged sequences or log events, e.g., those that are related to " Task exception" and "Speculative execution", as false positive results that affect the precision of the approaches. Therefore, in this question, we compare the two approaches in terms of precision. We also discuss how our approach facilitates the verification of the flagged execution sequences.

Approach

In this research question, we categorize the flagged logs sequences by the traditional/our approach into two classes: actual failures (true positives) and platform-related events (false positives). To identify the instances of the actual failure, we intentionally injected three different failures, which are commonly observed by BDA App developers [com], into our experimental environment. These three failures have often been encountered in our experience using *Hadoop* in large industrial clouds. The three injected failures are as follows:

1. **Machine failure.** A machine failure is one of the most common system errors in distributed computing. To inject this failure, we manually turn off one machine in the cluster.

2. **Missing supporting library.** A cluster administrator may decide to expand the size of the cluster. However, the new machines in the cluster may by missing supporting libraries or the version of the supporting libraries may be outdated. We inject this failure by removing one required library.

3. **Lack of disk space.** Disks often run out of space while a BDA App is running on the platform due to the large amount of intermediate data. We manually fill up one of the machine's disks to inject this failure.

Next, we manually analyzed the log sequences in the HTML reports and identified any false positive instances.

Results

Table 6.6 summarizes the number of false positives, total number of flagged sequences/log lines, and the precision of both approaches. The precision of our approach is 21%, 38%, and 10% for WordCount, PageRank, and JACK respectively. The range of the number of false positive sequences to verify is 15 to 34 sequences (16-49 log events). However, the

Table 6.6: Number of false positives, true positives, and the precision of both our approach and the traditional keyword search.

	Using our approach			**Using keyword search**		
	false positive	true positive	precision	false positive	true positive	precision
WordCount	15	4	21%	432	35	7%
PageRank	34	21	38%	272	1467	84%
JACK	18	2	10%	650	76	10%

precision of the traditional approach is 7%, 84%, and 10% for WordCount, PageRank, and JACK respectively, while the range of the number of false positive log lines to verify is 432 to 650 log lines. A more detailed analysis of the results shows that the only case where the precision of our approach is outperformed by traditional approach is with JACK BDA App where one single exception appears in almost every log line. Though the traditional approach has a higher precision, the same exception produces 1,467 log lines that must be examined manually to determine their context and decided whether they are problematic or not. Unfortunately, because a keyword approach does not provide any abstraction all log lines would need to be examined carefully, even though they are instances of the same abstract problem.

Note that the recall for both approaches is 100%, because all instances of log lines and execution sequences related to the failures are identified by the keyword search. However, there are cases where it may not be possible to catch deployment failures by a keyword search. For example, a temporary network congestion may cause the pending queue to be very long, but logs may record that the pending queue is too long without making use of an “error” like string in the log line. In some cases, a node in the cloud may even fail without recording any error message [COR06]. In such situations, our approach is superior, because the traditional approach simply fails (as no “error” log lines are produced) and the developer would miss such problems unless he or she examines each log line. However, our recovered execution sequences still work, because it only depends on finding the delta set of sequences when switching from the small to large cloud.

Another interesting aspect of our approach, which was the initial motivation of this work, is the extra information (context of a log line) that our approach provides for deployment verification. Even after all the reduction that is performed by the keyword search approach, 467 to 1,739 log lines should be verified manually. As discussed earlier, there are false positives in the flagged log lines. Distinguishing them from true positives requires knowledge about the context of each line (otherwise both categories contain the failure-related keyword). Our approach provides such context by grouping the log events in one execution sequence, which speeds up the understanding and verification of the event.

The precision of our approach for assisting deployment verification of BDA Apps in the cloud is comparable with the precision of the traditional keyword-scan approach. However, our approach provides additional context information (execution sequences) that is essential in speeding up the manual investigation of flagged problems.

6.7 Discussion

In this section, we discuss other possible features of our approach. In particular, one feature is its ability to support developers in understanding the run-time differences when migrating BDA Apps from one platform to another.

To find the most optimal and economical platform for BDA Apps, a BDA App may need to be migrated from one Big-Data Analytics platform to another [FDCD12]. This type of re-deployment requires similar verifications as discussed in the research questions. Therefore, developers need an approach to help them in identifying any run-time behaviour changes, caused by the migration. Identifying the differences between the execution of the BDA App in the two environments may help in verifying the new deployment and flag any potential failures or anomalies.

To assess the ability of our approach to identify the potential redeployment problems

in the cloud, we migrated the *PageRank* program from *Hadoop* to Pig. Pig [ORS+08] is a Hadoop-based [Whi09] platform designed for analysis of massive amounts of data. To reduce the effort of coding in the MapReduce paradigm, Pig provides a high-level data processing language called Pig Latin [ORS+08]. Using Pig Latin, developers can improve their productivity by focusing on the process of data analysis instead of writing the boiler-plate MapReduce code [GNC+09].

We ran PageRank three times on *Hadoop* and three times on Pig. After examining the sequence reports we could note the following differences between both platforms:

1. Hadoop-based PageRank has more MapReduce steps than Pig-based PageRank

In our implementation, the Hadoop-based PageRank consists of four MapReduce steps, while the Pig-based PageRank has eight lines of Pig scripts (each line is one step in the pipeline of the data analysis). However, in the real-world execution, the Pig platform groups the eight steps of data process into three MapReduce steps. Since the *Hadoop* platform does not have a feature for grouping MapReduce steps, the execution of Hadoop-based PageRank has four MapReduce steps, just as it is written.

2. Hadoop-based PageRank has more tasks than Pig-based PageRank

We examine the distribution of sequences in both implementations of PageRank. The results show that the total number of log sequences from the Hadoop-based PageRank is much larger than the Pig-based PageRank. For example, one run of the Hadoop-based PageRank generates, in total, over 700 execution log sequences in the Task log, while this number by the Pig-based PageRank is less than 30. This result indicates that the Pig-based PageRank splits the execution into a significantly smaller number of tasks than the Hadoop-based one. The reason is that, based on the *Hadoop* instructions [Whi09], the number of Map Tasks should be set to a relatively large number. Therefore, in our case study, the number of Map Tasks is configured to 200. However, Pig optimizes the platform configurations at run-time and reduces the number of Map Tasks to a smaller number to get better performance.

Identifying such differences would be extremely difficult by only looking at the raw log

lines. Thus our approach not only assists developers of BDA Apps with the first deployment in the cloud but also helps them with any redeployments. We do note that our approach only works when the new platform is a derivative of the older platform (e.g., in the case of Pig, it provides a high-level abstraction to create programs that still run on MapReduce. Hence we can compare the MapReduce execution for the Pig program against the old MapReduce execution).

6.8 Limitations and Threats to Validity

We present the limitations and threats to validity for our approach in this section.

6.8.1 External validity

As a proof of concept, we illustrate the use of our approach to address the challenges encountered in our experience. However, there are still other challenges of developing and testing BDA Apps, such as choosing an architecture that optimizes the cost and performance [FDCD12]. Our approach may not be able to address other challenges. Additional case studies are needed to better understand the strengths and limitations of our approach.

We only demonstrate the use of our approach on Hadoop, one of the most widely adopted platforms for BDA Apps, with three injected failures. In practice, we have tried our approach on several commercial BDA Apps on other underlying platforms. The only effort for adapting our approach to other platforms of BDA Apps is to determine the parameters for abstracting and linking the platform logs. Additional studies on other open source and commercial platforms with other types of failures are needed to study the generalizability of our approach.

All our experiments are carried out on a small-scale private experimental cluster, which mimics the large cloud with 40 cores. However, a typical environment for BDA Apps has more than 1,000 cores, such as Amazon EC2 [ec2]. The logs of such large scale clouds

do lead to considerably more logs and more sequences. From our experiences using our approach in practice on such large clouds, we have found that our approach performs even better than grep since the abstraction and sequencing leads to a drastic reduction in the amount of data. Such observation was noted in our case study as well. One interesting note is that to support such large clouds we needed to re-implement our own approach to run in a cloud setting using the *Hadoop* platform (since we needed to process a very large amount of logs and summarize them into a small number of event sequences). Since our log linking is designed to be localized, for example, linking *Hadoop* task logs only needs the logs from one task, our approach is parallelizable with minimal effort.

6.8.2 Construct validity

We use abstracted execution logs to perform log clustering and to learn the run-time behaviour. However, the execution logs may not contain all the information of the run-time behaviour. Other types of dynamic information, such as execution tracing, may have more details about the execution of the BDA Apps. We use the execution logs rather than other more detailed dynamic information in this work, because execution logs are readily available and are widely used in practice (leading to no performance overhead). We leverage the logs from the underlying platform of the BDA Apps (e.g., Hadoop) instead of the logs from the Apps themselves. The purpose of this work is not to identify the bugs in the BDA Apps but rather assist in reducing the effort in deploying BDA Apps in a cloud environment. Therefore, the platform logs provide more and better information than application logs.

Identifying the re-occurrences of sub-sequences can also be used in our approach to reduce the event sequences, similar to our method of eliminating repetitions in Section 6.4. In our experience, we performed sub-sequence detection on the recovered event sequences and found that it did not suppress execution log sequences as good as our repetition elimination approach. In addition, the process of sub-sequence detection is very time consuming

and slows down the overall analysis. Therefore, we did not use the sub-sequence detection in practice. For other BDA Apps and other distributed platforms, sub-sequence detection may be effective in reducing the log sequences.

6.9 Related Work

In this section, we discuss the related work along two areas.

6.9.1 Dynamic software understanding

Fischer *et al.* [FOGG05] instrument the source code and produce different kinds of visualizations to track and to understand the evolution of software at the module level. Kothari *et al.* [KBMS08] propose a technique to evaluate the efficiency of software feature development by studying the evolution of call graphs generated by execution traces. Röthlisberger *et al.* [RGN08] implement an IDE called Hermion, which captures run-time information from an application under development. The run-time information is used to understand the navigation and browsing of source code in an IDE.

Recent work by Beschastnikh *et al.* [BBS+11] designed an automated tool that infers execution models from logs. The models can be used by developers to verify and diagnose bugs. Our techniques aim to provide context of logs when deploying BDA Apps in cloud.

In addition, Cornelissen *et al.* [CZvD+09] perform a systematic survey of using dynamic analysis to assist in program understanding and comprehension. FIELD is a development environment created by Reiss *et al.* [Rei95] that contains the features to dynamically understand the execution of a program. However, the environment is rather designed for traditional application development, rather than cloud deployment of BDA Apps.

6.9.2 Hadoop log analysis

Hadoop typically runs on large scale clusters of machines with hundreds or even thousands of nodes. As a result, large amounts of log data are generated by Hadoop. To collect and analyze the large amounts of log data from Hadoop, Boulon *et al.* built Chukwa[RK10]. This framework monitors *Hadoop* clusters in real-time and stores the log data in Hadoop's distributed file system (HDFS). By leveraging Hadoop's infrastructure, Chukwa can scale to thousands of nodes in both collection and analysis. However, Chukwa focuses more on collecting logs without the ability to perform complex analysis.

Tan *et al.* introduced SALSA, an approach to automatically analyze Hadoop logs to construct state-machine views of the platform's execution [TPK+08]. The derived state-machines are used to trace the data-flow and control-flow executions. SALSA computes the histograms of the durations of each state and uses these histograms to estimate the Probability Density Functions (PDFs) of the distributions of the durations. SALSA uses the difference between the PDFs across machines to detect anomalies. Tan *et al.* also compare the duration of a state in a particular node with its past PDF to determine if the duration exceeds a determined threshold and can be flagged as an anomaly.

Another related approach is the work of Xu *et al.* in [XHF+09b], which uses the source code to understand the structure of the logs. They create features based on the constant and variable parts of the log messages and apply the Principal Component Analysis (PCA) to detect the abnormal behaviour.

All the above approaches are designed for system operators in managing their large clusters. Our approach, on the other hand, aims to assist developers in comparing the deployed system on such large clusters against the development cloud.

6.10 Chapter Summary

Developers of BDA Apps typically first develop their application with a small sample of data in a pseudo cloud, then deploy the application in a large scale cloud environment. However, the larger data and more complex environments lead to unexpected executions of the underlying platform. Such unexpected executions and their context cannot be easily uncovered by traditional approaches.

In this chapter, we propose an approach to assist in verifying the deployment of BDA Apps. Our approach uncovers the different behaviour of the underlying platforms for BDA Apps between runs with small testing data and large real-world data in a cloud environment. To evaluate our approach, we perform a case study on Hadoop, a widely used platform, with three BDA Apps. The case study results show the strength of our approach in two aspects:

1. Our approach drastically reduces the verification effort by 86-97% when verifying the deployment of BDA Apps in the cloud.

2. The precision of our approach is comparable with the traditional keyword search approach. However, our approach reports fewer problematic logs than a keyword search. The smaller number of reported problematic logs makes it possible to manually explore them.

In addition, our approach provides additional context information (execution sequences). Based on the context information, developers can explore the execution sequences of the logs to rapidly understand the cause of problematic log lines.

Part IV

Conclusions and Future Work

CHAPTER 7

Summary, Contribution and Future Work

This chapter summarizes the main ideas presented in this thesis. In addition, we propose future work to leverage logs in other types of software engineering activities.

Logs are generated from statements inserted into the code during development to draw the attention of system operators and developers to important run-time behaviour. Such statements reflect the rich experience of system experts. The rich nature of logs has created a new market for log management applications (e.g., *Splunk*, *XpoLog* and *Logstash*) that assist in storing, querying and analyzing logs. Moreover, recent software research has demonstrated the importance of logs in understanding and improving software systems. However, practitioners often treat logs as textual data. We believe that logs have much more potential in supporting software engineering activities. To evaluate our hypothesis, we propose leveraging logs to support practitioners in maintaining and operating large scale systems. Through our case studies, we conclude that there are challenges in understanding logs in practice and that logs are co-evolving with the systems. We propose leveraging logs and their evolution to better prioritize software quality improvement efforts and assist in debugging Big Data Analytic applications. Our results and approaches to leveraging logs are valuable for software engineering practitioners.

7.1 Thesis contributions

The goal of this thesis is to better understand the challenges associated with logging statements and to propose automated approaches for leveraging logs, in a systematic fashion. We make several contributions towards this goal. These contributions are motivated by our survey of the state-of-the-art software log mining research in Chapter 2. From our literature review, we find that much software log mining research leverages logs using ad hoc approaches and focuses on assisting system operators. However, very little work focuses on assisting in software development activities. We now highlight the main contributions of

the thesis in more detail.

1. **Part 1: Studying the Challenges Associated with Understanding and Evolving Logging Statements.**

   (a) **What are the Challenges in Understanding Logging Statements?** We find that there are many challenges to understanding logs in practice. Around 80% of the logging statements provide the meaning of the log lines, while the cause, context, solution and impact of the log lines are typically not provided by the logging statements. We find that development knowledge can assist in providing more information about log lines. Our study shows that development knowledge can provide answers to resolve 24 out of 45 real-world inquiries. We also find that log line experts are often the ones who answer inquiries about log lines. We propose an automated approach that can successfully provide the development knowledge to answer real-world inquiries about 45 different log lines in the studied software systems.

   (b) **How Do Logging Statements Evolve?** We find that logs co-evolve with software systems. Our study results show that systems communicate more about their execution as they evolve. During the evolution of software systems, the logs also evolve. Especially when there are major source code changes (e.g., a new major release), the logs change significantly, although the changes of implementation ideally should not have an impact on logs. In addition, we observed 8 types of log modifications. Among the log modifications in the studied systems, less than 15% of the modifications are unavoidable and are likely to introduce errors into *Log Processing Apps*. We also find that short-lived logs typically contain system implementation-level details and system error messages. Our finding highlight the need for allocating more maintenance effort to *Log Processing Apps* and the need for tools and approaches (e.g., traceability techniques) to ease the

maintenance of *Log Processing Apps*.

2. **Part 2: Log Engineering Approaches to Support Software Development Activities.**

   (a) **Prioritizing Code Review and Testing Efforts Using Logs and Their Churn.** We are the first to establish an empirical link between logging characteristics and software defects. We find that source code files with logging statements have higher post-release defect densities than those without logging statements. We also find a positive correlation between source code files with log lines added by developers and source code files with post-release defects. Our log-related metrics complement traditional product and process metrics in explaining post-release defects. Our findings suggest that software maintainers should allocate more preventive maintenance effort on source code files with more logs and log churn, since such source code files might be the ones where developers and operators may have more doubts and concerns, and hence are more defect prone.

   (b) **Verifying the Deployment of Big Data Analytic Applications Using Logs.** We propose an approach to support large scale testing and deployment of large scale applications by assisting the deployment of Big Data Analytic applications using logs. Our approach uncovers the differing behaviour of the underlying platforms for BDA Apps between runs with small testing data and large field data in a cloud environment. Our case study results show that our approach drastically reduces the verification effort by 86-97% when verifying the deployment of BDA Apps in the cloud. The precision of our approach is comparable with the traditional keyword search approach. However, fewer problematic logs are reported by our approach compared to using keyword search, which makes it feasible to manually examine the problematic logs.

7.2 Future research

We believe that our thesis makes a major contribution towards the goal of systematically leveraging logs to support software engineering activities. However, there are many open challenges and opportunities to leverage logs in practice. We now highlight some avenues for future work.

7.2.1 Formally Investigating the Use of Logs in Software Engineering Activities

In this thesis, we relied on our literature review and our experience using logs to support software engineering activities. In the future, we intend to conduct more detailed developer and operator studies regarding the use of logs in practice.

7.2.2 Log Repository

Important software engineering data, such as code and bug reports, are systematically stored in software repositories, such as Git and JIRA. Although logs are extensively used in practice, logs are archived as raw data typically in shared network folders. How to efficiently store, manage, and further leverage such big log data is an open question. We plan to investigate approaches to systematically store such log data in a repository, manage the big log data in an efficient and effective way, and leverage the data for software practitioners.

7.2.3 Domain-specific Language for Log Mining

Logs are often analyzed using scripting languages, such as Perl and Python. Such analysis scripts are ad hoc and difficult to scale and maintain. Reusing log analysis techniques and scaling such techniques to support large scale log data is challenging. Research should explore the design of domain specific query and manipulation languages for logs to ease

the reuse of log analysis techniques. Designing the language on top of web platforms, such as Hadoop [Whi09] and Pig [ORS+08], is a promising direction, as it leads to efficient and scalable analysis.

7.2.4 System Test Planing Using Field Logs

In this thesis, we propose leveraging of logs to support code quality improvement efforts and large scale deployments. Logs can be leveraged in other software engineering activities. In particular, we plan to leverage field logs to understand system behaviour in the field. We plan to optimize system test plans based on such learned behaviour to achieve more realistic and effective system testing.

Appendix

APPENDIX A

Selection Protocol and Summary of Surveyed Papers

A.1 Selection Protocol of Surveyed Papers

There has been an extensive body of work that focuses on SLM. However, each study used different types of logs, transformation techniques and analysis techniques to achieve various goals. Therefore, there is a need to compare the prior work in order to better understand the underlying assumptions and implications. In this appendix, we briefly explain the selection protocol for our survey. The surveyed papers in this appendix are the prior research that is reviewed in our literature review chapter (Chapter 2). The related research of this thesis is beyond the selected papers for literature review. Those related research paper are presented in the related work section of each chapter (Section 3.2, 4.6, 5.3 and 6.9).

We select the following venues to search for relevant literature review:

- ACM symposium on Operating Systems Principles (SOSP)
- All conferences and workshops from the USENIX association. One of the most prestigious conferences is the symposium on Operating Systems Design and Implementation (OSDI)
- IEEE Transactions on Software Engineering (TSE)
- International Conference on Software Engineering (ICSE)
- ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)
- IEEE International Conference on Software Maintenance (ICSM)
- IEEE International Conference on Software Testing, Verification and Validation (ICST)
- International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)
- IEEE International Symposium on Software Reliability Engineering (ISSRE)

- IEEE Working Conference on Mining Software Repositories (MSR)

We then include SLM work that is cited by the papers selected from these aforementioned venues. A summary of the related work is shown in Table A.2.

A.2 Summary of Surveyed Papers

Xu *et al.* [XHF+09b] propose an approach to abstract logs by analyzing source code. They identify the logging statements from source code and create templates for each logging statement. They match log lines with the templates to perform log abstraction. Rakbin *et al.* [RXW+10] create a graphical representation to further assist in abstracting identifiers in logs.

However, source code of the systems is not always available. Jiang *et al.* [JHHF08a] propose leveraging data mining and clone detection techniques to abstract logs into events. Their approach mines large numbers of logs and group log lines by length of tokens. They analyze each group of log lines and treat the exact matched tokens as static parts of the logs. The non-matched tokens are treated as dynamic parts of the logs.

Nagappan *et al.* [NV10] leverage clustering techniques for log abstraction. They first build a frequency table that has the number of times a particular word occurs in a particular position in the log line. They identify the dynamic part of the logs by looking for the words with low frequency at each position.

Makanju *et al.* [MZHM09] use the Iterative Partitioning Log Mining algorithm to abstract logs from the BlueGene/L supercomputer logs. They first group the logs by number of tokens, then find the token position with the least number of unique values and split each group using the unique values in this token position. Afterwards, they search for bijective relationships between the set of unique tokens in two token positions to create a new partition. Finally, they abstract logs based on the number of unique tokens in each position of each partition. Due to the large number of logs and the evolution of systems, abstracting

logs repeatedly is time consuming. Therefore, Fisher *et al.* [ZFW10] propose an approach to learn the log formats incrementally.

The rich yet unstructured nature of logs has created a new market for log management applications (e.g., *Splunk* [BGSZ10], *XpoLog* [xpo] and *Logstash* [logb]) that assist in storing, querying and analyzing logs. Artemis [CCBG08] is a log mining platform that collects logs from distributed nodes, abstracts and stores data from logs into a database. The platform supports visualizations to explore the data and a plugin interface for user-defined analysis.

Lang *et al.* [Lan12] present a logging framework that gathers logs from all software or hardware in an IT company. The platform archives logs and supports future search and analysis on the logs. The platform is able is handle 100K lines of logs per second.

Jiang *et al.* [JHFH08; JHHF08b] leverage logs to detect abnormal system behaviours. Their approach transforms logs into event pairs. They identify the abnormal system behaviour by flagging the rarely occurring event pairs. Jiang *et al.* [JHHF09] also leverage logs to detect performance degradations. They link the logs into sequences and measure response times of the sequences using the time stamps associated with each log line. They flag the instances of log sequences that have higher then normal response time.

Xu *et al.* [XHF+09b; XHF+09a] propose an approach to detect system anomalies. They first abstract the logs by matching logs with the logging statements from source code. They distill the printed parameters and system state from the logs. Therefore, they extract a matrix with each vector indicating the values of parameters and the system state. They perform Principal Component Analysis to identify the usual and unusual correlations among features to detect system anomalies. They use their approach on Google's production logs. Their experience shows that their approach works well on production logs [XHF+10]. Their experiences of the advances and challenges of analyzing logs are reported [OGX12].

Salfner *et al.* [ST08] focus on processing error logs to predict system failures. Their approach includes three steps: 1) abstracting error logs into error IDs, 2) grouping error log

sequences and 3) filtering noise in the logs. Fulp *et al.* [FFH08] transform logs into support vector machines to predict system failures. They evaluate their technique by predicting the failure of nodes in a cluster with over 1000 nodes.

Zhang *et al.* [ZCHC09] model the availability of IT service using the information reported in the logs. They derive a Bayesian network structure to capture the causality between system components. Their approach is evaluated using a banking service named Douglas.

Tricaud [Tri08] the use of parallel coordinates to visualize logs for detecting security anomalies. A tool named Picviz is implemented to visualize logs by Parallel coordinates.

Lou *et al.* [LFY+10] propose transforming logs into a matrix. They first detect invariants by solving the matrix from historical logs. They analyze new logs by checking whether the vector extracted from the logs violates the invariants. The logs that violate the invariance are considered to indicate system failure.

Similarly Sandeep *et al.* [SSN+08] extract features from logs into vectors in a matrix. They leverage the matrix to train a model for system performance and use the model to predict performance based on new logs.

Tan *et al.* [TPK+08] propose SALSA, which transforms logs into a state-machine. They compare the probability distributions of the durations from one state to another across hosts to detect system anomalies. Tan *et al.* [TPK+09] further leverage the state-machine generated by SALSA to visualize control and data flow from distributed systems. The visualization is used to detect performance issues, especially in Hadoop clusters [TKGN10]. SALSA and the visualization techniques are included in a platform named ASDF [BKT+10] that aims to detect performance anomalies in large scale distributed platforms by analyzing logs and performance counters.

Yuan *et al.* [YMX+10] design a tool named SherLog to assist in the diagnosis of field failures. The tool analyzes the source code and the logs to generate what must or may happen in terms of both control flow and data flow.

Chukwa is a data collection system designed on top of *Hadoop* to collect logs and monitor system health [RK10],

Kavulia *et al.* [KTGN10] leverage logs to study the execution of a commercial super computing cluster based on Hadoop using field logs collected over a period of 10 months. They first characterize the workload and job patterns. They then study the errors during execution and find that better diagnosis and recovery approaches are needed. Finally, they propose approaches to predict performance problems and workload changes.

Wei *et al.* [XHF+09a] propose an online log analysis approach using frequent pattern mining and distribution estimation techniques to capture the dominant patterns from the system. Their technique can recover the frequent sequences of logs as well as the duration of the sequences.

Nagappan *et al.* [NWV09] propose an approach to recover the system workload from abstracted logs using Suffix Arrays and Longest Common Prefix. They first abstract the logs then build Suffix Array and Longest Common Prefix from the logs. They use the values in the Longest Common Prefix to detect patterns in logs. Their approach requires O(N) in space and time to discover all possible patterns in N events from logs.

Hassan *et al.* [HMF+08] propose leveraging the rich information in logs to generate workload information for capacity planning of large scale systems. Instead of recovering all scenarios of a system, their approach uncovers a limited set of high workload scenarios with high volume of repeating events. They compress the scenarios by compressing the repeated events in these scenarios and use the measure of compression ratio to determine the most critical scenarios.

Lou *et al.* [LFWL10] propose uncovering dependencies between system components using logs. Their approach first transforms logs into keys and then it uses co-occurrence analysis to detect dependent keys from each system component. Finally, their approach leverages Bayesian decision theory to determine the dependency between each pair of keys.

Beschastnikh *et al.* [BBE+11; BBS+11; BABE11] design a tool named Synoptic, which

infers concise and accurate system models from logs. Synoptic focuses on mining invariants from the logs, such as log A needs to follow log B. Developers used Synoptic-generated models to verify known bugs, diagnose new bugs, and increase their confidence in the correctness of their systems.

Nagappan *et al.* [NR11] transform logs into Directed Cyclic Graphs. They create a tool to transform the logs into graphs and apply existing graph-based algorithms to analyze the logs.

Fu *et al.* [FLL+13] recover system execution patterns by extracting a sequence of logs. They propose an algorithm to mine the execution patterns from the logs. From the extracted patterns, they learn essential contextual factors, such as parameter values in logs, which cause the execution of a branch of a system.

Weigert *et al.* [WHF11] propose transforming logs into a distributed graph to support large scale analysis. They perform experiments with a distributed cluster with 50 nodes and 39 million log lines, with a latency of only 10 ms.

Yuan *et al.* [YZP+11] propose a tool named Log Enhancer, which automatically adds more context to log lines. The tool analyzes the variable values that are directly included in the conditions for the logging statements. It then analyzes the data flow of these variable values to understand why these conditions hold. The tool enhances the logging statements by printing the values of variables that can still be accessed.

Jiang *et al.* [JAS+10] leverage logs to speed up the user acceptance testing. Their approach mines a repository of logs to estimate the reliability of the system under different execution scenario.

Recent work by Yuan *et al.* [YPZ12] studies the logging characteristics of four open source systems. They quantify the pervasiveness and the benefit of software logging. They also find that developers modify logs because they often cannot get the correct log message on the first attempt.

Yuan *et al.* [YPH+12] empirically study the efficacy of logging practices and find that

more than half of the failures could not easily be diagnosed using existing logs. Therefore, they design a tool called Errlog, which insert logging statements into source code in locations where the exceptions are not well-logged.

King *et al.* [KW13] perform an empirical study on the logging mechanism for electrical health record systems. They find 14 security events and 2 data elements in logs that are not explicitly addressed by system documents, highlighting the lack of traceability between logs and system documentation

Selsky *et al.* [SM05] propose a uniformed logging architecture, which uses XML to define the logging format and stores logs from multiple sources in a central storage. Such architecture potentially supports correlating logs from different sources.

Rouillard [Rou04] designs a program named SEC, Simple Event Corrector, which transforms logs into vectors of events. The program can perform various analyses on the events to detect problems during system execution.

Sah *et al.* [S+02] present a log managing system, which is designed specifically to manage large scale enterprise logs. The system collects logs from distributed nodes and supports SQL and Perl to analyze the logs.

Dunlap *et al.* [DKC+02] propose ReVirt, a virtual machine logging mechanism that provides enough information for instruction-by-instruction replaying a long-term execution of a virtual machine. Such logging mechanism is used to enable better intrusion analysis with little overhead.

Jiang *et al.* [JHP+09] study the characteristics of customer problem troubleshooting using logs of an industrial storage system. They observe that the customer problems with attached logs were resolved sooner than those without logs.

Girardin *et al.* [GB98] propose visualizing logs using a spring layout. The visualization is used to assist in the detection of anomalies and break-in attempts of a firewall system.

Glass [Gla02] designs a log monitor daemon for BSD UNIX. The log monitor can successfully recognize security holes.

Cordero *et al.* [CW08] design replayable logs for monitoring voting machines. The logs provide a comprehensive, trustworthy, replayable record of essentially everything the voter saw and did in the voting booth.

Krizak [Kri10] designs a platform that collects logs via syslog from distributed nodes. The platform supports log analysis plugins to perform various analyses on the collected logs.

Antonyan *et al.* [ADK+09] perform automated analysis on the voting machine logs. The logs are transformed into a State Diagram. Their analysis of the event logs shows that there are some deviations from the prescribed and expected use such voting terminals.

Bing *et al.* [BE00] design a program named SHARP to extend UNIX logging. The program improves monitoring of systems by extending the existing syslog infrastructure with programmable modules.

Logothetis *et al.* [LTWY11] present In-situ, a MapReduce based platform for log analysis. The platform aims to process up to 10 MB of logs per second.

Takada *et al.* [TK02] design MieLog, an interactive visual log browser. The browser supports visualization and interactive views of logs. The browser also abstract logs and support statistical analysis on logs, such as finding unusual log messages.

Stearley *et al.* [SBB12] design an approach that transforms logs into a state machine and leverages the state machine to identify failure-related logs.

Baxter *et al.* [BEO+12] design automated methods to detect errors in election audit logs. They perform statistical analysis on the voting flow recovered by logs and suggest resource allocation for the system.

Garduno *et al.* [GKT+12] propose Theia, a visualization approach that assists in program diagnosis of Hadoop cluster. Theia includes three visualization views: anomaly heat map, job execution stream and job execution detail.

Veeraraghavan *et al.* [VLW+11] propose an approach that improves logs using the simpler and faster mechanisms of single-processor record and replay, yet still achieve the scalability offered by multiple cores, by using an additional execution to parallelize the record and replay of an application.

Zawawy *et al.* [ZKM10] present a log mining framework for log reduction and interpretation. The platform abstracts and aggregates logs from distributed systems. The platform supports further analysis on the aggregated logs.

Kodre *et al.* [KZR08] leverage statistical sampling techniques on logs to identify a small fraction of test cases to execute while still retaining a high degree of confidence in the code-quality.

Mizan *et al.* [MF12] generate a layered queuing network based performance model from logs. The model can then be analyzed to locate bottlenecks in both the hardware and software.

Pecchia *et al.* [PR12] perform an empirical study based on a data set of 17,387 experiments where failures have been induced by means of software fault injection into three systems. The study shows that the system architecture, placement of the logging instructions and specific supports provided by the execution environment, significantly increase the accuracy of logs at run-time.

Banerjee *et al.* [BSC10] leverage web logs to assess the reliability of Software as a Service (SaaS) suites. They classify logs based on the effects of their outcomes on SaaS usability.

Mariani *et al.* [MP08] propose an approach to automatically identify failure causes from logs. Their approach abstracts logs into events and transforms logs into finite state machines. They compare the logs from system execution with and without failure in order to identify the failure-related logs.

Andrews *et al.* [AZ00] propose transforming software unit test logs into state machines and propose a log file analysis language for software unit testing.

Table A.1: Summary of log mining related work

Paper	Log collection	Log transformation	Log analysis	Goal
Lou[LFWL10]	platform logs	events	co-occurrence analysis bayesian decision theory	system model recovery
Lou[LFY+10]	platform logs	matrix	solving vectors	anomaly detection
Tan[TPK+08]	platform logs	state machine	comparing pair-wise distance	anomaly detection
Tan[TPK+09]	platform logs	state machine	visualization	anomaly detection
Tan[TKGN10]	platform logs	state machine	visualization	anomaly detection
Yuan[YMX+10]	logging statements application log		static analysis	anomaly detection
Yuan[YPZ12]	logging statements			empirical study
Yuan[YZP+11]	logging statements		static analysis	log improvement
Yuan[YPH+12]	logging statements		static analysis	anomaly detection
Jiang[JAS+10]	application logs	events	calculating probability	software testing
Jiang[JHHF09]	application logs	sequence	response time	software testing
Jiang[JHHF08b]	application logs	pairs	k-stat	software testing
Jiang[JHFH08]	run-time logs	events		log abstraction
Jiang[JHHF08a]	run-time logs	events		log abstraction
Boulon[RK10]	run-time logs			system monitoring

Bare[BKT+10]	platform logs	state machine	visualization	anomaly detection
Kavulya[KTGN10]	platform logs	sequence	linear regression	workload recovery anomaly detection
Oliner[OGX12]	platform logs			empirical study
Rakbin[RXW+10]	platform logs	events		log abstraction
Xu[XHF+09b]	platform logs logging statements	matrix, sequence	PCA, frequent pattern mining distribution estimation	anomaly detection log abstraction
Xu[XHF+09a]	platform logs logging statements	matrix, sequence	PCA, frequent pattern mining distribution estimation	anomaly detection log abstraction
Xu[XHF+08]	platform log logging statements	matrix	PCA	anomaly detection log abstraction
Xu[XHF+10]	platform log			anomaly detection
Bitincka[BGSZ10]	run-time logs	time series		log platform
Nagappan[NWV09]	run-time logs	suffix arrays	suffix array longest common prefix	workload recovery workload recovery
Nagappan[NR11]	run-time logs	graph	graph-based algorithms	system model recovery

Nagappan[NV10]	run-time logs	events		log abstraction
Fu[FLL+13]	run-time logs	graph	supervised classification	system model recovery
Weigert[WHF11]	run-time logs	distributed graph		system model recovery
Makanju[MZHM09]	platform log	events		log abstraction
Zhu[ZFW10]	run-time logs	events		log abstraction
Zhang[ZCHC09]	run-time logs	graph	baysian model analysis	anomaly detection
Sandeep[SSN+08]	run-time logs	matrix	random forest	anomaly detection
Cretu[CCBG08]	run-time logs	events		log analysis platform
Tricaud[Tri08]	platform log	parallel coordinates	visualization	anomaly detection
Salfner[ST08]	run-time log	sequences	filtering	anomaly detection
Fulp[FFH08]	platform logs	sequences	support vector machine	anomaly detection
Hassan[HMF+08]	run-time logs	sequences	compression	workload recovery
Beschastnikh[BABE11]	run-time logs	graph	mining invariants	system model recovery
Beschastnikh[BBS+11]	run-time logs	graph	mining invariants	system model recovery
Beschastnikh[BBE+11]	run-time logs	graph	mining invariants	system model recovery
Lang [Lan12]	platform logs			log platform
King [KW13]	application logs			empirical study
Selsky [SM05]	run-time logs	events		log platform
			correlation	

Rouillard [Rou04]	application logs	sequences	correlation	anomaly detection
			correlation	compression
Sah [S+02]	run-time logs			log platform
Dunlap [DKC+02]	platform logs			anomaly detection
Jiang [JHP+09]	platform logs			empirical study
Girardin [GB98]	run-time logs	events	visualization	anomaly detection
Glass [Gla02]	platform logs			system monitoring
Cordero [CW08]	application log			system monitoring
Krizak [Kri10]	platform logs	sequences		log platform
Antonyan [ADK+09]	application logs	state diagram		workload recovery
Rabkin [RK10]	platform logs			log platform
Bing [BE00]	platform logs			system monitoring
Logothetis [LTWY11]	run-time logs			log platform
Takada [TK02]	run-time logs	events	visualization	anomaly detection
		statistical analysis	correlation	
Stearley [SBB12]	platform logs	state machine		anomaly detection
Baxter [BEO+12]	application logs		statistical analysis	workload recovery
Garduno [GKT+12]	platform logs		visualization	anomaly detection
Veeraraghavan [VLW+11]	platform logs			log improvement

Zawawy [ZKM10]	run-time logs	events		log platform
Kodre [KZR08]	run-time logs	trees		software testing
Mizan [MF12]	run-time logs	layered queueing network		anomaly detection
Pecchia [PR12]	run-time logs			empirical study
Banerjee [BSC10]	platform logs			empirical study
Mariani [MP08]	run-time logs	finite state machine	graph diff	anomaly detection
Andrews [AZ00]	application log	finite state machine		software testing

Bibliography

[ADK+09] Tigran Antonyan, Seda Davtyan, Sotirios Kentros, Aggelos Kiayias, Laurent Michel, Nicolas Nicolaou, Alexander Russell, and Alexander Shvartsman. Automating voting terminal event log analysis. In *EVT/WOTE'09: Proceedings of the 2009 conference on Electronic voting technology/workshop on trustworthy elections*, pages 15–15, Berkeley, CA, USA, 2009. USENIX Association. [180, 185]

[AHM+08] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and William Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, 2008. [4, 11]

[AP01] Annie I. Antón and Colin Potts. Functional paleontology: system evolution as the user sees it. In *ICSE 2000: Proceedings of the 23rd International Conference on Software Engineering*, pages 421–430, Toronto, Ontario, Canada, 2001. IEEE Computer Society. [95]

[AZ00] James H. Andrews and Yingjun Zhang. Broad-spectrum studies of log file analysis. In *ICSE 2000: Proceedings of the 22nd international conference on Software engineering*, pages 105–114, New York, NY, USA, 2000. ACM. [181, 186]

[BABE11] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D. Ernst. Synoptic: Studying logged behavior with inferred models. In *ESEC/FSE '11: Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering Tool Demonstration Track*, pages 448–451, Szeged, Hungary, September 2011. [20, 177, 184]

[BBA+09] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130, New York, NY, USA, 2009. ACM. [133]

[BBE+11] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. Mining temporal invariants from partially ordered logs. In *SLAML '11: Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, pages 3:1–3:10, New York, NY, USA, 2011. ACM. [20, 177, 184]

[BBS+11] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277, New York, NY, USA, 2011. ACM. [17, 18, 20, 30, 108, 161, 177, 184]

[BDL10] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Are popular classes

more defect prone? In *FASE '10: Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering*, pages 59–73, Berlin, Heidelberg, 2010. Springer-Verlag. [111]

[BE00] Matthew Bing and Carl Erickson. Extending unix system logging with sharp. In *LISA '00: Proceedings of the 14th USENIX conference on System administration*, pages 101–108, Berkeley, CA, USA, 2000. USENIX Association. [180, 185]

[Bei84] Boris Beizer. *Software system testing and quality assurance*. Van Nostrand Reinhold Co., New York, NY, USA, 1984. [68]

[Bel99] Thoms Bell. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999. [4, 11]

[BEO+12] Patrick Baxter, Anne Edmundson, Keishla Ortiz, Ana Maria Quevedo, Samuel Rodríguez, Cynthia Sturton, and David Wagner. Automated analysis of election audit logs. In *EVT/WOTE'12: Proceedings of the 2012 international conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association. [180, 185]

[BGMD10] Christian Bird, Harald Gall, Brendan Murphy, and Premkumar Devanbu. An analysis of the effect of code ownership on software quality across windows, eclipse, and firefox. Technical Report 541, University of California, Davis, Davis, California, USA, 2010. [112]

[BGSZ10] Ledion Bitincka, Archana Ganapathi, Stephen Sorkin, and Steve Zhang. Optimizing data analysis with a semi-structured time series database. In *SLAML'10: Proceedings of the 2010 workshop on Managing systems via log analysis and machine learning techniques*, pages 7–7, Vancouver, Canada, 2010. USENIX. [1, 2, 16, 27, 61, 175, 183]

[BH10] Nicolas Bettenburg and Ahmed E. Hassan. Studying the impact of social structures on software quality. In *ICPC '10: Proceedings of the 18th International Conference on Program Comprehension*, pages 124–133, Washington, DC, USA, 2010. IEEE Computer Society. [111, 112, 116]

[BH13] Nicolas Bettenburg and AhmedE. Hassan. Studying the impact of social interactions on software quality. *Empirical Software Engineering*, 18(2):375–431, 2013. [112]

[BJ08] R.S. Brower and H.S. Jeong. Beyond description to derive theory from qualitative data. In Boca Raton, editor, *Handbook of Research Methods in Public Administration*, pages 823–839. Taylor Francis, 2008. [79]

[BKT+10] Keith Bare, Soila P. Kavulya, Jiaqi Tan, Xinghao Pan, Eugene Marinelli, Michael Kasick, Rajeev Gandhi, and Priya Narasimhan. Architecting dependable systems vii. chapter ASDF: an automated, online framework for diagnosing performance problems, pages 201–226. Springer-Verlag, Berlin, Heidelberg, 2010. [14, 176, 183]

[BKZ11] Liliane Barbour, Foutse Khomh, and Ying Zou. Late propagation in software clones. In *ICSM '11: Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, pages 273–282, Washington, DC, USA, 2011. IEEE Computer Society. [118]

[BLR10] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 375–384, 2010. [27]

[BMS03] Elisa L. A. Baniassad, Gail C. Murphy, and Christa Schwanninger. Design pattern rationale graphs: linking design to source. In *ICSE '03: Proceedings of the*

25th International Conference on Software Engineering, pages 352–362, 2003. [27]

[BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003. [88, 111]

[BSC10] Sean Banerjee, Hema Srikanth, and Bojan Cukic. Log-based reliability analysis of software as a service (saas). In *ISSRE '10: Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 239–248, Washington, DC, USA, 2010. IEEE Computer Society. [181, 186]

[BvDDT07] Magiel Bruntink, Arie van Deursen, Maja D'Hondt, and Tom Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 199–211, Vancouver, British Columbia, Canada, 2007. ACM. [32, 68]

[BW08] Raymond P.L. Buse and Westley R. Weimer. Automatic documentation inference for exceptions. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 273–282, 2008. [29]

[BW10] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42, 2010. [29]

[CCBG08] Gabriela F. Cretu-Ciocarlie, Mihai Budiu, and Moisés Goldszmidt. Hunting for problems with artemis. In *WASL'08: Proceedings of the First USENIX conference on Analysis of system logs*. USENIX Association, 2008. [175, 184]

[CMRH09] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb.

Software dependencies, work dependencies, and their impact on failures. *IEEE Transaction on Software Engineering*, 35:864–878, November 2009. [118]

[com] America's most wanted - a metric to detect persistently faulty machines in hadoop. http://hadoopblog.blogspot.com/2010/06/americas-most-wanted-metric-to-detect.html Last Verified on January 2014. [155]

[COR06] Domenico Cotroneo, Salvatore Orlando, and Stefano Russo. Failure classification and analysis of the java virtual machine. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, pages 17–, Washington, DC, USA, 2006. IEEE Computer Society. [156]

[Cor11] James R. Cordy. Excerpts from the txl cookbook. In *GTTSE '09: Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, pages 27–91, Berlin, Heidelberg, 2011. Springer-Verlag. [70]

[CTNH12] Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, and Ahmed E. Hassan. Explaining software defects using topic models. In *MSR 2012: Proceedings of 9th IEEE Working Conference of Mining Software Repositories*, pages 189–198. IEEE, 2012. [111]

[CW08] Arel Cordero and David Wagner. Replayable voting machine audit logs. In *EVT '08: Proceedings of the conference on Electronic voting technology*, pages 2:1–2:14, Berkeley, CA, USA, 2008. USENIX Association. [180, 185]

[CZD+09] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35:684–702, 2009. [1, 4, 11, 61]

[CZvD+09] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transaction on Software Engineering*, 35:684–702, September 2009. [161]

[DFL+12] Rui Ding, Qiang Fu, Jian-Guang Lou, Qingwei Lin, Dongmei Zhang, Jiajun Shen, and Tao Xie. Healing online service systems via mining historical issue repositories. In *ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 318–321, 2012. [30]

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. [66, 116, 142, 152]

[DKC+02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Operating System Review*, 36(SI):211–224, December 2002. [179, 185]

[DPH10] Wim De Pauw and Stephen Heisig. Visual and algorithmic tooling for system trace analysis: a case study. *SIGOPS Operating Systems Review*, 44(1):97–102, March 2010. [18]

[DR12] Barthélémy Dagenais and Martin P. Robillard. Recovering traceability links between an api and its learning resources. In *ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering*, pages 47–57, Piscataway, NJ, USA, 2012. IEEE Press. [27]

[ds2] Dell dvd store. http://linux.dell.com/dvdstore/ Last Verified on January 2014. [14, 68]

[eba] Ebay is powered by hadoop. http://wiki.apache.org/hadoop/PoweredBy Last Verified on January 2014. [138]

[ec2] Amazon ec2. https://aws.amazon.com/ec2/ Last Verified on January 2014. [159]

[FDCD12] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with big data analytics. *interactions*, 19(3):50–59, May 2012. [100, 138, 141, 152, 157, 159]

[FFH08] Errin W. Fulp, Glenn A. Fink, and Jereme N. Haack. Predicting computer system failures using support vector machines. In *WASL'08: Proceedings of the First USENIX conference on Analysis of system logs*, pages 5–5, Berkeley, CA, USA, 2008. USENIX Association. [18, 176, 184]

[FLL+13] Qiang Fu, Jian-Guang Lou, Qingwei Lin, Rui Ding, Dongmei Zhang, and Tao Xie. Contextual analysis of program logs for understanding system behaviors. In *MSR '13: Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 397–400, Piscataway, NJ, USA, 2013. IEEE Press. [18, 178, 184]

[FN99] N.E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. [110]

[FOGG05] Michael Fischer, Johann Oberleitner, Harald Gall, and Thomas Gschwind. System evolution tracking through execution trace analysis. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 237–246, Washington, DC, USA, 2005. IEEE Computer Society. [161]

[FWG07] Beat Fluri, Michael Wursch, and Harald C. Gall. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In

WCRE '07:Proceedings of the 14th Working Conference on Reverse Engineering, pages 70–79, Vancouver, BC, Canada, 2007. IEEE Computer Society. [95]

[FWGG09] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Control*, 17:367–394, December 2009. [95]

[GB98] Luc Girardin and Dominique Brodbeck. A visual approach for monitoring logs. In *LISA '98: Proceedings of the 12th USENIX conference on System administration*, pages 299–308, Berkeley, CA, USA, 1998. USENIX Association. [179, 185]

[GDG06] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views: Research Articles. *J. Softw. Maint. Evol.*, 18:425–456, November 2006. [12, 95]

[GG08] Michael W. Godfrey and Daniel M. Germán;. The past, present, and future of software evolution. In *FoSM: Frontiers of Software Maintenance*, pages 129–138, Beijing, China, October 2008. [96]

[Gil] Brian R Gilstrap. An introduction to the java logging api. http://www.onjava.com/pub/a/onjava/2002/06/19/log.html Last Verified on January 2014. [114]

[GJKT97] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software Evolution Observations Based on Product Release History. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 160–166, Bari, Italy, 1997. IEEE Computer Society. [94]

[GKMS00] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transaction on Software Engineering*, 26:653–661, July 2000. [113, 124]

[GKT+12] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Theia: visual signatures for problem diagnosis in large hadoop clusters. In *LISA '12: Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques*, pages 33–42, Berkeley, CA, USA, 2012. USENIX Association. [180, 185]

[Gla02] Brett Glass. Log monitors in BSD UNIX. In *BSDC '02: Proceedings of the BSD Conference 2002 on BSD Conference*, pages 14–14, Berkeley, CA, USA, 2002. USENIX Association. [179, 185]

[GNC+09] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, 2009. [158]

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in Open Source Software: A Case Study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, pages 131–142, San Jose, California, USA, 2000. IEEE Computer Society. [94]

[GWS06] RL Graham, TS Woodall, and JM Squyres. *The Practice of Programming*. 2006. [2, 104]

[hada] Hadoop. http://hadoop.apache.org Last Verified on January 2014. [116]

[hadb] Hadoop 0.18.0 release notes. http://hadoop.apache.org/docs/r0.18.0/releasenotes.html Last Verified on January 2014. [74, 79]

[hadc] Hadoop stream. http://hadoop.apache.org/common/ docs/r0.20.2/streaming.html Last Verified on January 2014. [76]

[Har01] F.E. Harrell. *Regression Modeling Strategies With Applications to Linear Models, Logistic Regression, and Survival Analysis*. Springer, 2001. [125]

[Has05] Ahmed E. Hassan. *Mining software repositories to assist developers and support managers*. PhD thesis, University of Waterloo, 2005. [52, 72, 117]

[Has08] Ahmed E. Hassan. Automated classification of change messages in open source projects. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 837–841, New York, NY, USA, 2008. ACM. [133]

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society. [111, 118]

[HBB+12] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012. [110]

[HH04] Ahmed E. Hassan and Ric C. Holt. Using development history sticky notes to understand software architecture. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 183–192, Washington, DC, USA, 2004. IEEE Computer Society. [27, 29]

[HH10] I. Herraiz and A.E. Hassan. Beyond lines of code: Do we need more complexity metrics? In Andy Oram and Greg Wilson, editors, *Making Software: What Really Works, and Why We Believe It?* OReilly Media, 2010. [112]

[HMF+08] 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 *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 713–723, Leipzig, Germany, 2008. ACM. [2, 18, 27, 33, 61, 177, 184]

[HP00] Idris His and Colin Potts. Studying the Evolution and Enhancement of Software Features. In *ICSM '00:Proceedings of the International Conference on Software Maintenance*, pages 143–151, San Jose, California, USA, 2000. IEEE Computer Society. [95]

[HW09] Daqing Hou and Yuejiao Wang. An empirical analysis of the evolution of user-visible features in an integrated development environment. In *CASCON '09:Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 122–135, Toronto, Ontario, Canada, 2009. ACM. [95]

[IBAH12] Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293–2304, 2012. [29, 113]

[ibm] Infosphere streams. http://www-03.ibm.com/software/products/en/infosphere-streams Last Verified on January 2014. [62]

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007. [139, 142]

[IG96] R. Ihaka and R. Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, pages 299–314, 1996. [117]

[JAS+10] Zhen Ming Jiang, Alberto Avritzer, Emad Shihab, Ahmed E. Hassan, and Parminder Flora. An industrial case study on speeding up user acceptance testing by mining execution logs. In *SSIRI '10: Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 131–140, Washington, DC, USA, 2010. IEEE Computer Society. [17, 178, 182]

[jboa] Jboss application server. http://www.jboss.org/jbossas Last Verified on January 2014. [116]

[jbob] Jbossprofiler. https://community.jboss.org/wiki/JBossProfiler Last Verified on January 2014. [116]

[jdt] Eclipse jdt. http://www.eclipse.org/jdt Last Verified on January 2014. [70]

[JH06] Zhen Ming Jiang and Ahmed E. Hassan. Examining the evolution of code comments in postgresql. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 179–180, Shanghai, China, 2006. ACM. [95]

[JHFH08] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *Proceedings of the 2008 The Eighth International Conference on Quality Software*, QSIC '08, pages 181–186, Washington, DC, USA, 2008. IEEE Computer Society. [175, 182]

[JHHF08a] 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: Research and Practice*, 20(4):249–267, 2008. [15, 16, 17, 69, 94, 147, 174, 182]

[JHHF08b] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *ICSM '08: Proceedings of 24th IEEE International Conference on Software Maintenance*, pages 307–316, Beijing, China, 2008. IEEE. [2, 13, 14, 15, 16, 18, 27, 61, 109, 150, 175, 182]

[JHHF09] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *ICSM '09: 25th IEEE International Conference on Software Maintenance*, pages 125–134, Edmonton, Alberta, Canada, 2009. IEEE. [2, 14, 15, 16, 17, 20, 27, 61, 109, 175, 182]

[JHP+09] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 43–56, Berkeley, CA, USA, 2009. USENIX Association. [109, 179, 185]

[jme] Apache jmeter. http://jmeter.apache.org/ Last Verified on January 2014. [68]

[JW91] J.E. Jackson and J. Wiley. *A user's guide to principal components*, volume 19. Wiley Online Library, 1991. [125]

[KBMS08] Jay Kothari, Dmitriy Bespalov, Spiros Mancoridis, and Ali Shokoufandeh. On evaluating the efficiency of software feature development using algebraic manifolds. In *ICSM '08: International Conference on Software Maintenance*, pages 7–16, Beijing, China, 2008. [95, 161]

[KKS06] Johannes Koskinen, Markus Kettunen, and Tarja Systa. Profile-based approach to support comprehension of software behavior. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 212–224, Washington, DC, USA, 2006. IEEE Computer Society. [12]

[Kri10] Paul Krizak. Log analysis and event correlation using variable temporal event correlator (vtec). In *LISA'10: Proceedings of the 24th international conference on Large installation system administration*, pages 1–11, Berkeley, CA, USA, 2010. USENIX Association. [180, 185]

[KTGN10] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *CCGRID '10: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 94–103, Washington, DC, USA, 2010. IEEE Computer Society. [15, 18, 20, 177, 183]

[KW13] Jason King and Laurie Williams. Cataloging and comparing logging mechanism specifications for electronic health record systems. In *HealthTech '13: Proceedings of the 2013 USENIX Workshop on Health Information Technologies*, Berkeley, CA, USA, 2013. USENIX Association. [179, 184]

[KZR08] Ravidutta Kodre, Hadar Ziv, and Debra Richardson. Statistical sampling based approach to alleviate log replay testing. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 533–536, Washington, DC, USA, 2008. IEEE Computer Society. [181, 186]

[Lan12] David Lang. Building a 100k log/sec logging infrastructure. In *LISA '12: Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques*, pages 203–214, Berkeley, CA, USA, 2012. USENIX Association. [175, 184]

[LFWL10] Jian-Guang Lou, Qiang Fu, Yi Wang, and Jiang Li. Mining dependency in distributed systems through unstructured logs analysis. *SIGOPS Operating System Review*, 44(1):91–96, March 2010. [17, 18, 177, 182]

[LFY+10] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association. [17, 176, 182]

[loga] Log4j. http://logging.apache.org/log4j/1.2/ Last Verified on January 2014. [30, 71, 113, 118]

[logb] logstash. http://logstash.net/ Last Verified on January 2014. [1, 19, 27, 175]

[LPF+09] Yixun Liu, D. Poshyvanyk, R. Ferenc, T. Gyimothy, and N. Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *ICSM 2009: Proceedings of the 2009 IEEE International Conference on Software Maintenance.*, pages 233–242, 2009. [110]

[LRW+97] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *Proceedings of the 4th International Symposium on Software Metrics*, pages 20–32, Albuquerque, NM, USA, 1997. IEEE Computer Society. [94, 96]

[LTWY11] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth Yocum. In-situ mapreduce for log processing. In *USENIXATC '11: Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association. [180, 185]

[MAL] Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu/ Last Verified on January 2014. [89]

[MF12] Ahmad Mizan and Greg Franks. Automated performance model construction through event log analysis. In *ICST '12: Proceedings of the 2012 IEEE Fifth*

International Conference on Software Testing, Verification and Validation, pages 636–641, Washington, DC, USA, 2012. IEEE Computer Society. [181, 186]

[MGF07] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007. [132]

[MH02] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proc. of the 24th International Conference on Software Engineering*, pages 503–512, 2002. [47, 56]

[Moc10] Audris Mockus. Organizational volatility and its effects on software defects. In *FSE '10: Proc. of the 18th ACM SIGSOFT International Symp. on Foundations of software engineering*, pages 117–126, New York, NY, USA, 2010. ACM. [128]

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering*, page 13, Washington, DC, USA, 2001. IEEE Computer Society. [84]

[MP08] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 117–126, Washington, DC, USA, 2008. IEEE Computer Society. [181, 186]

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE 2008: Proceedings of the 30th international conference on Software engineering*, pages 181–190, New York, NY, USA, 2008. ACM. [111, 134]

[MTF11] Prashanth Mundkur, Ville Tuulos, and Jared Flatow. Disco: a computing platform for large-scale data analytics. In *Erlang '11: Proc. of the 10th ACM SIGPLAN workshop on Erlang*, pages 84–89, New York, NY, USA, 2011. ACM. [142]

[MV00] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, pages 120–, Washington, DC, USA, 2000. IEEE Computer Society. [118, 133]

[MW00] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2000. [112]

[MWSO08] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, New York, NY, USA, 2008. ACM. [111]

[MZHM09] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. Extracting message types from BlueGene/l's logs. In *WASL 2008: Proceedings of the ACM SIGOPS SOSP Workshop on the Analysis of System Logss*, New York, NY, USA, 2009. ACM. [174, 184]

[NAH10] Thanh H. D. Nguyen, Bram Adams, and Ahmed E. Hassan. Studying the impact of dependency network measures on software quality. In *ICSM '10: Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. [110]

[NB05] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures

to predict system defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 284–292, New York, NY, USA, 2005. ACM. [111]

[NB07] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 364–373, Washington, DC, USA, 2007. IEEE Computer Society. [106, 111]

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 452–461, New York, NY, USA, 2006. ACM. [106, 110, 118, 124, 134]

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. [4, 11]

[NNP11] Tung Thanh Nguyen, Tien N. Nguyen, and Tu Minh Phuong. Topic-based defect prediction (nier track). In *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, pages 932–935, New York, NY, USA, 2011. ACM. [111]

[NR11] Meiyappan Nagappan and Brian Robinson. Creating operational profiles of software systems by transforming their log files to directed cyclic graphs. In *TEFSE '11: Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 54–57, New York, NY, USA, 2011. ACM. [16, 17, 178, 183]

[NV10] M. Nagappan and M.A. Vouk. Abstracting log lines to log event types for mining software system logs. In *MSR 2010: Proceedings of the 7th IEEE Working*

Conference on Mining Software Repositories, pages 114 –117, may 2010. [17, 174, 184]

[NWV09] Meiyappan Nagappan, Kesheng Wu, and Mladen A. Vouk. Efficiently extracting operational profiles from execution logs using suffix arrays. In *ISSRE'09: Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*, pages 41–50, Bengaluru-Mysuru, India, 2009. IEEE Press. [2, 16, 18, 27, 61, 177, 183]

[OA96] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transaction on Software Engineering*, 22:886–894, December 1996. [110]

[OGX12] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, February 2012. [175, 183]

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM. [158, 170]

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. [150]

[PDDF13] Daryl Posnett, Raissa D'Souza, Premkumar Devanbu, and Vladimir Filkov. Dual ecological measures of focus in software development. In *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, pages 452–461, Piscataway, NJ, USA, 2013. IEEE Press. [112]

[pgf] phfouine. http://pgfouine.projects.postgresql.org/ Last Verified on January 2014. [66]

[PNM08] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12, New York, NY, USA, 2008. ACM. [111]

[pos] Postgresql release notes of 8.3. http://www.postgresql.org/docs/8.3/static/release-8-3.html Last Verified on January 2014. [76]

[PR12] A. Pecchia and S. Russo. Detection of software failures through event logs: An experimental study. In *ISSRE '12: 2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 31–40, 2012. [181, 186]

[PTZ09] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers-Taxonomies and characteristics of comments in operating system code. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 331–341, Washington, DC, USA, 2009. IEEE Computer Society. [29]

[RD11] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500, New York, NY, USA, 2011. ACM. [112]

[RD13] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441, Piscataway, NJ, USA, 2013. IEEE Press. [111]

[Rei95] S.P. Reiss. *The Field programming environment: A friendly integrated environment for learning and development*, volume 298. Springer, 1995. [161]

[RGN08] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Exploiting Runtime Information in the IDE. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 63–72, Washington, DC, USA, 2008. IEEE Computer Society. [161]

[RK10] Ariel Rabkin and Randy Katz. Chukwa: a system for reliable large-scale log collection. In *LISA'10: Proceedings of the 24th international conference on Large installation system administration*, pages 1–15, Berkeley, CA, USA, 2010. USENIX. [20, 66, 76, 116, 162, 177, 182, 185]

[RM02] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002. [27]

[Rou04] John P. Rouillard. Real-time log file analysis using the simple event correlator (sec). In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 133–150, Berkeley, CA, USA, 2004. USENIX Association. [179, 185]

[RS11] Vladimir V. Rubanov and Eugene A. Shatokhin. Runtime verification of linux kernel modules based on call interception. In *ICST '11: Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 180–189, Washington, DC, USA, 2011. IEEE Computer Society. [12]

[rub] Rubbos. http://jmob.ow2.org/rubbos.html Last Verified on January 2014. [14]

[RXW+10] Ariel Rabkin, Wei Xu, Avani Wildani, Armando Fox, David Patterson, and Randy Katz. A graphical representation for identifier structure in logs. In

SLAML'10: Proceedings of the 2010 workshop on Managing systems via log analysis and machine learning techniques, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. [174, 183]

[S+02] Adam Sah et al. A new architecture for managing enterprise log data. In *LISA 02: Proceedings of the 16th USENIX conference on System administration*, volume 2, pages 121–132, 2002. [179, 185]

[SAH10] Weiyi Shang, Bram Adams, and Ahmed E. Hassan. An experience report on scaling tools for mining software repositories using mapreduce. In *ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 275–284, New York, NY, USA, 2010. ACM. [19]

[SAH11] Weiyi Shang, Bram Adams, and Ahmed E. Hassan. Using Pig as a data preparation language for large-scale mining software repositories studies: An experience report. *Journal of Systems and Software*, 2011. In Press. [52, 72, 116, 117]

[san] Sans consensus project - information system audit logging requirements. http://www.sans.org/security-resources/policies/info_sys_audit.pdf Last Verified on January 2014. [32]

[SBB12] Jon Stearley, Robert Ballance, and Lara Bauman. A state-machine approach to disambiguating supercomputer event logs. In *Proceedings of 2012 Workshop on Managing Systems Automatically and Dynamically*, 2012. [180, 185]

[SBS+10] Gehan M. K. Selim, Liliane Barbour, Weiyi Shang, Bram Adams, Ahmed E. Hassan, and Ying Zou. Studying the impact of clones on software defects. In *WCRE '10: Proceedings of the 2010 17th Working Conference on Reverse Engineering*, pages 13–21, Washington, DC, USA, 2010. IEEE Computer Society. [94]

[Sea99] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *Software Engineering, IEEE Transactions on*, 25(4):557 –572, Jul/Aug 1999. [31]

[Sha10] Weiyi Shang. Enabling large-scale mining software repositories (msr) studies using web-scale platforms. Master's thesis, Queen's University, 2010. [52, 72, 116, 117]

[Sha12] Weiyi Shang. Bridging the divide between software developers and operators using logs. In *ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering*, pages 1583–1586, Piscataway, NJ, USA, 2012. IEEE Press. [30]

[Shi12] Emad Shihab. *An Exploration of Challenges Limiting Pragmatic Software Defect Prediction*. PhD thesis, Queen's University, 2012. [105]

[SJA+11] 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 *WCRE '11: Proceedings of the 2011 18th Working Conference on Reverse Engineering*, pages 335–344, Washington, DC, USA, 2011. IEEE Computer Society. [27, 66, 104, 109, 145]

[SJA+13] 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. *Journal of Software: Evolution and Process*, pages n/a–n/a, 2013. [27, 109]

[SJAH09] Weiyi Shang, Zhen Ming Jiang, Bram Adams, and Ahmed E. Hassan. Mapreduce as a general framework to support research in mining software repositories (msr). In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 21–30, Washington, DC, USA, 2009. IEEE Computer Society. [52, 72, 116, 117]

[SJH+13] 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 2013 International Conference on Software Engineering*, ICSE '13, pages 402–411, Piscataway, NJ, USA, 2013. IEEE Press. [56]

[SJI+10] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *ESEM '10: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 4:1–4:10, New York, NY, USA, 2010. ACM. [94, 112, 116, 119, 124, 128]

[SM05] Matt Selsky and Daniel Medina. Gulp: a unified logging architecture for authentication data. In *LISA '05: Proceedings of the 19th conference on Large Installation System Administration Conference*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association. [179, 184]

[Smi03] Michael Smithson. *Confidence Intervals*. Sage Publications, Thousand Oaks, CA, USA, 2003. [133]

[SMK+11] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: a study of breakage and surprise defects. In *ESEC/FSE '11: Proc. of the 19th ACM SIGSOFT symp. and the 13th Euro. conf.*

on Foundations of software engineering, pages 300–310, NY, USA, 2011. ACM. [128]

[soa] Summary of sarbanes-oxley act of 2002. http://www.soxlaw.com/ Last Verified on January 2014. [2, 104]

[Sor09] S. Sorkin. Large-scale, unstructured data retreival and analysis using splunk. *Technical paper, Splunk Inc*, 2009. [141]

[spl] Splunk. http://www.splunk.com/ Last Verified on January 2014. [62]

[SPVS11] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *ICPC '11: Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, pages 71–80, 2011. [29]

[SSN+08] S. Ratna Sandeep, M. Swapna, Thirumale Niranjan, Sai Susarla, and Siddhartha Nandi. Cluebox: a performance log analyzer for automated troubleshooting. In *WASL'08: Proceedings of the First USENIX conference on Analysis of system logs*, pages 1–1, Berkeley, CA, USA, 2008. USENIX Association. [18, 176, 184]

[SSR+08] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. Defect categorization: making use of a decade of widely varying historical data. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 149–157, New York, NY, USA, 2008. ACM. [107]

[ST08] Felix Salfner and Steffen Tschirpke. Error log processing for accurate failure prediction. In *WASL'08: Proceedings of the First USENIX conference on Analysis*

of system logs, pages 4–4, Berkeley, CA, USA, 2008. USENIX Association. [17, 18, 175, 184]

[TJH+08] D. Thakkar, Zhen Ming Jiang, A.E. Hassan, G. Hamann, and P. Flora. Retrieving relevant reports from a customer engagement repository. In *ICSM 2008: Proceedings of the 24th IEEE International Conference on Software Maintenance,*, pages 117–126, 2008. [43]

[TK02] Tetsuji Takada and Hideki Koike. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *LISA '02: Proceedings of the 16th USENIX conference on System administration*, pages 133–144, Berkeley, CA, USA, 2002. USENIX Association. [180, 185]

[TKGN10] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *ICDCS '10: Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 795–806, Washington, DC, USA, 2010. IEEE Computer Society. [14, 176, 182]

[TPK+08] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: analyzing logs as state machines. In *WASL'08: Proceedings of the 1st USENIX conference on Analysis of system logs*, pages 6–6, San Diego, California, 2008. USENIX. [14, 16, 66, 108, 162, 176, 182]

[TPK+09] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Mochi: visual log-analysis based tools for debugging hadoop. In *HotCloud 2009: Proceedings of the 2009 conference on Hot topics in cloud computing*, Berkeley, CA, USA, 2009. USENIX Association. [14, 18, 176, 182]

[Tri08] Sebastien Tricaud. Picviz: finding a needle in a haystack. In *WASL'08: Proceedings of the First USENIX conference on Analysis of system logs*, pages 3–3, Berkeley, CA, USA, 2008. USENIX Association. [17, 176, 184]

[VLW+11] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *ASPLOS XVI: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 15–26, New York, NY, USA, 2011. ACM. [181, 185]

[vM03] Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society. [27]

[vMV95] A. von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995. [27]

[WHF11] Stefan Weigert, Matti Hiltunen, and Christof Fetzer. Mining large distributed log data in near real time. In *SLAML '11: Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, pages 5:1–5:8, New York, NY, USA, 2011. ACM. [178, 184]

[Whi09] T. White. *Hadoop: The Definitive Guide*. Oreilly & Associates Inc, 2009. [14, 20, 100, 138, 139, 142, 158, 170]

[Win] N Wingfield. Virtual product, real profits: Players spend on zynga's games, but quality turns some off. *Wall Street Journal*. [138]

[WSDN09] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In

ICSE '09: Proceedings of the 31st International Conference on Software Engineering, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society. [111]

[XHF+08] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Mining console logs for large-scale system problem detection. In *SysML 2008: 3rd Workshop on Tackling System Problems with Machine Learning Techniques*, pages 1–6, December 2008. [183]

[XHF+09a] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Online system problem detection by mining patterns of console logs. In *ICDM '09: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, pages 588–597, Washington, DC, USA, 2009. IEEE Computer Society. [15, 18, 19, 175, 177, 183]

[XHF+09b] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, Big Sky, Montana, USA, 2009. ACM. [15, 18, 19, 53, 108, 162, 174, 175, 183]

[XHF+10] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Experience mining google's production console logs. In *SLAML'10: Proceedings of the 2010 workshop on Managing systems via log analysis and machine learning techniques*, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association. [175, 183]

[xpo] Xpolog. http://www.xpolog.com/ Last Verified on January 2014. [1, 27, 175]

[YMX+10] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs.

In *ASPLOS XV: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 143–154, New York, NY, USA, 2010. ACM. [15, 176, 182]

[YPH+12] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 293–306, Berkeley, CA, USA, 2012. USENIX Association. [20, 178, 182]

[YPZ12] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering*, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press. [2, 15, 104, 109, 178, 182]

[YZP+11] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *ASPLOS '11: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 3–14, Newport Beach, California, USA, 2011. ACM. [15, 17, 19, 27, 30, 108, 133, 178, 182]

[ZCHC09] Rui Zhang, Eric Cope, Lucas Heusler, and Feng Cheng. A bayesian network approach to modeling it service availability using system logs. In *WASL'09: Proceedings of the Second USENIX conference on Analysis of system logs*, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association. [176, 184]

[ZFW10] Kenny Q. Zhu, Kathleen Fisher, and David Walker. Incremental learning of system log formats. *SIGOPS Oper. Syst. Rev.*, 44(1):85–90, March 2010. [175, 184]

[ZKM10] Hamzeh Zawawy, Kostas Kontogiannis, and John Mylopoulos. Log filtering and interpretation for root cause analysis. In *ICSM '10: Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–5, Washington, DC, USA, 2010. IEEE Computer Society. [181, 186]

[ZKZH12] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. An empirical study of the effect of file editing patterns on software quality. In *WCRE '12: Proceedings of the 2012 19th Working Conference on Reverse Engineering*, pages 456–465, Washington, DC, USA, 2012. IEEE Computer Society. [112]

[ZN08] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 531–540, New York, NY, USA, 2008. ACM. [110]

[ZNW10] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *ICST '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428, 2010. [132]

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society. [116]