IMPROVING THE LOGGING PRACTICES IN DEVOPS

BOYUAN CHEN

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

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

OCTOBER 2020

Abstract

DevOps refers to a set of practices dedicated to accelerating modern software engineering process. It breaks the barriers between software development and IT operations and aims to produce and maintain high quality software systems. Software logging is widely used in DevOps. However, there are few guidelines and tool support for composing high quality logging code and current application context of log analysis is very limited with respect to feedback for developers and correlations among other telemetry data.

In this thesis, we first conduct a systematic survey on the instrumentation techniques used in software logging. Then we propose automated approaches to improving software logging practices in DevOps by leveraging various types of software repositories (e.g., historical, communication, bug, and runtime repositories). We aim to support the software development side by providing guidelines and tools on developing and maintaining high quality logging code. In particular, we study historical issues in logging code and their fixes from six popular Java-based open source projects. We found that existing state-of-the-art techniques on detecting logging code issues cannot detect a majority of the issues in logging code. We also study the use of Java logging utilities in the wild. We find the complexity of the use of logging utilities increases as the project size increases. We aim to support the IT operation side by enriching the log analysis context. In particular, we propose a technique, LogCoCo, to systematically estimate code coverage via executing logs. The results of LogCoCo are highly accurate under a variety of testing activities. Case studies show that our techniques and findings can provide useful software logging suggestions to both developers and operators in open source and commercial systems.

To my mother, my father, and Rita.

Acknowledgements

This thesis would not be completed without the help and support from the extraordianry people to whom I am deeply grateful.

I would like to thank my parents and my family. No matter where I am, they are always there to support me. They always tell me not to worry about anything but only to pursue what I really love to do. Thank you for being there for me at every stage of my life.

I would like to thank my supervisor, Professor Zhen Ming (Jack) Jiang for his constant guidance, support, and encouragement through these years. He provides me invaluable help both in academic and personal life during this journey. He is not only a supervisor but also a life-long friend to me. I feel very lucky to work with him.

I would like to thank my PhD supervisory committee: Dr. Marin Litoiu and Dr Song Wang. Their continued guidance, critique, and support are very valuable to my work.

I am very lucky to work with many of the talented and diligent researchers during my PhD research. I would like to express my gratitude to my collegues and collaborators, including Ruoyu Gao, Yangguang Li, and Minke Xiu.

I am grateful to be able to have the opportunity to apply my research in practice. I would like to thank the Baidu company and members of Baidu Cloud Testing team. They provide me many help when I was doing the internship in Beijing. In particular, I want to thank Jian Song, Peng Xu, and Xing Hu for providing me with valuable feedback and critique. The environment gives me a lot of confidence in seeking to apply my research outcomes to industry practice.

During this journey, I received much love from my friends and their families, including Feihong Liu, Ruoxin Pan, Jeff Wang, Shinnosuke Okada, Jingbo Zhao, Jinfu Chen, Anni Siren, Yiming Yin, Shanshan Li, Dongliang Liao, Rui Zheng, Pengcheng Wang, Jianwei Hu, Xirui Tang, Sinan Zhang, and their families. Thank you for making my PhD journey so enjoyable.

Related Publications

The following publications are related to this thesis:

1. **Boyuan Chen**. *Improving the software logging practices in DevOps.* In Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pages 194-197. Montreal, Canada. May, 2019. [Chapter 1]

2. **Boyuan Chen** and Zhen Ming (Jack) Jiang. *Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems.* In Empirical Software Engineering (EMSE), pages 2285-2322. August, 2019. [Chapter 3]

3. **Boyuan Chen** and Zhen Ming (Jack) Jiang. *Studying the Use of Java Logging Utilities in the Wild.* In Proceedings of the 42nd International Conference on Software Engineering (ICSE), July 2020. [Chapter 4]

4. **Boyuan Chen**, Jian Song, Peng Xu, Xing Hu and Zhen Ming (Jack) Jiang. *An Automated Approach to Estimating Code Coverage Measures via Execution Logs.* In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), pages 305-316. September, 2018. [Chapter 5]

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

DevOps is a software development methodology that intends to automate the process between software development and IT operations. The goal is to reduce the time between committing a change to a system and placing it to production, while ensuring high quality [1]. Compared to traditional software development process, DevOps provides faster feedback between software development and IT operations so that new features and bug fixes can be released faster to the customers. To ensure the quality and the health of the deployed systems, software logging plays a central role.

Software logging in the context of DevOps refers to the practices of developing and maintaining logging code and analyzing the resulting execution logs. Logging code refers to the code snippets that developers inserted into source code (e.g., `LOG.info("User " + userName + " logged in")`) to monitor the behavior of systems during runtime. There are typically four types of components in a snippet of logging code: a logging object, a verbosity level, static texts, and dynamic contents. In the above example, the logging object is `LOG`, the verbosity level is `info`, the static texts are `User` and `logged in`, and the dynamic content is `userName`. Execution logs (a.k.a., logs), which are generated by logging code during runtime, are readily available in large-scale software systems for many purposes like system monitoring [2], problem debugging [3], workload characterization [4], and business decision making [5]. Stale or incorrect logging code may cause confusion [6] or even more serious issues like system crash [7]. In particular, there are three major challenges associated with the software logging practices in DevOps:

- C_1: **No existing guidelines on producing high quality logging code.** Recent empirical studies show that there are no existing logging guidelines for commercial [8] and open source systems [9, 10]. Developers write logging code solely based on domain expertise and revise them in an ad-hoc fashion [9, 10]. Unlike feature code, which can be examined through testing, it is very challenging to verify the correctness of logging code.

- C_2: **Difficulty in maintaining and evolving logging code.** As logging code tangles with source code, it is very challenging to maintain and update logging code along with feature code for constantly evolving systems. Although there are language extensions (e.g., AspectJ [11]) to support better management of logging code, many industrial and open source systems still choose to inter-mix logging code with feature code [9, 10].

- C_3: **Limited mechanism for quality feedback.** In the context of DevOps, the software testing process is completely changed compared to traditional software development process, as many testing activities are automated and occur in the field [12]. There is limited mechanism for quality feedback from the IT operation to the software development. This problem becomes even more serious, as in DevOps code base evolves more rapidly with usage scenarios being constantly added or modified.

Motivated by the importance and challenges, throughout the thesis, we propose systematic approaches to improving the logging practices to aid software development and IT operations by leveraging various types of software repositories. To note, the resulting techniques and findings are especially useful for systems that adopt DevOps practices, but they can be useful for other types of systems as well.

1.2 Research Hypothesis

Research Hypothesis: Software repositories (e.g., code repositories, communication repositories, runtime repositories, and bug repositories) which are readily available and contain rich information about software development and system behavior during runtime, can be leveraged to improve the logging practices by understanding the current logging practices of software systems.

We mainly rely on four types of repositories to tackle challenges in software logging practices. The *historical repositories* refer to the source code version control systems like GitHub and SVN. The *communication repositories* refer to the online communication data from StackOverflow and developer mailing list. The *runtime repositories* refer to the telemetry data generated in various scenarios. The *bug repositories* refer to the issue tracking systems such as JIRA and BugZilla. In the thesis, we attempt to improve logging practices from two dimensions: software development and IT operation. For the aforementioned three challenges (C_1 - C_3), we propose corresponding research outcomes (O_1 - O_3), which address these challenges. O_1 and O_2 address the first two challenges which are on the development side. O_3 address the last challenge which is on the IT operation side.

- (O_1): We mine the historical and bug repositories to extract a benchmark dataset which contains real-world issues in logging code so that interested researchers could develop and evaluate their techniques of automated detection of logging code issues.

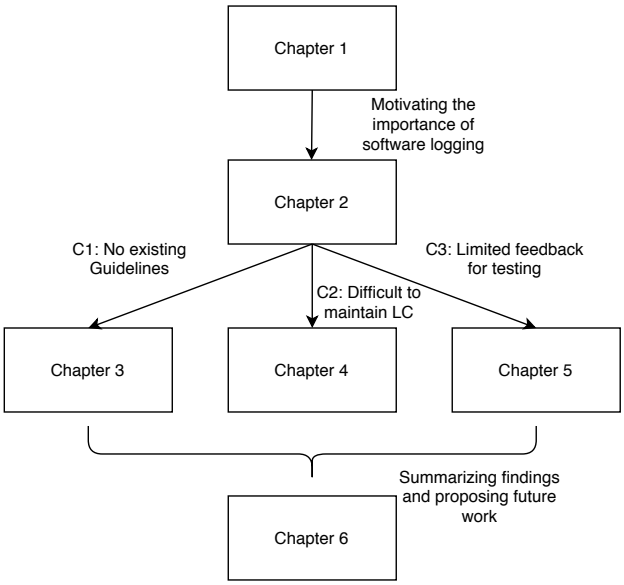

Figure 1.1: Overview of the thesis.

- (O_2): We mine the communication and historical repositories to explore the rationle behind various types of logging utilities (e.g., general-purpose, LU interaction, internationalization, and modularization).
- (O_3): We propose automated techniques to estimate code coverage measures(e.g., statement coverage, branch coverage) by correlating source code with the logs stored in the runtime repositories.

1.3 Thesis overview

In this section, we provide an overview of the presented work in this thesis. This thesis has four main chapters. As every chapter tackles a specific problem, we ensure that the contents of each chapter is self-contained. Hence, repetitions could be found among different chapters. In addition, for each chapter, the related work are summarized for the specific objective. The overview is shown in Figure 1.1. In Chapter 1(this chapter), we describe the motivation of this thesis. In Chapter 2, we survey the current practice of software log instrumentation and summarize three challenges. In Chapter 3, 4,and 5, we conduct studies for each challenge. At last, in chapter 6, we summarize our findings and propose future work.

- Chapter 2: A Survey of the Instrumentation Techniques Used in Software Logging

  Execution logs have been used widely for many software systems for a variety of purposes (e.g., monitoring, debugging, and security compliances). There are two phases in software logging: log instrumentation and log management. Log instrumentation

refers to the practice that developers insert logging code into source code to record runtime information. Log management refers to the practice that operators collect the generated log messages and conduct data analysis techniques to provide valuable insights of runtime behavior. Unlike log management, which is supported by many open source and commerical tools, there are no well-defined guidelines on log instrumentation. The quality of instrumented logging code snippets plays a vital role in software logging, as it impacts the quality of generated log messages, which will be used for different objectives. Hence, we conducted a systematic survey on state-of-the-art research on software log instrumentation by studying 69 papers over 23 years. In particular, we have focused on the problems and proposed solutions used in the three steps of log instrumentation: (1) logging approach; (2) logging utility integration; and (3) logging code composition. This survey will be useful to DevOps engineers and researchers who are interested in software logging.

- Chapter 3: Extracting and Studying the Logging-Code-Issue-Introducing Changes in Java-based Large-Scale Open Source Software Systems

  Execution logs, which are generated by logging code, are widely used in modern software projects for tasks like monitoring, debugging, and remote issue resolution. Ineffective logging would cause confusion, lack of information during problem diagnosis, or even system crash. However, it is challenging to develop and maintain logging code, as it inter-mixes with the feature code. Furthermore, unlike feature code, it is very challenging to verify the correctness of logging code. Currently developers usually rely on their intuition when performing their logging activities. There are no well established logging guidelines in research and practice.

  In this chapter, we intend to derive such guidelines through mining the historical logging code changes. In particular, we have extracted and studied the Logging-Code-Issue-Introducing (LCII) changes in six popular large-scale Java-based open source software systems. Preliminary studies on this dataset show that: (1) both co-changed and independently changed logging code changes can contain fixes to the LCII changes; (2) the complexity of fixes to LCII changes are similar to regular logging code updates; (3) it takes longer for developers to fix logging code issues than regular bugs; and (4) the state-of-the-art logging code issue detection tools can only detect a small fraction (3%) of the LCII changes. This highlights the urgent need for this area of research and the importance of such a dataset.

- Chapter 4: Studying the Use of Java Logging Utilities in the Wild

  Software logging is widely used in practice. Execution Logs have been used for a variety of purposes like debugging, monitoring, security compliance, and business analytics. Instead of directly invoking the standard output functions, developers usually prefer to use logging utilities (LUs) (e.g., `SLF4J`), which provide additional functionalities like thread-safety and verbosity level support, to instrument their source code. Many of the previous research work on software logging are focused on the log printing code. There

are very few work studying the use of LUs, although new LUs are constantly being introduced by companies and researchers. In this chapter, we conducted a large-scale empirical study on the use of Java LUs in the wild. We analyzed the use of 3,856 LUs from 11,194 projects in GitHub and found that many projects have complex usage patterns for LUs. For example, 75.8% of the *large*-sized projects have implemented their own LUs in their projects. More than 50% of these projects use at least three LUs. We conducted further qualitative studies to better understand and characterize the complex use of LUs. Our findings show that different LUs are used for a variety of reasons (e.g., internationalization of the log messages). Some projects develop their own LUs to satisfy project-specific logging needs (e.g., defining the logging format). Multiple uses of LUs in one project are pretty common for *large* and *very large*-sized projects mainly for context like enabling and configuring the logging behavior for the imported packages. The findings and the implications presented in this chapter will be useful for developers and researchers who are interested in developing and maintaining LUs.

- Chapter 5: An Automated Approach to Estimating Code Coverage Measures via Execution Logs

  Software testing is a widely used technique to ensure the quality of software systems. Code coverage measures are commonly used to evaluate and improve the existing test suites. Based on our industrial and open source studies, existing state-of-the-art code coverage tools are only used during unit and integration testing due to issues like engineering challenges, performance overhead, and incomplete results. To resolve these issues, in this chapter we have proposed an automated approach, called *LogCoCo*, to estimating code coverage measures using the readily available execution logs. Using program analysis techniques, LogCoCo matches the execution logs with their corresponding code paths and estimates three different code coverage criteria: method coverage, statement coverage, and branch coverage. Case studies on one open source system (HBase) and five commercial systems from Baidu and systems show that: (1) the results of LogCoCo are highly accurate ($> 96\%$ in seven out of nine experiments) under a variety of testing activities (unit testing, integration testing, and benchmarking); and (2) the results of LogCoCo can be used to evaluate and improve the existing test suites. Our collaborators at Baidu are currently considering adopting LogCoCo and use it on a daily basis.

1.4 Thesis Contribution

This thesis makes the following contributions:

1. We conduct the first survey that systematically covers the techniques used in all three software log instrumentation approaches: (1) conventional logging, (2) rule-based

logging, and (3) distributed tracing. The replication package for this chapter is provided in [13] (Chapter 2).

2. We provide the first dataset (to the best of our knowledge) on logging code issue introducing changes, combining both co-changed logging code changes, and independent logging code changes. We conduct a comprehensive study on evaluation of state-of-art logging suggestion tools. It includes machine learning models, static code rules, and code clone. Study shows that they are complementary to each other while future improvements can be explored. The replication package for this chapter is provided in [14] (Chapter 3).

3. We conduct the first large-scale and comprehensive study on characterizing the use of logging utility in Java-based open source projects. We show that as the complexity of projects increase, the problem of designing logging utilities become more challenging. We also extracted the most popular logging utilities based on the usages of open source projects. The replication package for this chapter is provided in [15] (Chapter 4).

4. We conduct the first work to leverage execution logs to estimate code coverage. We propose a research prototype, LogCoCo. Results show that LogCoCo achieves satisfying performance. (Chapter 5)

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 presents our literature review on software log instrumentation. Chapter 3 describes our study on Logging-Code-Issue-Introducing changes. Chapter 4 presents our study on the use of logging utilities in the wild. Chapter 5 explains our technique to estimate code coverage using execution logs. Chapter 6 concludes this thesis and discusses future research directions.

Chapter 2

A Survey of the Instrumentation Techniques Used in Software Logging

2.1 Introduction

Software logging is a common programming practice that developers use to track and record the runtime behavior of software systems. Software logging has been used extensively for monitoring [3, 16], failure diagnosis [17, 18], performance analysis [19, 20, 21, 22], test analysis [23, 24], security and legal compliance [25, 26, 27], and business analytics [5, 28].

As shown in Figure 2.1, software logging [29] consists of two phases: (1) **Log Instrumentation**, and (2) **Log Management**. The log instrumentation phase, which concerns about the development of the logging code, consists of three steps: *Logging Approach*, *Logging Utility (LU) Integration*, and *Logging Code (LC) Composition*. The log management phase, which focuses on processing the generated log messages once the system deploys, consists of three steps: *Log Generation*, *Log Collection*, and *Log Analysis*.

On one hand, many industrial-strength tools (e.g., LogStash [30] and Splunk [31]) are available already to aid effective log management. On the other hand, there are no well-defined

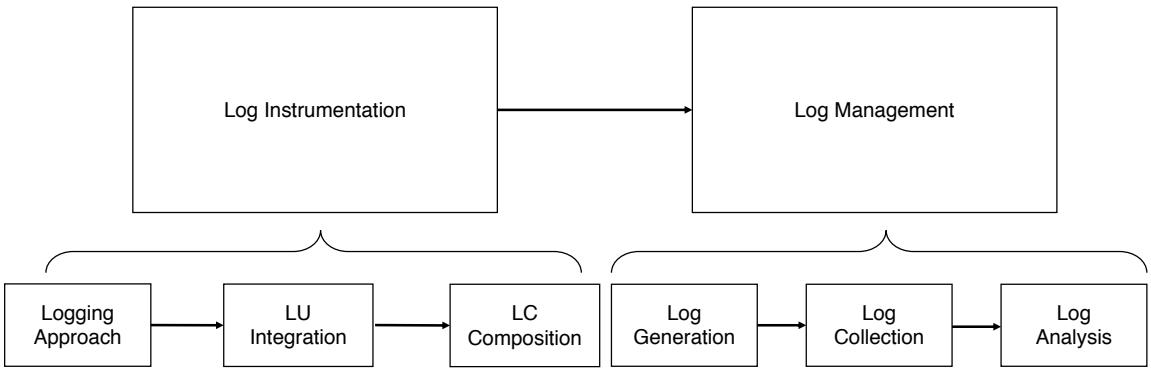

Figure 2.1: The overall process of software logging.

guidelines or tooling support for systematic log instrumentation [32, 5, 8]. Compared to code instrumentation in general, there are three differences from log instrumentation which makes the process more challenging : (1) existing process for log instrumentation is ad-hoc and often relies on developers' intuitions and domain expertise [10]; (2) many tools levearging code instrumentation such as Jacoco [33] are at bytecode/binary level, while log instrumentation is mainly done at source code level, which is more intrusive and difficult to maintain; (3) while code instrumentation at bytecode/binary level mostly record machine readable data, log instrumentation also needs to consider the human aspect. For example, JBoss logging [34] provides utilities for translating log messages to multiple human languages.

Low quality logging code may cause various issues in problem diagnosis [35], high maintenance efforts [36, 37, 38], performance slow-down [21], or even system crashes [7]. As more systems are migrating to the cloud [39] with increasing layers of complexity [40], new software development paradigms like Observability-Driven Development (ODD) [41] are introduced. ODD, in which log instrumentation plays a key role, emphasizes the exposure of the state and the behavior of a System Under Study (SUS) during runtime. Hence, in this chapter, we have conducted a systematic survey [42] on the instrumentation techniques used in software logging.

The contributions of this chapter are:

- This is the first survey that systematically covers the techniques used in all three software log instrumentation approaches: conventional logging, rule-based logging, and distributed tracing.
- Through the process of systematic survey ([43]), we have identified four main challenges associated with software log instrumentation and described their proposed solutions throughout the three steps in the log instrumentation phase. We have also discussed the limitations and future work associated with these state-of-the-art solutions if applicable. The challenges, the solutions, and the discussions will be useful for both practitioners and researchers who are interested in developing and maintaining software logging solutions.

Chapter Organization

The structure of this chapter is organized as follows: Section 2.2 provides some background regarding software log instrumentation. Section 2.3 describes the overview of our systematic survey process and the summary of studied papers. Section 2.4 discusses three approaches to software log instrumentation. Section 2.5 discusses studies on LU integration. Section 2.6 presents studies on LC composition. Section 2.7 concludes this survey.

2.2 Background on Software Log Instrumentation

In this section, we will provide some background information regarding software log instrumentation. Software logging consists of two phases: (1) **Log Instrumentation**, which

concerns about the development of the logging code, and (2) **Log Management**, which concerns about the collection and the analysis of the generated log messages. Here we further explain the three steps in the log instrumenation phase, which is the focus of this survey:

1. *Logging Approach*: Logging is a cross-cutting concern, as the LC snippets are scattered across the entire system and tangled with the feature code [44]. In addition, logging incurs performance overhead [21] and if not careful may slow down the system execution and impact user experience. Hence, additional logging approaches have been proposed to resovle some of these issues. However, they also introduce additional problem(s). For example, although Aspect-Oriented Programming (AOP) improves the modularity of the LC snippets, it introduces steep learning curves of different programming paradigms and is difficult to generalize individual logging concerns into rules [45]. Developers have to first decide which logging approach to adopt for their SUS before instrumenting their SUS with LC.

2. *LU Integration*: Instead of directly invoking the standard output functions like `System.out.print`, developers prefer to instrument their systems using LUs like SLF4J [46] for Java and spdlog [47] for C++ for additional functionalities like thread-safety (synchronized logging in multi-threaded systems) and verbosity levels (controlling the amount of logs outputted). While integrating LUs, developers have to address the following two concerns: *what-to-adopt*: different LUs provide different functionalities [45]. Depending on the actual usage context, developers may integrate existing third party LUs or develop their own; and *how-to-configure*: each project may contain one or more LU(s), each of which has many different configuration options. It is important to configure LUs effectively, so that the SUS can produce high quality log messages during runtime.

3. *LC Composition*: Once the developers integrate the LU(s), they need to insert LC into the SUS in order to expose various state and behavior of the SUS during runtime. While composing LC, developers have to address the following three concerns: *where-to-log*: determining the approprite logging points; *what-to-log*: providing sufficient information in the LC; and *how-to-log*: developing and maintaining high quality LC.

2.3 An Overview of Our Systematic Survey

In this section, we will first describe our systematic survey process in Section 2.3.1. Then we will summarize our survey results in Section 2.3.2. Finally, we will explain the differences of our survey against existing work in Section 2.3.3.

2.3.1 Our Process

A systematic survey is a type of literature review, which uses systematic approaches to identifying and analyzing the primary studies related to a particular research topic [43]. The

main benefits of conducting a systematic survey are : (1) the results of selected studies are less likely to be biased, since it applies a pre-defined search strategy; (2) the search process is documented so that the study could be easily replicated.

Here we briefly describe our systematic survey process. In addition to "logging" and "instrumentation", we also include the word "tracing" as our search keywords, as "logging" and "tracing" are used interchangeably in the research literature and practice. For example, "trace" is usually a common verbosity level defined in many LUs to capture information flow through the SUS [48, 49]. In addition, many tracing frameworks [50, 51, 22] also use the term "logging" to record various runtime behavior of the SUS. Hence, in order to capture all the possible related work, we search IEEE Xplore, ACM, and DBLP publication repositories with the root form of these three words: *log*, *trace*, and *instrument*. We then manually check the search results by employing the following inclusion and exclusion critera:

1. We exclude all the papers that only study the issues in the log management phase. For example, there are many studies focusing on log analysis (e.g., [52, 53, 19]) and log abstraction (e.g., [54, 55]), which are not relevant to this survey.

2. We only include the papers that are published in software engineering or computer systems related venues, as our target audience is software practitioners or researchers.

3. Since we are only foucusing on SUS, which sits on top of operating systems and are connected by computer networks, we will exclude papers that focus on the logging at the kernel (e.g., [56]) or network levels (e.g., [57]).

After we gather the first batch of papers, we further apply the snowballing method [58] from the references of these papers as well as looking through the papers which cite them. This process results in a total of 69 papers, which match our criteria. To verify the completeness of the surveyed papers, the final results include all the papers we knew beforehand that are related to software log instrumentation (e.g., [10, 36, 8]).

2.3.2 Summary

We performed our paper collection process on October 30, 2019. Figure 2.2 illustrates the number of related papers between these 23 years (1997 - 2019), as the first research paper in this area [59] appears in 1997. There is a clear increasing trend in terms of number of research papers over the years on this topic. In particular, the research interests in this area spiked after 2012, as 62% (43) of the studied papers have been published since then.

After carefully studying each paper, we have identified nine challenges, which are further grouped into the following four major categories. Note that some papers may touch on multiple challenges.

1. **Usability** refers to the log instrumentation techniques that facilitate the adoption of various logging techniques. There are two specific challenges in this category:

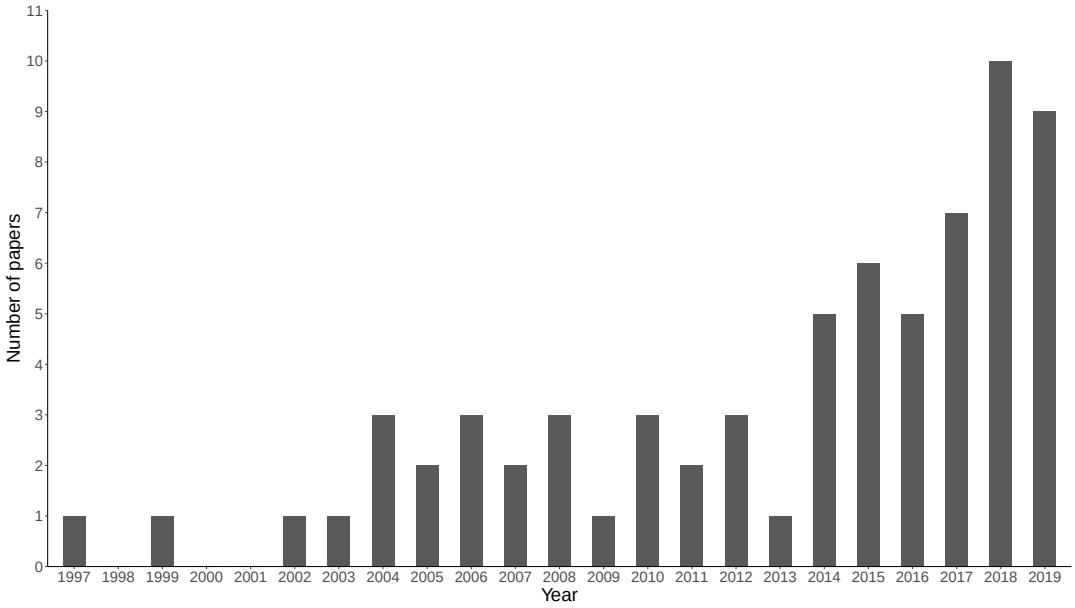

Figure 2.2: Software log instrumentation papers from 1997 to 2019.

(a) *Configurability* (2 papers) refers to the challenge on whether the studied paper provides support to ease the configuration process of various log instrumentation techniques.

(b) *Performance Overhead* (4 papers) refers to the challenge on whether the study aims to minimize the slowdown caused by logging.

2. **LC Quality** refers to the log instrumentation techniques that improve various development aspects of LC. This category consists of the following three challenges:

   (a) *Clarity* (3 papers) refers to the challenge on making LC easy to understand and less ambiguous to both developers and operators.

   (b) *Maintainability* (17 papers) refers to the challenge on supporting the maintenance and evolution of LC, as LC is scattered across the entire system and tangled with the constant evolving feature code.

   (c) *Consistency* (10 papers) refers to the challenges on ensuring uniform styles of logging across different components of SUS.

3. **Diagnosability** refers to the log instrumentation techniques that support the analysis and debugging tasks of various functional and non-functional problems. This category consists of the following two challenges:

   (a) *Failure Diagnosis* (12 papers) refers to the challenges associated with providing sufficient logging information to diagnose functional failures.

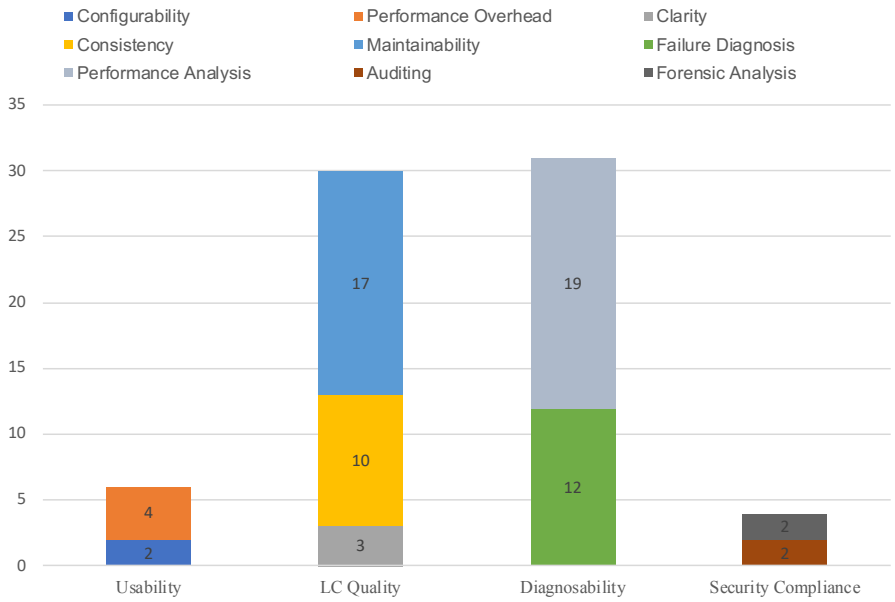

Figure 2.3: Paper distribution classified by associated challenges.

(b) *Performance Analysis* (19 papers) refers to the challenges associated with providing sufficient logging information to detect and debug performance problems.

4. **Security Compliance** refers to the log instrumentation techniques that address the safety or legal concerns of SUS. This category consists of the following two challenges:

   (a) *Auditing* (2 papers) refers to the challenge on recording a serial of security relevent events in order to meet various legal regulations like Sarbanes-Oxley Act of 2002 [25].

   (b) *Forensic Analysis* (2 papers) refers to the challenges on recording the user activities to support investigations on criminal activities like intrusion or fraud detection.

Figure 2.3 shows the distribution of studied papers by the challenges they tackle. Note that they do not add up to 69, as some papers tackle multiple challenges. Majority of them focus on the diagnosability (45%) or LC quality (43%) aspects. Few papers tackle the usability (9%) or security (6%) challenges associated with logging.

In the next three sections (Section 2.4,2.5, and 2.6), we will explain the proposed solutions and discuss their limitations throughout the three steps of the log instrumentation phase.

2.3.3 Comparing Against Existing Surveys

There are three existing work [60, 61, 62] which are related to our survey.

Rong et al. [60] conducted a systematic review on the log instrumentation practices of conventional logging. However, they did not cover other logging approaches or discuss the

Table 2.1: Comparison among three logging approaches.

	Conventional Logging	**Rule-based Logging**	**Distributed Tracing**
Who	SUS developers	SUS developers	Library developers
Filtering	Verbosity level	Verbosity level	Sampling
Format	Free form	Free form	Structured
Domain	Genearl	General	Distributed systems
Flexibility	High	Low	Medium
Scattering	High	Low	Low

techniques in LU intergration. Furthermore, compared to [60], we have placed each studied paper in the context of associated challenges so that practitioners or researchers can easily locate the relevant techniques to adopt and understand their limitations if any.

Sambasivan et al. [61] conducted a survey on distributed tracing systems, which are logging frameworks to support monitoring and diagnose problems for distributed systems. They examined the features of 15 frameworks (e.g., preserving causal relationships or visualization) and focused on the diagnosability aspect of the log instrumentation techniques. Our survey covers all the general logging approaches, which include distributed tracing. In addition, on top of diagnosability, we have identified and examined additional challenges (e.g., usability and security) associated with logging.

Candido et al. [62] conducted a systematic review on logging techniques for contemporary software monitoring. They have mainly discussed four dimensions: log engineering, log infrastructure, log analysis, and log platforms. Log engineering is only focusing on logging code composition of conventional logging, while our survey not only discusses other logging approaches, but also covers LU integration. The rest of the three studied dimensions are all related to the phase of log management, which is not the focus of this survey. Furthermore, monitoring is only one task that logging deals with, while we examined other challenges such as configurability and clarity.

2.4 Logging Approach

There are three general logging approaches: conventional logging, rule-based logging, and distributed tracing. Section 2.4.1 provides an overview of these three approaches. Section 2.4.2 discusses the proposed solutions from these logging approaches that address various logging challenges. Section 2.4.3 summarizes our findings and presents some open problems associated with logging approaches.

2.4.1 An Overview of Three General Logging Approaches

Table 2.1 compares these approaches among the following six dimensions:

- *Who* refers to the type of developers responsible for performing the log instrumentation tasks. The SUS developers are mainly responsible for the log instrumtation tasks if they adopt the conventional or rule-based logging approaches. If the SUS imports third party libraries, they may need to configure the LUs within these libraries in order to gain the full picture of the SUS behavior during runtime. Please refer to Section 2.5 for details. On the contrary, third-party library developers are the ones responsible for most of the log instrumenation tasks if they adopt distributed tracing approach. Only if needed, SUS developers may add additional LC in the SUS.

- *Filtering* refers to the process of removing the unwanted log messages during runtime in order to reduce overhead. The verbosity level is used in conventional and rule-based logging to indicate the severity of LC. While in distributed tracing, sampling is adopted to filter log messages. The sampling decision can be controlled by pre-defined probability, rate or even adaptive. Please refer to Section 2.5.2 for details.

- *Format* refers to the requirements on the structure and the contents of the generated log messages. Instrumenting free form LC is mostly adopted in conventional and rule-based logging. On the other hand, distributed tracing utilities record the information in a more structured way. For example, key-value pair is a popular paradigm to structure the generated log messages.

- *Domain* refers to the categories of applicable SUS for each logging approach. Conventional and rule-based logging can be applied in almost any types of SUS, while distributed tracing is mostly adopted in distributed systems.

- *Flexibility* refers to the feasibility and the effort needed for a particular logging approach to be applied under various instrumentation scenarios. Conducting conventional logging is most flexible, as SUS developers can insert LC in any program points. Instrumenting LC in distributed tracing is less flexible, as most of the LC snippets are in the boundaries of software components. Rule-based logging is the least flexible, because the locations of LC snippets need to follow pre-defined rules.

- *Scattering* refers to the spread of the instrumented LC snippets across the code base by adopting a particular logging approach. The degree of scattering of LC in conventional logging is high, because of its high flexibility. On the other hand, LC snippets in rule-based logging and distributed tracing is less scattered, as its locations follow designated rules(e.g., beginning of a method) or certain patterns (e.g., before an RPC call).

To better describe the three logging approaches, we will explain them by going through a running example. The log instrumentation scenario is to record the behvior of a web server while trying to authenticate users. We have realized this logging concern with all three approaches as illustrated in Figure 2.4.


```
1  import org.apache.logging.log4j.LogManager;
2  import org.apache.logging.log4j.Logger;
3  class MyServer {
4    Logger logger = Logger.getLogger(MyServer.class);
5    void authentication(Request req, ...) {
6      logger.info("Receive from client " + req.userName);
       // actual authentication process ...
10     reply(response, ...)
11     logger.info("Send response to " + req.IP);
12   }
13   private void start() {
14     Server server = new Server();
15   }
```

```
Apr 02, 2020 12:22:41 PM mycompany.MyServer Receive from client alice
...
Apr 02, 2020 12:22:42 PM mycompany.MyServer Send response to 192.168.0.1
...
Apr 02, 2020 12:22:50 PM mycompany.MyServer Receive from client bob
...
Apr 02, 2020 12:22:52 PM mycompany.MyServer Send response to 192.168.0.2
...
Apr 02, 2020 12:23:01 PM mycompany.MyServer Receive from client tom
...
Apr 02, 2020 12:23:02 PM mycompany.MyServer Send response to 192.168.0.2
...
...
```

```
1  import org.apache.logging.log4j.LogManager;
2  import org.apache.logging.log4j.Logger;
                                    ...
5  @Around("execution(* MyServer.authentication(..))")
6  public Object logAround(ProceedingJointPoint pjp, Request req) {
7    logger.info("Receive from client " + req.userName);
8    pjp.proceed()
9    logger.info("Send response to " + req.IP)
10 }
```

```
3  class MyServer {
4    void authentication(Request req, ...) {
     // actual authentication process ...
10   }
```

```
Apr 02, 2020 12:22:41 PM mycompany.MyServer Receive from client alice
...
Apr 02, 2020 12:22:42 PM mycompany.MyServer Send response to 192.168.0.1
...
Apr 02, 2020 12:22:50 PM mycompany.MyServer Receive from client bob
...
Apr 02, 2020 12:22:52 PM mycompany.MyServer Send response to 192.168.0.2
...
Apr 02, 2020 12:23:01 PM mycompany.MyServer Receive from client tom
...
Apr 02, 2020 12:23:02 PM mycompany.MyServer Send response to 192.168.0.2
...
...
```

```
1  import io.opentracing.*
2  class MyServer {
3  Tracer tracer = GlobalTracer.get();
4    private start() {
5      Server server = new TracedServer(tracer);
...
20   }
21   void authentication(Request req, ...) {
      // actual authentication process  ...
25     reply(response, ...)
26   }
```

```
1  import io.opentracing.*
2  class TracedServer extends Server {
                                    ...
8    @Override
9    public void onReceive() {
10     SpanContext parentSpan = tracer.extract(HTTP_HEADERS, headers);
11     spanBuilder = spanBuilder.asChildOf(parentSpan);
12     span = spanBuilder.start();
13     span.log(Map.put("Client", req.userName)
14                 .put("Message", "receive from client"))
                                    ...
20 }
21   @Override
22   public void onSend(Response response) {
                                    ...
32     span.log(Map.put("IP", req.IP)
33                 .put("Message", "Send response"));
                                    ...
```

```
"data" : [
  {
    "traceID": "1242029787ec9011"
    "spans": [
      ...
      {
         "traceID": "1242029787ec9011",
         "spanID": "1a481c39c9e66ac6",
         "parentSpanID": "c53ac490f028963a",
         "duration": 277146,
         ...
         "logs":[
            {
            "timestamp": 1585844561219000,
            "Client": "bob",
            "Message": "Receive from client"
            },
            {
            "timestamp": 1585844561230000,
            "IP": "192.168.0.1",
            "Message": "Send Response"
            },
         ...
         ],
      }
    ]
  },
  ...
]
```

Figure 2.4: An example of user authentication scenario instrumented with three general logging approaches.

Conventional Logging

The code snippet using the conventional logging approach is shown in Figure 2.4(a). Before we can instrument the system with LC snippets, we have to first import the LU(s), which provides functionalities of conventional logging. As shown from line 1 and 2 of the code snippet, we use `Log4J 2` library [63], a popular LU for Java-based systems. Then a logging object, which is responsible for performing the log instrumentation in the rest of this example, is created at line 4. Line 6 and 11 show two lines of LC snippets, which record the user names and their IP addresses before and after the authentication process. Similar to standard I/O methods like `System.out.println` or `System.err`, an LC snippet contains static texts and dynamic contents. In addition, it also contains a logging object as well as a verbosity level to control the amount of outputted log messages. Take the LC snippet at line 6 as an example. The four components are highlighted in different colors: the logging object (`logger`) in red, the verbosity level (`info`) in yellow, the static texts (`"Received from client"`) in green, and the dynamic contents (`req.userName`) in grey.

A sample snippet of the generated log messages are shown in Figure 2.4(d). In addition to the static texts and dynamic contents, each log message also contains basic information like timestamp and the location of the logging code. Similar to the natural language text, the resulting log messages are usually loosely formatted and cannot be easily parsed by the computer programs [64].

Conventional logging is very easy to setup and the resulting LC snippets can be placed in almost anywhere in the SUS. However, there are two main issues concerning conventional logging: (1) *Cross-cutting Concerns*: the resulting LC snippets are scattered across the entire system and tangled with the feature code [65, 66, 67, 44]. This results in challenges on developing and maintaining high quality LC, while the SUS evolves. To resolve this issue, rule-based logging approach is introduced (Section 2.4.1). (2) *Lack of Execution Context*: It is very challenging to correlate log messages from different processes or even machines [68]. This is especially the case for large-scale distributed systems. Hence, distributed tracing is introduced (Section 2.4.1).

Rule-based Logging

Different from the conventional logging approach, in which the LC inter-mixes with the feature code, the rule-based logging approach generalizes the logging behavior by specifying a set of rules. This greatly improves the modularity of the LC and hence provides a much better support for developers to track and maintain their LC while the SUS evolves.

Aspect-Oriented Progamming(AOP)-based logging is one of the most commonly used rule-based logging techniques. AOP is a programming paradigm, which is designed to improve modularity by reducing the amount of cross-cutting concerns [44], one of which is software logging. AOP-based logging has been used to support diagnosing functional failures [65, 67]. Developers define rules through aspect files. A typical aspect file consists of pointcuts and advice. A pointcut is to define the point of execution where the cross-cutting concern (e.g. logging) needs to be applied. An advice is the additional code (e.g. LC) instrumented.

Figure 2.4(b) continues our running example by using the `AspectJ` LU, which is a very popular AOP-based approach for Java-based systems. The file `LogAspect.java` acts as the aspect file. The rules (a.k.a., instrumented points) are defined through the Java annotation at line 5. In this example, the annotation `@Around` means both the beginning and the end of the methods will be instrumented. The value within the brackets specify the instrumented methods. In this example, the method with name `authentication` within class `MyServer` will be instrumented. The instrumented code is defined at line 7 and line 9, which outputs the same log messages as the conventional logging example. The file `Server.java` does not include any LC, as all the LC is modularized and specified in the aspect files. As shown in Figure 2.4(e), the same log messages will be outputted during runtime.

On one hand, rule-based logging enables developers to separate rules with actual instrumentation. Updating LC is easy, as developers only need to revise the rules without modifying code at multiple locations. This improves the modularity of the LC. On the other hand, rule-based logging lacks flexbility, as LC cannot be instrumented at anywhere due to the implementation limitation [45, 69, 70].

Distributed Tracing

Although it's flexible and easy to perform conventional logging, the resulting log messages are free-formed text, which cannot be easily cross-linked across different process or even machines. This will be a major problem for distributed systems, in which one scenario may be executed on mulitple machines. To cope with this challenge, distributed tracing is introduced. Different from conventional logging, in which developers of SUS mainly perform the log instrumentation activities, library developers are mainly responsible for the task of log instrumentation. Although developers of SUS can perform additional log instrumentations, their main task is to import the tracing library and perform some setup actions. As a result, the generated log messages are structured and can be connected by a set of common variables. In the context of distributed tracing, these structured log messages are connected together also referred as an end-to-end *trace*. For brevity, we call this as *trace* in the rest of this section.

Figure 2.4(c) continues our running example by using distributed tracing as our log instrumentation approach. This example uses `OpenTracing`, a very popular LU supporting distributed tracing-based logging approach [71]. For the code snippet at the top of Figure 2.4(c), a tracer object and a traced server instance are created at line 3 and line 5, respectively. Other than this, there are no additional log instrumentation efforts required from SUS developers. The actual LC composition is done by the library developers whose code snippet is shown at the bottom of Figure 2.4(c). They implement `TracedServer`, which extends the original `Server` class and overrides two important methods: `onReceive` and `onSend`. These two methods will be invoked when receiving a request and sending a reply, respectively.

A typical *trace* in the context of distributed tracing consists of multiple log messages which are connected together. These connected log messages form a complete request workflow. In

`OpenTracing`, such log messages are called `spans`. In both the client side and the server side, library developers compose the span log using APIs like `span.log`. These spans are passed from the client side to the server side by certain communication protocols (e.g., HTTP), so that spans from both ends can be connected. In this way, a complete trace recording information from both ends can be generated. In our example, we only show the code at the server side because the tracing code at the client side is similar. To notice, inside method `onReceive`, where we receive the client's request, we extract the context data that is sent from the client at line 10. At line 11, we create the span as a child of the span that we extract from the client. At line 13, we store the same message as the example in conventional and rule-based logging. After the request is processed, inside the `onSend` method, the span log at line 32 will be sent to the client.

The generated traces are shown in Figure 2.4(f). As we can see, the traces are structured in JSON format. JSON stores information in a key-value fashion. For example, the recorded information of the LC snippet at line 13 has two keys: `Client` and `Message`. The key (`Client`) corresponds to the runtime information: the `userName` of the request, and the key (`Message`) corresponds to the static texts describing the logging context. Compared to conventional and rule-based logging, log messages generated by distributed tracing are more structured. The related log messages can easily be linked across different machines by the associated `traceID`.

2.4.2 Challenges and Proposed Solutions

Below we present various challenges and their proposed solutions during the logging approach step:

- **Usability - Performance Overhead** To reduce the I/O cost associated with conventional logging, techniques have been proposed to delay the output of log messages only when needed. NanoLog [72] is a nanosecond scale logging system implemented in C++. It first statically analyzes the source code during compilation time and generates a compression function for each LC snippet. During runtime, only the compact log messages would be generated. The full textual version of the log messages will only be generated during the post-processing time. In this way, NanoLog can achieve 1-2 order of magnitude faster than conventional logging libraries (e.g., Log4J 2 [63], spdlog [47]). Similarly, `Log++` is a logging system which optimizes the logging performance for the Node.js platform by postponing log generation offline.

- **LC Quality - Maintainability** One of the main issues associated with rule-based logging is the steep-learning curve associated with the supporting LUs [45]. To cope with this issue, two techniques have been proposed:

  - *Rule Generalization*: Cinque et al. [73, 74] summarize eight general rules for log instrumentation by studying system design artifacts such as architectural models and UML diagrams. They abstract these artifacts into a system representation

model, which consists of the interactions among a set of entities in the system. Then they propose a set of rules to log key interactions which could cover the needed data for failure diagnosis. These rules are proposed to meet the needs of diagnosing four types of functional failures: (1) service error, (2) service complaint, (3) interaction error, and (4) crash error. For instance, service errors refer to errors that a service is unable to reach an exit point. To handle such kind of errors, the event of service start and end must be logged. Crash errors refer to any abnormal stop of an entity. To handle such errors, a heartbeat log needs to be periodically invoked to monitor the service execution. Such rules can be used to guide developers to perform their their rule-based log instrumentation activities.

– *Specifying Logging Concerns*: In order to ease the migration for conventional logging to rule-based logging approach, Bruntink et al. [75] and Mohammadian et al. [76] propose frameworks to easily express logging concerns in a moduralized manner.

2.4.3 Summary and Open Problems

In this section, we have presented three logging approaches: conventional logging, rule-based logging, and distributed tracing. In addition, we also describe serveral proposed solutions to address the usability and maintainability related challenges. Below we present two open problems in this area:

- **Best Practices**: There are no well documented guidelines on suggesting the appropriate logging approaches under specific scenarios. It is worthwhile to extract and generalize all the logging needs by studying the existing commerical and open source projects. Different logging approaches can be evaluated under these different logging needs in order to provide a systematic guideline on the best practices of using different logging approaches.

- **Migration**: Although rule-based and distributed tracing-based logging bring additional benefits compared to conventional logging, a study [45] shows that majority of the software projects still adopt the conventional logging approaches. The key issue is associated with steep learning curve and high migration effort [70, 69]. It is worthwhile to investigate the issues and concerns associated with migrating of logging approaches so that researchers and logging framework developers can provide better tool support for the migration process.

2.5 LU Integration

Instead of directly invoking the standard output functions like `System.out.print`, developers prefer to instrument their SUS using LUs (e.g., `SLF4J` [46] for Java and `spdlog` [47] for C++) due to additional functionalities like thread-safety (synchronized logging in multi-threaded

systems), data archival configuration (automated rotation of the log files), and verbosity levels (controlling the amount of log messages outputted). There are generally two concerns associated with LU integration:

- **what-to-adopt**: There are many LUs available in the wild. Developers need to decide which or whether existing LU(s) are need for their SUS. Furthermore, for projects with LU(s) integrated already, developers have to determine if they would like to migrate to other or newer LU(s).
- **how-to-configure**: Each LU contains a set of configuration options ranging from controlling the amount of output to the location of the log files. Developers need to properly configure LUs for their SUS in order to gather enough logging data while minimizing the performance overhead and storage requirements.

In the rest of this section, we present the challenges and their proposed solutions associated with each of the two concerns in the LU intergration step.

2.5.1 What-to-Adopt

With the increasing amount of LUs available in the wild [45], integrating appropriate LUs according to the requirements of individual SUS is important. The problem of what-to-adopt focuses on this matter with regard to the maintainability and security compliance of LUs. After that, it is also important to configure these LUs to increase their usability.

Modern software often leverage the functionalities provided by the LUs to instrument their SUS. A study [45] on 11,194 Java-based GitHub projects shows that there are more than 3,000 LUs being adopted in the wild. For example, many developers adopt LUs like Log4j [77] and Apache Commons Logging [78] to instrument their Java-based SUS [79]. Furthermore, many of these projects adopt multiple LUs or even implement their own LUs in order to address one or more of the following challenges:

- **Usability - Configurability**: One of the main reasons behind the adoption of multiple LUs in many large-sized projects is due to the import of third party libraries, which internally use LUs [45]. Additional LUs are developed in order to configure and control the logging behavior from third party libraries when using conventional and rule-based logging. In distributed tracing, the third-party libraries are pre-instrumented with LUs. Developers only need to import the instrumented versions of third-party libraries to enable logging, which is easy to configure because all the traced libraries can adopt the same interface [71]. It also works well within an industry environment. As illustrated in [50], almost all of the applications in Google use the same threading model, control flow and RPC libraries, which makes the instrumentation applicable to every SUS which adopts these libraries.
- **LC Quality - Maintainability**: Although many projects use general-purpose LUs, some LUs are used specifically for improving the maintainability of the LC. For example,

LUs like AspectJ [44], which realizes the rule-based logging approaches, support better modularization of the LC. Other LUs like JBoss logging [34] that enables the SUSs to easily support multiple human readable languages in the log messages. LU migration [80] is quite common in open source projects. Developers migrate the LUs for better flexibility, performance, and maintenance. However, 28.6% of the LU migration attempts are aborted due to various reasons. Over 70% of the successfully migrated projects may suffer from post-migration bugs.

- **Security - Auditing**: Some projects (e.g., Hadoop) develop their own LUs to better support auditing. Auditing log messages are generally more structured compared to the regular log messages.

2.5.2 How-to-Configure

LUs contain many different configuration options. They can be related to controlling the amount of log messages outputted, or the location of the log files, and the size of the log files. Solutions are proposed for the following challenges:

- **Usability - Configurability**: Zhi et al. [81] anazlye the evolution of logging configurations of 10 open source Java-based projects in GitHub and 10 industrial projects. They find that the names of loggers are changed frequently due to inconsistencies. Inspired by this finding, they propose a static checking technique to identify inconsistent loggers by comparing the contents of the loggers in the configuration files and in the source code files.

- **Usability - Performance Overhead**: One of the main issues associated with logging is the performance overhead incurred to the SUS. Many LUs (e.g., Google's Dapper [50] or Facebook's Canopy [51]) implementing distributed tracing-based logging approaches usually support sampling, which is a technique to selectively generate and preserve log messages in order to reduce the runtime overhead. There are three types of sampling techniques: head-based sampling; tail-based sampling; and unitary sampling [82, 61]:

  1. *Head-based Sampling*: The sampling decision is made at the beginning of every trace. It either preserves the whole trace (including every trace point), or no trace at all. Head-based sampling techniques can be further divided into:
     - *Probability sampling* makes the sampling decision based on a pre-defined probability. For example, the rate of 0.1% means one out of one thousand traces are preserved. This is the most basic sampling technique, which is used by many distributed tracing-based LUs.
     - *Rate-limit sampling* makes sampling decision based on a pre-defined sampling rate. For example, if the sampling rate is 100 traces per minute, only 100 traces will be preseved per minute regardless of the current throughput.

   - *Adaptive sampling* dynamically adjusts the sampling decisions during runtime. For example, Dapper [50] is able to tune the probability of sampling for each individual service based on the workload traffic.

2. *Tail-based sampling* makes the sampling decision at the end of each trace. Compared to head-based sampling, it can make a more informed decision after all the collected traces are available. Hence, developers can pay more attention to the traces that may contain anomalies, and discard the normal traces. However, tail-based sampling is not supported by many LUs due to its high resource requirements on memory/disks for temporarily storing all the generated traces.
3. *Unitary sampling* makes the sampling decision at every trace point. Hence, a complete trace cannot be recovered through this approach. This technique only has very limited usage scenarios.

2.5.3 Summary and Open Problems

In this section, we have discussed the challenges and proposed solutions associated with two common aspects on LU integration: what-to-adopt and how-to-configure. Below we present two open problems for this area:

- **LU Recommendation**: There are many LUs available in the wild, which provides various functionalities about software logging. In addition, new LUs are also constantly introduced. It is very important to provide developers' suggestions on the appropriate LU(s) for the SUS in order to ensure all the logging needs are satisfied. Furthermore, as the SUS evolves over time, more suggestions are also needed on incorporating additional LUs or LU migrations.
- **LU Management**: Many large-scale projects use third party packages, in which imports LUs. It is necessary to effectively configure the LUs for the SUS as well as the LUs for their imported packages in order to gain full observability of the entire systems. Further research is urgently needed to develop tools or techniques to automatically manage the logging behavior across mutliple LUs in one SUS.

2.6 LC Composition

The last step of log instrumentation is LC composition. There are three sub-steps in LC composition:

1. The step of **where-to-log** is about deciding the appropriate logging points. Various studies [10, 9, 83, 32, 8] have shown that logging is pervasive in software development process. Developers usually rely on their experience or gut feelings when deciding on the logging points in the source code. On one hand, logging too little will hinder the diagnosability of log messages. For example, missing logging statements in exception

blocks will cause incomplete information of failures, making failure diagnosis more difficult [18]. Incomplete LC snippets also hinder developers' understanding, hurting LC quality, since they can only recover ambigious execution paths from the execution log messages [2]. On the other hand, although excessive logging, in which LC snippets are inserted everywhere in the source code, will provide rich runtime information, it will bring in huge performance overhead and high storage cost associated with the generated log messages. In addition, it is very challenging to diagnose problems by analyzing large volumes of log messages, most of which are not related to the problematic scenarios [84].

2. The step of **what-to-log** is about providing sufficient information in the three components of each LC snippet:

   - *Verbosity level* specifies whether an LC snippet should be outputted during the execution of SUS. Choosing an appropriate verbosity level for an LC snippet is important. For example, if an LC snippet records information about a failed execution, the verbosity level should be set as `error` or `fatal`. If it is mistakenly set as `debug`, such log messages may not be outputted or even if they do, developers may neglect them. Such neglection could impact customer experience and negatively impact the product quality.
   - *Static texts* describe the logging context in a human readable manner. Currently, developers are responsible for manually composing the static texts in the LC snippets. Poorly written or outdated static texts may cause confusion of the DevOps engineers and impact their various log analysis tasks.
   - *Dynamic contents* reflect the state of SUSs during runtime. They are the results of executing variables and method invocations included in each LC snippet. It is important to record the necessary runtime information in order to satisfy various logging needs from the developers and operators.

3. The step of **how-to-log** is about developing and maintaining high quality LC, which is scattered across the entire system and tangled with the feature code. Although the rule-based logging approach provides better management of LC, many industrial and open source systems still choose to inter-mix LC with feature code [36, 85]. A study [37] shows that 20% - 45% of the LC has been changed at least once during their lifetime. The median number of days between an LC snippet is introduced and its first change ranges from 1 to 17 days. Unlike feature code, whose quality can be verified via testing, the correctness of LC is very difficult to verify. This can hinder program understanding or even cause runtime issues like crashes [7].

Many solutions have been proposed to address various challenges in these three substeps. Each solution focuses on a different type of challenge. We demonstrate all the proposed solutions, their associated challenges, and their applicable steps in Table 2.2. For example, to improve failure diagnosis, which is a type of usability challenge, program

analysis techniques are proposed during the *where-to-log* and *what-to-log* sub-steps. The research papers that propose these techniques are [18, 86, 87, 35]. In the rest of this section (Section 2.6.1, 2.6.2, 2.6.3, and 2.6.4), we will discuss each challenge and their associated solution(s) in details.

2.6.1 Usability - Performance Overhead

The main challenges associated with usability during the LC composition step is performance overhead. Various cost optimization-based techniques have been proposed to determine the optimal logging points (a.k.a., *where-to-log*) in the SUSs:

- *Information Theory*: To better recover execution paths using log messages, Zhao et al. [88, 2] propose *Log20*, which is a technique to automatically insert LC snippets. In order to evaluate such capability, the concept of entropy is used. Entropy is originated from Shannon's information theory, The higher entropy, the more uncertain execution paths exist in a code block. At the same time, the performance costs of the instrumented LC snippets should not exceed a customized threshold to minimize performance overhead. The best logging points are those that resolve the most uncertainty during problem diagnosis within an acceptable range of performance overhead.

- *Constraint Solving*: To dynamically control the performance overhead, Ding et al. [21] propose a constraint solving method to determine the optimal logging points which incur minimum performance overhead with maximum amount of runtime information. This approach provides a configuration, which dynamically adjusts the types of log messages outputted during runtime based on the performance of SUS.

- *Statistical Modeling*: To better monitor the performance of SUS, Yao et al. propose Log4Perf [89] to suggest logging points. They first build performance models by running performance tests. Through these models, source code snippets that are performance influencing are identified. For all the methods in the source code, the entry points and the exit points are instrumented with LC snippets. After re-executing the performance tests, the methods that cost constant execution time are identified and their corresponding LC snippets are removed. The remaining set of LC snippets will assist developers for diagnosing and optimizing system performance by increasing the visibility of performance issues.

2.6.2 Diagnosability

There are two objectives within the diagnosability challenge: failure diagnosis and performance analysis. Program analysis-based solutions are proposed to address the challenges of improving failure diagnosis, whereas end-to-end based solutions are used for better performance analysis. Both types of proposed solutions are used during the *where-to-log* and *what-to-log* sub-steps.

Table 2.2: The challenges and the proposed solutions in LC composition step.

Category	Objectives	Techniques	Steps	References
Usability	Performance overhead	Cost optimization	where-to-log	[88, 2, 89, 21]
	Failure diagnosis	Program analysis	where-to-log	[18, 86, 87]
			what-to-log	[35, 90]
Diagnosability	Performance analysis	Causality tracking	where-to-log	[59, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 50, 20, 102, 22, 51, 103]
			what-to-log	
LC Quality	Clarity	NLP/Visualization	what-to-log	[104, 105]
	Maintainability	History-based	how-to-log	[36, 38, 106]
			where-to-log	[8, 107, 108, 109, 110, 111]
	Consistency	ML/Cloning	what-to-log	[48, 112, 113]
			how-to-log	[10, 114]
SecurityCompliance	Forensic analysis	Heuristics	what-to-log	[115, 116]

Failure Diagnosis

Program analysis is a technique, which analyzes the behavior of SUSs automatically through static or dynamic analysis [117, 118]. Both types of program analysis techniques have been used to support better diagnosis of functional failures:

- *Static analysis*-based solutions analyze source code without executing the SUSs. There are many program analysis tools, which automatically scan through the source code of SUS, output abstract representations like ASTs (abstract syntax trees) and call graphs, and reveal deficiencies in the source code. For example, call graphs can be analyzed further in order to identify logging points which are suitable for failure diagnosis.

  Yuan et al. have conducted two prior work to improve failure diagnosis by leveraging static analysis [18, 35]. For example, they investigate 250 bug reports and characterize exception patterns that need additional logging [18]. A static checking tool, Errlog, is proposed to scan the code base for these types of exception blocks and automatically instrument LC snippets to record the error locations and error context. They also propose LogEnhancer [18], to instrument additional variables in the exsiting LC snippets. These variables are extracted by statically analyzing the control flow and data flow of the source code of SUS, so that log messages can contain complete runtime information to closely replay and diagnose the failure context. Different from the above approaches, in which the logging points are suggested manually one-by-one. SmartLog [119] is proposed to automatically instrument LC snippets by leveraging the data mining techniques on the static analysis results. The context (e.g., residing functions) of LC snippets are analyzed to generate log intention models. The log intention models represent the logging descisions (a.k.a., whether this LC snippet is logged or not logged) of a code snippet. Data mining models are then trained on such dataset and used subsequently to suggest program points for log instrumentation.

  Static analysis-based solutions can improve the diagnosability of the generated log messages. The advantage is that they can be applied off-line without running SUS. It is also easier for developers to understand the reasons behind each logging point. The disadvantage is that it requires great manual efforts in terms of deriving the patterns.

- *Dynamic Analysis*-based solutions suggest logging points by analyzing the runtime behavior of SUSs. Compared to static analysis-based solutions, dynamic analysis-based solutions need to execute the SUS. The output data generated by the execution is then analyzed for various tasks. The ambiguous and missing information in the outputs lead back to logging suggestions. Hence, dynamic analysis-based solution usually consists of three steps:

  1. *Running SUS under different settings*: Developers can inject customized faults into the SUS, or just run with common workloads. The goal of this step is to collect output data, such as log messages, stack traces, and memory dumps.

2. *Log analysis*: The goal of this step is to check whether the current output data is capable of diagnosing failures. If not, the missing information indicates the potential logging points and what key variables need to be recorded.
3. *Update instrumentation*: Additional instrumentation will be performed on the SUS. Then the same experiments from step 1 will be executed again. The goal of this step is to evaluate if the newly instrumented LC can improve the failure diagnosis process.

There are three proposed techinques that leverage dynamic analysis-based solutions to suggest additional logging points. For example, Cinque et al. [86] propose a technique to increase the failure diagnosability of log messages. They first inject faults into three popular open source systems and execute the SUS to collect log messages and memory dumps. Analyzing the output data, they summarize the top 10 frequent executed functions from halt failures and silent failures. Additional LC snippets are inserted into these functions. Crameri et al. [87] also propose similar techniques to suggest which branches to log. They first repeatedly execute the SUS with different inputs using a symbolic engine. After each run terminates, they record the constraints between the symbolic variables and the executed branches. Additional log instrumentation is performed by identifying the associations of executed branches and variables. Jia et al. [90] propose an approach to inserting LC snippets to ease fault localization. They first run the SUS with injected faults. Then they compare the log messages between successful runs and failed runs to identify key variables to log.

Performance Analysis

Modern software systems are complex, consisting of multiple software components across various layers. With the increasing popularity of cloud native applications and microservices, one system could contain hundreds or thousands of small services, many of which are developed by different engineering teams. Conducting performance analysis through conventional or rule-based logging on this type of software system is very hard, as the resulting log messages may lack contextual information to build the causality among the logged events. Hence, to enable thorough end-to-end performance analysis, the causality tracking technique is proposed. In particular, causality tracking is conducted in two ways [82, 61]:

- *Schema-based techniques*: Schema-based techniques correlate the related log messages based on manually designed rules [91, 93, 94, 92]. Developers need to design event schemas to join individual event to recover a complete request. For example, event `A` and event `B` shares the same value of variable `x`, event `B` and event `C` share the same value of variable `y`, then event `A`,`B`, and `C` will be joined. The causality is decided by the timestamp of the events.
- *Propagation-based techniques*: Propagation-based techniques track the causality within a request by passing the context metadata between instrumented components. A

complete trace of a request can be comprised of multiple log messages linked by the metadata. The metadata contains a unique global trace ID. The metadata is propagated as the request flows from one component to another one. Apart from the global trace ID, it usually records the necessary information such as parent ID to keep the causality relations between individual trace. The format of metadata needs to follow a certain standard or protocol so that both the senders and receivers can pack and unpack the information.

Most of the modern distributed tracing frameworks adopt propagation-based techniques instead of schema-based techniques. The reasons are three folds:

- *Performance Overhead*: Schema-based techniques do not support sampling, because they cannot decide which log messages to be discarded without compromising the ability to conduct the join operations. Hence, for SUS that generates a large volumes of log messages every day, it would be too expensive to adopt schema-based techniques.
- *Generalizability*: Propagation-based techniques are general across different SUSs. On the contrary, for schema-based techniques, developers need to implement the join schema for every different SUS based on its characteristics, which can be time consuming and error-prone.
- *Real-time feedback*: Schema-based techniques are mostly conducted offline after all the log messages have been collected. For systems that need monitoring or online analysis and detection, propagation-based techniques are more appropriate.

2.6.3 LC Quality

Various solutions have been proposed at different sub-steps to address the three specific challenges (clarity, maintainability, and consistency) associated with LC quality:

Clarity

The static texts in LC act as descriptions for developers and operators when they read log messages for understanding what is going on under the hood. Hence, it is important to clearly describe the context of specific logging scenarios in each LC snippet to avoid confusion.

- *Natural Language Processing (NLP)*: He et al. [104] conduct a study on characterizing the static texts of logging code. They use n-gram language model to calculate the repetitiveness of static texts. They find that static texts in logging code are endemic, i.e., LC within the same file or in the same context tend to use similar static texts to describe the program behavior. Inspired by this finding, they propose a technique to automatically generate static texts in each LC snippet. They first extract the code snippet that contains the LC snippet. Then they search for the most similar code snippet in the code base by comparing the edit distances. The static texts of the LC snippet in that code snippet is then used as the texts of the candidate LC snippet.

- *Visualization*: Other than misleading static texts, missing variables can also cause difficulty in understanding runtime behavior. As many post analysis, manual or automated, require that log messages can be correlated through a set of variables. A technique [105] has been proposed to add missing variables of LC snippets by analyzing the visualization of log messages. The first step is to create a graphical representation of log messages, i.e., the identifier graph, since each log message may contain various identifiers. The log messages with same identifers are correlated. As the identifiers can be missing, inconsistent, or ambiguous, the identifier graph [105] is used to visualize the deficiencies. For example, a log message with insufficient identifiers will lead to missing edges in the graph, where the semantics of the source code show that the edge should exist. Based on the observation, suggestions can be proposed to improve the quality of LC snippets.

Maintainability

Outdated LC exists for a variety of reasons. First, as LC is mostly done manually, human errors (e.g., typos in the static texts) can happen. Furthermore, as LC scatters across the entire code base and cross-cuts with the feature code, it is hard to keep track of them efficiently. As the SUS evolves rapidly, developers may forget to update the LC accordingly along with feature code. Hence, techniques to automatically maintain LC are needed. Existing work [36, 38, 106] mainly rely on the past development history to guide the maintenance of LC. They usually consist of the following steps:

1. Examine the development history, e.g., code changes and issue reports.
2. Characterize the code changes that are fixing logging code issues.
3. Extract anti-patterns (a.k.a., common problems) from the previous step and implement automated tools to detect them.

Depending on the type of studied development artifacts, there are three kinds of techniques:

- *Commit-based*: Chen and Jiang [36] propose the first work on characterizing and detecting anti-patterns in the logging code. They find six anti-patterns in the logging code by carefully studying the code commits of popular and well maintained open source projects. Their anti-patterns are implemented inside LCAnalyzer, which is a static analyzer to flag outdated LC snippets.
- *Source code-based*: Li et al. [38] focus on a specific type of anti-patterns in LC: duplicate logging code smells. They uncover five types of duplicate logging code smells by studying the source code of four popular open source projects. They propose a tool, DLFinder, to detect the instances of duplicate logging code smells.
- *Issue report-based*: Instead of analyzing the commit history or the source code releases, Hassani et al. [106] study log-related issue reports and propose seven root causes of these issues. They subsequently implement a tool to detect these issues.

Consistency

Another method to maintain the quality of LC is to ensure the consistency between the existing and the newly added LC snippets. The rationale is that many mature projects with long history tend to have high quality of LC, as their resulting log messages are extensively used and examined already. There are two approaches proposed to ensure the consistency of the LC:

- *Machine Learning:* Recently, machine learning techniques become widely adopted in software engineering tasks [120]. It makes suggestions or predictions based on data [121]. As logging is pervasive in software [10, 9], datasets are readily available for applying machine learning techniques to automatically make decisions.

  Depending on the objectives of tasks, there are two common types of ML: supervised learning and unsupervised learning. Supervised learning requires the training data to have readily available labels and predicts the labels of new data. On the other hand, unsupervised learning does not require the training data. It is mostly used for deriving patterns in a dataset.

  - *Supervised learning* techniques are adopted for assisting LC composition, as the objectives are labelling if a code snippet should be logged (where-to-log) or if a particular variable/verbosity level should be used for logging (what-to-log). The general process usually consists of the following four steps: (1) *Data gathering:* This step is to collect training data for performing the tasks. The training data consists of a set of instances with labels; (2) *Feature engineering:* This step is to extract features that are to describe the instances. These features are the input for the machine learning models; (3) *Model building:* This step is to build machine learning models from the labeled training data. The parameters are tuned in this step to improve the model performance; and (4) *Making predictions:* This step is to apply the model on new data to predict the labels. This step is the final output of the supervised learning process.

    Supervised learning techniques have been applied in the following two sub-steps in the LC composition phase:

    * *Predicting where-to-log*: Fu et al. [8] propose a technique to predict if a code snippet should be logged. In particular, they focus on two types of code snippets: `catch` blocks and `return-value-check` blocks. Each logged or unlogged code snippets are collected and labeled. Contextual keywords, such as the residing function name, are then collected as the features. A decision tree model is built for the task based on the collected features. Zhu et al. [107] propose LogAdviosr, which improves the previous technique by collecting more types of features. Their features include structural features, which are the contextual keywords, the textual features, which are generated by transforming the code snippet into stemmed words, and the syntactic features, which are

the properties of the code snippets such as the total lines of the code snippet. Lal et al. also propose a technique with a different feature set to predict addition of LC snippets in the `catch blocks` [108] and `if` code blocks [109]. Li et al. [110] propose a technique to infer if a method needs to be logged. They first extract the topic of a method using topic models. The topics are automatically created using the co-occurances of words in code snippets. They then find that the topic of a method is an important feature for the model, which provides additional explanatory power for ML models, improving both AUC and accuracy. They [111] also propose similar techniques to infer if a commit need just-in-time change for LC snippets.

* *Predicting what-to-log*: Other than being used for predicting if a code snippet should be logged, supervised learning techniques can also be used to predict the content of a LC snippet at the step of what-to-log. There are two areas of work depending on the types of logging components:
  - *Verbosity levels*: Li et al. [48] propose to recommend the most appropriate verbosity level for newly-added LC snippet. Since there are more than two verbosity levels for an LC snippets, they build an ordinal regression model for this task. They gather the training data from development histories. File metrics, change metrics, and historical metrics are used for features. Results show that the model performs better than random guessing. Important features that impact the verbosity levels include the characteristics of the containing block of a newly-added LC snippet and the number of existing LC snippets in the containing file. Kim et al. [113] also propose a classifier to validate verbosity levels. They use the `word2vec` model to generate the feature vectors.
  - *Dynamic Contents*: Liu et al. [112] propose a technique to recommend variables in LC snippets. Different from predicting a verbosity level, which have a fixed set of labels, variables in LC snippets are dynamic. This means that for each instance, the set of possible labels are not the same. To solve this issue, they first use neural networks to learn the proper representation of each program token, and then use a binary classifier to predict if the program token should be logged.

– *Unsupervised learning* is applied to assist the modification of LC snippets during the sub-step of how-to-log. Li et al. [114] propose that LC snippets with similar context tend to share similar modifications. Driven by this assumption, they implement LogTracker, a tool to automatically learn log revision rules. For each instance, the features are generated from the semantics of context. Then they apply agglomerative hierarchical clustering algorithm to group similar LC snippets. If a new instance's features are similar to the features of instances within a cluster, a similar modification is then suggested.

- *Code Cloning:* The idea of the code cloning-based approach is to validate the quality of

LC by searching for similar LC snippets. For example, Yuan et al. [10] propose a code cloning-based technique to fix inconsistent verbosity levels in LC snippets. They first extract all the groups of code clones and identify the LC snippets within. If within the same clone group the LC snippets have inconsistent verbosity levels, at least one of them is incorrect. However, although straightforward and easy to implement, the capability of code cloning-based approaches is limited, as not every code snippet containing LC snippets has clones.

2.6.4 Security Compliance

King et al. [115] propose a heuristics-driven technique to identify whether a user event should be logged or not from the forensic perspective. Computer forensic is the practice of collecting data for legal purposes. These data is further exploited for investigation of crimes. Log messages are one of the important sources of the evidence [122]. They first extract verb-object pairs from natural-language artifacts such as specifications and requirement documents. Then they propose 12 heuristics-driven rules to identify the mandatory logging events (MLEs) from these verb-object pairs. They follow up by evaluating three methods [116] to identify MLEs: standards-driven, resource-driven, and heuristics-driven methods. A controlled experiment is conducted on 103 computer science students. Unfortunately, their results show that there is no recommended method, which out-performs the other two methods in a statistical significant level. Their study shows that more research is needed towards identifying correct MLEs.

2.6.5 Summary and Open Problems

There are various solutions proposed to address the challenges in the three sub-steps of LC composition: where-to-log, what-to-log, and how-to-log. Below we describe a few open problems in this area:

- *Can we effectively bootstrap logging practices?* So far, many studies aiming to improve logging practices are dependent on the existing logging practices of the studied projects. However, there are many software projects with few or even no LC snippets. It remains a challenge how to bootstrap logging in these projects.

- *Can we evaluate the effectiveness of logging practices?* Currently, there are no standards in evaluating the effectiveness of logging practics. Defining metrics, such as test coverage to testing effectiveness, should be developed to evaluate logging practices from different dimensions such as usability, diagnosability, etc. Such metrics can shed lights on identifying opportunities to improve the quality of logging.

- *Can we provide benchmarks for improving logging code composition?* Chen et al. [123] have conducted the first attempt to extract the Logging-Code-Issue-Introducing(LCII) changes by mining the projects' historical data. Every LCII change corresponds to a potential logging issue. Such a dataset can be very useful for interested researchers

to develop new techniques in the area of *how-to-log*. However, more benchmarks are needed for evaluating solutions in the other two sub-steps of LC composition solutions.

2.7 Conclusion

Software logging is used widely by developers for a variety of purposes. Log instrumentation is about the development and maintenance of logging code. This is the first phase in software logging and happens before log management. Unfortunately, unlike log management, which has extensive tool support, there are little research and practice done in the area of log instrumentation. This survey aims to summarize the challenges that developers face when conducting log instrumentation. In particular, for each step (logging approach, LU interaction, and LC composition) in the log instrumentation phase, we describe their challenges and the proposed solution techniques. To ease replicability and further study on this survey, we have provided our dataset at [13].

Chapter 3

Extracting and Studying the Logging-Code-Issue-Introducing Changes in Java-based Large-Scale Open Source Software Systems

3.1 Introduction

Execution logs, which are usually readily available for large-scale software systems, have been widely used in practice for a variety of tasks (e.g., system monitoring [16], problem debugging [18], remote issue resolution [3], test analysis [19], and business decision making [5]). Execution logs are generated by executing the logging code (e.g., `Logger.info("User " + userName + " logged in")`) that developers have inserted into the source code. There are typically four types of components in a snippet of logging code: a logging object, a verbosity level, static texts, and dynamic contents. In the above example, the logging object is `Logger`, the verbosity level is `info`, the static texts are `User` and `logged in`, and the dynamic content is `userName`.

It is very challenging to develop and maintain high quality logging code for constantly evolving systems for the following two reasons: (1) **Management**: software logging is a cross-cutting concern, which tangles with the feature code [44]. Although there are language extensions (e.g., AspectJ [11]) to support better management of logging code, many industrial and open source systems still choose to inter-mix logging code with feature code [9, 32, 10]. (2) **Verification**: unlike feature code or other types of cross-cutting concerns (e.g., exception handling or configuration), whose correctness can be verified via software testing; it is very challenging to verify the correctness of the logging code. Figure 3.1 shows one such issue in the logging code. In the Hadoop DFSClient source code, the variable name was changed from `LEASE_SOFTLIMIT_PERIOD` to `LEASE_HARDLIMIT_PERIOD` in version 4078. However, the developer forgot to update the static text from `soft-limit` to `hard-limit`. Such issues are very hard to be detected using existing software verification techniques, except conducting

careful manual code reviews. Developers have to rely on their intuition to compose, review, and update logging code. Most of the existing issues in logging code (e.g., inconsistent or out-dated static texts, and wrong verbosity levels) are discovered and fixed manually [9, 10].

DFSClient.java from *Hadoop (HDFS-5800)*

V 4078:
```
LOG.warn("Failed to renew lease for " + clientName + " for "+ (elapsed / 1000)+ "
seconds (>= soft-limit ="+ (HdfsConstants.LEASE_HARDLIMIT_PERIOD / 1000)+ " seconds.)
"+ "Closing all files being written ...",e)
```

V 5956:
```
LOG.warn("Failed to renew lease for " + clientName + " for "+ (elapsed / 1000)+ "
seconds (>= hard-limit ="+ (HdfsConstants.LEASE_HARDLIMIT_PERIOD / 1000)+ " seconds.)
"+ "Closing all files being written ...",e)
```

Figure 3.1: An example of an issue in the logging code [124].

Most of the existing research on logging code focuses on "where-to-log" (a.k.a., suggesting where to add logging code) [21, 8, 107] and "what-to-log" (a.k.a., providing sufficient information in the logging code) [80, 6, 35]. There are very few work tackling the problem of "how-to-log" (a.k.a., developing and maintaining high quality logging code). Low quality logging code can hinder program understanding [6], cause performance slow-down [125], or even system crashes [7]. Unlike other software engineering processes (e.g., refactoring [126] and release management [127]), there are no well-established logging practices in industry [8, 32]. Our previous work [36] is the first study, which characterizes and detects anti-patterns (common issues) in logging code by manually examining a sample of the logging code changes from three open source projects. Six anti-patterns in the logging code (ALC) were identified. The majority (72%) of the reported ALC instances, on the most recent releases of the ten open source systems, have been accepted or fixed by their developers. This clearly demonstrates the need for this area of research. However, one of the main obstacles facing researchers is the lack of the available dataset, which contains the Logging-Code-Issue-Introducing (LCII) changes, as many issues in the logging code are generally not documented in the commit logs or in the bug reports. An LCII change is analogous to a bug introducing change [128]. It is a type of code change, which will lead to future changes (e.g., changing the static texts for clarification or lowering the verbosity levels to reduce the runtime overhead) to the corresponding logging code. For example, in Figure 3.1, the logging code change at version 4078 is the LCII change. This change introduced a bug in the logging code. The fix to this LCII change is at version 5956. Logging code changes which are co-evolved with feature code are not included as LCII changes.

In this chapter, we have developed a general approach to extracting the LCII changes by mining the projects' historical data. Our approach analyzes the development history (historical code changes and bug reports) of a software system and outputs a list of LCII changes for further analysis. We have performed a few preliminary studies on the resulting LCII change dataset and presented some open problems in this area. The contributions of

this chapter are as follows:

1. Compared to [36], using our new approach, the resulting LCII changes are more complete. In [36], the authors assumed that only the independently changed logging code changes (a.k.a., the logging code changes which are not co-changed with any feature code changes) may contain fixes to the LCII changes. In this chapter, we have found that this assumption is invalid, as some fixes to the LCII changes may require feature code changes as well. Thus, rather than only focusing on the independently changed logging code, we extract the LCII changes from all the logging code changes.

2. Instead of manually identifying LCII changes as in [36], we have developed an adapted version of the SZZ algorithm [129], called LCC-SZZ (Logging Code Change-based SZZ), to automatically locate the LCII changes from their fixes. By leveraging this algorithm, we can extract the LCII changes among various code revisions.

3. To ease replication and encourage further research in the area of "how-to-log", we have provided a large-scale benchmarking dataset, which contains 8,748 LCII changes from six large-scale open source systems, each consisting of six to ten years of development history [13]. Such a dataset, which is the first of its kind to the authors' knowledge, can be very useful for interested researchers to develop new techniques to characterize and detect ALCs or to derive coding guidelines on effective software logging.

4. We have conducted some preliminary studies on the extracted LCII change dataset and reported four main findings: (1) both co-changed and independently changed logging code can contain fixes to LCII changes; (2) the complexity of the LCII changes and other logging code changes are similar; (3) it takes significantly longer to fix a logging code issue than a regular bug, and (4) although none of the existing techniques on detecting logging code issues perform well ($< 3\%$ recall), their detection results complement each other. Our findings clearly indicate the need in this area of research and demonstrate the usefulness of our approach and the provided dataset.

Chapter Organization

The rest of the chapter is organized as follows. Section 3.2 gives an overview about the extraction process for the LCII changes. Section 3.3, 3.4, and 3.5 illustrate the three phases of our extraction approach. Section 3.6 describes our preliminary studies on the extracted dataset. Section 3.7 presents the related work. Section 3.8 discusses the threats to validity. Section 3.9 concludes the chapter and describes some of the future work.

3.2 An Overview of Our Approach to Extracting the LCII Changes

In this section, we will explain our approach to extracting the LCII changes from the software development history. As shown in Figure 3.2, our approach consists of the following three phases:

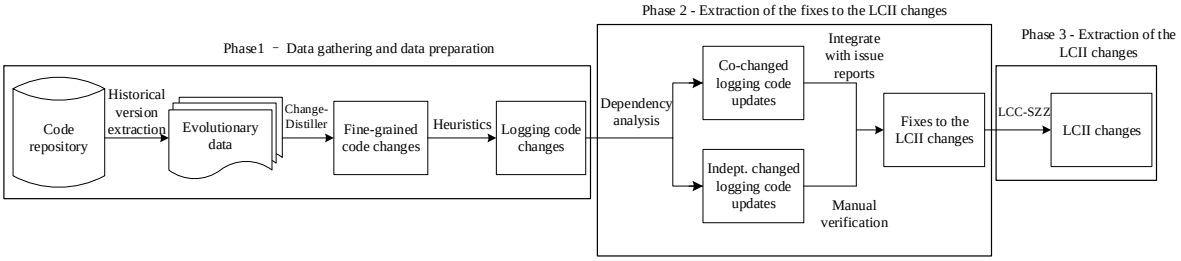

Figure 3.2: Overall process.

1. **Phase 1 - Data gathering and data preparation** (Section 3.3): this is the data gathering and pre-processing phase. First, the versions of each source file are extracted from the source code repositories. Then, all the bug reports are downloaded for the studied project. Fine-grained code changes (e.g., function update, variable renaming, etc.) are identified between adjacent versions of the same files. Finally, heuristics are applied to automatically identify the changes which are related to the logging code.

2. **Phase 2 - Extraction of the fixes to the LCII changes** (Section 3.4): in order to identify the LCII changes, we first need to identify their fixes among all the logging code changes. In this phase, we extract the fixes to the LCII changes by carefully examining both the independently changed and co-changed logging code changes.

3. **Phase 3 - Extraction of the LCII changes** (Section 3.5): unlike the feature code changes in which each changed line can be traced back to a previous version, the process of identifying the LCII changes is different. As the logging code is tangled with the feature code, one line of logging code changes can be related to multiple feature code changes. For example, in one code commit, developers may update the static texts (due to method renaming) as well as the method invocations (due to changes in the method signatures) in one single line of the logging code. Hence, in this phase, we have developed an adapted version of the SZZ algorithm, called LCC-SZZ, to automatically identify LCII changes.

The next three sections (Section 3.3, 3.4, and 3.5) will explain the above three phases in details.

Table 3.1: Studied projects.

Project	Description	Bug Repository	Code History	LCC
Elasticsearch	Distributed analysis engine	GitHub	(2010-02-08, 2017-03-30)	2,781
Hadoop	Distributed compute platform	Jira	(2009-05-19, 2017-08-22)	2,652
HBase	Distributed database	Jira	(2007-04-03, 2017-05-05)	3,638
Hibernate	ORM framework	Jira	(2007-06-29, 2017-07-25)	2,619
Geronimo	Server runtime	Jira	(2003-08-07, 2013-03-21)	1,019
Wildfly	Application server	Jira, GitHub	(2010-05-28, 2017-03-30)	3,401

3.3 Phase 1 - Data Gathering and Data Preparation

We will first briefly describe the studied projects. Then we will explain our process to extract the logging code changes and bug reports.

3.3.1 Studied Projects

In this chapter, we focus our study on six popular Java-based open source software projects: Elasticsearch, Hadoop, HBase, Hibernate, Geronimo, and Wildfly. Table 3.1 provides an overview of the studied projects. These projects are from different application domains. The selected projects are either server-side or supporting component-based projects, since the previous study [9] showed that software logging is more pervasive and more actively maintained in these two categories than in client-side projects. Each studied project has one or more bug repositories (GitHub or Jira) as shown in the third column of Table 3.1. As shown in the fourth column of the table, all the selected projects have a relatively long development history: ranging between six to ten years. The fifth column (LCC) shows the total number of logging code changes throughout the development history. LCC stands for the Logging Code Changes. It refers to the code changes which are directly related to software logging. Additional/replacement/removal of a variable inside a debug statement, changes to the verbosity levels, or adding and removing the static texts are three examples of LCCs. Here we only focus on the direct changes to the logging code. For example, if only the condition outside a logging statement is changed, such change is not considered as an LCC, as the changes may related to some feature code changes. As we can see the logging code from all six projects are actively maintained: each containing thousands of logging code changes.

There are two different types of data that we need to extract from the historical repositories in order to identify the LCII changes: the logging code changes, and the issue reports.

3.3.2 Logging Code Changes

All six studied projects use GitHub as their source code repositories. We wrote a script to automatically extract the code version history from the master branch. We only focused on the code changes committed to the master branch, as the logging code changes committed there had been carefully reviewed. The extracted data includes every version of every source code file , along with the meta information of each code version (e.g., the commit hash, the commit date, the authors, etc.). For every file, we then sorted these code versions by their commit timestamps. For example, if the file `Foo.java` was changed in two commits (*hash:f4b214* and *hash:a7cc6b*), which correspond to the first and the second commits, it would result in two extracted versions named `Foo_v1.java` and `Foo_v2.java`.

We ran ChangeDistiller (CD) [130] to get the source code level changes between two adjacent versions. CD parses two adjacent versions (e.g., `Foo_v1.java` and `Foo_v2.java`) of the same the file into Abstract Syntax Trees (ASTs) and compares the ASTs using a tree differencing algorithm. The output of CD contains a list of fine-grained code changes from four categories: insertion, deletion, update, and move. Since our focus is on the changes to the existing logging code, we only analyzed the updated code changes like update to a particular method invocation or update to a variable assignment.

Once we obtained the source code level changes, we applied keyword-based heuristics to filter out the non-logging code changes. We searched for commit messages which include the words like "log", "trace", and "debug". We ruled out the changes which contained the mismatched words like "dialog", "login", etc. We also excluded the logging code which did not print any messages (e.g., logging verbosity level setting statement like `log.setLevel(org.apache.log4j.Level.toLevel(level))`). Such keyword-based heuristics have been used in many of the previous studies (e.g., [9, 36, 8, 131, 10]) to identify logging code changes with high precision.

3.3.3 Issue Reports

The studied projects use two kinds of bug tracking systems: Jira and GitHub. For Jira-based projects (Hadoop, HBase, Hibernate, Geronimo, and Wildfly), we followed a similar approach in [9] to crawl the Jira reports using the provided Jira APIs. For Elasticsearch and Wildfly, their issues are managed through GitHub in the form of pull requests and GitHub issues. We used the public APIs to crawl the related GitHub data.

3.4 Phase 2 - Extraction of the Fixes to the LCII Changes

We have categorized the logging code changes into the following two categories:

- *co-changed logging code*, in which the logging code is updated together with the corresponding feature code. In [36], there is an assumption that the fixes to the LCII changes only exist in the independently changed logging code. It is not clear whether such an assumption is valid or not.
- *independently changed logging code*, refers to the logging code changes which are not classified as co-changed logging code changes. Unlike the co-changed logging code changes, which are generally updated along with the feature code, independently changed logging code is usually about fixing LCII changes. However, it is not clear whether all the independently changed logging code is actually used to fix the LCII changes.

In the previous study [9], we have identified the following eight scenarios of the co-changed logging code changes: (1) co-change with condition expressions, (2) co-change with variable declarations, (3) co-change with feature methods, (4) co-change with class attributes, (5) co-change with variable assignment, (6) co-change with string method invocations, (7) co-change with method parameters, and (8) co-change with exception conditions. We encoded these rules to automatically identify the co-changed logging code changes. The remaining logging code changes belong to the independently changed logging code changes.

In the rest of this section, we will explain our process to extract fixes to the LCII changes by mining the independently changed and co-changed logging code changes. During this process, we will validate the above two assumptions.

3.4.1 Extracting Fixes to the LCII Changes from the Co-changed Logging Code Changes

We extract fixes to the LCII changes from the co-change logging code changes by leveraging the information contained in the filed bug reports. Usually, developers would include the bug report IDs in the commit logs for traceability [132] and code review [133]. For example, in Hadoop, there is a commit with the commit log stating the following: "*HADOOP-15198. Correct the spelling in CopyFilter.java. Contributed by Mukul Kumar Singh.*". This commit refers to the bug ID `HADOOP-15198`, whose details can be found by searching this Jira ID online (`https://issues.apache.org/jira/browse/HADOOP-15198`). Hence, we extracted the bug IDs mentioned in the commit logs to link to the corresponding Jira issues. Then we performed keyword-based filtering to only include the bug reports which addressed the logging issues by searching the subject of the issue reports with keywords like "logging", "logs", and "logging code".

We did not search the same keywords in the bug description and the comments sections. Many of the reported issues include logs or logging code in their bug descriptions or comments as an artifact to help facilitate developers to understand or reproduce the reported issues. Thus, searching through these two sections would cause too much noise in the resulting dataset. For example, Hadoop issue `HADOOP-12666` [134] is about "Support Microsoft Azure Data Lake" and in the discussion, someone pasted a code snippet which contained logging

code. However, this issue report is obviously not related to the issues in logging code. Hence, to avoid too many false positives, we chose to only match those keywords in the title of the bug reports.

The six studied projects contain a total of 7,092 co-changed logging code changes. There are 553 (8%) co-changed logging code changes which are linked to the log-related issue reports. For each of these changes, we manually categorized their change types based on their intentions and whether they belong to the fixes to the LCII changes. There are 130 logging code changes, which are not related to the LCII changes. Figure 3.3 shows one such example from the `Server.java` file in the Hadoop project. This code change is linked to the Hadoop issue `HADOOP-7358` [135]. The attribute of the `call` object was changed from `id` to `callId`. However, the bug report was about improving "log levels when exceptions caught in RPC handler", which is not related to this logging code changes. After filtering such irreverent changes, we ended up with 423 co-changed logging code changes, which are fixes to the LCII changes.

Server.java from *Hadoop (HADOOP-7358)*

V 1413:
```
LOG.debug(getName() + ": responding to #" + call.callId + " from " + call.connection);
```

V 1630:
```
LOG.debug(getName() + ": responding to #" + call.id + " from " + call.connection);
```

Figure 3.3: An example of the co-changed logging code, which is linked to a log-related bug report. However, it is not a fix to an LCII change.

3.4.2 Extracting Fixes to the LCII Changes from the Independently Changed Logging Code Changes

There are a total of 9,018 independently changed logging code changes. Due to its sheer size, we can only manually examine a few sampled instances. We randomly selected 369 independently changed logging code changes for manual investigation. This corresponds to a confidence level of 95% with a confidence interval of 5%.

During our analysis, we found one scenario of the independently changed logging code changes, which are not fixes to the LCII changes. This type of logging code changes modifies the printing format (e.g., adding or removing spaces) without changing the actual contents. Figure 3.4 shows one such example. The only change for that logging statement in the new revision was adding a space after the word `client` in the static texts. There were no changes in any of the four logging components (logging library, verbosity level, static texts, or dynamic information). Hence, we automatically filtered out all the printing format changes from all the independently changed logging code changes using a script. We ended up with 8,325 (92%) independently changed logging code changes, which are fixes to the LCII changes.

We have further investigated the relation between the independently changed logging code changes and the reported log-related issues. We have located the issue IDs referenced in

the commit logs of these changes and tried to verify if these issue reports contained any log related keywords in their titles. It turns out that only 10.4% of the independently changed logging code changes have corresponding log-related issue reports. This verified our argument in Section 5.1 that many issues in the logging code are generally not documented in the commit logs or in the bug reports.

RpcProgramNfs3.java from *Hadoop (HDFS-7423)*

V 8351:
```
LOG.debug("GETATTR for fileId: " + handle.getFileId() + " client:"+ remoteAddress);
```

V 8428:
```
LOG.debug("GETATTR for fileId: " + handle.getFileId() + " client: "+ remoteAddress);
```

Figure 3.4: An example of the printing format change.

3.5 Phase 3 - Extraction of the LCII Changes

In the previous phase, we have generated a dataset which contains the fixes to the LCII changes. In this phase, we will extract the code commits containing the LCII changes.

3.5.1 Our Approach to Extracting the LCII Changes

We used an adapted approach of the SZZ algorithm [129] to automatically identify the LCII changes from their fixes. The SZZ algorithm [129] is an approach which automatically identifies the bug introducing changes from their fixes. It first tags code changes to bug fixing changes by searching for bug reports. Then it traces through the code revision history to find when the changed code was introduced (bug introducing changes). There are many follow-up work to further enhance the effectiveness of this approach [136, 128, 129]. However, all these approaches are focused on identifying bugs in the feature code, directly applying these approaches on the fixes to LCII changes may lead to incomplete or incorrect results. Different from the feature code changes, one line of logging code changes can be related to multiple previous lines of the logging code changes.

We will illustrate the problem of using the SZZ algorithm to locate the LCII changes using an example shown in Figure 3.7. V_n represents the version of the file. The file was changed through V_1 to V_{40} while the logging code was introduced at V_1 and subsequently was changed at V_{20} and V_{40}. The change at V_{40} is a fix to the LCII change since it corrected the verbosity level from `info` to `fatal` in order to be consistent with the text `Fatal error`. In addition, in the same change, the developer corrected a typo from `adddress` to `address` in the static texts. By using the SZZ algorithm, we find out that the problematic logging code (at V_{39}) was introduced at V_{20}. Therefore, we considered the LCII change to be at V_{20}. However, after examining the entire code revision history related to this logging code snippet, we noticed that the logging code at V_{20} is not the only LCII change. There are two problems associated

with the logging code at V_{39}: the verbosity level and the static texts. The typo of the static texts was introduced in (V_{20}), and the verbosity level `info` was introduced when the logging code was initially added (V_1). Hence, for fix version V_{40}, there are two LCII change versions: V_1 and V_{20}.

The main problem with the original SZZ algorithm is that it would consider logging code as one entity instead of treating the various components (logging object, verbosity level, static texts, and dynamic contents) of the logging code separately. To cope with this problem, we have developed an adapted version of the SZZ algorithm, called LCC-SZZ. There are two inputs for the LCC-SZZ algorithm: (1) a list containing the historical revisions of a particular snippet of the logging code, and (2) the versions at which the logging code are fixed to resolve the LCII change. The output of the LCC-SZZ algorithm is the version(s) of LCII change(s). The pseudo code of this algorithm is shown in Figure 3.5.

The whole process contains two steps: (1) formulating a list of component chains; and (2) finding the version(s) of the LCII change(s) for each fixed component.

In step 1, LCC-SZZ breaks down each version of that logging code snippet into various components. Each component has a type and a string representation. The type could be logging library, verbosity level, static texts, or dynamic information (e.g. variables, method invocations, etc.). The string representation is the value of that component. For example, the string representation of a variable is the variable name. We then track the historical changes for each component and formalize them as component chains. Therefore, for a list of logging code, we have multiple component chains and grouop them together as the component chain list. This step is done by the procedure `CHAIN_FORMULATION`. In the beginning, the component chain list is created as an empty list on line 2. From line 3 to 6, the components for each version of that logging code snippet are extracted. Then this extracted data is transformed through the `CHAIN_FORMULATION` procedure. The inputs for `CHAIN_FORMULATION` are the extracted components from the logging code and the component chain list. The `for` loop on line 15 is used to iterate through all components to match with the existing component chains. The `for` loop on line 17 is used to iterate through all the component chains to see if the current component can be matched to any of them. As shown on line 18, the component needs to have the same type with the component chain. If the component type is logging library or verbosity level, it will be put into the corresponding component chain (shown from line 19 to 22); as each logging code snippet only contains one logging library and one verbosity level. For other types of components, they are analyzed using their string representations. The old and the new string representations of the component are checked to see if they are similar from line 24 to 29 based on the two following criteria: (1) if the string edit distance between the two component string representations is less than 3; or (2) if the length of the longest substring from the two component string representations are larger than 3. We choose threshold value to be 3 based on a trial and error process. If either one of the above criteria is true, they are considered to be similar, and the current component is inserted to this chain as a new node. If there is no match found, we will initialize a new component chain with this component to be the head of the chain. This process is implemented on line 33. This component chain is then added to the component chain list. In the end, for a list of logging

code, we have formulated a list of component chains (i.e., the `component_chain_list`). In our example shown in Figure 3.7, the logging code list contains three lines of logging code: V_1, V_{20}, and V_{40}. There are two fix versions: V_{20} and V_{40}. There are four component chains in the component chain list for this example. The component chain for the verbosity level is: "info(V_1) ← info(V_{20}) ← fatal(V_{40})". The component chain for the variable is: "ipAddress(V_1) ← ipComplexAddress(V_{20}) ← ipComplexAddress(V_{40})". All four resulting component chains are shown in Figure 3.6.

After the formulation of the component chain list, the next step (a.k.a., step 2) is to find the issue introducing version given a specified fix version. This process is explained in the procedure `EXTRACT_ISSUE_INTRODUCING_VERSION` starting from line 38. All the components which are changed in the fix version are retrieved. Then the components from the previous version of the fix version are picked, as they contain these issues. This step is implemented from line 40 to 43. For the components with issues in the previous version, the search continues until the first version when the issue appears is found. This process is implemented through the `while` loop on line 44. In our example, the fix versions are V_{40} and V_{20}. In fix version V_{40}, both the verbosity level, and the static texts are changed. Hence, the previous version (V_{20}) is examined. In V_{20}, the verbosity level is `info` whose first appearance is in the initial version of this logging code snippet, V_1. Similarly, at V_{40}, the spelling of `adddress` is changed to `address`, and the first appearance of this issue is at V_{20}. Therefore, for fix version V_{40}, there are two issue introducing versions: V_1 and V_{20}. In fix version V_{20}, the variable `ipAddress` is changed to `ipComplexAddress`. The issue introducing version is V_1. Hence, the LCII change versions for fix version V_{40} are V_1 and V_{20}. The LCII change version for fix version V_{20} is V_1.

3.5.2 Evaluation

To evaluate and compare the effectiveness of the LCC-SZZ and the SZZ algorithms, we used an approach similar to [137]. First, we applied both algorithms on the fixes to the LCII changes. Then we compared the results of the two algorithms from the following three dimensions: (1) disagreement ratio between the results from the two algorithms; (2) the earliest appearance of the LCII changes; and (3) manual verification. The first dimension is a general estimation of how these two algorithms differ. The second dimension is concerned with the discrepancies between the two algorithms when compared to the estimates given by the development team. The third dimension is to estimate the accuracy by comparing against a human oracle.

The disagreement ratio between SZZ and LCC-SZZ

We calculated the disagreement ratio between the introducing versions generated by SZZ and LCC-SZZ. The results are shown in Table 3.2. In total, 12.7% of the results are different. Among the six studied projects, HBase has the largest difference ratio (16.8%) and Geronimo has the lowest difference ratio (8.8%).

Algorithm 1 The pseudo code of the LCC-SZZ algorithm.

```
    Input: logging_code_list, fix_version_list
 1: procedure LCC_SZZ(logging_code_list, fix_version_list)
 2:     component_chain_list ← [ ]
 3:     for logging_code in logging_code_list do
 4:         extracted_components ← extract_components(logging_code)
 5:         CHAIN_FORMULATION(extracted_components,component_chain_list)
 6:     end for
 7:     for fix_version in fix_version_list do
 8:         for component_chain in component_chain_list do
 9:             EXTRACT_ISSUE_INTRODUCING_VERSION(fix_version, component_chain)
10:         end for
11:     end for
12: end procedure
13:
14: procedure CHAIN_FORMULATION(extracted_components, component_chain_list)
15:     for component in extracted_components do
16:         find_chain ← false
17:         for component_chain in component_chain_list do
18:             if component.type=component_chain.type then
19:                 if component.type = LEVEL or component.type = LIB then
20:                     component_chain.add(component)
21:                     find_chain ← true
22:                     break
23:                 end if
24:                 latest_component ← latest_component(component_chain)
25:                 if similar_string(component,latest_component) = true  then
26:                     component_chain.add(component)
27:                     find_chain ← true
28:                     break
29:                 end if
30:             end if
31:         end for
32:         if find_chain = false then
33:             init_new_chain(component, component_chain_list)
34:         end if
35:     end for
36: end procedure
37:
38: procedure EXTRACT_ISSUE_INTRODUCING_VERSION(fix_version, component_chain)
39:     for component in component_chain do
40:         if component.version = fix_version then
41:             issue_component ← component.previous
42:             issue_component_content ← component.previous.content
43:             if issue_component_content != component.content then
44:                 while issue_component.content = issue_component_content do
45:                     if issue_component.previous = null  then
46:                         break
47:                     end if
48:                     issue_component ← issue_component.previous
49:                 end while
50:             end if
51:             print issue_component.version
52:         end if
53:     end for
54: end procedure
```

Figure 3.5: Algorithm 1 - The pseudo code of the LCC-SZZ algorithm

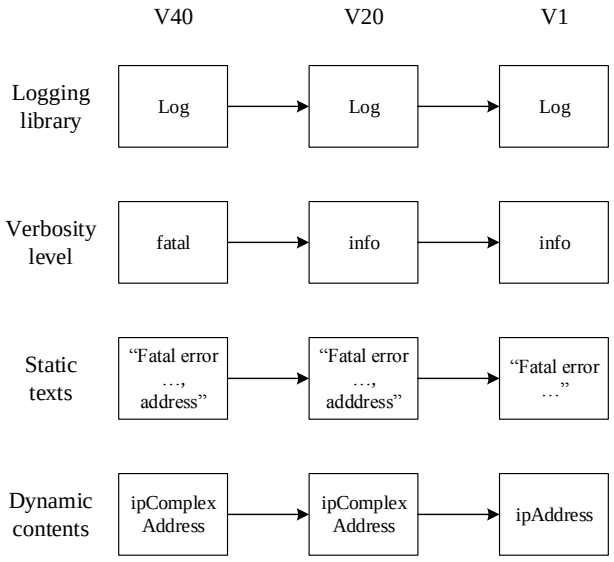

Figure 3.6: The resulting component chains.

V 1 : `Log.info("Fatal error occur in execution: " + ipAddress);`
(Logging code first introduced)
V 20: `Log.info("Fatal error occur in execution, adddress: " + ipComplexAddress);`
(Update info and typo introduced)
V 40: `Log.fatal("Fatal error occur in execution, address: " + ipComplexAddress);`
(Fix level and typo)

Figure 3.7: An example of the logging code changes.

Table 3.2: Difference ratio of computed introducing versions by SZZ and LCC-SZZ.

Project	**Same**	**Different**	**Total**
Elasticsearch	758 (90.5%)	80 (9.5%)	838
Hadoop	905 (87.8%)	125 (12.2%)	1,030
HBase	1,276 (83.2%)	257 (16.8%)	1,533
Hibernate	1,657 (83.5%)	328 (16.5%)	1,985
Geronimo	489 (91.2%)	47 (8.8%)	536
Wildfly	2,553 (90.3%)	273 (9.7%)	2,826
Total	7,638 (87.3%)	1,110 (12.7%)	8,748

Table 3.3: Comparing the time from the earliest bug appearance and the LCII code commit timestamp.

Project	After affected version (SZZ)	After affected version (LCC-SZZ)	Bug reports with affected version	# of bug report
Hadoop	41	29	286	407
HBase	27	25	87	397
Hibernate	2	2	141	332
Geronimo	0	0	21	32
Total	70 (13%)	56 (10%)	535	1,168

The earliest appearance of the LCII changes

The evaluation on the earliest LCII changes is to compare the results from the SZZ and LCC-SZZ algorithms with the estimates provided by the development community. This evaluation dimension is not meant to tell whether SZZ/LCC-SZZ is absolutely correct. Rather, it aims to point out the obviously incorrectly outputted changes. Within each bug report, there is a field called "affected-version" showing the versions of the project that the logging issue impacted. For example, in HDFS, a component of the Hadoop system, the Jira issue `HDFS-4122` [138] shows the affected HDFS versions are 1.0.0 and 2.0.0-alpha. As the issue introducing date cannot be after the earliest affected version, we can use such a dataset to evaluate the SZZ/LCC-SZZ results. We consider the results from SZZ/LCC-SZZ to be incorrect if the resulting date reported by either algorithm is after the earliest affected version date. To notice, if the LCC-SZZ yields multiple version results, we use the date of the earliest version. For example, if the date of the earliest affected version is January 1, 2012 and the LCII changes outputted by the SZZ algorithm is October 1, 2012, we consider this output by the SZZ algorithm to be incorrect. However, if the resulting date is before the earliest affected version, we cannot judge the correctness of the outputted changes. Table 3.3 shows the details.

For Hadoop, HBase, Hibernate, and Geronimo, less than half ($\frac{535}{1168} \times 100\% = 46\%$) of the bug reports have valid data in the "affected-version" field. Using the SZZ algorithm, 13% of the computed versions are after the earliest affected versions (a.k.a., incorrectly identified LCII changes). Using the LCC-SZZ algorithm, this number reduces to 10%, which is a 30% improvement compared to the original SZZ algorithm. The majority of these improvements come from Hadoop, which also happens to contain the largest number of bug reports with valid data in the "affected-version" field.

Manual verification

To further evaluate the results, we did a stratified sampling on all the fixes to the LCII changes, and went through the logging code change history to find the commits where the

Table 3.4: Consistency compared to manual oracles.

Project	SZZ	LCC-SZZ	Total
Elasticsearch	25	29	35
Hadoop	37	41	43
HBase	53	60	64
Hibernate	75	76	84
Geronimo	20	22	23
Wildfly	105	115	119
Total	315	343	368

```
V 658:  logger.debug("Index [" + index + "]: Update mapping ["+ type+ "]
        (dynamic) with source ["+ updatedMappingSource+ "]")
V 715:  logger.debug("index [" + index + "]: Update mapping ["+ type+ "]
        (dynamic) with source ["+ updatedMappingSource+ "]")
V 736:  logger.debug("[{}] update mapping [{}] (dynamic) with source
        [{}]",index,type,updatedMappingSource)
```

MetaDataService.java in *ElasticSearch*

Figure 3.8: Examples of both SZZ and LCC-SZZ failed.

logging code snippets became issues. In this way, we generated a manually verified oracle dataset for the correctly identified LCII changes.

We calculated the difference ratio between the oracle and results from the SZZ/LCC-SZZ algorithms. The results are shown in Table 3.4. In total, we manually examined 368 fixes to the LCII changes and derived the oracle for code commits containing the LCII changes. 85% of the SZZ detecting results agree with the oracle while 93% of the LCC-SZZ results agree with the oracle. All of the LCII changes correctly identified by the SZZ algorithm are also correctly identified in the LCC-SZZ results.

For the case that both LCC-SZZ and SZZ failed, we manually examined a few of them. We found that the misclassified instances are mainly related to logging style changes. Figure 3.8 shows an example. The logging code style was changed to format printing from string concatenation. The logging code applied string concatenation style since introduction at version 658. Therefore, version 658 should be the LCII change version. However, there is a change from "Index " to "index" (fixing typos) from version 658 to 715. Both SZZ and LCC-SZZ mistakenly label 715 as the version containing the LCII changes.

Summary: our evaluation results show that: (1) 13.6% of the identified LCII changes obtained from the LCC-SZZ algorithm are different from the original SZZ algorithm; (2) when evaluating under the earliest appearance of the LCII changes, there are more incorrectly flagged results (3%) by the SZZ algorithm compared to the LCC-SZZ algorithm; (3) the LCC-SZZ algorithm achieves 93% accuracy when compared to the oracle, which is an 8% improvement compared to results from the SZZ algorithm.

3.6 Preliminary Studies

In the previous sections (Sections 3.3, 3.4, 3.5), we have explained our approach to extracting the LCII changes and evaluated the quality of the resulting dataset (93% accuracy based on manual verification). In this section, we will perform a few preliminary studies on this dataset to highlight the usefulness of such data and discuss some of the open problems. We want to study the characteristics of LCII changes by characterizing the intentions behind the fixes to the LCII changes (RQ1), studying the complexity of their fixes (RQ2), the duration to address these issues (RQ3), and the effectiveness of existing automated approaches to detecting logging code issues (RQ4). For each of the research question, we will describe our extraction process for the required dataset, explaining the data analysis process, and discuss its findings and implications.

3.6.1 RQ1: What are the intentions behind the fixes to the LCII changes?

In this RQ, we will characterize the intentions behind the fixes to the LCII changes. We intend to do this based on the two types of logging code changes from two dimensions:(1) what-to-log vs. how-to-log, and (2) co-changed logging code changes vs. independently changed logging code changes.

Data Extraction

Table 3.5 shows the break down of the total number of logging code changes and the number of fixes to the LCII changes. TL shows the total number of logging code changes; IND shows the number of independently changed logging code changes; IND_FIX shows the number of IND which are fixes to LCII changes; CO shows the number of co-changed logging code changes; CO_FIX shows the number of CO which are fixes to the LCII changes; and T_FIX shows the total number of fixes to LCII changes. For example, in Hadoop, out of 2,652 logging code changes, 1,030 (38.8%) are fixes to the logging code changes. Out of these 1,030 fixes, 913 (88.6%) are independently changed logging code changes and 117 (11.4%) are co-changed logging code changes. Most of the independently changed logging code changes (92.3%) are fixes to the LCII changes. However, when we look at the co-changed logging code changes, majority of them (94.0%) are co-changes with the features code instead of fixes to

Project	TL	IND	IND_FIX	CO	CO_FIX	**T_FIX**
Elasticsearch	2,781	1,004	818	1,777	20	**838**
Hadoop	2,652	1,121	913	1,531	117	**1,030**
HBase	3,638	1,497	1,413	2,141	120	**1,533**
Hibernate	2,619	1,958	1,921	661	64	**1,985**
Geronimo	1,019	652	527	367	9	**536**
Wildfly	3,401	2,786	2,733	615	93	**2,826**
Total	16,110	9,018	8,325	7,092	423	**8,748**

Table 3.5: Summary of the fixes to LCII changes.

the LCII changes. Among the total 8,748 fixes to LCII changes, majority of which (95.0%) are independently changed logging code changes.

An Overview of the Data Analysis Process

Since there are only 423 co-changed logging code changes, which are fixes to the LCII changes, we manually studied all of them. On the other hand, there are much more independently changed logging code changes than the co-changed logging code changes (8,325 vs. 423). It would take too much time for us to examine all the instances manually. Hence, we only examined 367 randomly sampled instances. This sample size corresponds to a confidence level of 95% with a confidence interval of 5%. When selecting the sample instances for manual inspection, we applied the stratified sampling technique [9] to ensure the representative samples are selected and studied from each project. Using the stratified sampling approach, the portion of sampled logging code changes from each project is equal to the relative weight of the total number of independently changed logging code changes. For example, among the total 8,325 independently changed logging code changes from the six studied projects, 913 of them are from Hadoop. Thus, 40 (a.k.a., $367 \times \frac{913}{8325}$) of the manually studied independently changed code changes were selected from Hadoop. The detailed breakdown of our characterization results is shown in Table 3.6.

In total, we have characterized 4 intentions and 7 intentions behind the what-to-log vs. how-to-log dimension. Among co-changed logging code changes, 27% ($\frac{114}{423} \times 100\%$) are "Add Logging Guards (ALG)", which is the most common intention. Among independently changed logging code changes, the majority of the fixes ($\frac{255}{367} \times 100\% = 70\%$) are for "Updating Logging Style (ULS)". There are also other intentions. However, since the number of these intentions is small, we just grouped them into a category called "Others". We will explain below using real-world code examples for each characterized intention.

Category	Rationale	Co-changed	Independently
What-to-log	Adding More Information (AMI)	64	24
	CLArification (CLA)	84	25
	Fixing Language Problems (FLP)	14	38
	Avoid Excessive Logging (AEL)	38	15
How-to-log	Checking Nullable Variables(CNV)	0	1
	Changing Object Casting (COC)	0	1
	Refactoring Logging Code (RLC)	0	1
	Changing Output Format (COF)	0	1
	Updating Logging Style (ULS)	82	255
	Adding Logging Guards (ALG)	114	0
	Mess-up Commits in VCS (MCV)	0	6
Others	-	27	0
Total	-	423	367

Table 3.6: Our manual characterization results on intentions behind the fixes to LCII changes

Detailed Characterization of the Intentions behind the LCII changes

Here we will explain our detailed characterization of the intentions behind the LCII changes from the following three categories: what-to-log, how-to-log, and others.

Figure 3.9 shows the real world code examples for each intention for the fixes of the LCII changes behind the what-to-log category:

- *Adding More Information (AMI)*: extra information is added to the logging code for the purpose of providing additional contextual information. Both co-changed and independently changed logging code changes can have this intention. Figure 3.9(a) shows an example for co-changed logging code changes. A method `getClientIdAuditPrefix` is added in the newer version in order to provide additional runtime context [139]: "this (client IP/port of the balancer invoker) is a critical piece of admin functionality, we should log the IP for it at least ...". An independently changed logging code changes, as shown in Figure 3.9(b), `this.serverName` was added to provide more information about the SplitLogWorker.

- *CLArification (CLA)*: to clarify the runtime context, some of the outdated or imprecise dynamic information (e.g., local variables or class attributes) is fixed. Both co-changed and independently changed logging code changes can have this intention. Figure 3.9(c) shows an example of the co-changed logging code changes. The variable `numEdits` is updated to `numEditsThisLog` to output a more accurate number of edits required for this logging context [140]. Figure 3.9(d) shows an example for the independently changed

<table>
<tr><th>Intention</th><th>Change Type</th><th>Example</th></tr>
<tr><td rowspan="2">AMI</td><td>Co</td><td>

```
LOG.debug("Submitting snapshot request for:" +
        ClientSnapshotDescriptionUtils.toString(request.getSnapshot()));
```

```
String getClientIdAuditPrefix() {
  return "Client=" + RequestContext.getRequestUserName() + "/" +
        RequestContext.get().getRemoteAddress(); }
LOG.info(getClientIdAuditPrefix() + " snapshot request for:" +
        ClientSnapshotDescriptionUtils.toString(request.getSnapshot()));}
```

(a) HMaster.java *in HBase*
</td></tr>
<tr><td>Ind</td><td>

```
LOG.info("SplitLogWorker starting");
```

```
LOG.info("SplitLogWorker " + this.serverName + " starting");
```

(b) SplitLogWorker.java *in HBase*
</td></tr>
<tr><td rowspan="2">CLA</td><td>Co</td><td>

```
int numEdits = new FSEditLogLoader(namesystem).loadFSEdits(new EditLogFileInputStream(editFile));
System.out.println("Number of edits: " + numEdits);
```

```
int numEditsThisLog = loader.loadFSEdits(new EditLogFileInputStream(editFile), startTxId);
System.out.println("Number of edits: " + numEditsThisLog);
```

(c) TestEditLogRace.java *in Hadoop*
</td></tr>
<tr><td>Ind</td><td>

```
LOG.error("Can't make a speculator -- check " + AMConstants.SPECULATOR_CLASS + " " + ex);
```

```
LOG.error("Can't make a speculator -- check " + MRJobConfig.MR_AM_JOB_SPECULATOR, ex);
```

(d) MRAppMaster.java *in Hadoop*
</td></tr>
<tr><td rowspan="2">FLP</td><td>Co</td><td>

```
private static final String AUTHZ_SUCCESSFULL_FOR = "Authorization successfull for ";
AUDITLOG.info(AUTHZ_SUCCESSFULL_FOR + user + " for protocol="+protocol);
```

```
private static final String AUTHZ_SUCCESSFUL_FOR = "Authorization successful for ";
AUDITLOG.info(AUTHZ_SUCCESSFUL_FOR + user + " for protocol="+protocol);
```

(e) ServiceAuthorizationManager.java *in Hadoop*
</td></tr>
<tr><td>Ind</td><td>

```
MetricsUtil.LOG.error("Unexpected attrubute suffix");
```

```
MetricsUtil.LOG.error("Unexpected attribute suffix");
```

(f) MetricsDynamicMBeanBase.java *in Hadoop*
</td></tr>
<tr><td rowspan="2">AEL</td><td>Co</td><td>

```
LOG.error("Couldn't verify if the referenced file still exists, keep it just in case")
```

```
if (LOG.isDebugEnabled()) {
  LOG.debug("Couldn't verify if the referenced file still exists, keep it just in case: " +
          hfilePath)
```

(g) HFileLinkCleaner.java *in HBase*
</td></tr>
<tr><td>Ind</td><td>

```
LOG.info("Attempt #" + numAttempt + "/" + numTries + " failed all ops, trying resubmit," +
       " tableName=" + tableName + ", location=" + location);
```

```
LOG.info("Attempt #" + numAttempt + "/" + numTries + " failed all ops, trying resubmit," +
        location);
```

(h) AsyncProcess.java *in HBase*
</td></tr>
</table>

Figure 3.9: The intentions behind various fixes to the LCII changes, which are related to **what-to-log**. For each intention, we have included real-world code examples.

logging code change. The variable `AMConstants.SPECULATOR_CLASS` was replaced by `MRJobConfig.MR_AM_JOB_SPECULATOR` to better reflect the reported error.

- *Fixing Language Problems (FLP)*: the logging code is updated to fix typos or grammar mistakes in texts or dynamic information (e.g., class attributes, local variables, method names, and class names). Both co-changed and independently changed logging code changes can have this intention. Figure 3.9(e) shows an example for the co-changed logging code changes. The class attribute `AUTHZ_SUCCESSFULL_FOR` was misspelled, as reported in [141], and was corrected in the next revision. Figure 3.9(f) shows an example for the independently changed logging code changes. The word `attrubute` in the static texts was misspelled and corrected to `attribute` in the next version.

- *Avoid Excessive Logging (AEL)*: the logging code is updated to avoid excessive logging to reduce the runtime overhead. Both co-changed and independently changed logging code changes can have this intention. Figure 3.9(f) shows an example for the co-changed logging code changes. The verbosity level of the logging code was changed from `error` to `debug` and the logging guard was also added. This was to reduce the amount of logging to lessen the runtime overhead and the effort to analyze the log files [142]. Figure 3.9(h) shows an example for the independently changed logging code changes. The variable and the text `tableName` were deleted since the variable `location` already contained this information.

Figure 3.10 shows the real world code examples for each intention for the fixes of the LCII changes behind the how-to-log category:

- *Adding Log Guards (ALG)*: the condition statements are added to ensure that the logs are only generated for some scenarios. Only co-changed logging code changes have this intention. Figure 3.10(a) shows one such example. The log guard `LOG.isTraceEnabled()` is added to ensure the corresponding generated logs got printed when the trace level logging is enabled in the configuration settings [143]. Such logging code changes are considered as fixes to the LCII changes, since these changes can improve software performance by preventing unnecessary creation of strings.

- *Updating Logging Style (ULS)*: the logging code is updated due to the changes in the logging library/method/API. Both co-changed and independently changed logging code changes can have this intention. Figure 3.10(b) shows an example for the co-changed logging code changes. The new version of the logging code uses log-specific API (`methodInvocationFailed`) to generate logs as explained in the pull request [144] of the Wildfly project. This style is commonly used in some of the third-party logging libraries like JBoss [34]. Figure 3.10(c) shows an example for the independently changed logging code changes. The invoked method was changed from `trace` to `tracev`. The latter method requires the parameter to be formatted as `{0}` instead of string concatenation. Such changes can improve the readability of logging code and therefore ease the future maintenance activities on the logging code.

Intention	Change Type	Example

ALG	Co	`LOG.trace("Add sequence generator with name: " + idGen.getName());`
		`if (LOG.isTraceEnabled()) {`
		`  LOG.tracev("Add sequence generator with name: {0}", idGen.getName() ); }`
		(a) AnnotationBinder.java *in Hibernate*

ULS	Co	`log.error(MESSAGES.methodInvocationFailed(t.getLocalizedMessage()),t)`
		`WSLogger.ROOT_LOGGER.methodInvocationFailed(t,t.getLocalizedMessage())`
		(b) AbstractInvocationHandler.java *in Wildfly*

	Ind	`LOG.trace("Returning null as column " + names[0]);`
		`LOG.tracev("Returning null as column {0}", names[0] );`
		(c) EnumType.java *in Hibernate*

CNV	Ind	`LOG.info("Reassigning " + hris.size() + " region(s) that " + serverName + …);`
		`LOG.info("Reassigning " + (hris==null?0:hris.size()) + " region(s) that " + serverName + …);`
		(d) EnumType.java in *HBase*

COC	Ind	`logger.debug("binding server bootstrap to: {}", addresses);`
		`logger.debug("binding server bootstrap to: {}", (Object)addresses);`
		(e) NettyTransport.java in *Elasticsearch*

RLC	Ind	`LOG.warn("Error executing shell command " + Arrays.toString(shexec.getExecString()) + ioe);`
		`LOG.warn("Error executing shell command " + shexec.toString() + ioe);`
		(f) ProcessTree.java in *Hadoop*

COF	Ind	`LOG.debug("removing meta region " + info.getRegionName() +`
		`  " from online meta regions");`
		`LOG.debug("removing meta region " + Bytes.toString(info.getRegionName()) +`
		`  " from online meta regions");`
		(g) ProcessServerShutdown.java in *HBase*

MCV	Ind	`LOG.info("Timed out on waiting for region:" + hri.getEncodedName() + " to be assigned.")`
		`LOG.info("Timed out on waiting for" + hri.getEncodedName() + " to be assigned.")`
		`LOG.info("Timed out on waiting for region:" + hri.getEncodedName() + " to be assigned.")`
		(h) AssignmentManager.java in *HBase*

Figure 3.10: The intentions behind various fixes to the LCII changes, which are related to **how-to-log**. For each intention, we have included real-world code examples.

- *Checking Null Variable (CNV)*: the logging code is updated to check if the variable is null to prevent runtime failure. Only independently changed logging code changes have this intention. Figure 3.10(d) shows one such example. The null check of the variable `hris` was added to avoid the null pointer exception. Such changes improve the robustness of the logging code by preventing runtime failures caused by null pointer exceptions.

- *Changing Object Cast (COC)*:

  the logging code is updated to change the object cast of a variable. Only independently changed logging code changes have this intention. Figure 3.10(e) shows one such example. An explicit cast was added to the variable `addresses` to improve the formatting of the output string. Such changes can improve the readability of logging code and therefore ease the future maintenance activities on the logging code.

- *Refactoring Logging Code (RLC)*: the logging code is updated for refactoring purposes. Only independently changed logging code changes have this intention. Figure 3.10(f) shows one such example. For the variable `shexec`, both `Arrays.toString(shexec.getExecString())` and `shexec.toString()` would output the same string. Similar as the above fixes, such changes can improve the readability of logging code and therefore ease the future maintenance activities on the logging code.

- *Changing Output Format (COF)*: the logging code is updated to correct the malformed output. Only independently changed logging code changes have this intention. Figure 3.10(g) shows one such example. The type of return value of `info.getRegionName()` is `Byte`. If it is outputted directly, it will not be human readable. Hence, it was wrapped by the method `Bytes.toString()` to improve the readability and maintainability of the logging code.

- *Mess-up Commits in VCS (MCV)*: the logging code is updated due to mess-up commits in version control systems (VCS). Only independently changed logging code changes have this intention. Figure 3.10(h) shows one such example. The static text `region` was deleted at first and then added back. This is because developers first merged commits from another branch and later reverted the change. Such changes are mainly logging code maintenance activities. They may or may not improve the quality of the logging code.

Some fixes to the LCII changes are due to other reasons. Figure 3.11 shows one such example. The system functionality has been changed from renaming to moving as stated in the bug report [145]. Thus, the logging texts are updated as well. However, the numbers for these intentions are very small. Hence, we grouped these together into one category ("Others").

Intention	Change Type	Example

OTH	Co	`LOG.error("The endTxId of the temporary file is not less than the " + "last committed transaction id. Aborting renaming to final file" + finalFile)`
		`Files.move(tmpFile.toPath(), finalFile.toPath(), StandardCopyOption.ATOMIC_MOVE);`
		`LOG.error("The endTxId of the temporary file is not less than the " + "last committed transaction id. Aborting move to final file" + finalFile)`
		(a) Journal.java in *Hadoop (HDFS-11448)*

Figure 3.11: An example of fixes to LCII changes due to other reasons.

Breakdown for fixes to LCII changes per project

We further break down the characterization of the fixes to the LCII changes for each of the studied project. The results are shown in Table 3.7.

In general, ULS is the biggest group in three out of six projects (Hibernate, Geronimo, Wildfly). CLA is the biggest group in Elasticsearch (27%) and HBase (24%). In Hadoop, the biggest group is ALG. When we look among the intentions in the what-to-log category, CLA is the biggest category in five out of the six studies projects and AMI is the second biggest. This shows that developers tend to clarify or add additional information when they fix logging code issues related to what-to-log. In how-to-log, ULS is the biggest category in four out of six projects. The second biggest intention is ALG. There are less than 1% MCV instances due to the merging issue in the version control systems.

Summary

Findings: Contrary to our previous study [36], both co-changed and independently changed logging code changes can contain fixes to the LCII changes. The majority of the fixes to the LCII changes are from the independently changed logging code changes. In total, there are ten different intentions behind the fixes to the LCII changes: four are related to what-to-log, and seven are related to how-to-log. Among them, "Clarification" and "Updating Logging Style" are the top two intentions.
Implications: Our current approach to characterizing the fixes to the LCII changes is done manually. This is very time consuming and prevents us from studying larger datasets. Advanced text analysis approaches like topic modeling or natural language processing can be helpful to automatically cluster related code fixes and provide summarizations.

3.6.2 RQ2: Are the fixes to the LCII changes more complex than other logging code changes?

In the previous RQ, we have characterized the intentions behind different fixes to the LCII changes. In this RQ, we intend to compare the complexity between the fixes to the LCII

Table 3.7: Our manual characterization results on the intentions behind the fixes to the LCII changes

Category	Rationale	Elasticsearch	Hadoop	HBase	Hibernate	Geronimo	Wildfly
What-to-log	AMI	8	29	42	0	8	1
	CLA	15	30	44	9	1	10
	FLP	10	14	19	2	1	6
	AEL	12	15	24	0	1	1
How-to-log	CNV	0	0	1	0	0	0
	ROC	1	0	0	0	0	0
	RLC	0	1	0	0	0	0
	COF	0	0	1	0	0	0
	ULS	10	23	13	77	19	195
	ALG	0	38	17	57	2	0
	MCV	0	2	2	1	0	1
Others	-	0	5	22	0	0	0
Total	-	56	157	185	146	32	214

changes and other logging code changes. Since there are no existing metrics available to quantify the logging code change complexity, we have defined a complexity metric (the number of changed components of the logging code) to characterize how complex a logging code change is. Each logging code snippet contains four components: the logging library, the verbosity level, statics texts, and dynamic invocations. A logging code change is more complex, if there are more types of components changed together. The most complex logging code change is that all the components of a logging code snippet have been changed. On the other hand, some logging code changes only involve component position changes without actual content changes. For example, the following logging code snippet `log.info("text" + var);` could be changed to `log.info("text", var);`, since none of the components' contents have been changed. For such changes, the value of the complexity metric is "0".

Data Extraction

For each logging code change, we tracked whether any logging components have been changed using the heuristics similar to our previous work [9, 36] and calculated their change complexity metrics defined above.

Data Analysis

Table 3.8 shows the breakdown of different components for the fixes to the LCII changes and other logging code changes. Generally, there are large differences in three out of four components. 27.5% of the fixes to LCII changes contain logging library changes, while only 5.1% of the other logging code changes contain logging library changes. The portion of verbosity level changes in the fixes to the LCII changes and other logging code changes are 46.4% versus 8.2%. While the fixes to LCII changes contain more logging library changes and verbosity level changes, only 14.4% of them contain dynamic content changes. On the other hand, 78.6% of other logging code changes contain dynamic content changes. For static text changes, fixes to the LCII changes and other logging code changes are similar (42.9% versus 47.1%). For fixes to the LCII changes, the verbosity level changes rank first while dynamic content changes rank first in other logging code changes. Static text changes are ranked second in both types of logging code changes.

Table 3.9 shows the complexity of the logging code changes for each project. The majority of logging code changes are "1 type" changes (72.9% in the fixes to the LCII changes and 66.8% in other logging code changes). The second largest category is "2 types" logging code changes, 21.9% and 27.9% for the fixes to the LCII changes and other logging code changes, respectively. For "3 types" and "4 types" logging code changes, they only make up 4.8% and 5.3% of the fixes to the LCII changes and other logging code changes. There are 0.5% fixes to the LCII changes categorized as "0 type" changes. Hence, in terms of change complexity, fixes to the LCII changes and other logging code changes are similar.

Here we further examined the top two most types of most frequently occurred complexity changes: "1 type" and "2 types" changes. The results for the "1 type change" are shown in Table 3.10. For each component, there are two columns. The column on the left shows the

Table 3.8: Number of individual logging component changes.

Project	Lib	Level	Static	Dynamic	Total
Elasticsearch_LCII	35 (4.2%)	132 (15.8%)	603 (72.0%)	236 (28.2%)	838
Elasticsearch_Other	18 (0.9%)	47 (2.4%)	933 (48.0%)	1,633 (84.0%)	1,943
Hadoop_LCII	182 (17.7%)	360 (35.0%)	479 (46.5%)	240 (23.3%)	1,030
Hadoop_Other	28 (1.7%)	53 (3.3%)	738 (45.5%)	1,253 (77.3%)	1,622
HBase_LCII	122 (8.0%)	407 (26.5%)	875 (57.1%)	461 (30.1%)	1,533
HBase_Other	11 (0.5%)	48 (2.3%)	847 (40.2%)	1,858 (88.3%)	2,105
Hibernate_LCII	742 (37.4%)	1,292 (65.1%)	1,089 (54.9%)	84 (4.2%)	1,985
Hibernate_Other	251 (39.6%)	346 (54.6%)	443 (69.9%)	296 (46.7%)	634
Geronimo_LCII	123 (22.9%)	291 (54.3%)	141 (26.3%)	63 (11.8%)	536
Geronimo_Other	18 (3.7%)	24 (5.0%)	269 (55.7%)	293 (60.7%)	483
Wildfly_LCII	1,206 (42.7%)	1,575 (55.7%)	565 (20.0%)	177 (6.3%)	2,826
Wildfly_Other	48 (8.3%)	87 (15.1%)	236 (41.0%)	453 (78.8%)	575
Total_LCII	2,410 (27.5%)	4,057 (46.4%)	3,752 (42.9%)	1,261 (14.4%)	8,748
Total_Other	374 (5.1%)	605 (8.2%)	3,466 (47.1%)	5,786 (78.6%)	7,362

Table 3.9: The complexity of logging code changes grouped by each project.

Project	0 type	1 type	2 types	3 types	4 types	Total
Elasticsearch_LCII	1 (0.1%)	668 (79.7%)	169 (20.2%)	0 (0.0%)	0 (0.0%)	838
Elasticsearch_Other	0 (0.0%)	1,287 (66.2%)	625 (32.2%)	30 (1.5%)	1 (0.1%)	1,943
Hadoop_LCII	9 (0.9%)	799 (77.6%)	205 (19.9%)	16 (1.6%)	1 (0.1%)	1,030
Hadoop_Other	0 (0.0%)	1,209 (74.5%)	381 (23.5%)	27 (1.7%)	5 (0.3%)	1,622
HBase_LCII	1 (0.1%)	1,217 (79.4%)	298 (19.4%)	16 (1.0%)	1 (0.1%)	1,533
HBase_Other	0 (0.0%)	1,477 (70.2%)	598 (28.4%)	29 (1.4%)	1 (0.0%)	2,105
Hibernate_LCII	3 (0.2%)	952 (48.0%)	838 (42.2%)	189 (9.5%)	3 (0.2%)	1,985
Hibernate_Other	0 (0.0%)	181 (28.5%)	219 (34.5%)	219 (34.5%)	15 (2.4%)	634
Geronimo_LCII	15 (2.8%)	431 (80.4%)	84 (15.7%)	5 (0.9%)	1 (0.2%)	536
Geronimo_Other	0 (0.0%)	372 (77.0%)	104 (21.5%)	4 (0.8%)	3 (0.6%)	483
Wildfly_LCII	13 (0.5%)	2,307 (81.6%)	318 (11.3%)	172 (6.1%)	16 (0.6%)	2,826
Wildfly_Other	0 (0.0%)	393 (68.3%)	125 (21.7%)	47 (8.2%)	10 (1.7%)	575
Total_LCII	42 (0.5%)	6,374 (72.9%)	1,912 (21.9%)	398 (4.5%)	22 (0.3%)	8,748
Total_Other	0 (0.0%)	4,919 (66.8%)	2,052 (27.9%)	356 (4.8%)	35 (0.5%)	7,362

counts of co-changed logging code changes which change that component. The column on the right shows the counts of independently changed logging code changes which change that component. For example, there are 58 "1 type"co-changed logging code changes which changes static texts and 1661 for independently changed logging code changes. Among all components, changes related to verbosity levels are the most, accounting for 36.8%. None of the co-changed logging code changes are related to logging library and verbosity level, which is a big difference from indpendently changed logging code changes. The detail results for the "2 types" changes are shown in Table 3.11. Changes related to static texts and verbosity levels are the two most common changes. 84.5% of the changes modify texts and 68.1% of the changes modify verbosity levels. The independently changed logging code and co-changed logging code changes have the similar distribution.

Summary

Findings: the majority of the fixes to the LCII changes are changing the verbosity level, while the majority of the other logging code changes are changing dynamic contents. The two types of logging code changes (LCII changes and co-evolving logging code changes) are similar in terms of change complexity. The majority of both types of logging code changes only have one or two component changes (a.k.a., "1 type" and "2 types" logging code changes). Among the "1 type" logging code changes, the most common changes are about verbosity level changes, followed by the logging library and static text changes tied at the second place. Among the "2 types" logging code changes, the top two most common changes are changes in the verbosity level and static texts.
Implications: correcting logging code issues just needs to change one or two components in most of the cases with particular focuses on the verbosity levels and the static texts. Although there are existing automated approaches (e.g., [36, 35]) to helping developers determine the appropriate verbosity levels for each logging code snippet, there are no techniques focusing on detecting and improving the static texts component.

3.6.3 RQ3: How long does it take to fix an LCII change?

In this RQ, we will measure how long it takes to resolve a LCII change. We are going to compare the resolution time between LCII changes and reported bugs.

Data Extraction

We first extracted the bug creation date and the bug resolution date from the bug reports in order to compute the bug resolution time. To compute the resolution time for each LCII change, we computed the time differences between the LCII changes and their fixes.

Table 3.10: Component changes for the "1 type" changes

Project	Lib		Level		Text		Dynamic		Total
	Co	Ind	Co	Ind	Co	Ind	Co	Ind	
Elasticsearch	0	30	0	103	2	441	8	84	668
Hadoop	0	120	0	261	42	273	23	80	799
HBase	0	69	0	312	10	601	50	175	1,217
Hibernate	0	530	0	297	1	70	4	50	952
Geronimo	0	71	0	231	2	97	5	25	431
Wildfly	0	899	0	1,144	1	179	43	41	2,307
Total	0	1719(27.0%)	0	2,348(38.0%)	58(0.9%)	1,661(26.1%)	133(2.1%)	454(7.1%)	6,374

Table 3.11: Component changes for the "2 types" changes

Project	Lib		Level		Static		Dynamic		Total
	Co	Ind	Co	Ind	Co	Ind	Co	Ind	
Elasticsearch	0	5	0	29	9	151	9	135	169
Hadoop	0	55	18	65	38	109	28	97	205
HBase	0	51	14	65	44	203	30	189	298
Hibernate	0	33	55	748	54	772	1	13	838
Geronimo	0	49	0	54	0	36	0	29	84
Wildfly	11	124	2	251	14	185	15	34	318
Total	11(0.6%)	317(16.6%)	89(4.7%)	1,212(63.4%)	159(8.3%)	1,456(76.2%)	83(4.3%)	497(26.0%)	1,912

Project	# of LCII fixes	# of bug reports
Elasticsearch	838	N/A
Hadoop	1,030	31,420
HBase	1,533	15,486
Hibernate	1,985	9,009
Geronimo	536	5,594
Wildfly	2,826	14,469

Table 3.12: The number of fixes to the LCII changes and the number of bug reports in each project.

Data Analysis

Table 3.12 shows the numbers of fixes to LCII changes and regular bug reports for each studied project. There are much more reported bugs than fixes to LCII changes. For example, in Hadoop, there are $1,030$ fixes to LCII changes while there are $31,420$ bug reports (almost 30 times more). Elasticsearch does not have an issue tracking system, so we only show the number of fixes to LCII changes.

Table 3.13 shows the median resolution time for both LCII and regular bugs in each project. For example, in Hadoop, the median resolution time for LCII changes is 210.9 days and it is 15.5 days for regular bugs. For all five studied projects, the median resolution time of the LCII changes is longer than that of regular bugs. In particular, the resolution time can be as long as 379 days for the fixes to the LCII changes (Wildfly) while the longest median resolution time for regular bugs is only 30.9 days (Hibernate).

Figure 3.12 visually compares the distribution of the resolution time for the fixes to the LCII changes, and the duration for fixing regular bugs. Each plot is a beanplot [146]. The left part of the plot shows the distributions of the durations for fixing the LCII changes, while the right part of the plot shows the distribution of the bug resolution time. The vertical scale is shown under the unit of the natural logarithm of days. Among the five studied projects, the left part of the plots are always higher (a.k.a., containing higher extreme points) than the right part.

To statistically compare the resolution time for LCII changes and regular bugs across all five projects, we performed the non-parametric Wilcoxon rank-sum test (WRS). Table 3.13 shows our results. The two-sided WRS test shows that the resolution time of LCII changes is significantly different from that of regular bugs ($p < 0.001$) in all five projects. To assess the magnitude of the differences between the resolution time for LCIIs and regular bugs, we have also calculated the effect sizes using Cliff's Delta for all five projects in Table 3.13. The strength of the effects and the corresponding range of Cliff's Delta (d) values [147] are defined as follows:

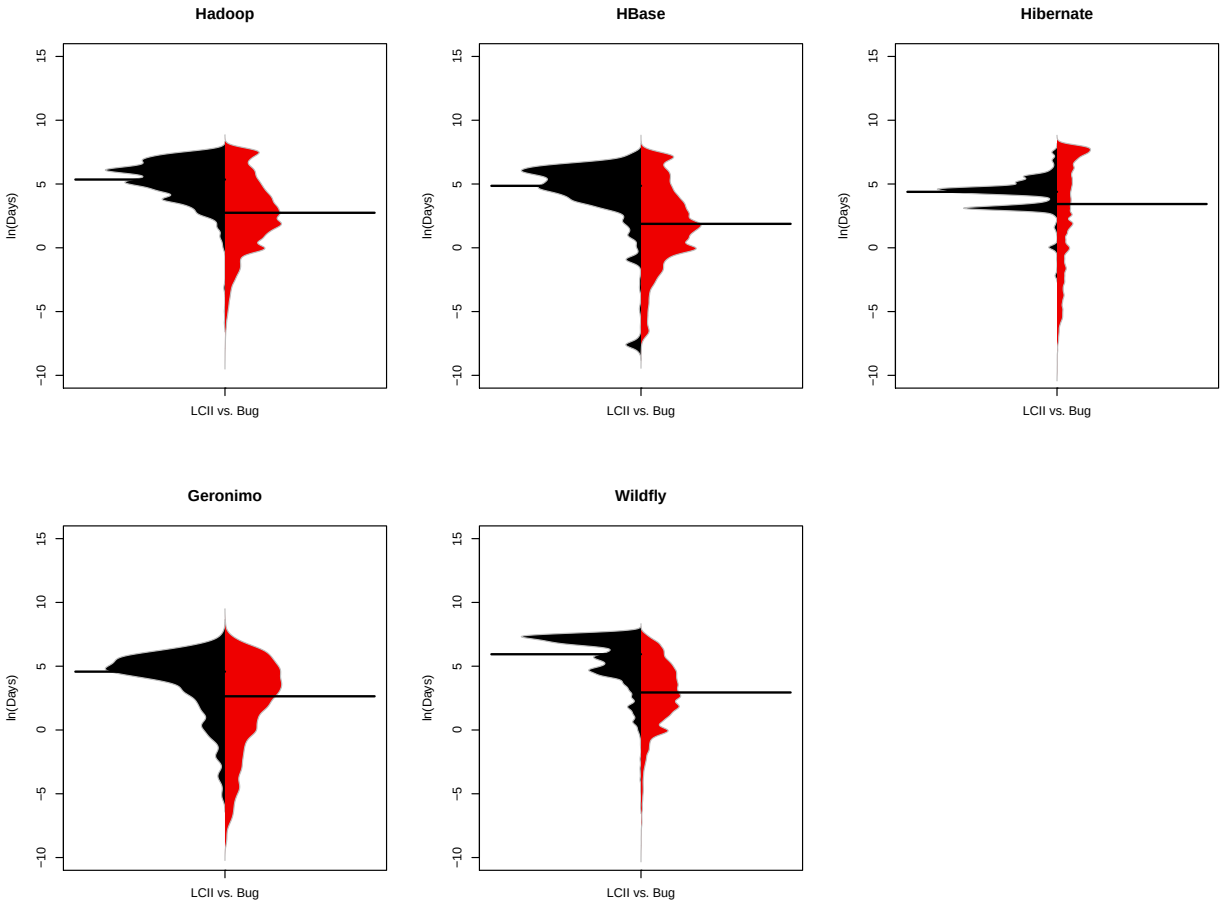

Figure 3.12: Comparing the resolution time of LCII and regular bugs.

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

Our results show that the effect sizes for three out of five projects (Hadoop, HBase, and Wildfly) are large, whereas the other projects have negligible (Hibernate) or small (Geronimo) effect sizes. However, the actual rationales behind the longer resolution time for logging code issues are not clear, as there are both high priority and low priority issues in regular bugs and logging code issues. In the future, we plan to investigate this further by surveying the developers or the domain experts of these projects.

We further compared the resolution time between the LCII changes from co-changed logging code changes and the independently changed logging code changes using the similar method as described above. The results are shown in Table 3.14. Each row corresponds

Table 3.13: Comparing the resolution time between the LCII changes and regular bugs. The unit of resolution time for both the LCII changes and regular bugs are in days.

Project	LCII	Bugs	p-values for WRS	Cliff's Delta(d)
Hadoop	210.9	15.5	<0.001	-0.54 (large)
HBase	128.6	6.5	<0.001	-0.5 (large)
Hibernate	80.4	30.9	<0.001	-0.13 (negligible)
Geronimo	96.9	14.0	<0.001	-0.33 (small)
Wildfly	379.2	18.9	<0.001	-0.63 (large)

Table 3.14: Comparing the resolution time between the co-changed LCII changes and independently changed LCII changes. The unit of resolution time for both the co-changed and independently changed changes are in days.

Project	Co-changed	Independently	p-values for WRS	Cliff's Delta(d)
Elasticserach	103.9	84.1	0.04	-0.3 (small)
Hadoop	210.9	216.8	0.73	0.02 (negligible)
HBase	183.5	126.1	0.14	-0.08 (negligible)
Hibernate	163.2	80.3	<0.001	-0.72 (large)
Geronimo	21.3	96.9	0.27	0.22(small)
Wildfly	219.3	397.3	<0.001	0.20 (small)

to one projet. For example, in Hadoop, the median resolution time of fixes by co-changed LCII changes is 210.9 days and it is 216.8 days for independently changed LCII changes. For Elasticsearch, HBase and Hibernate, the median resolution time of the co-changed LCII changes is longer. The differences between the two types of LCII changes are only statistically significant in Hibernate and Wildfly ($p < 0.001$). The magnitude of differences is either small or negligible for five out of six projects.

We also studied the distribution of the resolution time with respect to the complexity of the LCII changes. The results are visualized in Figure 3.13. Each plot contains 2 to 5 violin plots for each project. Some plots are missing because there are not enough instances. Take Elasticsearch for example, there are no enough instances for "0 type", "3 types" and "4 types" changes. Hence, there are no corresponding plots for this project. The vertical scale for all the plots is in the unit of the natural logarithm of days. As we can see across the violin plots, the distribution differs from project to project. It demonstrates that the complexity of the fixes to the LCII changes do not necessarily impact the resolution time.

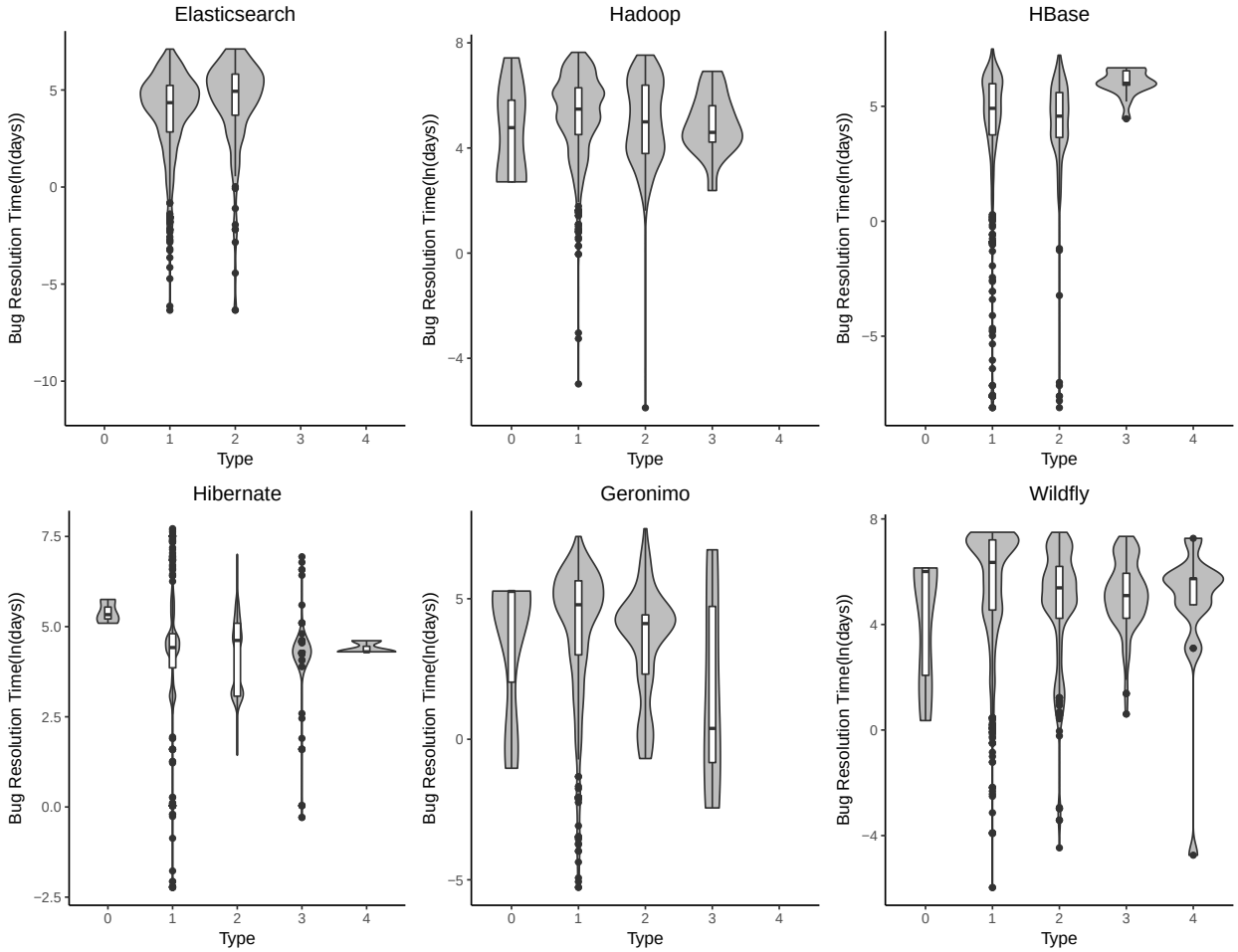

Figure 3.13: Comparing the resolution time of different types of LCII changes.

Summary

Findings: the resolution time of the LCII changes and regular bugs are statistically different in all studied projects. The median resolution time of the LCII changes is much longer than that of regular bugs and the magnitude of differences is large in three out of the five studied projects. Within the LCII changes, there is no clear statistical difference between the types of changes (co-changed vs. independently changed) or the complexity (1 type vs. 2 types vs. 3 types vs. 4 types) of the changes.
Implications: although software logging is used widely in many contexts (e.g., debugging, runtime monitoring, business decision making, and security), the correctness of the logging code cannot be easily verified using conventional software verification techniques. The issues in the logging code changes can take much longer to be detected and fixed than regular bugs, as most of the existing issues in the logging code can only be detected and fixed manually. Simple logging code changes may take as much time as complex logging code changes. Automated approaches to validating and improving the quality of the logging code is becoming an increasingly important research area.

3.6.4 RQ4: Are state-of-the-art code detectors able to detect logging code with issues?

In this RQ, we want to study the effectiveness of the state-of-the-art techniques on detecting issues in the logging code. There are two techniques in the existing research literature attempting to flag issues in the logging code:

- **Rule-based Static Analysis (LCAnalyzer)**: this technique is proposed by Chen et al [36]. It scans through source code searching for six anti-pattern instances using a set of rules.
- **Code Clones (Cloning)**: this technique is proposed by Yuan et al [10]. It uses a code clone detection tool to find all the clone groups and checks if the logging code in the clone groups have the same verbosity levels. If the levels are inconsistent, at least one of them have incorrect verbosity levels.

Data Extraction

Our extracted dataset contains all the LCII changes and their fixes throughout the entire development history. However, both techniques listed above need to be applied on the entire source code from one release of each project, since LCAnalyzer needs to extract code dependency information and Cloning needs to scan all the source code to find code clones. Therefore, we selected bi-monthly snapshots of the studied projects and ran both techniques on them.

We extracted bi-monthly snapshots for all the studied projects. For each snapshot, we computed the number of existing logging code issues using our dataset. For each logging code issue, we kept track of the issue introduction and fixed time/version, so that we can track the number of logging code issues in each snapshot by checking its timestamp against various commits: a logging code issue exists in a snapshot if it is introduced before the release time of the snapshot and fixed after the release time of the snapshot.

To avoid repeated counts, we counted each unique logging code issue once. For example, if a piece of logging code with issue exists in two snapshots, and is detected by the above techniques, we only count it once. Across all the snapshots, we computed the total number of unique logging code issues and aggregated the total number of detected ones from the two approaches.

In order to further characterize the capability of these two detecting techniques, we classified the logging code with issues into four categories: (1) issues in the logging library, (2) issues in the verbosity level, (3) issues in the static text, and (4) issues in the dynamic contents. Note that Cloning can only detect issues in the verbosity level. We obtained the LCAnalyzer from [148] and implemented the Cloning technique by ourselves.

Data Analysis

The detected results of LCAnalyzer and Cloning are shown in Figure 3.14. We calculated the recall of the detection results as $\frac{Number\ of\ unique\ LCII\ detected}{Number\ of\ all\ unique\ LCII} \times 100\%$. In total, only 2.1% of the logging code issues have been detected by LCAnalyzer, and only 0.1% of them have been detected by Cloning technique. We then split the issues based on its problematic components. For example, 0.3% of verbosity level related issues are detected by Cloning technique while 1.5% of them are detected by LCAnalyzer.

In general, LCAnalyzer performs the best on issues related to dynamic contents, while it is only able to detect 5.1% of all issues in the total 60 snapshots. The worst performance of LCAnalyzer is 1.0% recall, when the issues are related to the logging library. On the other hand, Cloning technique can only detect verbosity level related issues due to its design, yet it is still outperformed by LCAnalyzer (0.3% vs. 1.5%).

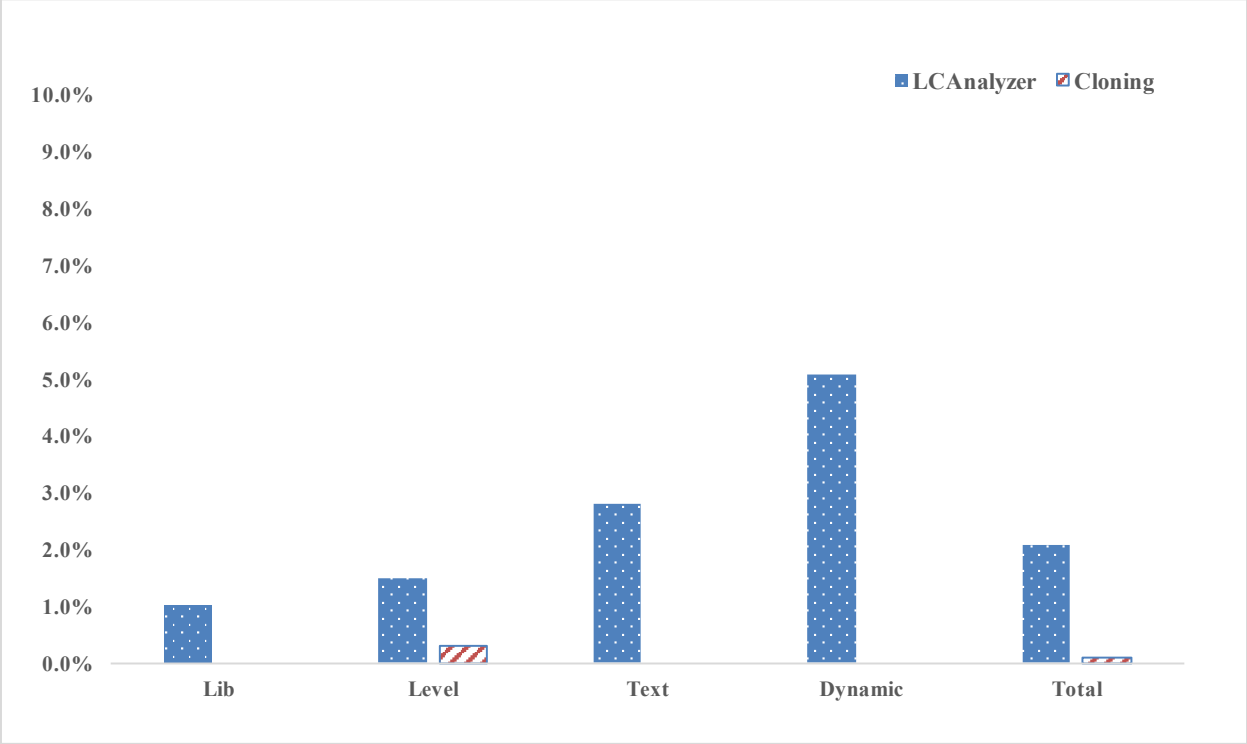

Figure 3.14: Comparing the recall of the detection results for the two studies techniques.

When we compared the correctly detected instances from the two studied tools, we found that the two techniques complement each other. We noticed that all the logging code with issues found by the Cloning techniques are not detected by the LCAnalyzer, and vice versa. To demonstrate the differences of these two approaches, we show two real world examples in Figure 3.15. LCAnalyzer can only detect the issue in the top, whereas Cloning can only detect the issue in the bottom. The reason that LCAnalyzer can detect the logging code issue in the top is because the verbosity level and the static content are inconsistent. In the

static text, it shows that the logging code is for debugging purpose while the verbosity level is `info`. In the fix version, the verbosity level is changed to `debug` and the text `DEBUG` is deleted. For the issue in the bottom, two pieces of logging code snippets (highlighted in red) are considered as clones. The verbosity level of one logging code is `debug` whereas the other one is `info`. Therefore, the `info` level should be changed to `debug`. However, even combining the power of these two techniques, most of the issues in logging code are still undetected.

Tool	Code Example
LCAnalzyer	V 1255: `LOG.info("DEBUG --- getStagingAreaDir: dir=" + path);` V 1256: `LOG.debug("getStagingAreaDir: dir=" + path);` ResourceMgrDelegate.java in *Hadoop*

Code Clone	`if (currentSplitSize + srcFileStatus.getLen() > nBytesPerSplit`
	`  && lastPosition != 0) {`
	`…`
	`if (LOG.isDebugEnabled())`
	`  LOG.debug("Creating split : " + split + ", bytes in split: " +`
	`  currentSplitSize);`
	`…`
	`if (LOG.isDebugEnabled())`
	`  LOG.info("Creating split : " + split + ", bytes in split: " +`
	`  currentSplitSize);`
	UniformSizeInputFormat.java in *Hadoop*

Figure 3.15: Examples of the detected issues in the logging code from the two studied techniques.

Summary

Findings: both the LCAnalyzer and the Cloning technique can only detect a small fraction ($< 3\%$) of the issues in logging code. The results outputted by the two techniques complement each other.
Implications: there are still many logging code issues, which cannot be detected by existing automated techniques. Leveraging our provided dataset, researchers can develop and benchmark their new techniques on automatically detecting issues in the logging code and deriving effective logging guidelines. As shown in RQ2, the majority of the LCII changes are related to verbosity level changes and static texts, researchers are recommended to prioritize their effort on detecting and improving the issues in these two categories.

3.7 Related Work

In this section, we will discuss three areas of related research: (1) empirical studies on software logging, (2) research on automated suggestions on the logging code, and (3) research on identifying the bug introducing changes.

3.7.1 Empirical Studies on Software Logging

Empirical studies show that there are no well-established logging practices used in industry [8, 32] as well as open source projects [9, 10]. Although most of the logging code has been actively maintained, some of the existing logging code can be confusing and difficult to understand [6]. Various studies have been conducted to study the relationship between software logging and code quality. Shang et al. [131] found that the amount of logging code is correlated with the amount of post-release bugs. Kabinna et al. [80] studied the migration of logging libraries and its rationales. Over 70% of the migrated systems have been found buggy afterwards. In [2], the authors analyzed the logging code changes from three open source systems (Hadoop, HBase, and ZooKeeper). They found that the majority of the logging code changes are due to adding logging statements. Many modifications to the exsting logging code are related to the verbosity level changes.

In this chapter, we studied the historical logging code changes from six open source systems. We have performed three empirical studies on the commits with a focus on the issues in the logging code and their fixes. First, we compared the change characteristics between the fixes to the logging code and regular logging code changes. Second, we studied the resolution time for fixing logging code issues and compared the duration against the resolution time for bugs. Finally, we assessed the effectiveness of the state-of-the-art techniques on detecting issues in the logging code. All the above three research questions, which had never been studied before, leveraged our extracted dataset presented in this chapter.

3.7.2 Research on Automated Suggestions on the Logging Code

Orthogonal with the feature code, the logging code is considered as a cross-cutting concern. Adding and maintaining high quality logging code is very challenging and usually requires a large amount of manual effort. Various research has been conducted in the area of automated suggestions of the logging code:

- **where-to-log** focuses on the problem of providing automated suggestions on adding logging code. Yuan et al. [18] leveraged program-analysis techniques to suggest code locations to add logging code for debugging purposes. Zhu et al. [107] learned common logging patterns from existing code and made automated suggestions based on the resulting learned models. Zhao et al. [2] introduced Log20, which can automate the placement of logging code under certain overhead threshold.

- **what-to-log** is related to the problem of adding sufficient information into the logging code. Yuan et al. [35] proposed a program analysis-based approach to suggesting adding

additional variables into the existing logging code to improve the diagnosability of software systems. Li et al. [111] learned from the development history to automatically suggest the most appropriate level for each newly-added logging code.

- **how-to-log** is about maintaining high quality logging code. Li et al. [48] learned from the code level changes to predict whether the logging code is a particular code commit requiring updates (a.k.a., just-in-time logging code changes). However, their technique did not pin-point the exact logging code snippets to be updated within each code commit. Yuan et al. [10] leveraged code cloning techniques to detect inconsistent verbosity levels among similar feature code segments. Chen et al. [36] inferred six anti-patterns in the logging code by manually sampling a few hundred logging code changes.

This chapter fits in the area of "how-to-log" and is close to [36]. However, the technique to extract issues in the logging code and their fixes have been improved in this chapter so that the resulting dataset is more complete and more accurate. Furthermore, instead of only focusing on a small set of sampled instances, this chapter provided a dataset containing all the historical issues in the logging code for six popular open source projects. In addition, we have also conducted three empirical studies on the resulting dataset to demonstrate the usefulness of this dataset and presented some open research problems in the area of "how-to-log".

3.7.3 Research on Identifying the Bug Introducing Changes

It is important to identify bug introducing changes so that developers and researchers can learn from them and develop tools to detect and fix them automatically. There are generally two steps in identifying bug introducing changes:

- **Step 1 - Identifying bug fixing commits**: keyword heuristics based approaches have been used to identify bug fixing commits by searching through relevant keywords (e.g., "Fixed 42233" or "Bug 23444", etc.) in the code commit logs [132, 149].
- **Step 2 - Identifying bug introducing commits based on their fixes**: Śliwerski et al. [129] were the first to develop an automatic algorithm to identify bug introducing changes based on their fixes. Their algorithm, called the SZZ algorithm, initially started at the code commits which fixes these issues, then tracks backward to the previous commits which touched those source code lines. Afterwards, there are various modifications to the SZZ algorithms [128, 150, 151] to improve its precision and to evaluate its effectiveness [137].

This chapter is different from the above, as it aims to extract and study the LCII changes instead of the software bugs. Software bugs are usually reported to the bug tracking systems and their fixes are logged in the code commit messages. However, issues in the logging code are usually undocumented. Thus, in this chapter, we have proposed an approach to first identifying fixes to the LCII changes and then automatically identifying the LCII changes

based on their fixes. We have analyzed both of the co-changed and independently changed logging code changes to flag fixes to the LCII changes. Since there are multiple components (e.g., logging library, verbosity level, static texts, and dynamic contents) in each logging code snippets related to more than one line of feature code, there can be multiple issues from different code commits in one line of the logging code. Hence, we have developed an adapted SZZ algorithm (LCC-SZZ) to identify issues in the logging code changes.

3.8 Threats to Validity

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

3.8.1 Internal Validity

There are two general types of logging code changes: independently and co-changed logging code changes. We have identified the fixes to the LCII changes by carefully analyzing the contents of both types of changes. For analysis of co-changed logging code changes, we manually examined 533 instances. We inspected the source code files which contain those co-changed logging code changes, and examined the context to verify whether these are fixes to LCII changes. The resulting outputs were compared and discussed to generate a reconciled dataset in the end. The resulting LCII changes identified using the LCC-SZZ algorithm have also been verified to be highly accurate (93% accuracy).

3.8.2 External Validity

In this chapter, we extracted the LCII changes from six Java-based systems. We do feel our approach is generic and can be applied to identify LCII changes for systems implemented in other programming languages (e.g., C, C#, and Python).

We have conducted three empirical studies on the resulting dataset extracted from the development history of six Java-based projects (Elasticsearch, Hadoop, HBase, Hibernate, Geronimo, and Wildfly), which come from different domains (big data platform, web server, database, middleware, etc.). All of these projects have relatively long development history and are actively maintained. However, our findings in the research questions may not be generalizable to other programming languages.

3.8.3 Construct Validity

We have used ChangeDistiller [130]to extract the fine-grained code changes across different code versions. ChangeDistiller has been used in many of the previous work [9, 36] and is proven to be highly accurate and very robust.

We have demonstrated that LCC-SZZ is more effective than SZZ in terms of identifying LCII changes across the following three dimensions: (1) the disagreement ratio between the extraction results from the two algorithms; (2) comparing the earliest appearance of the

LCII changes against the bug reporting time; and (3) manual verification. Our evaluation approaches are similar to many of the existing studies in this area (e.g. [137, 152, 153, 154]).

3.9 Conclusions and Future Work

Software logging has been used widely in large-scale software systems for a variety of purposes. It is hard to develop and maintain high quality logging code, as it is very challenging to verify the correctness of a particular logging code snippet. To aid effective maintenance of logging code, in this chapter we have extracted and studied the historical issues in logging code and their fixes from six popular Java-based open source projects. To demonstrate the effectiveness of our dataset, we have conducted four preliminary case studies. We have found that both the co-changed and the independently changed logging code changes can contain fixes to the LCII changes. The change complexity metrics between the fixes to the LCII changes and other logging code changes are similar. It usually takes much longer to address an LCII change than a regular bug. Existing state-of-the-art techniques on detecting logging code issues cannot detect a majority of the issues in logging code. In the future, we plan to further leverage our derived dataset to: (1) develop better techniques to automatically detect issues in logging code, and (2) derive best practices in terms of developing and maintaining high quality logging code.

Chapter 4

Studying the Use of Java Logging Utilities in the Wild

4.1 Introduction

Execution logs (a.k.a., logs) have been used widely in practice for a variety of purposes (e.g., system monitoring [3, 16], failure diagnosis [17, 18], and business analytics [5, 28]). Logs are generated during runtime by the output statements that developers insert into the source code. Instead of directly invoking the standard output functions like `System.out.print`, developers prefer to instrument their systems using logging utilities (LUs) (e.g., `SLF4J` [46] for Java and `spdlog` [47] for C++) due to additional functionalities like thread-safety (synchronized logging in multi-threaded systems), data archival configuration (automated rotation of the log files), and verbosity levels (controlling the amount of logs outputted).

Unlike many of the software engineering tasks (e.g., code refactoring [126] and release management [127]), there are no well-defined guidelines for software logging. Recently, there have been many research work devoted to the area of where-to-log (deciding the appropriate logging points) [18, 8, 21, 107, 2], what-to-log (providing sufficient execution context in the logging code) [35, 48, 104], and how-to-log (developing and maintaining high quality logging code) [10, 111, 36, 38, 123]. However, all of these work focus on improving the quality of log printing code (e.g., `LOG.info("User " + username + " authenticated")`). There are only two research work on the migration [80] and the configuration [81] of the LUs. It is important to study the use of LUs due to these three reasons:

Measuring the Adoptions of the LUs: Although there are already many LUs available (e.g., [77, 46, 63, 78]), new LUs are continuously introduced by companies (e.g., Flogger from Google [155]) and researchers (e.g., NanoLog [72] and Log++ [156]). It is not clear whether these LUs are adopted and used in the wild.

Understanding the Complex Use of the LUs: Incorporating multiple LUs in one project may cause various issues during compilation [157, 158], deployment [159, 160], and runtime [161, 162]). However, many software systems still use more than one LUs in their projects [163]. For example, `Hadoop`, which is a very well-maintained popular open

source Big Data platform, contains not only six external LUs (`Apache Commons Logging`, `java.util.logging`, `Log4j 1.x`, `Log4j 2`, `SLF4J`, `Jetty logging`) and implements their own LU in their project. `IntelliJ Idea`, which is a very popular IDE, uses 12 LUs in their project. It is important to study this phenomena in order to suggest best practices for the system developers and to identify future directions for the LU designers and researchers.
Assessing the Impact of LUs on the Logging Code: On one hand, the structure of logging code is a result of adopting certain LUs. For example, it is generally recommended to specify the logging needs as rules via Aspect Oriented-Programming (AOP) constructs to improve system modularity [44] and use centralized logging [34] to support internationalization (a.k.a., outputting the log messages in different human lanaguages). On the other hand, extra care are needed in order to cope with the complex use of LUs in one project. For example, many projects nowadays reuse existing functionalities by importing third-party packages, which also use LUs. However, there is no empirical study to assess the impact of the logging code due to the use of the LUs.

In this chapter, we have performed a large-scale empirical study on the use of Java LUs in the wild. We focus on Java, because it is currently ranked as the most popular programming language in the world based on the TIOBE index [164]. Many popular software systems (e.g., Android-based mobile applications [165, 166], IDEs [167, 168], web servers [169, 170], and Big Data platforms [171, 172]) are implemented in Java and logging is prelevant in these systems [9, 83]. We have examined 11,194 open source Java-based projects in GitHub, which uses 3,856 LUs. We have uncovered four important findings and implications. To ensure the usefulness and generalizability of our findings from open source systems, we also interviewed 13 industrial developers on the use of LUs. We summarize our main contributions and findings as follows:

- This is the first empirical study on the complex use of LUs in the wild.
- 3,856 LUs are currently being used by 11,194 projects in GitHub. The number of used LUs increases as the systems grow bigger, as larger-sized projects usually reuse existing functionalities by importing third-party packages, which also use LUs. Many projects also implement their own LUs to satisfy project-specific needs. Our findings raise developers' awareness on the complexity of the LUs in their systems as well as the need to manage these LUs to better monitor and debug systems' runtime behavior.
- In addition to the quantitative study, we have also conducted a qualitative study to understand and characterize the rationales behind the complex use of LUs for different projects. Such results can guide researchers to propose more general logging solutions for various logging context.
- To support independent verification or further research on the use of Java LUs, we have provided a replication package [13] in this chapter. This package can be useful for other researchers who are interested in studying and improving the logging practices.

Chapter Organization. The rest of this chapter is organized as follows. Section 4.2 provides an overview of our approach and describes our studied projects. Section 4.3 and 4.4 quantitatively and qualitatively study the use of the LUs in the wild. Section 4.5 explains our interview process and describes our observations. Section 4.6 discusses the significance of our findings and presents some future work. Section 4.7 describes the related work. Section 4.8 explains the threats to validity and Section 4.9 concludes the chapter.

4.2 Overview

In this section, we will provide an overview of our approach to empirically studying the use of Java LUs in the wild and describe our studied projects.

4.2.1 Our Approach

We followed a mixed-methods approach characterized by a sequential explanatory strategy [173] to analyze the use of LUs. We first extracted the source code from popular Java-based GitHub projects (Section 4.2). Then we performed a quantitative study on measuring the degree of adoption of different LUs as well as comparing the number of used LUs across different projects (Section 4.3). We also tracked the number of projects which also implement their own LUs. Afterwards, we performed a qualitative study (Section 4.4) by manually studying the use of LUs among different projects. Since our study is performed on open source projects, we also cross-validated our findings by interviewing 13 experienced developers who work on commerical systems (Section 4.5).

4.2.2 Studied Projects

To study the use of Java LUs in the wild, we focused on analyzing Java-based projects from GitHub. GitHub is currently the largest code hosting site with more than 100 million projects as of April 2019 [174]. We built a local GHTorrent [175] database from the MySQL dump. This database, which was last updated on 2019-06-01, contains the meta information of a project such as the corresponding GitHub URL, the project name, and the main programming language(s). We extracted a list of GitHub URLs for all the Java-based projects for post-processing.

We further filtered the list of GitHub projects to avoid potential perils in our analysis [176]. One GitHub project may be forked or cloned by others, whose source code can be identical or very similar to the orignal one. This would introduce noise into our study. Hence, we only selected projects which are neither a fork nor a clone of other projects. Furthermore, the number of stars that a project has indicates its popularity. Similar to prior work [177, 178, 179], we further filterd the list of projects to ensure they have at least 30 or more stars. We ended up with $25,611$ Java projects. We downloaded the soure code for the most recent releases of these projects by invoking the GitHub APIs.

4.3 Quantitative Study

In this section, we first analyzed the selected projects to identify the list of LUs that are used. Then we measured the degree of LUs which are adopted and used. Finally, we compared the use of LUs among different projects.

4.3.1 Identifying LUs in Each Project

We developed a heuristic-based technique to identify LUs in each project. First, we excluded Java files, which were either in a `test` folder or containing the keyword of "test" in their file names, as they are probably not related to the core features of the projects under study. Then we used JDT [180] to parse the remaining Java files to identify a list of imported statements and function invocations for each Java class. We made sure these imported classes are used by checking if there is one or more methods been invoked in that class. We further filtered the list of import statements in each Java class, so that only packages or Java classes whose names contain patterns like "logging", "logger" or ".log." were kept. These packages or classes are considered as LUs. For logging in Aspect-Oriented Programming (AOP), we first identified Java files with import statements that contain "aspectj". Since `AspectJ` can do more than logging, we subsequently parsed the Java files to check whether they use any of the LUs identified by the above rules.

If a project used more than one LUs and their package names were similar, we merged them as one LU. For example, if the following two LUs, `Foo.Bar.Baz.ConsoleLogger` and `Foo.Bar.Baz.FileLogger`, were identified in one project, we merged them into one LU (`Foo.Bar.Baz`).

To verify the correctness of our heuristic-based technique, we randomly sampled 441 files and manually examined the identified LUs. Our technique yield an precision of 96%. Some of the Java classes are misclassified as LUs as the identified log-related Java classes did not implement log printing functions. For example, `LoggerProvider.java` in the `Ninja` framework [181] was not considered as an LU, since this class did not provide any log printing functions other than a utility function that returns an `SLF4J` logging object.

4.3.2 Measuring the Adoptions of LUs

We further classified the size of these projects into the following five bins based on their number of Java classes using the definitions from [182]. Table 4.1 shows the results. For example, there are a total of 761 *large*-sized projects, whose number of Java classes is between $1,000$ and $5,000$. Among them, 728 projects (a.k.a., 95.7%) adopt one or more LUs. There is a clear trend of increasing adoptions of LUs as the size of the projects get bigger.

For the subsequent studies, we removed all the projects which did not use any LUs, since they are not relevant to this chapter. We also excluded the projects whose size was *very small*, as many of them are not considered as useful projects (e.g., trial projects for self-studying, and collections of code snippets). After filtering, there were $11,194$ Java projects remaining,

which used 3,856 distinct LUs.

Table 4.1: Measuring the adoptions of LUs among the Java-based GitHub projects.

Bins (# of classes)		**Total # of Projects**	**% of Projects Used LUs**
Very Small	$[0, 20)$	11,390	43.2%
Small	$[20, 100)$	7,781	72.5%
Medium	$[100, 1000)$	5,576	84.3%
Large	$[1000, 5000)$	761	95.7%
Very Large	$[5000, \infty)$	103	97.1%
Total		25,611	63.1%

4.3.3 Comparing the Use of LUs Among Projects

Depending on where these LUs were implemented, we further classified the list of used LUs in each project as *External LU (ELU)*s or *Internal LU (ILU)*s. It is an ELU, if the implementation of this LU is not inside the project under study. Otherwise, it is an ILU. For example, `Hadoop` uses five different ELUs: `Apache Commons Logging`, `java.util.logging`, `Log4j 1.x`, `Log4j 2`, `SLF4J`, `Jetty logging` and implements its own ILU, which is mainly used for auditing purposes. Table 4.2 shows the results of the percentage of projects that use ELUs, ILUs, or both. For example, among all the 728 Large-sized projects that used LUs, 95.3% of them are ELUs, 75.8% of them implemented their own ILUs in their projects, and 71.2% used both types of LUs. Overall, there are 866 ELUs used by 11,194 projects. 26.7% of the studied projects have implemented their own ILUs.

The percentage of projects that adopted ELUs and ILUs shown in the second and the third columns increases as the size of projects increases. When comparing the LUs within the same bins, there are always more projects using ELUs than ILUs. However, almost all of the *very large*-sized projects implemented their own LUs. The combined use of ILUs and ELUs also increase dramatically as project sizes become *large* or *very large*.

We further examined the number of LUs that were used in each project. For each project, we counted the number of LUs, which included both ELUs and ILUs. We grouped projects based on their sizes and visualized their distributions using boxplots in Figure 4.1. The width of the boxplot is proportional to the number of projects in that group. Since there are much more *small*-sized projects in our study, it is the widest. The red dashed line connects the median points for each bin. The median usage for the *small*-sized projects is one. It increases as the size of the project increases. For the *very large*-sized projects, the median number of LUs used is four. The project that adopts the biggest number of LUs is within the group of *very large*-sized projects. It is an enterprise web platform ([183]), in which 21 LUs are used!

We conducted a Kruskal-Wallis test [184] to statistically check if their distributions are identical. The *p-value* was smaller than 0.05, which indicated statistically signficant differences

Table 4.2: Comparing the complexity of the uses of LUs among projects.

Project Size	**% of Projects Using** (*ELU*	*ILU*	*Both*)
Small	(97.2%	12.7%	9.9%)
Medium	(95.3%	34.6%	29.9%)
Large	(95.3%	75.8%	71.2%)
Very Large	(97.0%	92.0%	89.0%)
Total	(96.3%	26.7%	23.0%)

among the distributions of LUs across different project sizes. This clearly demonstrates the complexity of the use of LUs in our studied projects.

The findings in our quantitative studies motivated us to conduct the subsequent qualitative studies (Section 4.4) to figure out the rationales behind them.

Findings: (1) There are 3,856 LUs, of which 866 are ELUs, being used by over 11,000 projects. (2) 96.3% of the studied projects adopt at least one ELU and 26.7% adopt at least one ILU. (3) As the project size becomes larger, more LUs will be integrated into the project. This is especially the case for *large* and *very large*-sized projects.
Implications: The LUs are used widely among different Java projects. However, as the size of the project increases, the complexitity of the use of LUs also increases. Multiple LUs are used in many of the *medium* to *very large*-sized projects. This is contradictive to the common understanding of developers [185, 186, 163, 187] and researchers [80], as only a handful of popular Java LUs (e.g., `Log4j 1.x`, `Log4j 2`, and `SLF4J`) are compared and discussed.

4.4 Qualitative Study

In this section, we conducted a qualitative study on the use of the LUs. We manually examined their source code in order to answer the following RQs:

- **RQ1: What are the external LUs being used in the wild?** As shown in Section 4.3, there were 866 ELUs being used by 11,194 projects. The goal of this RQ is to characterize the rationales of different ELUs.
- **RQ2: Why do developers implement ILUs in their projects?** 26.7% of the studied projects have implemented their own internal LUs. This is especially the case for the *large* or *very large*-sized projects, in which 75.8% and 92.0% of them contain ILUs. The goal of this RQ is to extract the project-specific logging needs by studying the use of ILUs.
- **RQ3: How are multiple LUs used in the wild?** Many of the studied projects, especially for the *large* or *very large*-sized projects, use multiple LUs. The goal of this

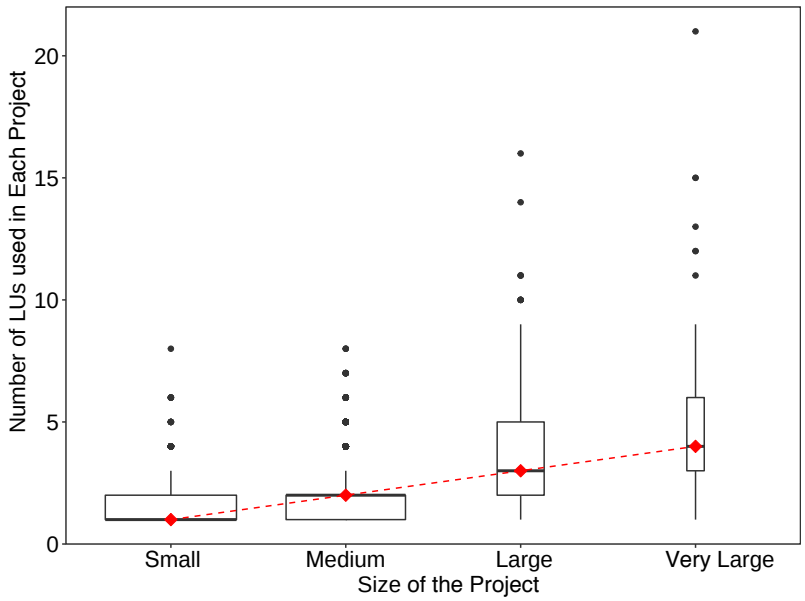

Figure 4.1: The distritbutions of the number of adopted LUs in each project grouped by the size of the projects.

RQ is to characterize the usage context associated with using multiple LUs in one project.

4.4.1 RQ1: What are the external LUs being used in the wild?

In this RQ, we will characterize the rationales behind the use of individual ELUs. For each ELU, we calculated the number of projects that adopted it. We sorted ELUs by their popularities (a.k.a., the number of projects that used them) and selected the top-100 most popular ELUs, which are used by 95% of the studied projects.

For each of these 100 ELUs, we manually examined the online documentation, release notes, and blog posts for these ELUs to understand their features and capabilities. Then we studied the source code of the projects which used them to extract their usage context. Our findings are summarized in Figure 4.2. Generally, there are four reasons behind the use of ELUs: (1) General-purpose logging, (2) LU interactions, (3) Internationalization, and (4) Modularization.

General-purpose Logging

General-purpose logging refers to the most common logging usage context, which requires: (1) *verbosity levels*, which are to control the amount of log message outputted; (2) *configurations*, which are used to specify various options and policies to format and output the logs; and (3) *thread safety*, which is to ensure the logs are recorded in sequence for a multi-threaded system.

<table>
<tr><th>Rationales</th><th>% of Top-100 ELUs</th><th>% of Projects</th><th>Top 2 LUs (where applicable)</th><th>Code Examples</th></tr>
<tr><td>General-Purpose</td><td>12%</td><td>89.9%</td><td>android.util.Log (1)

SLF4J(2)</td><td>
<pre>static final Logger LOG = LoggerFactory.getLogger(Configuration.class);
...
LOG.debug("parsing File " + file);</pre>
Coniguration.java (Hadoop, using SLF4J)
</td></tr>
<tr><td>LU Interactions</td><td>83%</td><td>14.3%</td><td>okhttp3.logging.HttpLoggingInterceptor (8)

io.netty.handler.logging.LoggingHandler (11)</td><td>
<pre>HttpLoggingInterceptor loggingInterceptor = new HttpLoggingInterceptor();
...
final OkHttpClient.Builder httpClientBuilder = new OkHttpClient.Builder();
...
httpClientBuilder.addInterceptor(loggingInterceptor);</pre>
Constants.java (bitcoin-wallet, using okhttp3.logging.HttpLoggingInterceptor)
</td></tr>
<tr><td>Internationalization</td><td>4%</td><td>1.0%</td><td>JBoss Logging (15)

org.glassfish.jersey.logging (29)</td><td>
<pre>@LogMessage(level = WARN)
@Message(value = "I/O reported cached file could not be found : %s : %s",
id = 23)
void cachedFileNotFound(String path, FileNotFoundException error);</pre>
CoreMessageLogger.java (Hibernate-ORM, using JBoss Logging)
<pre>...
catch (FileNotFoundException e) {
  log.cachedFileNotFound(serFile.getName(), e);
}</pre>
CacheableFileXmlSource.java (Hibernate-ORM, using JBoss Logging)
</td></tr>
<tr><td>Modularization</td><td>1%</td><td>1.7%</td><td>AOP Logging (10)</td><td>
<pre>@Around("execution(* org.unitedinternet.cosmo.service.
                    ContentService.getRootItem(..)) &&" + "args(user)")
public Object checkGetRootItem(ProceedingJoinPoint pjp, User user)
    throws Throwable {
  LOG.debug("in checkGetRootItem(user)");
...</pre>
SecurityAdvice.java (cosmo using AOP Logging)
<pre>public class StandardContentService implements ContentService {
...
  public HomeCollectionItem getRootItem(User user) ...</pre>
StandardContentService.java (cosmo using AOP Logging)
</td></tr>
</table>

Figure 4.2: The rationales behind the use of Top-100 most used ELUs.

Among the top-10 most popular LUs, 8 are for general-purpose logging. Six ELUs (`SLF4J`, `Log4j 1.x`, `Log4j 2`, `java.util.logging`, `Apache Commons Logging`, and `LogBack`) are considered as general-purpose LUs for desktop or server-based systems and two ELUs (`android.util.Log` and Timber [188]) are general-purpose LUs for Android-based systems. As shown in Figure 4.2, a log printing function from a general-purpose ELU generally consists of four parts: the logging object (`LOG`), the verbosity level (`debug`), the static texts describing the logging context (`"parsing File"`), and the dynamic contents revealing the runtime information (`file`).

Among all the desktop or server-based ELUs, `SLF4J` is the most popular ELU due to its improved performance and compatibility with other LUs [185, 186, 163]. There are many Android projects (4,477) in our study. Most of them are *small*-sized projects that used the default ELU from Android SDK (i.e., `android.util.Log`).

LU Interactions

LU interactions concerns about configurating and controlling the logging behavior for the imported packages. Many Java-based projects are built on top of existing packages, many of which contain ILUs. In order to configure and control the logging behavior for one imported package, developers have to invoke the logging APIs from this package. For example, 161 projects use the `okhttp3` [189] to send and receive data from network. In order to enable logging for the `okhttp3` package, developers of the `bitcoin-wallet` project [190], which is a Bitcoin payment app for Android, invokes the `addInterceptor` method from `okhttp3`'s ILU as shown in Figure 4.2. 83 out of the top-100 ELUs in this study are used for this reason. Hence, *LU interactions* is the main reason why there are so many ELUs used in the wild.

Internationalization

Internationalization concerns about adapting the logs to other human languages (e.g., German or Chinese). In order to support internationalization, developers first need to create a centralized file, which stores a list of pre-defined log message templates. Each of the log message template comprises four parts: the verbosity level, the parameterized string, the logging methods, and the message ID.

The third row of Figure 4.2 shows one such example. This code snipeet is from `Hibernate-ORM`, which is a popular Object-Relational Mapping (ORM) framework. Inside the centralized file, `CoreMessageLogger.java`, there are two annotations (`@LogMessage` and `@Message`) for each logging method and the message ID. `@LogMessage` contains a key/-value pair, which defines the verbosity level (`level = WARN`) of this log message. `@Message` provides the parameterized string, which contains the static texts (`I/O reported cached file could not be found`) and placeholders for parameters `%s`. The message ID is an integer, that can be used to uniquely identify this log message. The logging method which implements this log message is `cachedFileNotFound`. It takes in two parameters: the file path and the error message. In order to output this log message, the `cachedFileNotFound` method needs to be invoked. The code snippet in `CacheableFileXmlSource.java` shows

one example of how this log message can be invoked during runtime. It is used within a catch block to handle an exception.

To translate the log messages into different human languages, developers need to create a translation property file and define internationalized labels for each log message. For example, the property file `log_en_FR.properties` would contain all the French labels for all of the above English log messages.

The most common ELU for *Internationalization* is JBoss Logging [34]. Although there are three other ELUs, which also support this usage context, they are just wrappers of the JBoss Logging.

Modularization

Modularization concerns about improving the modularity of the logging code. Logging code is a cross-cutting concern, as it inter-mixes with the feature code. Only one ELU, AOP-based logging, is used for this reason. AOP is a programming paradigm, which is designed to improve modularity by reducing the amount of cross-cutting concerns [44]. Logging is considered as one of its common use cases.

In order to perform AOP-based logging, developers need to provide rules through aspect files. A typical aspect file consists of pointcuts and advice. A pointcut is to define the point of execution where the cross-cutting concern (e.g. logging) needs to be applied. An advice is the additional code (e.g. logging code) instrumented.

Figure 4.2 shows a real-world example from `cosmo`, a calendar server that implements CalDAV protocol. The file `SecurityAdvice.java` is the aspect file. The instrumented points (pointcuts) are defined through the annotation. In this example, the annotation `@Around` means both the beginning and the end of the methods will be instrumented. The value within the brackets specify the instrumented methods. In this example, any methods with the name `getRootItem` within package `org.unitedinternet.cosmo.service.ContentService` and parameter type as `user` will be instrumented. The instrumented code (advice) is defined via method `checkGetRootItem`. It contains a log printing statement with the message `in checkGetRootItem` and the user name. The code snippet in `StandardContentService.java` shows how a log message is actually generated. This class implements the interface `ContenService`, which is within the specified package. It contains a method `getRootItem`, of which the parameter is `user`. Therefore, this method qualifies the pre-defined instrumented rule above. Hence, during runtime, the above logging statment will be executed while entering and exiting this method. In general, only 2% of our studied projects adopted AOP-based logging. This matches with previous empirical studies [69, 70], which indicated the very limited use of AOP in the wild.

Findings: There are four main reasons behind the use of ELUs: (1) General-purpose logging, (2) LU interactions, (3) Internationalization, and (4) Modularization. Majority of the projects use ELU for *general-purpose logging*. 83 out of the top-100 mostly used ELUs are used for *LU interactions*. Some ELUs provide unique features like *internationalization* and *modularization*, but they are only used by a very small number of projects.
Implications: Many ELUs are used, as developers need to enable and configure the logging behavior for the imported packages. It is important to manage these LUs to better monitor and debug systems' runtime behavior. An existing study [106] shows that log configurations are one of the main source of errors for Java-based projects. Unfortunately, only one tool [81] is developed to detect invalid loggers in the configuration files. To effectively monitor and debug systems' behavior, developers need to figure out how to configure the logging behavior for the imported packages and how to interact these LUs with the LUs in their own projects. Hence, techniques to recover the logging architecture and conduct automatic configuration are needed.

4.4.2 RQ2: Why do developers implement ILUs in their projects?

It is not clear why many projects have still implemented their ILUs even there are 866 ELUs available in the wild. To investigate the rationales behind this, out of 2,990 projects which have implemented ILUs, we randomly selected 341 of them for close manual examination. This corresponds to a confidence level of 95% with a confidence interval of ±5%. We used the stratified sampling technique [191] to ensure representative samples are selected from projects of different sizes. The portion of the sampled projects within different sized projects is equal to the relative weight of the total number of the projects with that size. For example, there are a total of 721 *small*-sized projects which contain ILUs. Hence 82 ($\frac{721}{2990} \times 341$) *small*-sized projects are selected.

For each selected project, we carefully studied how their ILUs are being used by reading through the source code. We also examined the relevant commit logs, issue reports, and pull requests [192] to look for the rationales on why ILUs are implemented. In the end, we have identified three project-specific logging needs as shown in Figure 4.3: (1) Defining the logging format, (2) Compatibility with other LUs, and (3) Ease of configuration and dependency management. Note that the percentage of projects with different sizes do not add up to 100% because: (1) one project could contain multiple ILUs; (2) few ILUs ($< 4\%$) are mislabeled.

Defining the Logging Format

Log messages are generally loosely structured, which contain free formed texts. Many projects define their own format of the log messages, so that they can be automatically parsed and analyzed. For example, the `Hadoop` project introduces audit logging in order to satisfy the security compliance requirements. The format of the auditing logs vary across different `Hadoop` components. For example, in `hadoop-common` component shown in Figure 4.3, an interface (`KMSAuditLogger`) is first defined with a method `logAuditEvent`, which defines the

<table>
<tr><th>Rationales</th><th>% of Projects</th><th>Code Examples</th></tr>
<tr><td>Defining the Logging Format</td><td>Small (64.6%)
Medium (53.5%)
Large (47.6%)
Very Large (72.7%)</td><td>

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
class SimpleKMSAuditLogger implements KMSAuditLogger {
  final private Logger auditLog = LoggerFactory.getLogger(KMS_LOGGER_NAME);
  public void logAuditEvent(final OpStatus status, final AuditEvent event) {
    switch (status) {
      case OK:
      auditLog.info("{}[op={}, key={}, user={}, accessCount={}, interval={}ms] {}",
      status, event.getOp(), event.getKeyName(), event.getUser(),
      event.getAccessCount().get(),
      (event.getEndTime() - event.getStartTime()), event.getExtraMsg());
      break;
```

SimpleKMSAuditLogger.java (Hadoop)
</td></tr>
<tr><td>Compatibility with other LUs</td><td>Small (7.3%)
Medium (18.4%)
Large (25.4%)
Very Large (27.3%)</td><td>

```
public interface LogDelegate {
...
  void error(Object message);
}
```

LogDelegate.java (vert.x)

```
public class JULLogDelegate implements LogDelegate {
  private final java.util.logging.Logger logger;
  JULLogDelegate(final String name) {
    logger = java.util.logging.Logger.getLogger(name);
  }
  ...
  public void error(final Object message) {
    log(Level.SEVERE, message);
  }
```

JULLogDelegate.java (vert.x)
</td></tr>
<tr><td>Ease of Configuration and Dependency Management</td><td>Small (24.4%)
Medium (30.3%)
Large (33.3%)
Very Large (27.3%)</td><td>

```
public class DefaultLogSystem implements LogSystem {
  public static PrintStream out = System.out;
  ..
  public void error(Throwable e) {
    out.println(new Date()+" ERROR:" +e.getMessage());
    e.printStackTrace(out);
  }
  ...
```

DefaultLogSystem.java (Slick2D)
</td></tr>
</table>

Figure 4.3: The rationales behind why ILUs are implemented.

auditing methods to be invoked. To facilitate code reuse, `SimpleKMSAuditLogger` implements this interface by using the adapter pattern. It implements the `logAuditEvent` method by invoking the `info` method from an `SLF4J` logger object. The `logAuditEvent` method contains a switch statement, in which depending on the actual event, different audit logs will be outputted. The resulting audit logs are much more structured compared to the regular loosely defined log messages. Other ILUs like `Cassandra`'s `StatusLogger` also fall into this case.

Compatibility with other LUs

Many of the studied projects can be packaged and used by other projects. For reusable packages, it is preferred to be flexible and compatible with different ELUs, as developers always want to use the most up-to-date LUs in their projects. Since LU migrations require high manual efforts and are error-prone [80], developers usually implement their ILUs to separate the coupling of their logging code with the LUs.

`Vert.x` [193] is a popular tool-kit for building reactive applications on the JVM. It receives more than 10,000 stars on GitHub. It suppports four general purpose Java ELUs: `java.util.logging`, `Log4j 1.x`, `Log4j 2`, and `SLF4J`. This functionality is realized by implementing the strategy design pattern to unify the APIs among these four ELUs. These four ELUs do not share the same set of verbosity levels. Therefore, the ILU needs to provide a unified interface (`LogDelegate`) for their logging methods. The `LogDelegate` interface defines a set of common logging APIs (e.g., `info`, `error`). Separate implementation classes are introduced to wrap around the four popular general purpose ELUs: `java.util.logging`, `Log4j 1.x`, `Log4j 2`, and `SLF4J`. Figure 4.3 shows a code snippet for `JULLogDel`
`egate`, which adapts the functionalities of `java.util.logging` to the common interface defined by `LogDelegate`. To implement the `error` method, it calls the `log` method along with the verbosity level `SEVERE` and the variable `message` from `java.util.logging`.

Ease of Configuration and Dependency Management

One of the common errors associated with logging is the configuration of LUs [106]. The main cause of this is due to complex dependency structures [157, 158, 159, 160, 161, 162]. Hence, about 29.3% of the ILUs are implemented to ease the configuration and dependency management issues. They are usually built from the ground up using only the standard JDK libraries and are not dependent on any ELUs to minimize the effort to manage dependencies. Figure 4.3 shows one example. This code snippet is from `Slick2D`, which is a 2D Java game library. This ILU is designed to be lightweighted and easy to use. It logs the complete error information with timestamps to the console.

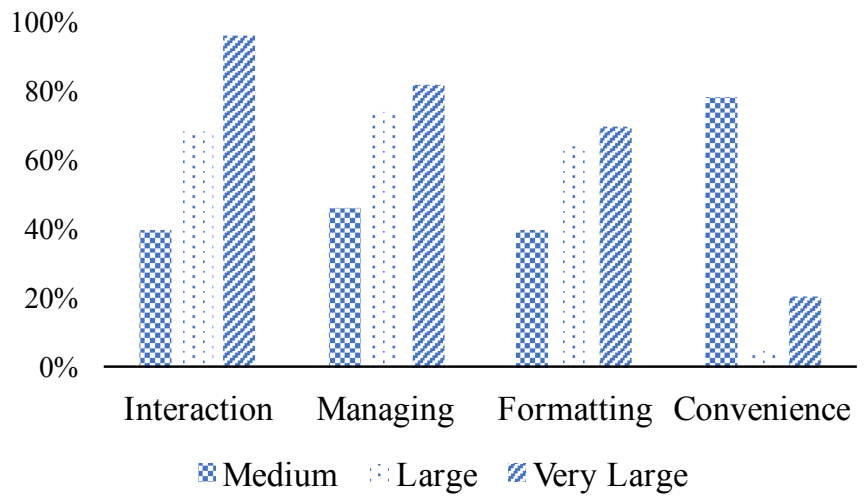

Figure 4.4: The usage context behind the Top-100 projects, which contain the most LUs.

Findings: There are three project specific needs, which lead to the implementation of ILUs: (1) defining the logging format, (2) compatibility with other LUs, and (3) ease of configuration and dependency management. Among these three needs, *defining the logging format* is the most common one. 55.3% of the sampled projects implemented ILUs for this specific need. As the size of the projects grow larger, more projects implement ILUs to satisfy the needs of *compatability with other LUs*.
Implications: Although more than 20% of the sampled projects implemented their own ILUs from the ground up. These ILUs are mainly used internally for debugging purposes. They are usually not intended to be used by other projects, since they might not satisfy the general logging needs (e.g., not thread-safe). Additional tools and techniques need to be developed to automatically warn developers who imported packages which implement ILUs.

4.4.3 RQ3: How are multiple LUs used in the wild?

The goal of this RQ is to understand and characterize the uses of multiple LUs in one project. We sorted our studied projects by the number of LUs that are used. Then we selected the top-100 projects, which used the most LUs, for close manual examinations. The number of LUs for these projects ranges from 7 (`Hadoop`) to 21 (`Liferay-portal`). Figure 4.4 summarizes our findings. There are generally four usage context behind the use of multiple LUs: (1) Interaction with LUs from the imported packages; (2) Managing the logging contents; (3) Formatting logging messages across different components; and (4) Developer convenience. As the project sizes grow bigger, the usage context of multiple LUs also become more complex. Note that there is no *small*-sized projects among them, as they generally use much fewer LUs.

```
pipeline.addLast("logger", new LoggingHandler(LogLevel.INFO));
```

(a) InboundConnectionInitiator.java(Cassandra, using Netty LoggingHandler)

```
KieRuntimeLogger logger=KieServices.Factory.get().
       getLoggers().newFileLogger(session,"log/correlation");
```

(b) CorrelationExample.java (OpenNMS, using KieRuntimeLogger)

Figure 4.5: An example of using multiple LUs for *interaction with LUs from the imported packages*.

Interaction with LUs from the imported packages

Many Java-based software systems use third party packages, which also contain LUs. In order to have full observability of the resulting systems, it is important to enable logging across all the components. Figure 4.5 shows two different examples on how to configure or enable logging for the imported packages. However, due to the API variabilitiy of these LUs, different techniques are needed in order to enable or configure their logging behavior. For example, `Netty` uses event-based programming. To enable the logging of the `Netty` package, the `Cassandra` developer needs to add the `LoggingHandler`. In order to enable logging for `Kie`, the `OpenNMS` developer needs to create a new logger to output the logs to a file. The more third party packages are used, the more interactions there are with the LUs from these imported packages. This is one of the main reasons why many *large* or *very Large*-sized projects use more LUs than the smaller sized projects.

Managing the logging contents

In addition to configuring and enabling the logging behavior for the imported packages, sometimes the studied projects also use the LUs from the imported packages for additional logging. This is mainly to ensure the relevant logging contents are stored in the same location. `Maven` is a popular tool to build and manage Java projects. The core of `Maven` consists of a set of plugins. Each plugin is reponsible for a particular functionality. For example, `clean`, which is used to clean up the build artificats, and `compiler`, which is used to compile Java sources, are two default plugins. During execution, the logs generated from these plugins will be redirected to the same storage location. If other projects intend to develop `Maven` plugins, they are advised to use the LUs from `Maven` to generate build related logs [194], so that `Maven`-related information can be aggregated to one centralized location, which is easy to analyze and archive. Figure 4.6 shows one such example. On one hand, `karaf` uses their own LU to record system-specific information. On other hand, it uses the LU from `Maven` to record build related logs.

```
import org.apache.maven.plugin.logging.Log;
...
public File resolveById(String id, Log log)
                     throws MojoFailureException {
...
  log.debug("Resolving artifact " + id + " from ”
                                    + projectRepositories);
```

Dependency31Helper.java (karaf)

Figure 4.6: An example of using multiple LUs for *managing the logging contents*.

```
for (InstanceStatus status : instanceStatusList) {
  if (INSTANCE_RUNNING !=
        status.getInstanceState().getCode()){
    LOGGER.debug("Instances are up but not
                all of them are in running state.");
    return false;
  }
 }
```

ASGroupStatusCheckerTask.java (cloudbreak)

Figure 4.7: An example of using multiple LUs for *developer convenience*.

Formatting logging messages across different components

In some cases, one LU may not satisfy all the logging needs for one project. For example, as shown in Figure 4.3, `Hadoop` uses `SLF4J` for generating execution logs, which are used for debugging and monitoring purposes. It also uses ILU to generate audit logs to satisfy security requirements.

Developer convenience

Some of the top-100 projects use AOP-based logging. However, the developers also include logging code, which is instrumented using general-purpose ELUs. This is mainly due to developer convenience. For example, `cloudbreak` uses AOP-based logging, but it also uses SLF4J. Figure 4.7 shows one such example. Since checking the instance state is a very localized concern, it is much faster to instrument with the general purpose LU than AOP-based logging. Similar cases also apply to LUs, which are used for internationalization. For example, `Hibernate` uses the JBoss logging to support internationalization. However, instead of putting the log messages into the centralized file, the developer chose to place their log message along with the feature code for debugging purposes.

Findings: There are four usage contexts behind the use of multiple LUs in one project: (1) interaction with LUs from the imported packages; (2) managing the logging contents; (3) formatting logging messages across different components; and (4) developer convenience. Except for *developer convenience*, the percentage of these usage contexts increases as the project sizes increase.
Implications: Logging is considered as a cross-cutting concern, as the logging code scatters across the entire system and inter-mixes with the feature code. To cope with this challenge, AOP is introduced. However, AOP cannot be used to satisfy the current logging needs, due to their inconvenience on specifying localized logging context. More importantly, for large-scale projects which use many third party packages, it is necessary to enable and configure the logging behavior for these imported packages in order to gain full observability of the entire systems. The management of the logging behavior for these packages is rather complex and introduces another form of cross-cutting concerns. Further research is urgently needed to develop tools or techniques to automatically manage and modularize such concerns.

4.5 Evaluation

Although we have analyzed 425 Java-based projects, our study focuses on Java-based open-source projects in GitHub. In order to assess the generalizability of our findings, we conducted a semi-structured inteview with 13 experienced industrial developers. We decided to conduct semi-structured interviews instead of surveys so that we can interact with the participants in a more flexible way. For example, we have prepared a set of questions before-hand. During the interview, we sometime asked follow-up questions based on participants' answers on the fly [195].

4.5.1 Setup

Similar to [58], the participants are from the authors' personal contacts. All of them are working at large-scale software companies. The development experience of the participants ranges from one to eight years. The types of projects that participants are working on vary from server-side projects (9), frameworks (3), and client application (1). All these projects have been widely used by millions of users worldwide. The programming languages that the participants use daily include Java, PHP, C++, C#, Python, and Go.

4.5.2 Findings

All of the participants have inserted, deleted, updated logging code in their development activities. It reaffirms that software logging is a pervasive practice [9].

Cross-validation of our findings

We presented our findings on the rationales behind the use of ELUs and ILUs, as well as the usage context for multiple LUs. Then we asked the participants: (1) whether their projects adopt one or more of the studied LUs; (2) whether our characterized usage context and the logging needs would be useful for them; and (3) if there are any additional information to add or comment on.

Some participants mentioned that they did adopt one or more of the studied LUs in this chapter. However, the rationales (e.g., JBoss-style logging) is a bit different. All participants felt that the findings and code examples from this study can help them better configure and manage the logging behavior for their projects.

- **ELUs**: All participants acknowledged that the general-purpose logging is the most commonly used logging need. However, they also acknowledge the challenges on co-evolving the logging code with the rapidly changed feature code, as the logging code is inter-mixed with the feature code. Although the participants aggreed that AOP-based logging is a great idea, only two of the interviewed participants used AOP-based logging in their projects. This is mainly due to (1) high learning curve of a different programming paradigm, (2) difficult to translate logging concerns into rules (a.k.a., advice), and (3) lack of automated support for legacy logging code.

  Compared to open-source projects, there is a much higher portion of industrial projects (7/13) that adopted the LUs that centralize the log messages (a.k.a., JBoss-style logging). One participant mentioned that in addition to internationalization, the centralized logging-style can: (1) improve the accuracy and the speed of the log processing task [64]; and (2) attach additional meta information (e.g., issue resolution strategies) with the message ID for better field support.

- **ILUs**: The participants mentioned that the choice of LUs were decided by the software architects or senior developers. Once the LUs were set up, only under very rare cases that they will migrate or modify LUs. None of them have directly modified the existing LUs. We asked the participants what type of LUs were used in their projects. Three of them said they directly use open source LUs such as `Log4j` [77] (Java) and `log4net` [196] (C#). The rest of them use ILUs included in their project. The rationales behind implementing ILUs are similar to our findings from the open source projects.

- **The use of Multiple LUs**: The participants are surprised about the use of multiple LUs. Five of them did not realize the need to do this until we presented our findings. They complained about the challenges on debugging and monitoring their projects, which use many third-party packages. By leveraging our findings and code examples, they can enable and manage the logging behavior from these imported packages, which will greatly improve the observability of the overall systems.

Beyond Logging

Five participants mentioned that they have integrated tracing tools (e.g., opentracing [71]) or Application Performance Monitoring (APM) tools (sentry [197], Google firebase [198], etc.) into their systems. They applied these tools to replace some of the logging functionalities such as crash reporting and collection of the profiling data. Unlike logging, most of these tools do not need developers to arbitrarily write code to record the desired information. For example, one participant mentioned that they adopted Spring-sleuth [199]. It is a distributed tracing tool which can automatically record the interactions between multiple Sprint Boot microservices. These tools pre-instrument output statements to record information such as RPC calls and stack traces, and they need minimum configuration efforts. The participants also mentioned that although these tools are useful, they still cannot completely replace the needs for software logging in their projects.

4.6 Discussions

Logs have been used extensively in practice. Instead of invoking output functions like System.out.println, developers prefer to use logging utilities (LU) to instrument their systems. This chapter categorizes the complex use of LUs in the wild and presents multiple implications which are useful for developers and researchers. In this section, we will discuss the significance of our findings and present some future work.

4.6.1 Complex Use of LUs

There are generally four reasons to use LUs: (1) general-purpose logging, (2) LU interactions with imported packages, (3) internationalization of the log messages, and (4) modularization of the logging code. Larger projects tend to use multiple LUs in one project to satisfy one or more of the following usage context: (1) interaction with LUs from various imported packages, (2) managing the logging contents, (3) formatting logging messages across different components, and (4) developer convenience. Many projects also implement their own LUs to satisfy project specific needs like defining the logging format, compatibility, and ease of configuration and dependency management. Such findings would be useful for developers and researchers who are interested in developing and maintaining LUs:

- Our findings raise developers' awareness that their systems can contain multiple LUs due to the imported packages. It is important to manage these LUs to better monitor and debug systems' runtime behavior.
- To effectively monitor and debug systems' behavior, developers need to figure out how to configure the logging behavior for the imported packages and how to interact these LUs with the LUs in their own projects. Hence, techniques to recover the logging architecture are needed.

- To ensure QoS, best practices and anti-patterns for managing LUs are needed, particularly in the following areas: (1) selecting the appropriate default verbosity levels for LUs for good observability with low overhead; and (2) correlating logging information across different LUs in one project.
- In addition to common LUs (e.g., Log4j or SLF4J), many projects also use additional ELUs or implement ILUs. These LUs are project-dependent and require implementation/maintenance effort. The findings in the chapter can guide researchers to propose more general techniques to satisfy such logging needs.

4.6.2 Beyond LUs

Certain findings in this chapter may be applicable to other utilities. It is important to study the use of different utilities in the wild due to: (a) multiple utilities (e.g., database/testing frameworks and ad libraries) with same/similar functionalities can be used in one project; and (b) cross-cutting concerns (e.g., logging, analytics and security) are implemented not only in the projects' code but also in the imported packages. Effectively managing such concerns is an open problem, which is important for development and maintenance of such systems [200, 201].

4.7 Related Work

In this section, we discuss two realted research areas.

4.7.1 Empirical Studies on Logging Practices

There are no well-established logging guidelines in neither industrial [8, 32] nor open source systems [10, 9, 131]. Hence, it is important to derive best practices and common mistakes by studying the current logging practices. Shang et al. [131] found that the amount of logging code is correlated with the amount of post-release bugs. Kabinna et al. [80] studied the logging library migrations for 33 Apache-based Java projects. They found that migrating logging libraries requires high manual efforts and is error-prone. Zhi et al. [81] conducted an exploratory study on the configuration aspects of the LUs.

Our work is different from the above studies in the following three aspects: (1) this chapter is the first study on the use of LUs, which includes both ELUs (a.k.a., logging libraries) as well as ILUs. (2) The scale of our study is much bigger than the previous ones, as we have studied over 11,194 Java-based GitHub projects, which uses 3,856 LUs. (3) Different from all of the above work, which are mainly quantitative studies, we have performed both the quantitative and the qualitative studies on the use of Java-based LUs.

4.7.2 Improving the Quality of the Logging Code

This area can be further divided into three parts: (1) where-to-log, (2) what-to-log, and (3) how-to-log.

- **where-to-log** is related to the problem of selecting appropriate logging points in the source code. Fu et al. [8] proposed a data-mining based approach to automatically extract the important attributes which affect the locations of the logging points. Zhu et al. [107] proposed a machine-learning based technique to learn common logging points based on the code structures. Yuan et al. [18] proposed a program-analysis based method to add logging points for debugging purposes. Ding et al. [21] proposed a constraint solving based method to select the optimal logging points which incur minimum performance overhead while keeping the maxmium runtime information. Zhao et al. [2] came up with a tool Log20, which automatically places the logging points under certain overhead threshold.

- **what-to-log** is related to the problem of providing complete runtime information in the logging code. Yuan et al. [35] proposed a program analysis based approach to add additional variables into existing logging statements so that more complete execution paths can be recovered, thus improving the diagnosability. He et al. [104] characterized the static texts inside logging statements for 10 Java and 7 C# projects. They provided an NLP-based description generation tool to automatically generate static texts in logging statements.

- **how-to-log** is related to the problem of designing and maintaining high quality logging code. Yuan et al. [10] partially studied the inconsistent verbosity level problem through a clone based approach. Li et al. learned from code changes in history to predict just-in-time logging code changes [111] and to suggest the most approapriate verbosity levels [48] through machine learning-based approaches. Chen et al. [36] summarized six types of anti-patterns within logging statements and proposed a tool to automatically detect them. Li et al. [38] proposed an automated tool to detect duplicate logging code smells. Chen et al. [123] proposed an approach to extract the Logging-Code-Issue-Introducing changes.

Our study lies within the category of **how-to-log**, but differs with the above work in the sense that our focus is on the LUs instead of the log printing code.

4.8 Threats to Validity

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

4.8.1 External Validity

We conducted our study only on 11,194 Java-based open source projects. We cross-validated our findings by interviewing with industrial developers. Although our approach can be easily adapted to another study on the use of LUs, our finding may not be generalizable to projects written in other programming languages.

4.8.2 Internal Validity

In our study, we removed all of the *very small*-sized projects, since only a small fraction of them use LUs and most of them are trial projects for self-studying and collections of code snippets. We also removed forked projects, since they are likely duplications of the base ones. We also removed unpopular projects which contain few stars. This practice is similar to many of the previous empirical studies on GitHub-based proejcts [177, 178, 179].

4.8.3 Construct Validity

In our study, we used a heuristics-based approach to count the adoptions of LUs. We made our best efforts to avoid bringing in any false positives. Through manual verification of the randomly selected samples, our approach yields an precision of 96%.

4.9 Conclusion

In this chapter, we conducted a large-scale empirical study on the use of Java LUs in the wild. We studied $11,194$ Java-based projects in the GitHub. These projects use $3,856$ LUs. Our findings suggest that the complexity of the use of LUs increases as the project size increases. Although many projects only use LUs for general-purpose logging, the actual logging needs vary from project to project. The main reason behind multiple use of LUs is to enable or manage the logging behavior of the imported packages. Many projects choose to implement their own ILU mainly for defining the project-specific format of their logging code. The findings and the implications presented in this chapter will be useful for LU designers and researchers as well as system developers.

Chapter 5

An Automated Approach to Estimating Code Coverage Measures via Execution Logs

5.1 Introduction

A recent report by Tricentis shows that software failure caused $1.7 trillion in financial losses in 2017 [202]. Therefore, software testing, which verifies a system's behavior under a set of inputs, is a required process to ensure the system quality. Unfortunately, as software testing can only show the presence of bugs but not their absence [203], complete testing (a.k.a., revealing all the faults) is often not feasible [204, 205]. Thus, it is important to develop high quality test suites, which systematically examine a system's behavior.

Code coverage measures the amount of executed source code, when the system is running under various scenarios [206]. There are various code coverage criteria (e.g., statement coverage, condition coverage, and decision coverage) proposed to measure how well the test suite exercises the system's source code. For example, the statement coverage measures the amount of executed statements, whereas the condition coverage measures the amount of true/false decisions taken from each conditional statement. Although there are mixed results between the relationship of code coverage and test suite effectiveness [207, 208], code coverage is still widely used in research [209, 210, 211] and industry [212, 213, 214] to evaluate and improve the quality of existing test suites.

There are quite a few commercial (e.g., [215, 216]) and open source (e.g., [33, 217]) tools already available to automatically measure the code coverage. All these tools rely on instrumentation at various code locations (e.g., method entry/exit points and conditional branches) either at source code [218] or at binary/bytecode levels [217, 219]. There are three main issues associated with these tools when used in practice: (1) **Engineering challenges**: for real-world large-scale distributed systems, it is not straight-forward to configure and deploy such tools, and collect the resulting data [220, 221]. (2) **Performance overhead**: the heavy instrumentation process can introduce performance overhead and slow down the

system execution [222, 223, 224]. (3) **Incomplete results**: due to various issues associated with code instrumentation, the coverage results from these tools do not agree with each other and are sometimes incomplete [23]. Hence, the application context of these tools is generally very limited (e.g., during unit and integration testing). It is very challenging to measure the coverage for the system under test (SUT) in a field-like environment to answer questions like evaluating the representativeness of in-house test suites [225]. Such problem is going to be increasingly important, as more and more systems are adopting the rapid deployment process like DevOps [226].

Execution logs are generated by the output statements (e.g., `Log.info(''User'' + user + ''checked out'')`) that developers insert into the source code. Studies have shown that execution logs have been actively maintained for many open source [10, 9] and commercial software systems [8, 32] and have been used extensively in practice for a variety of tasks (e.g., system monitoring [16], problem debugging [18, 3], and business decision making [5]). In this chapter, we have proposed an approach, called *LogCoCo* (Log-based Code Coverage), which automatically estimates the code coverage criteria by analyzing the readily available execution logs. We first leverage program analysis techniques to extract a set of possible code paths from the SUT. Then we traverse through these code paths to derive the list of corresponding log sequences, represented using regular expressions. We match the readily available execution logs, either from testing or in the field, with these regular expressions. Based on the matched results, we label the code regions as `Must` (definitely covered), `May` (maybe covered, maybe not), and `Must-not` (definitely not covered) and use these labels to infer three types of code coverage criteria: method coverage, statement coverage, and branch coverage. The contributions of this chapter are:

1. This work systematically assesses the use of the code coverage tools in a practical setting. It is the first work, to the authors' knowledge, to automatically estimate code coverage measures from execution logs.

2. Case studies on one open source and five commercial systems (from Baidu) show that the code coverage measures inferred by LogCoCo is highly accurate: achieving higher than 96% accuracy in seven out of nine experiments. Using LogCoCo, we can evaluate and improve the quality of various test suites (unit testing, integration testing, and benchmarking) by comparing and studying their code coverage measures.

3. This project is done in collaboration with Baidu, a large scale software company whose services are used by hundreds of millions of users. Our industrial collaborators are currently considering adopting and using LogCoCo on a daily basis. This clearly demonstrates the usefulness and the practical impact of our approach.

Chapter Organization: the rest of this chapter is structured as follows. Section 5.2 explains issues when applying code coverage tools in practice. Section 5.3 explains LogCoCo by using a running example. Section 5.4 describes our experiment setup. Section 5.5 and 5.6 study two research questions, respectively. Section 5.7 introduces the related work. Section 5.8 discusses the threats to validity. Section 5.9 concludes the chapter.

5.2 Applying Code Coverage Tools in Practice

We interviewed a few QA engineers at Baidu regarding their experience on the use of the code coverage tools. They regularly used code coverage tools like JaCoCo [33] and Cobertura [217]. However, they apply these tools only during the unit and integration testing. It turned out that there are some general issues associated with these state-of-the-art code coverage tools, which limit their application contexts (e.g., during performance testing and in the field). We summarized them into the following three main issues, which are also problematic for other companies [220, 221]: (1) *Engineering challenges:* depending on the instrumentation techniques, configuring and deploying these tools along with the SUT can be tedious (e.g., involving recompilation of source code) and error-prone (e.g., changing runtime options). (2) *Performance overhead*: although these tools can provide various code coverage measures (e.g., statement, branch, and method coverage), they introduce additional performance overhead. Such overhead can be very apparent, when the SUT is processing hundreds or thousands of concurrent requests. Therefore, they are not suitable to be running during non-functional testing (e.g., performance or user acceptance testing) or in the field (e.g., to evaluate and improve the representativeness of the in-house test suites). (3) *Incomplete results*: the code coverage results from these tools are sometimes incomplete.

In this section, we will illustrate the three issues mentioned above through our experience in applying the state-of-the-art code coverage tools on HBase [172] in a field-like environment.

5.2.1 The HBase Experiment

HBase, which is an open source distributed NoSQL database, has been used by many companies (e.g., Facebook [227], Twitter, and Yahoo! [228]) serving millions of users everyday. It is important to assess its behavior under load (a.k.a., collecting code coverage measures) and ensure the representativeness of the in-house test suites (a.k.a., covering the behavior in the field).

YCSB [229] is a popular benchmark suite, originally developed by Yahoo!, to evaluate the performance of various cloud-based systems (e.g., Cassandra, Hadoop, HBase, and MongoDB). YCSB contains six core benchmark workloads (A, B, C, D, E, and F) which are derived by examining a wide range of workload characteristics from real-world applications [230]. Hence, we use this benchmark suite to simulate the field behavior of HBase.

Our HBase experiment was conducted on a three-machine-cluster with one master node and two region server nodes. These three machines have the same hardware specifications: Intel i7-4790 CPU, 16 GB memory, and 2 TB hard-drive. We picked HBase version 1.2.6 for this experiment, since it was the most current stable release by the time of our study. We configured the number of operations to be 10 million for each benchmark workload. Each benchmark test exercised all the benchmark workloads under one of the following three YCSB thread number configurations: 5, 10, and 15. Different number of YCSB threads indicates different load levels: the higher the number of threads, the higher the benchmark load.

5.2.2 Engineering Challenges

Since HBase is implemented in Java, we experimented with two Java-based state-of-the-art code coverage tools: JaCoCo [33] and Clover [231]. Both tools have been used widely in research (e.g., [23, 232, 233]) and practice (e.g., [220, 221]). These two tools use different instrumentation approaches to collecting the code coverage measures. Clover [231] instruments the SUT and injects its monitoring probes at the source code level, while JaCoCo [219] uses the bytecode instrumentation technique and injects its probes during runtime.

Overall, we found the configuration and the deployment processes for both tools to be quite tedious and error-prone. For example, to enable the code coverage measurement by JaCoCo, we had to examine various HBase scripts to figure out the command line options to startup HBase and its required jar files. This process was non-trivial and required manual effort, as the command line options could differ from systems to systems and even different versions of the same systems. The process for Clover was even more complicated, as we had to reconfigure the Maven build system to produce a new set of instrumented jar files. In addition, we could not simply copy and replace the newly instrumented jar files into the test environment due to dependency changes. It required a thorough cleanup of the test environment before re-deploying SUT and running any tests. We considered such efforts to be non-trivial, as we had to repeat this process on all three target machines. This effort could be much higher if the experiments were done on tens or hundreds of machines, which is considered as a normal deployment size for HBase [234].

We decided to proceed with JaCoCo. Its instrumentation and deployment process were less intrusive, as the behavior of HBase needed to be assessed in a field like environment.

5.2.3 Performance Overhead

We ran each benchmark test twice: once with JaCoCo enabled, and once without. We gathered the response time statistics for each benchmark run and estimated the performance overhead introduced by JaCoCo. Figure 5.1 shows the performance overhead for the six different workloads (workload $A, \ldots, F$). Within each workload, the figure shows the average performance overhead (in percentages) as well as the confidence intervals across different YCSB thread numbers. For example, the average performance overhead for workload A is 16%, but can vary from 10% to 22% depending on the number of threads.

Depending on the workload, the performance impact of JaCoCo varies. Workload B has the highest impact (79% to 106%) with JaCoCo enabled, whereas workload E has the smallest impact (4% to 13%). Overall, JaCoCo does have a negative impact on the SUT with a noticeable performance overhead ($> 8\%$ on average) across all benchmark tests. Hence, it is not feasible to deploy JaCoCo in a field-like environment with the SUT, as it can significantly degrade the user experience.

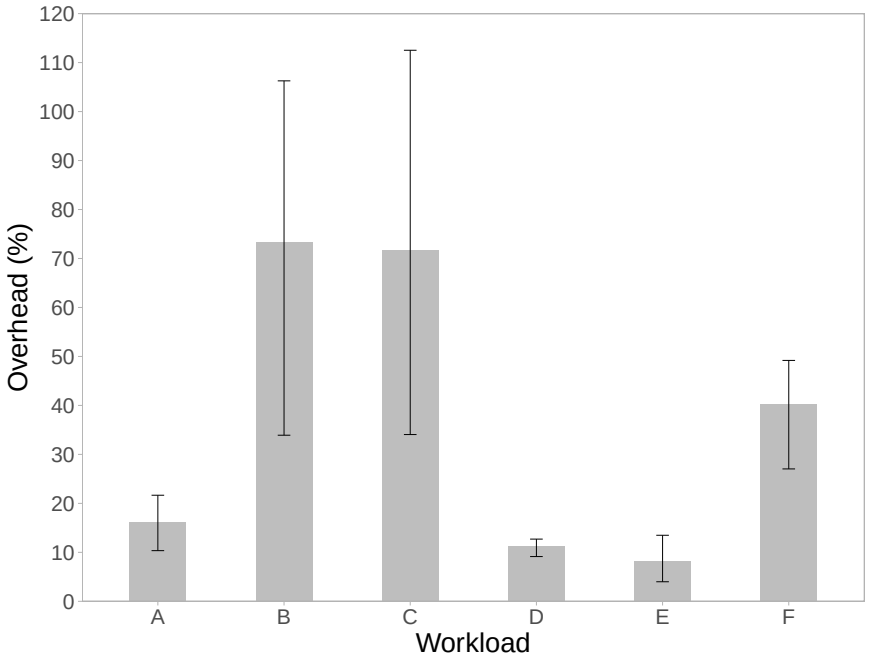

Figure 5.1: The JaCoCo overhead for the HBase experiment.

5.2.4 Incomplete Results

We sampled some of the code coverage data produced by JaCoCo for manual verification. We found that JaCoCo did not report the code coverage measures for some modules. JaCoCo only instrumented the HBase modules (a.k.a., the `hbase-server` module) in which the YCSB benchmark suite directly invoked. If the `hbase-server` module invokes another module (e.g., `client`) not specified during the HBase startup, the `client` module will not be instrumented by JaCoCo and will not have any coverage data reported. For example, during our experiment, the logging statement from the method `setTableState` in `ZKTableStateManager.java` was outputted. Hence, `setTableState` should be covered. Since `setTableState` calls `setTableStateInZK`, which calls `joinZNode` in `ZKUtil.java`, the method `joinZNode` should also be covered. However, the `joinZNode` method was marked as not covered by JaCoCo. A similar problem was also reported in [23].

To resolve the three issues mentioned above, we have proposed a new approach to automatically estimating the code coverage measures by leveraging the readily available execution logs. Our approach estimates the code coverage by correlating information in the source code and the log files, once the tests are completed. It imposes little performance overhead to the SUT, and requires no additional setup or configuration actions from the QA engineers.

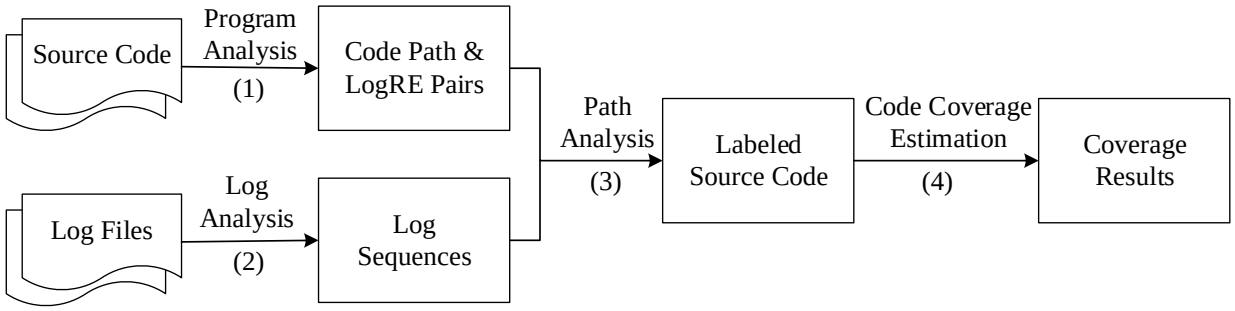

Figure 5.2: An overview of LogCoCo.

5.3 LogCoCo

In this section, we will describe LogCoCo, which is an automated approach to estimating code coverage measures using execution logs. As illustrated in Figure 5.2, our approach consists of the following four phases: (1) during the program analysis phase, we analyze the SUT's source code and derive a list of possible code paths and their corresponding log sequences expressed in regular expressions (LogRE). (2) During the log analysis phase, we analyze the execution log files and recover the log sequences based on their execution context. (3) During the path analysis phase, we match each log sequence with one of the derived LogRE and highlight the corresponding code paths with three kinds of labels: `May`, `Must`, and `Must-not`. (4) Based on the labels, we estimate the values for the following three code coverage criteria: method coverage, statement coverage, and branch coverage. In the rest of this section, we will explain the aforementioned four phases in details with a running example shown in Figure 5.3.

5.3.1 Phase 1 - Program Analysis

Different sequences of log lines will be generated if the SUT executes different scenarios. Hence, the goal of this phase is to derive the matching pairs between the list of possible code paths and their corresponding log sequences. This phase is further divided into three steps:

Step 1 - Deriving AST for Each Method

we derive the per method Abstract Syntax Tree (AST) using a static analysis tool called Java Development Tools (JDT) [180] from the Eclipse Foundation. JDT is a very robust and accurate program analysis tool, which has been used in many software engineering research papers (e.g., bug prediction [149], logging code analysis [36], and software evolution [235]).

Consider our running example shown on the left part of Figure 5.3. This step will generate two ASTs for the two methods: `computation` and `process`. Each node in the resulting AST is marked with the corresponding line number and the statement type. For example, at line 3, there is a logging statement and at line 4 there is an if statement. There are also edges

Log Sequences		Seq 1	Seq 2	Final Code Coverage
Branch selection set		[if@4:*true*,for@12:*true*,if@16: *false*]	[if@4:*false*,for@12:*true*,if@16: *false*]	
LogRE		(Log@3)(Log@51)(Log@14)+	(Log@3)(Log@14)+	
Code Snippets		Intermediate Code Coverage		
1	`void computation(int a, int b) {`	Must	Must	Must
2	`  int a = randomInt();`	Must	Must	Must
3	`  log.info("Random No: " + a);`	Must	Must	Must
4	`  if (a < 10) {`	Must	Must	Must
5	`  a = process(a);`	Must	Must-not	Must
6	`  } else {`	Must-not	Must	Must
7	`  a = a + 10;`	Must-not	Must	Must
8	`  }`	-	-	-
9	`  if (a % 2 = 0) {`	Must	Must	Must
10	`  a ++;`	May	May	May
11	`  }`	-	-	-
12	`  for (;b < 3; b++) {`	Must	Must	Must
13	`  a ++;`	Must	Must	Must
14	`  log.info("Loop: " + a);`	Must	Must	Must
15	`  }`	-	-	-
16	`  if (a > 20) {`	Must	Must	Must
17	`  log.info("Check: " + a);`	Must-not	Must-not	Must-not
18	`  }`	-	-	-
19	`}`	-	-	-
⋮	⋮	⋮	⋮	⋮
50	`int process(int num) {`	Must	Must	Must
51	`  log.info("Process: " + (++num));`	Must	Must	Must
52	`  return num;`	Must	Must	Must
53	`}`	-	-	-

Figure 5.3: The code snippet of our running example.

connecting two nodes if one node is the parent of the other node.

Step 2 - Deriving Call Graphs

the resulting ASTs from the previous step only contain the control flow information at the method level. In order to derive a list of possible code paths, we need to form call graphs by chaining the ASTs of different methods. We have developed a script, which automatically detects method invocations in the ASTs, and links them with the corresponding method body. In the running example, our script will connect the method invocation of the `process` method at line 5 with the corresponding method body starting at line 50.

Step 3 - Deriving Code Paths and LogRE Pairs

based on the resulting call graphs, we will derive a list of possible code paths. The number of resulting code paths depends on the number and the type of control flow nodes (e.g., if, else, for, and while), which may contain multiple branching choices. Consider the if statement at line 4 in our running example: depending on the true/false values for the conditional variable `a`, there can be two call paths generated: `[4, 5, 50, 51, 52]` and `[4, 6, 7]`.

We leverage the Breadth-First-Search (BFS) algorithm to traverse through the call graphs in order to derive the list of possible code paths and their corresponding LogREs. When visiting each control flow node, we pick one of the decision outcomes for that node and go to the corresponding branches. During this process, we also keep track of the resulting LogREs. Each time when a logging statement is visited, we add it to our resulting LogRE. If a logging statement is inside a loop, a "+" sign will be appended to it indicating that this logging statement could be printed more than once. For the logging statement, which is inside a conditional branch within a loop, it will be appended with a "?" followed by a "+". In the end, we will generate a branch selection set for this particular code path and its corresponding LogRE. There can be some control flow nodes, under which there is no logging statement node. In this case, we cannot be certain if any code under these control flow nodes will be executed. For scalability concerns, we do not visit the subtrees under these control flow nodes.

In our running example, there are four control flow nodes: line 4 (`if`), 9 (`if`), 12 (`for`), and 16 (`if`). If the conditions are true for line 4 and 12, and false for line 16, the branch selection set is represented as `[if@4:true, for@12:true, if@16: false]`. The value for the `if` condition node at line 9 is irrelevant, as there is no logging statement under it. We will not visit its subtree, as no changes will be made to the resulting LogRE. The resulting code path for this branch selection is `1,2,3,4,5,50,51,52,9,(10),12,13,14,16`. The corresponding LogRE is `(Log@3)(Log@51)(Log@14)+`. Line 10 in the resulting code path is shown in brackets, because there is no logging statement under the `if` condition node at line 9. Thus, we cannot tell if line 10 is executed based on the generated log lines.

```
1  2018-01-18 15:42:18,158 INFO  [Thread-1] [Test.java:3] Random No: 2
2  2018-01-18 15:42:18,159 INFO  [Thread-2] [Test.java:3] Random No: 4
3  2018-01-18 15:42:18,159 INFO  [Thread-1] [Test.java:51] Process: 3
4  2018-01-18 15:42:18,162 INFO  [Thread-2] [Test.java:14] Loop: 6
5  2018-01-18 15:42:18,163 INFO  [Thread-1] [Test.java:14] Loop: 4
6  2018-01-18 15:42:18,163 INFO  [Thread-1] [Test.java:14] Loop: 5
```

Figure 5.4: Log file snippets for our running example.

5.3.2 Phase 2 - Log Analysis

Execution logs are generated when logging statements are executed. However, since there can be multiple scenarios executed concurrently, logs related to the different scenario executions may be inter-mixed. Hence, in this phase, we will recover the related logs into sequences by analyzing the log files. Suppose after executing some test cases for our running example, a set of log lines, shown in Figure 5.4, is generated. This phase is further divided into the following three steps:

Step 1 - Abstracting Log Lines

each log line contains static texts, which describe the particular logging context, and dynamic contents, which reveal the SUT's runtime states. Logs generated by modern logging frameworks like Log4j [77] can be configured to contain information such as file name and line number. Hence, we can easily map the generated log lines to the corresponding logging statements. In our example, each log line contains the file name `Test.java` and the line number. In other cases, if the file name and the line number is not printed, we can leverage existing log abstraction techniques (e.g., [**JiangJSME08**, 236]), which automatically recognize the dynamically generated contents and map log lines into the corresponding logging statements.

Step 2 - Grouping Related Log Lines

each log line contains some execution contexts (e.g., thread or session or user IDs). In this step, we group the related log lines into sequences by leveraging these execution contexts. In our running example, we group the related log lines by their thread IDs. There are in total two log sequences in our running example, which correspond to `Thread-1` and `Thread-2`.

Step 3 - Forming Log Sequences

in this step, we replace the grouped log line sequences into sequences of logging statements. For example, the log line sequence of line `1,3,5,6` grouped under `Thread-1` becomes `Log@3, Log@51, Log@14, Log@14`.

5.3.3 Phase 3 - Path Analysis

Based on the obtained log line sequences from the previous phase, we intend to estimate the covered code paths in this phase using a four-step process.

Step 1 - Matching Log Sequences with LogREs

we match the sequences of logging statements obtained in Phase 2 with the LogREs obtained in Phase 1. The two recovered log sequences are matched with the two LogREs, which are shown on the third row in Figure 5.3. The sequence of logging statement in our running example `Log@3, Log@51, Log@14, Log@14` (a.k.a., *Seq 1*) will be matched with LogRE `(Log@3)(Log@51)(Log@14)+`.

Step 2 - Labeling Statements

in the second step, based on each of the matched LogRE, we apply three types of labels to the corresponding source code based on their estimated coverage: `Must`, `May`, and `Must-Not`. The lower part of Figure 5.3 shows the results for our working example. For *Seq 1*, we label lines `1,2,3,4,5,9,12,13,14,16,50,51,52` as `Must`, as these lines are definitely covered if the above log sequence is generated. Line 10 is marked as `May`, because we are uncertain if the condition of the `if` statement at line 9 is satisfied. Lines `6,7,17` are marked as `Must-Not`, because the branch choice is true at line 4 and false at line 17. Due to the page limit, we do not explain the source code labeling process for *Seq 2*.

Step 3 - Reconciling Statement-level Labels

as one line of source code may be assigned with multiple different labels from different log sequences, in the third step, we reconcile the labels obtained from different log sequences and assign one final resulting label to each line of source code. We use the following criteria for our assignment:

- **At least one `Must` label**: since a particular line of source code is considered as "covered", when it has been executed at least once. Hence, if there is at least one `Must` label assigned to that line of source code, regardless of other scenarios, it is considered as covered (a.k.a., assigning `Must` labels as the final resulting label). In our running example, line 5 is marked as `Must` in *Seq 1* and `Must-not` in *Seq 2*. Therefore, it will be marked as `Must` in the final label.

- **No `Must` labels, and at least one `May` label**: in this particular case, we may have a specific line of source code assigned with all `May` labels, or a mixture of `May` and `Must-not` labels. As there is a possibility that this particular line of source code can be covered by some test cases, we assigned it to be `May` in the final resulting label. In our running example, line 10 is marked `May`.

- **All `Must-not` labels**: in this particular case, since there are no existing test cases covering this line of source code, we assigned it to be `Must-not` in the final resulting label. In our running example, line 17 is marked `Must-not` in both *Seq 1* and *Seq 2*. It will be marked as `Must-not` in the final label.

Step 4 - Inferring Labels at the Method Levels

based on the line-level labels, we assign one final label to each method using the following criteria:

- for a particular method, if there is at least one line of source code labeled as `Must`, this method will be assigned with a `Must` label. In our running examples, both methods will be labeled as `Must`.
- for methods without `Must` labeled statements, we apply the following process:
  - **Initial Labeling**: all of the logging statements under such methods should already be labeled as `Must-not`, since none of these logging statements are executed. If there is at least one logging statement which is not under any control flow statement nodes in the call graph, this method will be labeled as `Must-not`.
  - **Callee Labeling**: starting from the initial set of the `Must-not` labeled methods, we search for methods that will only be called by these methods, and assign them with the `Must-not` labels. We iteratively repeat this process until no more methods can be added to the set.
  - **Remaining Labeling**: we assign the `May` labels to the remaining set of unlabeled methods.

Similarly, each branch will be assigned with one final resulting label based on the statement-level labels. Due to space constraints, we will not explain the process here.

Phase 4 - Code Coverage Estimation

In this phase, we estimate the method/branch/statement-level code coverage measures using the labels obtained from the previous phase. As logging statements are not instrumented everywhere, there are code regions labeled as `May`, which indicates uncertainty of coverage. Hence, when estimating the code coverage measures, we provide two values: a minimum and a maximum value for the above three coverage criteria. The minimum value of statement coverage is calculated as $\frac{\#\ of\ \texttt{Must}\ labels}{Total\#\ of\ labels}$, and the maximum value is calculated as $\frac{\#\ of\ \texttt{Must}\ labels + \#\ of\ \texttt{May}\ labels}{Total\#\ of\ labels}$. In our running example, the numbers of `Must`, `May`, and `Must-not` statements are 15, 1, and 1, respectively. Therefore, the range of the statement coverage is from 88% ($\frac{15}{15+1+1} \times 100\%$) to 94% ($\frac{15+1}{15+1+1} \times 100\%$).

Similarly, since the number of `Must`, `May`, and `Must-not` branches is 5, 1, and 2 in our running example, the range of branch coverage is from 62.5% ($\frac{5}{5+1+2} \times 100\%$) to 87.5% ($\frac{5+2}{5+1+2} \times 100\%$).

The number of `Must`, `May`, and `Must-not` methods are 2, 0, and 0. Hence, the method level coverage is 100%.

5.4 Case setup

To evaluate the effectiveness of our approach, we have selected five commercial projects from Baidu and one large-scale open-source project in our case study. Table 5.1 shows the general information about these projects in terms of their project name, project descriptions, and their sizes. All six projects are implemented in Java and their domains span widely from web services, to application platforms and NoSQL databases. The main reason why we picked commercial projects to study is that we can easily get hold of QA engineers for questions and feedback. The five commercial projects (C_1, C_2, C_3, C_4, and C_5) were carefully selected based on consultations with Baidu's QA engineers. We also picked one large-scale popular open source project, HBase [172], because we can freely discuss about the details. We focus on HBase version 1.2.6 in this study, since it is the most recent stable release by the time of the study. All six studied projects are actively maintained and being used by millions or hundreds of millions of users worldwide. We proposed the following two research questions (RQs), which will be discussed in the next two sections:

Table 5.1: Information about the six studied projects.

Projet	**Descriptions**	**LOC**
C_1	Internal API library	24K
C_2	Platform	80K
C_3	Cloud service	12K
C_4	Video streaming service	35K
C_5	Distributed file system	228K
HBase	Distributed NoSQL Database	453K

- **RQ1: (Accuracy)** *How accurate is LogCoCo compared to the state-of-the-art code coverage tools?* The goal of this RQ is to evaluate the quality of the code coverage measures derived from LogCoCo against the state-of-the-art code coverage tools. We intend to conduct this study using data from various testing activities.
- **RQ2: (Usefulness)** *Can we evaluate and improve the existing test suites by comparing the LogCoCo results derived from various execution contexts?* The goal of this RQ is to check if the existing test suites can be improved by comparing the estimated coverage measures using LogCoCo from various system execution contexts.

5.5 RQ1: Accuracy

On one hand, existing state-of-the-art code coverage tools (e.g., Jacoco [33], Cobertura [217]) collect the code coverage measures by excessively instrumenting the SUT either at the source code [218] or at the binary/bytecode levels [217, 219]. The excessive instrumentation (e.g., for every method entry/exit, and for every conditional and loop branch) ensures accurate measurements of code coverage, but imposes problems like deployment challenges and performance overhead (Section 5.2), which limit their application context. On the other hand, LogCoCo is easy to setup and imposes little performance overhead by analyzing the readily available execution logs. However, the estimated code coverage measures may be inaccurate or incomplete, as developers only selectively instrument certain parts of the source code by adding logging statements. Hence, in this RQ, we want to assess the quality of the estimated code coverage measures produced by LogCoCo.

5.5.1 Experiment

We ran nine test suites for the six studied projects as shown in Table 5.2. The nine test suites contained unit and integration tests. Since the unit test suites were not configured to generate logs for C_1, C_4, and C_5, we did not include them in our study. C_5 and HBase are distributed systems, so we conducted their integration tests in a field-like deployment setting.

Each test suite was run twice: once with the JaCoCo configured, and once without. We used the code coverage data from JaCoCo as our oracle and compared it against the estimated results from LogCoCo. For all the experiments, we collected data like generated log files and code coverage measures from JaCoCo. JaCoCo is a widely used state-of-the-art code coverage tool, which is used in both research [23, 232, 233] and practice [220, 221]. We picked JaCoCo to ensure that the experiments could be done in a field-like environment. Code coverage tools, which leverage source code-level instrumentation techniques, require recompilation and redeployment of the SUT. Such requirements would make the SUT's testing behavior no longer closely resemble the field behavior. JaCoCo, a bytecode instrumentation based code coverage tool, is less invasive and instruments the SUT during runtime. For each test, we gathered the JaCoCo results and the log files. Depending on the tests, the sizes of the log files range from 8 MB to 1.1 GB.

5.5.2 Data Analysis

We compared the three types of code coverage measures (method, statement, and branch coverage) derived from LogCoCo and JaCoCo. Since LogCoCo marks the source code for each type of coverage using the following three labels: `Must`, `May`, and `Must-not`, we calculated the percentage of correctly labeled entities for three types of labels.

For the `Must` labeled methods, we calculated the portion of methods which are marked as *covered* in the JaCoCo results. For example, if LogCoCo marked five methods as `Must` among which four were reported as *covered* in JaCoCo, the accuracy of the LogCoCo method-level

Table 5.2: Comparing the performance of LogCoCo against JaCoCo under various testing activities. The numbers above shows the amount of overlap between LogCoCo and JaCoCo.

Project	Type of Testing	Size of the Logs	Method Coverage			Statement Coverage			Branch Cov.		
			(Must	Must-not	May)	(Must	Must-not	May)	(Must	Must-not	May)
C_1	Integration	20 MB	100%	100%	16%	99%	100%	62%	100%	100%	67%
C_2	Unit	600 MB	100%	100%	78%	100%	100%	75%	100%	100%	76%
	Integration	550 MB	100%	100%	30%	100%	100%	90%	100%	100%	83%
C_3	Unit	8 MB	100%	88%	15%	84%	100%	77%	100%	100%	50%
	Integration	26 MB	100%	-	33%	83%	100%	89%	50%	100%	60%
C_4	Integration	29 MB	100%	99%	18%	97%	100%	49%	100%	100%	55%
C_5	Integration	1.1 GB	100%	90%	6%	97%	100%	45%	96%	100%	39%
HBase	Unit	527 MB	100%	83%	83%	99%	100%	83%	100%	100%	76%
	Integration	193 MB	100%	89%	50%	99%	100%	71%	100%	100%	63%

coverage measure would be $\frac{4}{5} \times 100\% = 80\%$. Similarly, for the `Must-not` labeled entities, we calculated the percentage of methods which were marked as *not covered* by JaCoCo.

For the `May` labeled methods, we calculated the portion of methods which are reported as *covered* by JaCoCo. Note that this calculation is ***not*** to assess the accuracy of the `May` covered methods, but to assess the actual amount of methods which are indeed covered during testing.

When calculating the accuracy of the statement and branch level coverage measures from LogCoCo, we only focused on code blocks from the `Must` covered methods. This is because all the statement and branch level coverage measures will be `May` or `Must-not` for the `May` or `Must-not` labeled methods, respectively. It would not be meaningful to evaluate these two cases again at the statement or branch level.

The evaluation results for the three coverage measures are shown in Table 5.2. If a cell is marked as "-", it means there is no source code assigned with the label. We will discuss the results in details below.

5.5.3 Discussion on Method-Level Coverage

As shown in Table 5.2, all methods labeled with `Must` are 100% accurate. It means that LogCoCo can achieve 100% accuracy when detecting covered methods. Rather than instrumenting all the methods like the existing code coverage tools do, LogCoCo uses program analysis techniques to infer the system execution contexts. For example, only 13% of the methods in C_1 have logs printed. The remaining 87% of the `Must` covered methods are inferred indirectly.

The methods labeled with `Must-not` are not always accurate. Three commercial projects (C_3, C_4, and C_5), and HBase have some methods which are actually covered in the tests but are falsely flagged as `Must-not` covered methods. Except for three cases, the accuracy of the `Must-not` labeled methods are all above 90%. We manually examined the misclassified instances and found the following two main reasons: (1) we have limited the size of our call graphs to 20 levels deep or a maximum of $100,000$ paths per AST tree due to memory constraints of our machine. Therefore, we missed some methods, which had deeper call chains. (2) Our current technique cannot handle recursive functions properly.

The amount of `May` covered methods that are actually covered is highly dependent on the type of projects and can range from 6% to 83%. In addition, this number seems to be irrelevant of the types of testing conducted. In order to obtain a more accurate estimate of the code coverage measures using LogCoCo, additional logging statements need to be added into the SUT to reduce the amount of `May` labeled methods. However, the logging locations should be decided strategically (e.g., leveraging techniques like [21, 2]) in order to minimize performance overhead.

5.5.4 Discussion on the Statement and Branch Coverage

For statement and branch coverage, the accuracy of the `Must-not` labels is 100% for all the experiments. However, the accuracy of the `Must` covered labels ranges from 83% to 100% for statement coverage and 50% to 100% for branch coverage. In seven out of the nine total experiments, the accuracy of the `Must` covered statements is 97% or higher. We manually examined the cases where the LogCoCo results are different from JaCoCo. We summarized them as follows:

1. *limitations on static analysis (LogCoCo issue)*: Java supports polymorphism. The actual type of certain objects are unknown until they are being executed. LogCoCo infers the call graphs statically and mistakenly flags some of the method invocations.

2. *new programming constructs (JaCoCo issue)*: the lambda expression is one of the new programming language constructs introduced in Java 8. JaCoCo mistakenly tags some statements containing lambda expressions as not covered.

The accuracy of the `Must` covered branches is generally above 95%, except one case: during the integration testing of C_3, LogCoCo detected two `Must` covered branches being executed, one of which was falsely labeled. The rationales for the differences of the branch coverage measures are the same as the statement coverage measures.

The amount of `May` actually covered statements and branches are generally higher than the amount of actually covered `May` methods. However, similar to the method-level coverage, we cannot easily guess the actual coverage information for a `May` labeled statement or branch.

5.5.5 Feedback from the QA Engineers

We demonstrated LogCoCo to the QA engineers at Baidu. They agreed that LogCoCo can be used for their daily testing activities, due to its ease of setup, wider application context, and accurate results. In particular, instead of treating all the source code equally, they would pay particular attention to the coverage of the methods, which have logging statements instrumented. This was because many of the logging statements were inserted into risky methods or methods which suffered from past field failures. Having test cases cover these methods is considered a higher priority. LogCoCo addressed this task nicely. In addition, they agreed that LogCoCo can also be used to speed up problem diagnosis in the field by automatically pin-pointing the problematic code regions. Finally, they were also very interested in the amount of `May` labeled entities (a.k.a., methods, statements, and branches), as they knew little about the runtime behavior of these entities. They considered reducing the amount of `May` labeled entities as one approach to improving their existing logging practices and were very interested to collaborate further with us on this topic.

Findings: The accuracy of `Must` and `Must-not` labeled entities from LogCoCo is very high for all three types of code coverage measures. However, one cannot easily infer whether a `May` labeled entity is actually covered in a test.
Implications: To further improve the accuracy, one must reduce the amount of `May` labeled entities through additional instrumentation. Researchers and practitioners can look into existing work (e.g., [21, 2]), which improve the SUT's logging behavior with minimal performance overhead.

5.6 RQ2: Usefulness

Existing code coverage tools are usually applied only during unit or integration testing due to various challenges explained in Section 5.2. LogCoCo, which analyzes the readily available execution logs, can work on a much wider application context. In this RQ, we intend to check if we can leverage the LogCoCo results from various execution contexts to improve the existing test suites. To tackle this problem, we further split this RQ into the following two sub-RQs:

RQ2.1: Can we improve the in-house functional test suites by the comparison among each other?

In this sub-RQ, we will focus on unit and integration testing, as they have different testing purposes. Unit testing examines the SUT's behavior with respect to the implementation of individual classes, whereas integration testing examines whether individual units can work correctly when they are connected to each other. We intend to check if one can leverage the coverage differences to improve the existing unit or integration test suites using data from LogCoCo.

Experiment

To study this sub-RQ, we reused the data obtained from RQ1's experiments. In particular, we selected the data from two commercial projects: C_2 and C_3, as they contain data from both unit and integration test suites. The main reason we focused on the commercial projects in this sub-RQ was because we can easily get hold of the QA engineers of Baidu for feedback or surveys (e.g., whether they can evaluate and improve the unit or integration tests by comparing the coverage data).

Data Analysis and Discussion

It would be impractical to study all the coverage differences from the two types of tests due to their large size. We randomly sampled a subset of methods, where both types of testing covered but their statement and branch level coverage measures differed. We presented this dataset to QA engineers from the two commercial projects for feedback. After manual

examinations, the QA engineers agreed to add additional unit testing cases for all the cases where unit testing did not cover. However, adding additional integration tests is harder than adding additional unit tests. The QA engineers rejected about 85% of the cases where unit testing covered but integration testing missed, as they were considered as hard or lower priority. We summarized their rationales as follows:

- *Defensive Programming*: defensive programming is a programming style to guard against unexpected conditions. Although it generally improves the robustness of the SUT, there can be unnecessary code introduced to guard against errors that would be impossible to happen. Developers insert much error checking code into the systems. Some of these issues are rare or impossible to happen. It is very hard to come up with an integration test case whose input values can exercise certain branches.
- *Low Risk Code*: some of the modules are considered as low risk based on the experience of the QA engineers. Since they are already covered by the unit test suites, adding additional integration test cases is considered as low priority.

RQ2.2: Can we evaluate the representativeness of in-house test suites by comparing them against field behavior?

One of the common concerns associated with QA engineers is whether the existing in-house test suites can properly represent field behavior. We intend to check if one can evaluate the quality of the existing in-house test suites by comparing them against field coverage using data from LogCoCo.

Experiment

Due to confidentiality reasons, we cannot disclose the details about the field behavior for the commercial projects. Therefore, we studied the open source system, HBase, for RQ2.2. The integration test suite from HBase is considered as a comprehensive test suite and is intended for "elaborate proofing of a release candidate beyond what unit tests can do" [237]. Hence, we consider it as HBase's in-house test suite. There are two setup approaches to running the HBase's integration tests: a mini-cluster, or a distributed cluster. The mini-cluster setup is usually run on one machine and can be integrated with the Maven build process. The distributed cluster setup needs a real HBase cluster setup and is invoked using a separated command. In this experiment, we ran under both setups and collected their logs.

The workloads defined in YCSB are derived by examining a wide range of workload characteristics from real web applications [230]. We thus used the YCSB benchmark test suite to mimic the field behavior of HBase. However, as we discovered under default settings, HBase did not output any logs during the benchmarking process. We followed the instructions from [238] to change the log levels for HBase from INFO to DEBUG on the fly (a.k.a., without server reboot). The resulting log file size is around 270 MB when running the YCSB benchmark tests for one hour.

Data Analysis and Discussion

Based on the LogCoCo results, there are 12 methods which were covered by the YCSB test and not by the integration test under the mini-cluster setup. Most of these methods were related to the functionalities associated with cluster setup and communications. Under the distributed cluster setup, which is more realistic, all the covered methods in the YSCB test were covered by the integration test.

The log verbosity level for the unit and the integration tests of HBase in RQ1 was kept as the default INFO level. Both tests generated hundreds of megabytes of logs. However, under the default verbosity level, the YCSB benchmarking test generated no logs except a few lines at the beginning of the test. This is mainly because HBase does not perform logging for their normal read, write, and scan operations for performance concerns. The integration tests output more INFO level logs is because: (1) many of the testing methods are instrumented with INFO or higher level logs. Such logs are not printed in practice; and (2) in addition to the functionalities covered in the YCSB benchmark, integration tests also verify other use cases, which can generate many logs. For example, one integration test case is about region replications. In this test, one of the HBase component, ZooKeeper, which is responsible for distributed configuration and naming service, generated many INFO level logs.

We further assessed the performance impact of turning on the DEBUG level logs for HBase. We compared the response time under the DEBUG and the INFO level logging with YCSB threads configured at 5, 10, and 15, respectively. The performance impact was very small ($< 1\%$) under all three YCSB settings (a.k.a., three different YCSB benchmark runs). Thus, for HBase the impact of DEBUG level logging is much smaller than JaCoCo. Furthermore, compared to JaCoCo, which requires server restart to enable/disable its process, the DEBUG level logging can easily be turned on/off during runtime.

Findings: LogCoCo results can be used to evaluate and improve the existing test suites. Multiple rationales are considered when adding a test case besides coverage.
Implications: There are mature techniques (e.g., EvoSuite [239] and Pex [240]) to automatically generate unit test cases with high coverage. However, there are no such techniques for other types of tests, which still require high manual effort to understand the context and to decide on a case-by-case basis. Further research is needed in this area.
The coverage information from LogCoCo highly depends on amount of generated logs. Researchers or practitioners should look into system monitoring techniques (e.g., sampling [50] or adaptive instrumentation [241]), which maximize the obtained information with minimal logging overhead.

5.7 Related work

In this section, we will discuss two areas of related research: (1) code coverage, and (2) software logging.

5.7.1 Code Coverage

Code coverage measures the amount of source code executed while running SUTs under various scenarios [206]. They have been used widely in both academia and industry to assess and improve the effectiveness of existing test suites [211, 205, 242, 243, 244, 245]. There are quite a few open source (e.g., [33, 217]) and commercial (e.g., [215, 216]) code coverage tools available. All these tools leverage additional code instrumentation, either at the source code level (e.g., [218, 231, 246]) or at the binary/bytecode (e.g., [216, 217, 219]) level, to automatically collect the runtime system behavior in order to measure the code coverage measures. In [224], Häubl et al. derived code coverage information from the profiling data recorded by an the just-in-time (JIT) compiler. They also compared their coverage information against JaCoCo. They showed their results are more accurate than JaCoCo and yield smaller overhead. Häubl et al.'s approach is different from ours, as they relied on data from the underlying virtual machines, whereas we focus on the logging statements from the SUT's source code. Recently, Horváth et al. [23] compared the results from various Java code coverage tools and assessed the impact of their differences to test prioritization and test suite reduction. In this chapter, we have evaluated the state-of-the-art Java-based code coverage tools in a field like setting and proposed a new approach, which leverages the readily available execution logs, to automatically estimating the code coverage measures.

In addition to the traditional code coverage metrics (e.g., method, branch, decision, and MC/DC coverage), new metrics have been proposed to better assess the oracle quality [210], to detect untested code regions [233], and to compose test cases with better abilities to detect faults [247]. There are various empirical studies conducted to examine the relationship between the test effectiveness and various code coverage metrics. For example, Inozemtseva and Holmes [207] leveraged mutation testing to evaluate the fault detection effectiveness of various code coverage measures and found that there is a low to moderate correlation between the two. Kochhar et al. [208, 232] performed a similar study, except they used real bugs instead. Their study reported a statistically significant correlation (moderate to strong) between fault detection and code coverage measures. Gligoric et al. [248] compared the effectiveness of various code coverage metrics in the context of test adequacy. The work conducted by Wang et al. [225], which is the closest to our work, compared the coverage between in-house test suites and field executions using invariant-based models. Our work differs from [225] in the following two main areas: (1) they used a record-replay tool, which instruments the SUT to collect coverage measures. Our work estimates the coverage measures based on the existing logs without extra instrumentation. (2) While they mainly focused on the client and desktop-based systems, our focus is on the server-based distributed systems deployed in a field-like environment processing large volumes of concurrent requests. In our context, extra instrumentation would not be ideal, as it will have a negative impact on the user experience.

5.7.2 Software Logging

Software logging is a cross-cutting concern that scatters across the entire system and intermixes with the feature code [44]. Unfortunately, recent empirical studies show that there are no well-established logging practices for commercial [8, 32] and open source systems [10, 9]. Recently researchers have focused on providing automated logging suggestions based on learning from past logging practices [8, 36, 107, 80] or program analysis [2]. Execution logs are widely available inside large-scale software systems for anomaly detection [249, 52], system monitoring [16, 3], problem debugging [250, 18, 35, 251], test analysis [19, 252], and business decision making [5]. Our work was inspired by [35], which leveraged logs to infer executed code paths for problem diagnosis. However, this is the first work, to the author's knowledge, which uses logs to automatically estimate code coverage measures.

The limitation is that we rely on that the study systems contain sufficient logging. It is generally the case among server-side projects. For client-side or other projects with limited logging, our approach should be complementary by other code coverage tools.

5.8 Threats to validity

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

5.8.1 Internal Validity

In this chapter, we proposed an automated approach to estimating code coverage by analyzing the readily available execution logs. The performance of our approach highly depends on the amount of the logging and the verbosity levels. The amount of logging is not a major issue as existing empirical studies show that software logging is pervasive in both open source [10, 9] and commercial [8, 32] systems. The logging overhead due to lower verbosity levels is small for the HBase study. For other systems, one can choose to enable lower verbosity level for a short period of time or use advanced logging techniques like sampling and adaptive instrumentation.

5.8.2 External Validity

In this chapter, we focused on the server-side systems mainly because these systems use logs extensively for a variety of tasks. All these systems are under active development and used by millions of users worldwide. To ensure our approach is generic, we studied both commercial and open source systems. Although our approach was evaluated on Java systems, to support other programming languages, we just need to replace the parser for another language (e.g., Saturn [253] for C, and AST [254] for Python) in LogCoCo. The remaining process stays the same. Our findings in the case studies may not be generalizable to systems and tools which have no or very few logging statements (sometimes seen in mobile applications and client/desktop-based systems).

5.8.3 Construct Validity

When comparing the results between LogCoCo and the state-of-the-art code coverage tools, we focused on JaCoCo, which collects code coverage information via bytecode instrumentation. This is because: (1) JaCoCo is widely used in Baidu, so we can easily collect the code coverage for the systems and gather feedback from the QA engineers; and (2) we intend to assess the coverage measures in a field-like environment, in which the system is deployed in a distributed environment and used by millions of users. Source-code-instrumentation-based code coverage tools (e.g., [231]) are not ideal, as they require recompilation and redeployment of the SUT.

5.9 Conclusions and Future work

Existing code coverage tools suffer from various problems, which limit their application context. To overcome with these problems, this chapter presents a novel approach, LogCoCo, which automatically estimates the code coverage measures by using the readily available execution logs. We have evaluated LogCoCo on a variety of testing activities conducted on open source and commercial systems. Our results show that LogCoCo yields high accuracy and can be used to evaluate and improve existing test suites.

In the future, we plan to extend LogCoCo for other programming languages. In particular, we are interested in applying LogCoCo to systems implemented in multiple programming languages. Furthermore, we also intend to extend LogCoCo to support other coverage criteria (e.g., data-flow coverage and concurrency coverage). Finally, since the quality of the LogCoCo results highly depends on the quality of logging, we will research into cost-effective techniques to improve the existing logging code.

Chapter 6

Conclusions and Future work

Although software logging plays a key role in DevOps, there are various challenges associated with it. This thesis is the first step towards tackling the challenges and improving existing software logging practices through systematic analysis of software repositories. We derive guidelines for developing high quality logging code and empirically evaluate various approaches associated with logging code management. We analyze logs to provide quality feedback to the developers and perform deeper problem analysis. The resulting findings and techniques are beneficial for both software developers and IT operators. This chapter is organized as follows: first, we summarize our findings and contributions based on the presented work in this thesis (Section 6.1). Then, we discuss some future work (Section 6.2). At last, we provide our closing remarks (Section 6.3).

6.1 Thesis Findings and Contributions

- **A Survey of the Instrumentation Techniques Used in Software Logging**

  We have summarized log instrumentation into three steps: (1) logging approach; (2) logging utility integration; and (3) logging code composition. For each step, we have surveyed related papers and provided detailed analysis and comparison. The key findings are summarized as follows:

  - *Logging Approach.* There are three logging approaches: conventional logging, rule-based logging, and distributed tracing. Conventional logging is easy to setup and LC snippets can be instrumented in anywhere of the system. Rule-based logging is proposed to remediate the cross-cutting concern. Distributed tracing enriches the context information in logs.
  - *LU Integration.* There are many LUs available in the wild. Tools and support are needed to control the behavior of different LUs. Issues are found within the configuration process for softare logging.

  - *LC Composition.* Many techniques are proposed to solve three concerns: where-to-log, what-to-log, and how-to-log. Examples are machine learning-based, history-based, and program analysis-based techniques.

- **Extracting and Studying the Logging-Code-Issue-Introducing Changes in Java-based Large-Scale Open Source Software Systems**

  Ineffective logging would cause many issues for large-scale software systems, such as confusion, lack of information during program diangosis, or even system crash. However, developing and maintaining high quality logging code is very challenging, as logging code usually inter-mixes with feature code and its correctness is hard to verify. Unfortunately, there are no existing guidelines in research and practice on how to conduct effective logging. We propose to derive such guidelines through mining the historical logging code changes. In particular, we have extracted a dataset containing Logging-Code-Issue-Introducing changes from six Java-based open source software systems. Through this dataset, we have conducted several preliminary studies. We believe this dataset is very valuable and can be used by many future research works. The key findings are summarized as follows:

  - *Characteristics of fixes to LCII changes.* Both co-changed and independently changed logging code changes can contain fixes to the LCII changes. Ten intentions are found behind the fixes to LCII changes. "Clarification" and "Updating Logging Style" are the top two intentions.
  - *Complexity of fixes to LCII changes.* The two types of logging code changes (LCII changes and co-evolving logging code changes) are similar in terms of change complexity.
  - *Resolution time of fixes to LCII changes.* The resolution time of the LCII changes and regular bugs are statistically different in all studied projects
  - *Effectiveness of current techniques.* Both the LCAnalyzer and Cloning technique can only detect a small fraction($< 3\%$) of the issues in logging code.

- **Studying the Use of Java Logging Utilities in the Wild**

  Instead of directly invoking the standard output functions, developers usually prefer to use logging utilities (LUs) (e.g., SLF4J [46]), which provide additional functionalities like thread-safety and verbosity level support, to instrument their source code. However, very few studies discuss the use of LUs, although new LUs are constantly being introduced. We have conducted the first large-scale empirical study on the use of Java LUs in the wild and identified 3,856 LUs being used in the wild. In particular, there are two types of LUs: External LUs (ELUs) and Internal LUs(ILUs). The key findings are summarized as follows:

  - *ELU.* ELUs are mainly used for: (1) General-purpose logging, (2) LU interactions, (3) Internationalization, and (4) Modularization.

  - *ILU.* ILUs are mainly implemented for: (1) defining the logging format, (2) compatibility with other LUs, and (3) ease of configuration and dependency management.
  - *Multiple LUs.* As the project size becomes larger, developers tend to use multiple LUs to fulfil their needs. The common four usage context are: (1) interaction with LUs from the imported packages; (2) managing the logging contents; (3) formatting log- ging messages across different components; and (4) developer convenience.

- **An Automated Approach to Estimating Code Coverage Measures via Execution Logs**

  Code coverage measures are commonly used to evaluate and improve the existing test suites. However, existing code coverage tools suffer from various problems, which limit their application context. To overcome with these problems, we present a novel approach, LogCoCo, which automatically estimates the code coverage measures by using the readily available execution logs. We have evaluated LogCoCo on a variety of testing activities conducted on open source and commercial systems. The key findings are summarized as follows:

  - *Accuracy.* The accuracy of Must and Must-not labeled entities from LogCoCo is very high. However, one cannot easily infer whether a May labeled entity is actually covered in a test.
  - *Usefulness.* LogCoCo results can be used to evaluate and improve the existing test suites. Multiple rationales are considered when adding a test case besides coverage.

6.2 Future Work

We propose the future work as follows:

- Deriving the best practice of adopting logging approaches.

  There are no well documented guidelines on suggesting the appropriate logging approaches under specific scenarios. It is worthwhile to extract and generalize all the logging needs by studying the existing commerical and open source projects. Different logging approaches can be evaluated under these different logging needs in order to provide a systematic guideline on the best practices of using different logging approaches. By mining historical information, we can study the practices of two other logging approaches: rule-based logging and distributed tracing from various dimensions such as instrumentation efforts, usage context and common issues.

- Recommending the appropriate LUs based on different needs.

  There are many LUs available in the wild, which provides various functionalities about software logging. In addition, new LUs are also constantly introduced. It is very

important to provide developers' suggestions on the appropriate LU(s) for the SUS in order to ensure all the logging needs are satisfied. Furthermore, as the SUS evolve over time, more suggestions are also needed on incorporating additional LUs or LU migrations. We propose to mine the development history of popular LUs and study what are the features or issues the modifications attemp to implement or resolve. We can then manually explore the rationle of selecting LUs in popular open source projects by mining the communication repositories such as issue reports, pull requests, and mailing lists.

- Benchmarking existing techniques on improving logging practices.

  Existing state-of-the-art techniques on detecting logging code issues cannot detect a majority of the issues in logging code. In the future, we plan to further leverage our benchmark dataset to: (1) evaluate more techniques such as machine learning-based approaches to suggesting logging code modification; (2) develop better techniques to automatically detect issues in logging code, and (3) derive best practices in terms of developing and maintaining high quality logging code.

- Correlating Telemetry Data to Aid Problem Diagnosis.

  Heterogeneous and complex telemetry data. Besides execution logs, large scale distributed systems also adopt other mechanisms to monitor the health of systems. Some examples are distributed tracing and Application Performance Monitoring (APM) tools. Existing problem diagnosis techniques only focus on one type of telemetry data (e.g., logs or traces). Very few work try to enrich the analysis by correlating the information among different types of telemetry data. For example, microservice architecture-based projects become increasingly popular. These projects are usually deployed with one or more distributed tracing utilities. It remains unknown what the challenges are on analyzing the execution logs compared with traditional monolithic software systems.

6.3 Closing Remarks

DevOps is a software development methodology that intends to automate the process between software development and IT operations. Software logging plays an essential role in both sides of DevOps to ensure software quality. On the development side, developers need to develop and maintain high quality logging code. On the IT operation side, operators need to leverage execution logs to conduct different analysis tasks. In this thesis, we provide a benchmark dataset to aid software engineering researchers to derive the best logging practices and develop automated techniques to flag issues in logging code. We conduct an empirical study on the use of logging utilities in the wild. We also expand and enrich log analysis techniques for wider application context. We hope that this thesis be useful for both software enineering researchers and practitioners.

Bibliography

[1] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. 1st. Addison-Wesley Professional, 2015. ISBN: 0134049845, 9780134049847.

[2] Xu Zhao et al. "Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold". In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 2017.

[3] Adam Oliner, Archana Ganapathi, and Wei Xu. "Advances and Challenges in Log Analysis". In: *Communications of ACM* (2012).

[4] Ahmed E. Hassan et al. "An Industrial Case Study of Customizing Operational Profiles Using Log Compression". In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. 2008.

[5] Titus Barik et al. "The bones of the system: a case study of logging and telemetry at Microsoft". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 2016.

[6] Weiyi Shang et al. "Understanding Log Lines Using Development Knowledge". In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2014.

[7] *HBASE-750: NPE caused by StoreFileScanner.updateReaders*. `https://issues.apache.org/jira/browse/HBASE-750/`. Last accessed: 05/31/2020.

[8] Qiang Fu et al. "Where do developers log? an empirical study on logging practices in industry". In: *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*. 2014.

[9] Boyuan Chen and Zhen Ming (Jack) Jiang. "Characterizing logging practices in Java-based open source software projects - a replication study in Apache Software Foundation". In: *Empirical Software Engineering* (2017).

[10] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. "Characterizing logging practices in open-source software". In: *34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland, June 2-9, 2012*. 2012.

[11] *The AspectJ Project*. `https://eclipse.org/aspectj/`. Last accessed: 12/20/2019.

[12] Bram Adams and Shane McIntosh. "Modern Release Engineering in a Nutshell – Why Researchers Should Care". In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. 2016.

[13] *The replication package of chapter 2*. `https://www.eecs.yorku.ca/~chenfsd/resources/survey.zip`. Last accessed: 05/21/2020.

[14] *The replication package of chapter 3*. `http://www.cse.yorku.ca/~zmjiang/share/replication_package/emse2018_chen/replication_package.zip`. Last accessed: 10/08/2020.

[15] *The replication package of chapter 4*. `https://www.eecs.yorku.ca/~chenfsd/resources/icse2020_replication.zip`. Last accessed: 01/26/2020.

[16] Weiyi Shang et al. "An exploratory study of the evolution of communicated information about the execution of large software systems". In: *Journal of Software: Evolution and Process* (2014).

[17] Ding Yuan et al. "SherLog: Error Diagnosis by Connecting Clues from Run-time Logs". In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (2010).

[18] Ding Yuan et al. "Be Conservative: Enhancing Failure Diagnosis with Proactive Logging". In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012.

[19] Zhen Ming (Jack) Jiang et al. "Automated Performance Analysis of Load Tests". In: *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*. 2009.

[20] Raja R. Sambasivan et al. "Diagnosing Performance Changes by Comparing Request Flows". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. 2011.

[21] Rui Ding et al. "Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis". In: *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*. 2015.

[22] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. "Pivot tracing: dynamic causal monitoring for distributed systems". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 2015.

[23] Ferenc Horváth et al. "Code coverage differences of Java bytecode and source code instrumentation tools". In: *Software Quality Journal* (2017).

[24] Boyuan Chen et al. "An automated approach to estimating code coverage measures via execution logs". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 2018.

[25] *Summary of Sarbanes-Oxley Act of 2002*. `http://www.soxlaw.com/`. Last accessed 08/26/2015.

[26] Di Ma and Gene Tsudik. "A new approach to secure logging". In: *TOS* (2009).

[27] Daniel Le Métayer, Eduardo Mazza, and Marie-Laure Potet. "Designing Log Architectures for Legal Evidence". In: *8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, Pisa, Italy, 13-18 September 2010*. 2010.

[28] Jimmy J. Lin and Dmitriy V. Ryaboy. "Scaling big data mining infrastructure: the twitter experience". In: *SIGKDD Explorations* (2012).

[29] Anton Chuvakin, Kevin Schmidt, and Chris Phillips. *Logging and Log Management*. Syngress, 2013.

[30] *logstash - open source log management*. `http://logstash.net/`. Last accessed 04/04/2020.

[31] *Splunk*. `http://www.splunk.com/`. Last accessed 04/04/2020.

[32] Antonio Pecchia et al. "Industry Practices and Event Logging: Assessment of a Critical Software Development Process". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. 2015.

[33] JaCoCo. *JaCoCo Java Code Coverage library*. `http://www.eclemma.org/jacoco/`. Last accessed: 01/29/2018. 2018.

[34] JBoss Logging. `https://developer.jboss.org/wiki/JBossLoggingTooling`. Last accessed: 05/08/2020. 2019.

[35] Ding Yuan et al. "Improving software diagnosability via log enhancement". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. 2011.

[36] Boyuan Chen and Zhen Ming (Jack) Jiang. "Characterizing and detecting anti-patterns in the logging code". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017.

[37] Suhas Kabinna et al. "Examining the Stability of Logging Statements". In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. 2016.

[38] Zhenhao Li et al. "Dlfinder: characterizing and detecting duplicate logging code smells". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019.

[39] Rajkumar Buyya et al. "A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade". In: *ACM Comput. Surv.* (2019).

[40] Xiang Zhou et al. "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study". In: *IEEE Transactions on Software Engineering* (2018).

[41] Cindy Sridharan. *Distributed Systems Observability*. O'Reilly Media, Inc., 2018.

[42] Barbara A. Kitchenham, David Budgen, and O. Pearl Brereton. "Using mapping studies as the basis for further research - A participant-observer case study". In: *Inf. Softw. Technol.* (2011).

[43] Barbara Kitchenham and Stuart Charters. "Guidelines for performing Systematic Literature Reviews in Software Engineering". In: (2007).

[44] Gregor Kiczales et al. "Aspect-oriented programming". In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*. 1997.

[45] Boyuan Chen and Zhen Ming (Jack) Jiang. "Studying the Use of Java Logging Utilities in the Wild". In: *Proceedings of the 42nd International Conference on Software Engineering: Companion Proceedings, ICSE 2020, Seoul, South Korea, May 23-29, 2019*. 2020.

[46] *Simple Logging Facade for Java (SLF4J)*. `https://www.slf4j.org/`. Last accessed: 07/23/2019.

[47] *spdlog - Very fast, header-only/compiled, C++ logging library.* `https://github.com/gabime/spdlog`. Last accessed: 07/23/2019.

[48] Heng Li, Weiyi Shang, and Ahmed E. Hassan. "Which log level should developers choose for a new logging statement?" In: *Empirical Software Engineering* (2017).

[49] *The JavaDoc of Log4J 2*. `https://logging.apache.org/log4j/2.x/log4j-api/apidocs/org/apache/logging/log4j/Level.html`. Last accessed: 04/03/2020.

[50] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: `https://research.google.com/archive/papers/dapper-2010-1.pdf`.

[51] Jonathan Kaldor et al. "Canopy: An End-to-End Performance Tracing And Analysis System". In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 2017.

[52] Weiyi Shang et al. "Assisting developers of big data analytics applications when deploying on hadoop clouds". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013.

[53] Wei Xu. "System Problem Detection by Mining Console Logs". PhD thesis. University of California, Berkeley, USA, 2010. URL: `http://www.escholarship.org/uc/item/6jx4w194`.

[54] Zhen Ming (Jack) Jiang et al. "An automated approach for abstracting execution logs to execution events". In: *Journal of Software Maintenance* (2008).

[55] Pinjia He et al. "An Evaluation Study on Log Parsing and Its Use in Log Mining". In: *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. 2016.

[56] Mohamad Gebai and Michel R. Dagenais. "Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead". In: *ACM Comput. Surv.* (2018).

[57] Chuanxiong Guo et al. "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. 2015.

[58] Mohammed Sayagh et al. "Software Configuration Engineering in Practice: Interviews, Survey, and Systematic Literature Review". In: *IEEE Transactions on Software Engineering* (2018).

[59] Michael James Katchabaw et al. "Making Distributed Applications Manageable Through Instrumentation". In: *International Symposium on Software Engineering for Parallel and Distributed Systems, PDSE 1997, Boston, MA, USA, May 17-18, 1997*. 1997.

[60] Guoping Rong et al. "A Systematic Review of Logging Practice in Software Engineering". In: *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*. 2017.

[61] Raja R. Sambasivan et al. "Principled workflow-centric tracing of distributed systems". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. 2016.

[62] Jeanderson Candido, MaurÃcio Aniche, and Arie van Deursen. *Contemporary Software Monitoring: A Systematic Literature Review*. 2019. arXiv: `1912.05878 [cs.SE]`.

[63] *LOG4J 2 Apache Log4j 2*. `http://logging.apache.org/log4j/2.x`. Last accessed: 07/23/2019.

[64] Jieming Zhu et al. "Tools and benchmarks for automated log parsing". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019.

[65] Lionel C. Briand, Wojciech J. Dzidek, and Yvan Labiche. "Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging". In: *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. 2005.

[66] Marc Bartsch and Rachel Harrison. "An exploratory study of the effect of aspect-oriented programming on maintainability". In: *Software Quality Journal* (2008).

[67] Kimmo Kiviluoma, Johannes Koskinen, and Tommi Mikkonen. "Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*. 2006.

[68] Xu Zhao et al. "lprof: A Non-intrusive Request Flow Profiler for Distributed Systems". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 2014.

[69] Freddy Munoz et al. "Usage and Testability of AOP: An Empirical Study of AspectJ". In: *Information and Software Technology (IST)* (2013).

[70] Sven Apel et al. "How AspectJ is used: An analysis of eleven Aspectj programs". In: *Journal of Object Technology (JOT)* (2008).

[71] Opentracing: vendor-neutral APIs and instrumentation for distributed tracing. `https://opentracing.io/`. Last accessed: 07/23/2019. 2019.

[72] Stephen Yang, Seo Jin Park, and John K. Ousterhout. "NanoLog: A Nanosecond Scale Logging System". In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.* 2018.

[73] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. "A Logging Approach for Effective Dependability Evaluation of Complex Systems". In: *2009 Second International Conference on Dependability*. 2009.

[74] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. "Event Logs for the Analysis of Software Failures: A Rule-Based Approach". In: *IEEE Trans. Software Eng.* (2013).

[75] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. "Isolating Idiomatic Crosscutting Concerns". In: *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. 2005.

[76] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. "Correct Audit Logging: Theory and Practice". In: *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016.

[77] Log4J. *A logging library for Java*. `http://logging.apache.org/log4j/1.2`. Last accessed: 04/04/2020. 2020.

[78] Apache Commons Logging. `https://commons.apache.org/proper/commons-logging/`. Last accessed: 04/04/2020. 2020.

[79] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. "Large-scale, AST-based API-usage analysis of open-source Java projects". In: *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*. 2011.

[80] Suhas Kabinna et al. "Logging library migrations: a case study for the apache software foundation projects". In: *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. 2016.

[81] Chen Zhi et al. "An Exploratory Study of Logging Configuration Practice in Java". In: *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. 2019.

[82] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd., 2019.

[83] Yi Zeng et al. "Studying the characteristics of logging practices in mobile apps: a case study on F-Droid". In: *Empirical Software Engineering* (2019).

[84] Zhen Ming (Jack) Jiang et al. "Automatic identification of load testing problems". In: *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*. 2008.

[85] Boyuan Chen. "Improving the software logging practices in DevOps". In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019.

[86] Marcello Cinque et al. "Assessing and improving the effectiveness of logs for the analysis of software faults". In: *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*. 2010.

[87] Olivier Crameri, Ricardo Bianchini, and Willy Zwaenepoel. "Striking a new balance between program instrumentation and debugging time". In: *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*. 2011.

[88] Xu Zhao et al. "The Game of Twenty Questions: Do You Know Where to Log?" In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*. 2017.

[89] Kundi Yao et al. "Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*. 2018.

[90] Tong Jia et al. "Machine Deserves Better Logging: A Log Enhancement Approach for Automatic Fault Diagnosis". In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Memphis, TN, USA, October 15-18, 2018*. 2018.

[91] Joseph L. Hellerstein et al. "ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems". In: *Proceedings of the 19th International Conference on Distributed Computing Systems, Austin, TX, USA, May 31 - June 4, 1999*. 1999.

[92] Mike Y. Chen et al. "Pinpoint: Problem Determination in Large, Dynamic Internet Services". In: *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*. 2002.

[93] Paul Barham et al. "Magpie: Online Modelling and Performance-aware Systems". In: *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, May 18-21, 2003, Lihue (Kauai), Hawaii, USA*. 2003.

[94] Paul Barham et al. "Using Magpie for Request Extraction and Workload Modelling". In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. 2004.

[95] Mike Y. Chen et al. "Path-Based Failure and Evolution Management". In: *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*. 2004.

[96] Patrick Reynolds et al. "Pip: Detecting the Unexpected in Distributed Systems". In: *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings*. 2006.

[97] Eno Thereska et al. "Stardust: tracking activity in a distributed storage system". In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2006, Saint Malo, France, June 26-30, 2006*. 2006.

[98] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. "Whodunit: transactional profiling for multi-tier applications". In: *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*. 2007.

[99] Rodrigo Fonseca et al. "X-Trace: A Pervasive Network Tracing Framework". In: *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*. 2007.

[100] Rodrigo Fonseca, Michael J. Freedman, and George Porter. "Experiences with Tracing Causality in Networked Services". In: *2010 Internet Network Management Workshop / Workshop on Research on Enterprise Networking, San Jose, CA, USA, April, 2010*. 2010.

[101] Rodrigo Fonseca et al. "Quanto: Tracking Energy in Networked Embedded Systems". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 2008.

[102] Jingwen Zhou et al. "MTracer: A Trace-Oriented Monitoring Framework for Medium-Scale Distributed Systems". In: *8th IEEE International Symposium on Service Oriented System Engineering, SOSE 2014, Oxford, United Kingdom, April 7-11, 2014*. 2014.

[103] Jhonny Mertz and Ingrid Nunes. "On the practical feasibility of software monitoring: a framework for low-impact execution tracing". In: *Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019.

[104] Pinjia He et al. "Characterizing the natural language descriptions in software logging statements". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 2018.

[105] Ariel Rabkin et al. "A Graphical Representation for Identifier Structure in Logs". In: *Workshop on Managing Systems via Log Analysis and Machine Learning Techniques, SLAML'10, Vancouver, BC, Canada, October 3, 2010*. 2010.

[106] Mehran Hassani et al. "Studying and detecting log-related issues". In: *Empirical Software Engineering* (2018).

[107] Jieming Zhu et al. "Learning to Log: Helping Developers Make Informed Logging Decisions". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 2015.

[108] Sangeeta Lal and Ashish Sureka. "LogOpt: Static Feature Extraction from Source Code for Automated Catch Block Logging Prediction". In: *Proceedings of the 9th India Software Engineering Conference, Goa, India, February 18-20, 2016*. 2016.

[109] Sangeeta Lal, Neetu Sardana, and Ashish Sureka. "LogOptPlus: Learning to Optimize Logging in Catch and If Programming Constructs". In: *40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016*. 2016.

[110] Heng Li et al. "Studying software logging using topic models". In: *Empirical Software Engineering* (2018).

[111] Heng Li et al. "Towards just-in-time suggestions for log changes". In: *Empirical Software Engineering* (2017).

[112] Zhongxin Liu et al. "Which Variables Should I Log?" In: *IEEE Transactions on Software Engineering* (2019).

[113] Tae-young Kim et al. "An Automatic Approach to Validating Log Levels in Java". In: *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018*. 2018.

[114] Shanshan Li et al. "Guiding log revisions by learning from software evolution history". In: *Empirical Software Engineering* (2019).

[115] Jason King, Rahul Pandita, and Laurie A. Williams. "Enabling forensics by proposing heuristics to identify mandatory log events". In: *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, HotSoS 2015, Urbana, IL, USA, April 21-22, 2015*. 2015.

[116] Jason King et al. "To log, or not to log: using heuristics to identify mandatory log events - a controlled experiment". In: *Empirical Software Engineering* (2017).

[117] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[118] Nicholas Nethercote. "Dynamic Binary Analysis and Instrumentation". In: *PhD thesis, University of Cambridge, United Kingdom, November, 2004*. 2004.

[119] Zhouyang Jia et al. "SMARTLOG: Place error log statement by deep understanding of log intention". In: *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. 2018.

[120] Jie M. Zhang et al. "Machine Learning Testing: Survey, Landscapes and Horizons". In: *IEEE Transactions on Software Engineering* (2020).

[121] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[122] Joakim Kävrestad. *Fundamentals of Digital Forensics - Theory, Methods, and Real-Life Applications*. Springer, 2018.

[123] Boyuan Chen and Zhen Ming (Jack) Jiang. "Extracting and studying the Logging-Code-Issue- Introducing changes in Java-based large-scale open source software systems". In: *Empirical Software Engineering* (2019).

[124] *HDFS-5800: Typo: soft-limit for hard-limit in DFSClient*. `https://issues.apache.org/jira/browse/HDFS-5800`. Last accessed: 02/14/2018.

[125] *HBASE-10470: Import generates huge log file while importing large amounts of data*. `https://issues.apache.org/jira/browse/HBASE-10470`. Last accessed: 01/24/2018.

[126] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[127] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[128] Sunghun Kim et al. "Automatic Identification of Bug-Introducing Changes". In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. 2006.

[129] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When Do Changes Induce Fixes?" In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. 2005.

[130] Beat Fluri et al. "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction". In: *Software Engineering, IEEE Transactions on* (2007).

[131] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. "Studying the relationship between logging characteristics and the code quality of platform software". In: *Empirical Software Engineering (EMSE)* (2015).

[132] Christian Bird et al. "Fair and Balanced?: Bias in Bug-fix Datasets". In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. 2009.

[133] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. "Open Source Software Peer Review Practices: A Case Study of the Apache Server". In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. 2008.

[134] *HADOOP-12666: Support Microsoft Azure Data Lake - as a file system in Hadoop*. `https://issues.apache.org/jira/browse/HADOOP-12666`. Last accessed: 02/06/2018.

[135] *HADOOP-7358: Improve log levels when exceptions caught in RPC handler*. `https://issues.apache.org/jira/browse/HADOOP-7358`. Last accessed: 02/14/2018.

[136] Steven Davies, Marc Roper, and Murray Wood. "Comparing text-based and dependence-based approaches for determining the origins of bugs". In: *Journal of Software: Evolution and Process* (2014).

[137] Daniel Alencar da Costa et al. "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes". In: *IEEE Transactions on Software Engineering* (2017).

[138] *HDFS-4122: Cleanup HDFS logs and reduce the size of logged messages*. `https://issues.apache.org/jira/browse/HDFS-4122`. Last accessed: 02/07/2018.

[139] *HBASE-8754: Log the client IP/port of the balancer invoker*. `https://issues.apache.org/jira/browse/HBASE-8754`. Last accessed: 02/14/2018.

[140] *HDFS-1073: Simpler model for Namenode's fs Image and edit Logs*. `https://issues.apache.org/jira/browse/HDFS-1073`. Last accessed: 02/14/2018.

[141] *HADOOP-8347: Hadoop Common logs misspell 'successful'*. `https://issues.apache.org/jira/browse/HADOOP-8347`. Last accessed: 02/14/2018.

[142] *HBASE-12539: HFileLinkCleaner logs are uselessly noisy*. `https://issues.apache.org/jira/browse/HBASE-12539`. Last accessed: 02/14/2018.

[143] *HHH-6732: Some logging trace statements are missing guards against unneeded string creation*. `https://hibernate.atlassian.net/browse/HHH-6732`. Last accessed: 02/14/2018.

[144] *PR5906: Split all log messages into separate module project codes*. `https://github.com/wildfly/wildfly/pull/5906`. Last accessed: 02/14/2018.

[145] *HDFS-11448: JN log segment syncing should support HA upgrade*. `https://issues.apache.org/jira/browse/HDFS-11448`. Last accessed: 02/14/2018.

[146] Peter Kampstra. "Beanplot: A Boxplot Alternative for Visual Comparison of Distributions". In: *Journal of Statistical Software, Code Snippets* (2008).

[147] Jeanine Romano et al. "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?" In: *Annual meeting of the Florida Association of Institutional Research*. 2006.

[148] *Replication Package for the LCAnalyzer work.* `http://www.cse.yorku.ca/~zmjiang/share/replication_package/icse2017_chen/LCAnalyzer.zip`. Last accessed: 02/14/2018.

[149] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. "Predicting Defects for Eclipse". In: *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. 2007.

[150] Chadd Williams and Jaime Spacco. "SZZ Revisited: Verifying when Changes Induce Fixes". In: *Proceedings of the 2008 Workshop on Defects in Large Software Systems*. 2008.

[151] Chadd Williams and Jaime Spacco. "Branching and Merging in the Repository". In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. 2008.

[152] Naouel Moha et al. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Transactions on Software Engineering (TSE)* (2010).

[153] Fabio Palomba et al. "Mining Version Histories for Detecting Code Smells". In: *IEEE Transactions on Software Engineering (TSE)* (2015).

[154] Hitesh Sajnani et al. "SourcererCC: Scaling Code Clone Detection to Big-code". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 2016.

[155] *Google Flogger: A Fluent Logging API for Java.* `https://github.com/google/flogger`. Last accessed: 07/23/2019.

[156] Mark Marron. "Log++ logging for a cloud-native world". In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018, Boston, MA, USA, November 6, 2018*. 2018.

[157] Spring Boot pull request 4341. `https://github.com/spring-projects/spring-boot/issues/4341`. Last accessed: 08/21/2019. 2019.

[158] Logging dependency conflicts. `https://cloud.tencent.com/developer/ask/121135`. Last accessed: 08/21/2019. 2019.

[159] StackOverflow: Disable Logback in SpringBoot. `https://stackoverflow.com/questions/23984009/disable-logback-in-springboot/23991715`. Last accessed: 08/01/2019. 2019.

[160] Multiple logging implementations found in Spring Boot. `https://stackoverflow.com/questions/52911393/multiple-logging-implementations-found-in-spring-boot`. Last accessed: 08/21/2019. 2019.

[161] *How to enable logging in Jetty.* `https://stackoverflow.com/questions/25786592/how-to-enable-logging-in-jettyt`. Last accessed: 08/21/2019.

[162] *How to enable logging in dubbo?* `https://blog.csdn.net/JDream314/article/details/44620767`. Last accessed: 08/21/2019.

[163] What's Up with Logging in Java? `https://stackoverflow.com/questions/354837/whats-up-with-logging-in-java`. Last accessed: 07/23/2019. 2019.

[164] TIOBE Index for August 2019. `https://www.tiobe.com/tiobe-index/`. Last accessed: 07/23/2019. 2019.

[165] Omni-Notes: note-taking application for Android. `https://github.com/federicoiosue/Omni-Notes`. Last accessed: 08/21/2019. 2019.

[166] Telegram - a cloud-based instant messaging service. `https://github.com/DrKLO/Telegram`. Last accessed: 08/21/2019. 2019.

[167] Eclipse Java IDE. `https://www.eclipse.org/ide/`. Last accessed: 08/22/2019. 2019.

[168] IntelliJ IDEA. `https://www.jetbrains.com/idea/`. Last accessed: 08/22/2019. 2019.

[169] Apache Tomcat. `http://tomcat.apache.org/`. Last accessed: 08/22/2019. 2019.

[170] Red Hat Wildfly. `https://wildfly.org/`. Last accessed: 08/22/2019. 2019.

[171] Apache Hadoop. `https://hadoop.apache.org/`. Last accessed: 08/22/2019. 2019.

[172] HBase. `https://hbase.apache.org`. Last accessed: 01/29/2018. 2018.

[173] Steve Easterbrook et al. *Selecting Empirical Methods for Software Engineering Research*. 2008.

[174] GitHub features. `https://github.com/features`. Last accessed: 07/29/2019. 2019.

[175] Georgios Gousios. "The GHTorrent dataset and tool suite". In: *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), 2013*. 2013.

[176] Eirini Kalliamvakou et al. "The Promises and Perils of Mining GitHub". In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), 2014*. 2014.

[177] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. "A Large Scale Study of Multiple Programming Languages and Code Quality". In: *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016*. 2016.

[178] Baishakhi Ray et al. "A Large-scale Study of Programming Languages and Code Quality in GitHub". In: *Communications of the ACM* (2017).

[179] Lingfeng Bao et al. "A Large Scale Study of Long-Time Contributor Prediction for GitHub Projects". In: *IEEE Transactions on Software Engineering (TSE)* (2019).

[180] JDT Java Development Tools. `https://eclipse.org/jdt/`. Last accessed: 07/23/2019. 2019.

[181] *LogProvider in Ninja*. `https://github.com/ninjaframework/ninja`. Last accessed: 08/22/2019.

[182] Cristina Lopes and Joel Ossher. "How Scale Affects Structure in Java Programs". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2015*. 2015.

[183] Liferay Portal. *Liferay Portal - an open source enterprise web platform*. `https://github.com/liferay/liferay-portal`. Last accessed: 08/22/2019. 2019.

[184] William H. Kruskal and W. Allen Wallis. "Use of Ranks in One-Criterion Variance Analysis". In: *Journal of the American Statistical Association (JASA)* (1952).

[185] Ultimate Guide to Logging. `https://www.loggly.com/ultimate-guide/java-logging-basics/`. Last accessed: 08/21/2019. 2019.

[186] The State of Logging in Java. `https://stackify.com/logging-java/`. Last accessed: 08/21/2019. 2019.

[187] Why not use java.util.logging? `https://stackoverflow.com/questions/11359187/why-not-use-java-util-logging`. Last accessed: 07/23/2019. 2019.

[188] Timber - a logger with a small, extensible API which provides utility on top of Android's normal Log class. `https://github.com/JakeWharton/timber`. Last accessed: 08/23/2019. 2019.

[189] *OkHttp - an HTTP client for Android, Kotlin, and Java*. `https://github.com/square/okhttp`. Last accessed: 08/23/2019.

[190] Bitcoin Wallet. *Bitcoin Wallet app for your Android device*. `https://github.com/bitcoin-wallet/bitcoin-wallet`. Last accessed: 08/23/2019. 2019.

[191] Jiawei Han. *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609016.

[192] Ahmed E. Hassan and Richard C. Holt. "Using Development History Sticky Notes to Understand Software Architecture". In: *Proceedings of 12th International Workshop on Program Comprehension (IWPC), 2004*. 2004.

[193] Vert.x - a tool-kit for building reactive applications on the JVM. `https://github.com/eclipse-vertx/vert.x`. Last accessed: 08/23/2019. 2019.

[194] Maven plugin development guide. `http://maven.apache.org/guides/plugin/guide-java-plugin-development.html`. Last accessed: 08/23/2019. 2019.

[195] Brittany Johnson et al. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013*. 2013.

[196] log4net. `https://logging.apache.org/log4net/`. Last accessed: 07/23/2019. 2019.

[197] Sentry - cross-platform application monitoring, with a focus on error reporting. `https://sentry.io`. Last accessed: 08/23/2019. 2019.

[198] Firebase - A mobile and web application development platform. `https://firebase.google.com/`. Last accessed: 08/23/2019. 2019.

[199] Spring Cloud Sleuth. `https://spring.io/projects/spring-cloud-sleuth`. Last accessed: 08/23/2019. 2019.

[200] Mathieu Goeminne and Tom Mens. "Towards a survival analysis of database framework usage in Java projects". In: *Proceedings of 31st International Conference on Software Maintenance and Evolution (ICSME), 2015*. 2015.

[201] Ahmed Zerouali and Tom Mens. "Analyzing the evolution of testing library usage in open source Java projects". In: *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017*. 2017.

[202] Report: Software failure caused 1.7 trillion dollar in financial losses in 2017. `https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017`. Last accessed: 07/23/2020. 2018.

[203] Edsger W. Dijkstra. "Notes on Structured Programming". Apr. 1970.

[204] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. 3rd. Boston, MA, USA: Auerbach Publications, 2008.

[205] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd. New York, NY, USA: Cambridge University Press, 2017.

[206] Martin Fowler. `https://martinfowler.com/bliki/TestCoverage.html`. Last accessed: 01/29/2018. Apr. 2012.

[207] Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness". In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 2014.

[208] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems". In: *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015.

[209] Ajitha Rajan, Michael Whalen, and Mats Heimdahl. "The effect of program and model structure on mc/dc test adequacy coverage". In: *In Proceedings of the ACM/IEEE 30th International Conference on Software Engineering (ICSE)*. 2008.

[210] David Schuler and Andreas Zeller. "Assessing Oracle Quality with Checked Coverage". In: *In Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2011.

[211] Alessandro Orso and Gregg Rothermel. "Software Testing: A Research Travelogue (2000–2014)". In: *Proceedings of the on Future of Software Engineering (FOSE)*. 2014.

[212] Yoram Adler et al. "Code Coverage Analysis in Practice for Large Systems". In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 2011.

[213] Code coverage goal: 80% and no less! `https://testing.googleblog.com/2010/07/code-coverage-goal-80-and-no-less.html`. Last accessed: 01/29/2018. July 2010.

[214] Alan Page and Ken Johnston. *How We Test Software at Microsoft*. Microsoft Press, 2008. ISBN: 0735624259, 9780735624252.

[215] Semantic Designs. *Test Coverage tools*. `http://www.semdesigns.com/Products/TestCoverage/`. Last accessed: 01/29/2018. 2016.

[216] Microsoft. *Microsoft Visual Studio - Using Code Coverage to Determine How Much Code is being Tested*. `https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested`. Last accessed: 01/29/2018. 2016.

[217] Cobertuna. *Cobertura*. http://cobertura.github.io/cobertura/. Last accessed: 01/29/2018. 2018.

[218] Ira Baxter. *Branch Coverage for Arbitrary Languages Made Easy*. `http://www.semdesigns.com/Company/Publications/TestCoverage.pdf`. Last accessed: 01/29/2018. 2002.

[219] Jacoco. *JaCoCo Implementation Design*. `http://www.jacoco.org/jacoco/trunk/doc/implementation.html`. Last accessed: 01/29/2018. 2018.

[220] Google Zúrich Marko Ivanković. *Measuring Coverage at Google*. `https://testing.googleblog.com/2014/07/measuring-coverage-at-google.html`. Last accessed: 01/29/2018. 2014.

[221] XWiki. *XWiki Development Zone - Testing*. `http://dev.xwiki.org/xwiki/bin/view/Community/Testing`. Last accessed: 03/04/2018. 2018.

[222] IBM. *IBM Rational Test Realtie - Estimating Instrumentation Overhead*. `https://www.ibm.com/support/knowledgecenter/en/SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/tsciestimate.htm`. Last accessed: 01/29/2018. 2017.

[223] Mustafa M. Tikir and Jeffrey K. Hollingsworth. "Efficient Instrumentation for Code Coverage Testing". In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2002.

[224] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. "Deriving Code Coverage Information from Profiling Data Recorded for a Trace-based Just-in-time Compiler". In: *In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*. 2013.

[225] Wang Qianqian, Brun Yuriy, and Orso Alessandro. "Behavioral Execution Comparison: Are Tests Representative of Field Behavior?" In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2017.

[226] Gene Kim et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. 2016.

[227] Kannan Muthukkaruppan. *The Underlying Technology of Messages*. `https://www.facebook.com/note.php?note_id=454991608919`. Last accessed: 03/04/2018. Nov. 2010.

[228] HBase. *Powered By Apache HBase*. `http://hbase.apache.org/poweredbyhbase.html`. Last accessed: 03/04/2018. 2018.

[229] Yahoo. *Yahoo! Cloud Serving Benchmark*. `https://github.com/brianfrankcooper/YCSB/`. Last accessed: 01/29/2018. 2018.

[230] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. 2010.

[231] Clover. *Java and Groovy code coverage*. `https://www.atlassian.com/software/clover`. Last accessed: 03/04/2018. 2018.

[232] Pavneet Singh Kochhar et al. "PCode Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects". In: *IEEE Transactions on Reliability* (2017).

[233] Chen Huo and James Clause. "Interpreting Coverage Information Using Direct and Indirect Coverage". In: *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2016.

[234] Tumblr. *Tumblr Architecture - 15 Billion Page Views A Month And Harder To Scale Than Twitter*. `http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html`. Last accessed: 03/04/2018. 2018.

[235] Michael Würsch, Emanuel Giger, and Harald C. Gall. "Evaluating a Query Framework for Software Evolution Data". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2013).

[236] Pinjia He et al. "Towards Automated Log Parsing for Large-Scale Log Data Analysis". In: *IEEE Transactions on Dependable and Secure Computing* (2017).

[237] HBase. *Integration Tests for HBase*. http://hbase.apache.org/0.94/book/hbase.tests.html. Last accessed: 01/29/2018. 2012.

[238] Vincent Jiang. *How to change HBase log level on the fly?* `https://community.hortonworks.com/content/supportkb/49499/how-to-change-hbase-log-level-on-the-fly.html`. Last accessed: 01/29/2018. 2016.

[239] M. Moein Almasi et al. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application". In: *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE SEIP)*. 2017.

[240] Tao Xie, Nikolai Tillmann, and Pratap Lakshman. "Advances in Unit Testing: Theory and Practice". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. 2016.

[241] Emre Kiciman and Helen J. Wang. "Live Monitoring: Using Adaptive Instrumentation and Analysis to Debug and Maintain Web Applications". In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. 2007.

[242] Pavneet Singh Kochhar et al. "An Empirical Study on the Adequacy of Testing in Open Source Projects". In: *2014 21st Asia-Pacific Software Engineering Conference (APSEC)*. 2014.

[243] Hoan Anh Nguyen et al. "Interaction-Based Tracking of Program Entities for Test Case Evolution". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017.

[244] Earl T. Barr et al. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* (2015).

[245] Leonardo Mariani et al. "The central role of test automation in software quality assurance". In: *Software Quality Journal* (2017).

[246] Coverage.py. *A tool for measuring code coverage of Python programs.* `https://coverage.readthedocs.io/en/coverage-4.5.1/`. Last accessed: 01/29/2018. 2018.

[247] Michael Whalen et al. "Observable Modified Condition/Decision Coverage". In: *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. 2013.

[248] Milos Gligoric et al. "Comparing Non-adequate Test Suites Using Coverage Criteria". In: *In Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*. 2013.

[249] Shilin He et al. "Experience Report: System Log Analysis for Anomaly Detection". In: *In Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 2016.

[250] Ivan Beschastnikh et al. "Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (FSE)*. 2011.

[251] Qingwei Lin et al. "iDice: Problem Identification for Emerging Issues". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 2016.

[252] Mark D. Syer et al. "Continuous validation of performance test workloads". In: *Automated Software Engineering* (2017).

[253] Yichen Xie and Alex Aiken. "Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability". In: *ACM Trans. Program. Lang. Syst.* (2007).

[254] Python. *The AST module in Python standard library.* `https://docs.python.org/2/library/ast.html`. Last accessed: 04/02/2018. 2018.