# Studying and Detecting Log-related Issues

Mehran Hassani

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of  (Computer Science) at
Concordia University
Montréal, Québec, Canada

6 March 2018

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

|  |  |
|---|---|
| By: | **Mehran Hassani** |
| Entitled: | **Studying and Detecting Log-related Issues** |

and submitted in partial fulfillment of the requirements for the degree of

**(Computer Science)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

|  |  |
|---|---|
| Dr. C. Poullis | Chair |
| Dr. T.-H. Chen | Examiner |
| Dr. Y.-G. Gueheneuc | Examiner |
| Dr. Weiyi Shang | Supervisor |
| Dr. Nikolaos Tsantalis | Co-supervisor |

Approved by

Dr Volker Haarslev, Graduate Program Director

15 March 2018

Dr Amir Asif, Dean

Faculty of Engineering and Computer Science

# Abstract

Studying and Detecting Log-related Issues

Mehran Hassani

Logs capture valuable information throughout the execution of software systems. The rich knowledge conveyed in logs is leveraged by researchers and practitioners in performing various tasks, both in software development and its operation. Log-related issues, such as missing or having outdated information, may have a large impact on the users who depend on these logs. In this paper, we first perform an empirical study on log-related issues in two large-scale, open-source software systems. We found that the files with log-related issues have undergone statistically significantly more frequent prior changes, and bug fixes. We also found that developers fixing these log-related issues are often not the ones who introduced the logging statement nor the owner of the method containing the logging statement. Maintaining logs is more challenging without experts. Finally, we found that most of the defective logging statements remain unreported for a long period (median 320 days). Once reported, the issues are fixed quickly (median five days). Our empirical findings suggest the need for automated tools that can detect log-related issues promptly. We conducted a manual study and identified seven root-causes of the log-related issues. Based on these root causes, we developed an automated tool that detects four types of log-related issues. Our tool can detect 78 existing inappropriate logging statements reported in 40 log-related issues. We also reported new issues found by our tool to developers and 38 previously unknown issues in the latest release of the subject systems were accepted by developers.

# Acknowledgments

First and foremost, I want to express my deepest gratitude to my supervisors, Dr. Weiyi Shang and Dr. Nikolaos Tsantalis for their encouragement, supervision and guidance from the very beginning of this research work. This thesis would not have been possible without their continuous support.

Apart from my advisors, I would like to thank my thesis examiners, Drs. Peter Chen and Yann-Gaël Guéhéneuc, for their extremely valuable and constructive suggestions. Furthermore, I would like to acknowledge Dr. Emad Shihab for his valuable contribution and guidance in my research.

With a special mention to all my labmates and colleagues, Moiz Aref, Guilherme Padua, Sultan Wehaibi, Everton Maldonado, Jinfu Chen, Kundi Yao, Maxime Lamothe, Giancarlo Sierra, Armin Najafi, Mohammad Matin Mansoori, and everyone else. I learned many things from you and I am grateful for all the great memories throughout my time at Concordia.

I would also like to thank my brother-in-law, Dr. Davood Mazinanian for always being there for me as my friend, as my brother, as my colleague, and everything in between. Also, I would like to thank my sister, Mehrnoosh. Thanks for making the life apart from mom and dad easy for me. You were always had my back since I came to this world and I will always be grateful for all the love and support you gave me in my life.

It is almost impossible to express my feelings toward my parents. Mom and Dad, thanks for your unconditional love and support. You gave everything to me in life and gave up on your son to let him start his journey far away from you. I could never ask for more in my life. Thanks for being the best. Furthermore, I would like to thank my dear brother, Behnam for being the best elder brother. Thanks for being there for mom and dad.

# Contents

**List of Figures** **viii**

**List of Tables** **ix**

**1 Introduction** **1**

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.4 Thesis Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.5 Related Publication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Related Work** **5**

2.1 Log analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1.1 The importance of logs and log analysis . . . . . . . . . . . . . . . . . 5

2.1.2 Existing log analysis tools . . . . . . . . . . . . . . . . . . . . . . . . 7

2.1.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 Logging enhancement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2.1 Chen *et al.* [CJ17] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2.2 Yuan *et al.* [YPZ12a] . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.2.3 Errlog (Yuan *et al.* [YPH+12]) . . . . . . . . . . . . . . . . . . . . . . 12

2.2.4 LogEnhancer (Yuan *et al.* [YZP+12]) . . . . . . . . . . . . . . . . . . . 13

2.2.5 Yao *et al.* [KY18] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.2.6 *Log*$^2$ (Ding *et al.* [DZL+15]) . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.7 Log20 (Zhao *et al.* [ZRL+17]) . . . . . . . . . . . . . . . . . . . . . . 16

2.2.8 Fu *et al.* [FZH+14] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.2.9 LogAdvisor (Zhu *et al.* [ZHF+15]) . . . . . . . . . . . . . . . . . . . . 18

2.2.10 Li *et al.* [LSH17] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.2.11 Li *et al.* [LSZEH17] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.2.12 Li *et al.* [LCSH18] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.2.13 Kabinna *et al.* [KBSH16] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.2.14 Kabinna *et al.* [KSBH16] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.2.15 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**3 Study Design** **25**
3.1 Subject systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.2 Collecting log-related issues . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

**4 Empirical Study** **28**
4.1 RQ1: What are the characteristics of files with log-related issues? . . . . . . . . . . . 29
4.1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.1.2 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.1.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.2 RQ2: Who reports and fixes log-related issues? . . . . . . . . . . . . . . . . . . . . 31
4.2.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2.2 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.3 RQ3: How quickly are log-related issues reported and fixed? . . . . . . . . . . . . . 35
4.3.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3.2 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.4 RQ4: What are the root-causes of log-related issues? . . . . . . . . . . . . . . . . . 38
4.4.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.4.2 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.4.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**5 Automatic detection of inappropriate logging statements** **45**
5.1 Log level checker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.2 Catch block checker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.3 Log level guard checker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.4 Typo checker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.5 Results of applying the tool . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.6 Impact of the threshold on our checkers . . . . . . . . . . . . . . . . . . . . . . 53
5.6.1 Log level checker refinement. . . . . . . . . . . . . . . . . . . . . . . . . . . 53

5.6.2 Catch block checker refinement. . . . . . . . . . . . . . . . . . . . . . . . . . . 53

**6 Threats to Validity 55**

6.1 Internal Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

6.2 Construct Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

6.3 External Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

**7 Conclusions and Future Work 57**

7.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

7.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

**Bibliography 60**

# List of Figures

1 An overview of our approach to select the subject systems . . . . . . . . . . . . . . . 26

2 An overview of our approach to answer the research questions . . . . . . . . . . . . . 28

3 Lifetime of an example inappropriate logging statement . . . . . . . . . . . . . . . . 36

4 Cumulative distribution of issue report time in days. Outliers that are greater than 1.5 time of the value of the third quartile of the data are not shown in this figure. . . 39

5 Cumulative distribution of issue fix time in days. Outliers that are greater than 1.5 time of the value of the third quartile of the data are not shown in this figure. . . . . 40

6 An overview of log-related issue checker. . . . . . . . . . . . . . . . . . . . . . . . . . 46

7 A patch to fix inappropriate logging level . . . . . . . . . . . . . . . . . . . . . . . . 47

8 Using entropy to suggest appropriate logging level . . . . . . . . . . . . . . . . . . . 48

9 An example of faulty logging statements that fixed by adding exception message . . 50

# List of Tables

1 Comparing our work with prior research on logging suggestions and improvements . 24

2 The Number of Issues in Hadoop and Camel . . . . . . . . . . . . . . . . . . . . . . 26

3 Cliff's delta effect size interpretation. . . . . . . . . . . . . . . . . . . . . . . . . . . 30

4 Medians (*Med.*) and effect sizes (*Eff.*) of the normalized process and product metrics for files with and without log-related issues. Effect sizes are not calculated if the difference between files with and without log-related issues is not statistically significant. 32

5 Medians (*Med.*) and effect sizes (*Eff.*) of the normalized process and product metrics for files with and without log-related issues. Files without logging statements are excluded. Effect sizes are not calculated if the difference between files with and without log-related issues is not statistically significant. The percentage of files with log-related bugs shown in front of each subject system. . . . . . . . . . . . . . . . . . 33

6 Number of days before an inappropriate logging statement being reported . . . . . . 37

7 Number of changes before an inappropriate logging statement get fixed . . . . . . . . 37

8 Categories of log-related issues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

9 Distribution of the root causes of log-related issues . . . . . . . . . . . . . . . . . . . 42

10 A five-number summary for the lowest entropy of the phrases in each logging statement 49

11 The results of our tool on known issues . . . . . . . . . . . . . . . . . . . . . . . . . 52

12 Our findings on log-related issues and their implication. . . . . . . . . . . . . . . . . 58

# Chapter 1

# Introduction

## 1.1 Motivation

Developers write logging statements in the source code to expose valuable information of runtime system behavior. A logging statement, e.g., *LOG.warn("Cannot access storage directory " + rootPath)*[1], typically consists of a log level (e.g., trace/debug/info/warn/error/fatal), a logged event using a static text, and variables that are related to the event context. At runtime, the invocation of these logging statements generates logs that are often treated as the most important, sometimes only, source of information for debugging and maintenance of large software systems.

The importance of logs has been widely identified [KP99]. Logs are used during various software development activities such as bug fixing [XHF$^+$09], anomaly detection [TPK$^+$08], testing results analyses [MHH13], and system monitoring [YZP$^+$12, BKQ$^+$08]. The vast application and usefulness of the logs motivate developers to embed large amounts of logging statements in their source code. For example, the OpenSSH server contains 3,407 logging statements in its code base [YPZ12a]. Moreover, log processing infrastructures such as Splunk [Car12] and ELK stack [elk17] are developed for the ease of systematic log analyses.

To improve logging statements, similar to fixing bugs, developers report their issues with the logging statement and fix it by changing the source code or other artifacts (e.g., configuration) during development. For example, in an issue in Apache *Hadoop-HDFS*, HDFS-3326[2], with the title "Append enabled log message uses the wrong variable", developers replace the recorded variable in the logging statement to provide more meaningful information. We consider such issues that are fixed to improve logging statements as *log-related issues.*

Prior empirical studies examine the characteristics of logging practices [YPZ12a] and the places

[1]https://issues.apache.org/jira/browse/HDFS-4048
[2]https://issues.apache.org/jira/browse/HDFS-3326

where developers embed logging statements [FZH$^+$14]. Also, prior research aims to enhance logging statements by automatically including more information [YMX$^+$10, YZP$^+$12], and provide suggestions on where to log [ZHF$^+$15]. However, these empirical results and the above-mentioned approaches do not help developers write an issue-free logging statement. In fact, there exist limited guidelines that developers can follow to write appropriate logging statements.

## 1.2 Problem Statement

The issues with logging statements have become one of the major concerns, due to the vast usage of logs in practice. Examples of such issues include missing to embed important logging statements[3] have misleading text in logging statements[4], and generating overwhelming information[5]. Logging statements with issues may significantly reduce usefulness of the logs and bring extra overhead to practitioners. For example, missing logging statements in the critical part of the source code may cause developers not to have enough knowledge about the system execution; the misleading textual description in logging statements may lead to wrong decisions made by system operators; and overwhelming information in logs would prevent practitioners from identifying the truly needed information [LSH17]. Recent research on Github projects claims that over half of the Java logging statements are "wrong" [Hen17]. Moreover, for automated log analyses, the issues may have an even larger impact by rather simple mistakes like a typo. For example, in the issue HADOOP-4190[6] with *Blocker* priority, developers missed a dot in a logging statement, leading to failures in log analysis tools.

In this work, we conduct an empirical study on the real log-related issues from two large, open source software systems that extensively use logging statements, i.e., Hadoop and Camel. Studying log-related issues can lead us in devising an automated technique that will aid developers to improve logging statements. We extract 563 log-related issues from the JIRA issue tracking systems of the two subject systems and study these issue reports and their corresponding code changes. Our study aims to answer the following research questions:

**RQ1** *What are the characteristics of files with log-related issues?*

Files with log-related issues have undergone, statistically significantly more frequent changes and more frequent bug fixes. Developers should prioritize their efforts on such files to identify logging statements with potential issues.

**RQ2** *Who reports and fixes log-related issues?*

[3]https://issues.apache.org/jira/browse/HDFS-3607
[4]https://issues.apache.org/jira/browse/HDFS-1332
[5]https://issues.apache.org/jira/browse/CAMEL-6551
[6]https://issues.apache.org/jira/browse/HADOOP-4190

We found that in 78% the cases, logging statements are added and fixed by different people. There exists no systematic responsibility for developers to maintain logging statements in the subject systems. This may make it difficult to identify an expert to ensure whether a logging statement is appropriate.

**RQ3** *How quickly are log-related issues reported and fixed?*

By examining the time between the introduction of logging statements, the report, and fixed time of the log-related issues, we find that log-related issues are often reported a long time (on a median of 320 days) after the logging statements were introduced into the source code. Once reported, however, the issues are fixed in a short time (on a median of five days). Therefore, practitioners may benefit from automated tools that detect such issues promptly.

**RQ4** *What are the root-causes of the log-related issues?*

Through a manual analysis on log-related issues and their corresponding fixes, we identify seven root-causes of log-related issues, namely: inappropriate log messages, missing logging statements, inappropriate log level, log library configuration issues, runtime issues, overwhelming logs, and log library changes. Many root-causes (like typos in logs) of these issues are rather trivial, suggesting the opportunity of developing automated tools for detecting log-related issues.

Our empirical study results highlight the needs and opportunities for automated tooling support for detecting evident log-related issues. Therefore, we developed an automated tool[7] to detect four types of log-related issues. Our tool detected 40 of the 133 known log-related issues. Moreover, we reported 78 detected potential log-related issues from the latest releases of the subject systems. Out of 78, 38 of them had been accepted by their development team through issue reports and the rest of them are still under review.

## 1.3 Contributions

Our most significant contributions are listed as follows:

- We perform a characteristic study on different aspects of log-related issues, namely the files that contain log-related issues, report and fix time, and developers' involvement in the process.
- We manually identify seven root-causes of log-related issues.
- We propose an automated tool that can detect four different types of evident log-related issues from source code.

[7]https://mehranhassani.github.io/LogBugFinder/

## 1.4 Thesis Organization

The rest of the study is organized as follows. Chapter 2 discusses the related works of our study in two categories of log analysis and logging enhancement. Chapter 3 presents the subject systems under study and our approach to identify and collect log-related issues. Chapter 4 explains the empirical study on different aspects of log-related issues. We motivate each research question and demonstrate our approach to perform our analysis. Then, we present and discuss the results of our study and its implications. Chapter 5 demonstrates our proposed tool and its approach to automatically detects log-related issues. We present how these checkers are designed and the issues that we were able to detect using these them. We also compare our approaches with the state of the art tools where ever the they were applicable. Chapter 6 discusses the potential threats to the validity of our study. Finally, Chapter 7 includes the conclusion of this study and suggestion for future works.

## 1.5 Related Publication

Earlier version of the work done in this thesis has been published in the following paper:

- **Mehran Hassani**, Weiyi Shang, Emad Shihab, Nikolaos Tsantalis. Studying and Detecting Log-Related Issues. Empirical Software Engineering.

# Chapter 2

# Related Work

In this chapter, we discuss the prior research which is related to the work conducted in this thesis. We have divided the related work into two categories, namely 1) the works concerning *log analysis*, and 2) the studies that aim at devising techniques for *improving logs* in the source code.

## 2.1 Log analysis

Logs are widely used in software development and operation to ensure the quality of large software systems. Valuable information is extracted from logs, including event correlations [NKN12, FRZ$^{+}$12], component dependency [OA11], and causal paths [YZP$^{+}$12]. In this section, we first review the works that discuss the importance of log analysis, and then list the related work that shows the status of log analysis research in the literature.

### 2.1.1 The importance of logs and log analysis

**Barik *et al.* [BDDF16]**

In a study at Microsoft, Barik *et al.* [BDDF16] performed a mixed-method study on how teams at Microsoft use event data, which also includes logs. The authors interviewed software engineers in various roles in the company, including software developers, program managers, operations engineers, content creators, data scientists, and managers. Interview participants are selected based on a recruitment survey from 1,897 randomly-selected employees at Microsoft. Doing so, they made sure that participants are familiar with event data and have already use it in the past to fulfill their daily tasks. The authors used *grounded theory* guidelines to analyze the interview data, and also performed a follow-up survey to increase the reliability of their qualitative findings.

The results showed that participants in any role extensively use event data in their activities.

Furthermore, the study found that software engineers share event data among different roles, since event data is by nature flexible and can be utilized for different purposes. For instance, one can use event data to find the root cause of a crash, while the other can use the same event data to rank the most used feature by users. Finally, the study revealed several challenges associated with using event data, including the complexity of combining data from different sources, the high amount of clerical work required, the length of time needed for the activities which are based on event data, and the presence of limited or difficult-to-use tools.

This study emphasizes the use of log data in various software engineering activities, and the challenges that software engineers face in their activities when using log data. In this thesis, we focus on identifying and fixing log-related issues, the presence of which can add even more complexity to the already-challenging activities performed by software engineers using logs.

#### Oliner *et al.* [OGX12]

Oliner *et al.* [OGX12] discuss the most common applications of log analysis in practice, as summarized below:

- **Debugging.** Most logs are used to facilitate debugging. Developers commonly use simple approaches to use log information for debugging, like using `grep` to search for messages with specific pattern that may reveal useful information. For instance, if a server crashes because of network drop, developers might try to search for "connection dropped" message in the logs. However, in many scenarios, these simple approaches will not help users to solve their problem, since, among the other reasons, logs can become overwhelmingly large in practice. Thus, developers and operators might use more sophisticated approaches, like PCA (principal component analysis) or SVM (support vector machines) to battle this complexity. Such approaches are especially helpful in networked or large-scale distributed systems, *e.g.*, Hadoop.

- **Performance.** Log analysis also helps users in improving or debugging the performance of a software system. Logs can be used to find performance anomalies and pinpoint their root causes.

- **Security.** Oliner *et al.* also discuss the use of logs in security applications. For example, users can analyze logs to find evidence for security breaches or misbehaviors. Furthermore, logs can be utilized to perform *postmortem inspection* of security incidents. Log messages are evidence of various events, such as the execution of a code block or the creation or destruction of an SSH session. Log analysts can aid to investigate logs in order to find out if an SSH connection constitutes a security breach or not.

- **Prediction.** Oliner *et al.* mention *prediction* as another usage for log analysis. Logs can be used to predict and plan for the future. Predicative models are widely used in workload management, scheduling, configuration optimization, and other tasks.
- **Profiling and reporting.** Another use for log analysis is profiling and reporting the resources' or users' behavior. Log analysts can employ a verity of statistical techniques for profiling and reporting activities on log data.

Consequently, issue-free logs – which is the focus of this study – will improve the results of log analysis tools, considering their wide use.

### 2.1.2 Existing log analysis tools

#### DISTALYZER (Nagaraj *et al.* [NKN12])

Nagaraj *et al.* [NKN12] presented "DISTALYZER", a tool used to analyze logs of distributed software systems to find the components that cause performance degradation. DISTALYZER compares two sets of logs, generated before and after the system exhibits degraded performance, and generates a summary showing the events that diverge the most across the sets of logs, and therefore, potentially affect the overall system's performance. The tool uses machine learning techniques to automatically find the strongest associations between system components and performance. DISTALYZER also provides the ability to interact with the results and the exploration for extra analysis. The authors evaluated DISTALYZER on three large-scale distributed systems (namely TritonSort, HBase, and Transmission), and identified the root causes of six performance problems in these subject systems.

#### SherLog (Yuan *et al.* [YMX$^+$10])

Yuan *et al.* [YMX$^+$10] introduced SherLog, which is a postmortem error diagnosis tool that automatically infers what must or may have happened during a failed execution. SherLog takes the log output and the source code of the software system and tries to find the execution path that leads to the failure. To do so, SherLog first parses the log output. Then, using the information provided by logs, it statically walks through the execution path that leads to the failure. The authors used eight real-world failures from seven applications to evaluate their approach. They manually reproduced and diagnosed each failure and collocated the runtime logs. The authors then compared the results generated by SherLog with their manual diagnoses. If a subset of the information collected during the manual study is generated by SherLog, the authors consider it as *useful.* If all the essential information in the manual investigation is generated by SherLog, they consider it as *complete.* In all the experiments, SherLog generated *complete* information. The results demonstrate that SherLog is effective in zooming into the paths that are relevant to the failure. The authors also evaluated the performance of SherLog, and showed that the longest SherLog diagnostic took around 40 minutes.

#### LogMaster (Fu *et al.* [FRZ$^+$12])

Fu *et al.* [FRZ$^+$12] proposed a correlation mining system and an event prediction system called LogMaster. LogMaster parses the logs into event sequences and uses an algorithm, named Apriori-LIS that can mine event rules form these event sequences. They also designed an abstraction of the log events called Events Correlation Graphs (ECGs) to represent event rules, and used it for event prediction. They validated their approach on three logs of production cloud and HPC systems with average precisions of over 80%. The authors also showed that their approach needs significantly less time comparing to related studies.

#### Oliner *et al.* [OA11]

Oliner *et al.* [OA11] proposed an online method for analyzing large software systems, which can detect the interactions among the components of the target system and identify unexpected behavior. The approach uses logs for its analysis. Logs have *timestamps*, *i.e.*, they can be seen as time-varying *signals*. In this approach, consequently, the authors exploit this fact for devising a technique for identifying unexpected system behavior. Logs are converted into real-valued functions of time, called *anomaly signals*. In the first stage of the proposed approach, an online Principal Component Analysis (PCA) is employed to summarize these anomaly signals by finding correlated groups of signals. Then, the approach uses these compressed signals to identify time-delayed correlations (i.e., surprising behavior) in logs using an online approximation algorithm. The authors show that the results of their approach can be valuable for system administration tasks in real use cases from eight unmodified production systems.

### 2.1.3 Summary

The extensive usage of logs motivates our study, since quality logs are extremely important for the effectiveness of prior log analysis research. None of the mentioned works have looked into this issue specifically. The outcome of this thesis would help in reducing log-related issues, and hence improves the adoption of advanced log analysis techniques in practice.

## 2.2 Logging enhancement

### 2.2.1 Chen *et al.* [CJ17]

The closest recent research to this study is the work by Chen *et al.* [CJ17]. The authors studied the problem of *how to log*, by looking into anti-patterns in the logging code. They manually studied logging code changes in three popular open source software systems over the period of more than six years. The authors focused on changes that occurred independently of the *feature code* that contains the logging statement. In order to identify these changes, the authors extract the fine-grained code changes from source code repositories. Then, they used heuristics to automatically extract the code changes which contain logging code modifications. Finally, they used dependency analysis to categorize the logging code changes into 1) changes due to modifications on the feature code, and 2) independent logging code changes. After extracting the independent logging changes, they manually went through a statistically significant sample of the changes. The authors found, in total, nine reasons for independently changing the logging code, categorized into two groups: first, "what to log", and second, "how to log". In this study, the authors only focus on the rationales in the category of "how to log'. They found five rationales in this category, each corresponding to the different fixes to the anti-patterns in the logging code. Based on these five rationales, they describe five logging anti-patterns as follows:

1. **Nullable Objects.** Logging statements use variables to log the runtime information throughout the execution of the system. However, the objects used in the dynamic contents can be null. If not checked for, this would cause a NullPointerException and crash the system.

2. **Explicit Casting.** Explicit casting converts an object into a particular type. If the object is not convertible to the requested type, explicit casting will cause runtime type conversion errors and the system will crash.

3. **Wrong verbosity level.** Logging libraries use *verbosity levels* to control the type and amount of information appeared in the log. Wrong verbosity level can hide important information from the users, or in contrast, it can result in flooded log output with unwanted logs.

4. **Logging Code Smells.** Code smells are symptoms of bad design and implementation choices [FB99]. In a similar fashion, the logging code smells are defined as poor design and implementation choices when developing logging code.

5. **Malformed Output.** Some objects do not have a human-readable format defined. If they are printed directly to the log output, the generated logs can be useless and pollute the output. For instance, printing a byte array in the logging statement.

To show the usefulness of identifying these anti-patterns, the authors also proposed a static analysis tool called LCAnalyzer. LCAnalyzer uses Eclipse's Java Development Tools (JDT) to flag anti-patterns in the target source code's ASTs. For instance, in order to find *Wrong Verbosity Level* anti-pattern, their checker extracts the static part of the logging statement in the logging code and compares it against its logging level: if developers explicitly mention, in the static part of the logging statement, that it is a debug-level logging statement, they check the logging level of the statement and if it doesn't match the static part, they report it as an anti-pattern.

To evaluate LCAnalyzer, the authors conduct two case studies. First, they evaluated the performance of their tool using an *oracle* and reported the precision and recall. The overall average recall for LCAnalyzer in finding the mentioned anti-patterns is 95%. To calculate the precision, they manually examined the detected anti-patterns from the oracle. The average precision for LCAnalyzer is 60%.

The authors also applied their tool on the latest releases of ten different open source software systems. They found instances of the mentioned anti-patterns on various types of systems like clients, servers, and frameworks. They also found that there is a medium to strong correlation between the amount of logging code and the number of anti-pattern instances. Finally, the authors manually selected 64 representative instances of anti-patterns in the aforementioned ten open source systems and reported them as issues to the developers. Out of these, 46 instances (71.9%) were accepted by the developers.

Our study complements this work in many ways. Chen *et al.* explore anti-patterns in logging code by mining logging code changes in three open-source systems. In particular, they identified six anti-patterns and proposed static code checkers to detect these patterns. However, instead of detecting anti-patterns (like logging code smells [CJ17]) that have the *possibility* to be an issue, we focus on the *reported issues* that are more certain. In fact, by comparing our study to the research by Chen *et al.*, only one anti-pattern/root-cause (which is Wrong Log Level) overlaps across the two studies. The reason may be the different focus on anti-patterns and evident issues, and that we identify log-related issues from issue reports while Chen *et al.* leverage code changes to identify anti-patterns. Moreover, the results of our empirical study on log-related issues provide more insights on these issues.

### 2.2.2 Yuan *et al.* [YPZ12a]

Yuan *et al.* [YPZ12a] performed a characteristic study on log practices by mining the revision histories of four open-source software projects. This was the first attempt to study the practice of software logging. Based on their analysis on the historical data from four open-source software projects. They provided several insights on logging:

- Logging is a pervasive practice during software development. On average, every 30 lines of code contains a logging statement.
- Logging is beneficial for failure diagnosis. Logging, on average, reduces diagnoses time of run-time failures on a median of 2.2X times.
- Developers actively maintain logging statements. The average churn rate of logging code is almost two times (1.8) compared to the entire code base.
- Logging code takes a significant part of software evolution. Despite relatively smaller density compared to the other parts of the code, a significant amount of commits (18%) modify logging code.
- Developers usually change logging statements since they do not write the appropriate logging statement in the first attempt. 36% of all logging statements are changed after they introduced to the system.

Furthermore, the authors studied code changes that occurred on logging code. They used simple heuristics to filter out code changes related to the feature changes in the commit history. The authors found that developers do not delete or move the logging statements unless they introduce serious problems: only 2% of the logging modifications are deletions. In contrast, 98% of modifications are done on the content of the log messages, such as their verbosity level, static text, and variables. Due to the large number of modifications occurred on the verbosity level of the logs (2,389 of the changes are only on the verbosity level), the authors also investigated these changes in more detail. The results showed that 72% of the verbosity level changes are due to the change of developers' judgment about an *error* event. Thus, tools that automatically expose error conditions can help identifying logging behaviors. In other words, exposing code to different error conditions can help developers in making better decisions when selecting verbosity level for logs. Even for none-error logging statements, developers reconsider the trade-off between multiple verbosity levels. The authors propose that *adaptive logging verbosity* can help to reduce problems regarding verbosity levels. In another finding, the authors show that more than one fourth of the logging modifications are related to variables in logs, suggesting that automatic approaches can help developers in choosing what variables should be logged. Finally, the authors show that more than one third (39%) of the changes to the static part of the logging statements are because of inconsistencies between logs and the execution information intended to be recorded, motivating tool support for automatically fixing such inconsistencies.

By identifying these manual efforts for modifying logging statements, the authors motivated opportunities for tools, compiler and programming language support to improve the current logging practices. The authors also built a simple checker that can help developers to write better logging statements. We looked into the changes in logging statements by examining log-related issues. We provided insights into the nature of the faulty logging statements and the source code that contains them. Furthermore, we built static analysis tools to help developers improve their logging statements.

### 2.2.3 Errlog (Yuan *et al.* [YPH+12])

Yuan *et al.* [YPH+12] investigate a sample of 250 real-world failure reports from five large and widely used software systems (namely, Apache, squid, PostgreSQL, SVN, and Coreutils). Across each system, the authors manually extracted the log messages and their relation to the failure that was identified. The authors show that, under the default verbosity mode among the software systems, almost all (97%) logging statements are *Error* and *Warning* level messages, approving that Error/Warning are usually the only source of information in case of failures in the field. By comparing the duration of each report, the authors found that reports supported by log messages are fixed 2.2X faster on average. This confirms the hypothesis about the importance of appropriate logging. However, more than half (57%) of the failures do not have failure-related log messages. Furthermore, the programs themselves have caught early error-manifestations in the majority (61%) of the cases, meaning that programs decided to tolerate the errors incorrectly. Making it worse, in most of these cases (85%) developers did not log the incorrectly-tolerated error. These findings show the need for automated tools that inject logs to error handling logic of the programs. Driven by these findings, the authors introduced a tool called Errlog. Errlog analyzes the code for potential unlogged exceptions and inserts logging statements in them. To do so, Errlog looks for seven generic exception patterns, driven from the authors manual study on real-world failures. The tool then examines whether the exception check already exists, and whether it contains a log message. If there are no logging statements in the target exception, Errlog will insert appropriate code to log the exception. To provide more control to the users on the overhead of the logging inserted to the system, Errlog provides three logging modes that will control the overhead introduced by the tool.

The authors evaluated their proposed tool in two ways. First, they performed an in-lab experiment to see how many of the error logs proposed by the tool are also manually added by developers in the original source code. They show that, by configuring the tool to medium overhead mode, Errlog was able to automatically generate an average of 84% of existing log points across all the evaluated subjects. Even on the lowest overhead setting, 52% of the existing logs are generated by the tool with only 1% overhead. Second, they performed a controlled user study to measure the effectiveness of Errlog. The authors asked 20 programmers with extensive and recent experience in C/C++ to fix five failures as best as they could. They provided half of the participants with the error-log-inserted programs and half of them with the original one. They show that, on average, programmers took 60.7% less time to diagnose failures when they where using the logs added by Errlog.

In this thesis, we also proposed an exception log message checker, similar to Errlog. However, we were not able to compare Errlog with our proposed tool since Errlog works on C/C++ code. Notwithstanding, we used a different approach to suggest enhancements for logging statements in the exception handling code: our tool uses the source code to find the instances of each exception type and provides a suggestion based on previous decisions from the developers of the system. This approach is specially helpful for the developers with less experience since it helps them to make decisions based on the well-reviewed and tested source code. Our proposed tool also can suggest improvements to already-existing logging statements in the code. Yet, our tool cannot provide any suggestion if the code has no previous logging statements while Errlog can be used to add logs to the systems with no logging statement in them. We will discuss about our tool in more details in Chapter 5.

### 2.2.4 LogEnhancer (Yuan *et al.* [YZP+12])

The authors propose a tool, named LogEnhancer [YZP+12], to automatically detect valuable information from the source code and inject this information to logging statements. In essence, LogEnhancer is a enhancement tool that takes the existing source code as input and produces a newer version with improved logging statements. To accomplish this goal, LogEnhancer uses a three-step approach. In the first stage, it performs a *Uncertainty Identification* analysis. At this stage, the tool starts from the log and works its way backward on the code and finds the conditions that needed to be satisfied in the control flow to allow the program to execute the log. In the second stage, the tool identifies the key values that will help in solving the uncertain paths leading to the logging statement. Finally, in the last stage, LogEnhancer inserts a procedure before each log to recorded the variable values that are identified in the previous step.

To evaluate LogEnhancer, the authors enhanced the logging statements in 8 different real-world applications. The authors performed three experiments to ensure the usefulness of LogEnhancer. In the first experiment, the authors compared the results of the tool with the variables selected manually by the developers of the subject systems. They found that, on average, 95% of the variables selected by developers are also recorded by LogEnhancer. Furthermore, the authors evaluated the diagnostic effectiveness of the tool by analyzing 15 real-world failures. They found that in all the 15 cases, the original logs were not helpful enough to diagnose the failure since they faced many uncertainties in the path that caused the error. However, the variables added from LogEnhancer significantly reduced the uncertainty, leading to easier diagnosis of the failures.

Our exception checker tool proposed in this thesis also aims at enhancing logging statements by suggesting addition and deletion of variables in the logging statements. With our tool, we suggest developers adding the exception variable to a logging statement, if the thrown exception type always includes the exception variable in the logging statement all across the source code of the subject system. Even though we couldn't run LogEnhancer, from the description of their approach, we can infer that LogEnhancer is guaranteed to solve the issues fixed by adding variables to the logging statements, as well as the issues with missing exception variables (28 log-related issues). However, LogEnhancer can potentially produce noise to the log output. In fact, seven log-related issues were fixed by removing variables from the logging statements, *i.e.,* LogEnhancer may result in worsening the issues. Zhu *et al.* [ZHF$^+$15], also mention that adding all variables to logging statement pollutes the log output and is not recommended by developers.

### 2.2.5 Yao *et al.* [KY18]

Yao *et al.* [KY18] proposed an approach that recommends proper locations to place logging statements in order to improve performance monitoring on web-based software systems. In this approach, the authors use improvements of the explanatory power of statistical performance models as a heuristic to suggest logging statement placement. Their approach contains several steps. To reduce the performance overhead, the approach parses the existing web logs and identifies the web requests that have significantly higher performance impact on the system. Using the web logs, the authors built a statistical performance model and measured the significance of the log metrics on the model's output. Since the log metrics represent the number of times each code block is executed, the authors used them as a proxy to identify the performance-influencing code. In the next step, they automatically injected logging statements into the beginning of the methods in the source code identified in the previous step. After injecting the logging statements, they rebuilt and re-run the performance tests. Then, they made a similar statistical model using both existing web logs and the newly-injected logs. In this step, they use the new model to remove the methods that do not have a significant impact on the performance. Afterwards, they injected logging statements to every basic block of the remaining methods. Finally, they used the logs injected in basic blocks as well as the previous logs to make a newer version of the statistical performance model. Using the impact of the log metrics on the model, they identified performance-influencing basic blocks and suggest adding logging statements to them.

The authors evaluated their approach on three different aspects of two open source and one enterprise software system as subject systems. First, they evaluated the statistical model proposed by their approach. The authors measured the model fit to understand the quality of their statistical models. They showed that their models with $R^2$ between 26.9% and 90.2% and can well explain the system performance. Second, they evaluated the performance influence of the recommended logging locations in the code. To measure the influence, they set all the metrics in the model to their mean value and computed the predicted performance. Then, they doubled the value of one metric while keeping all the other ones in their median value and computed the difference in the model's output. They found that while their suggested logging locations have a significant influence on the performance, the impact can be positive or negative. Finally, the authors performed a manual study on the suggested locations to understand their characteristics. They found that the suggested locations are not in complex methods. Moreover, most of the suggested locations are not in the performance hotspots.

Our study finds that missing logging statements are one of the root-causes of log-related issues. However, the proposed approach by Yao *et al.* is only suitable when considering performance monitoring.

### 2.2.6 *Log*$^2$ (Ding *et al.* [DZL+15])

Ding *et al.* [DZL+15] proposed a novel cost-aware logging mechanism called *Log*$^2$. Ding *et al.* performed a preliminary survey on logging practices in Microsoft. They found that developers at Microsoft believe that the overhead of logging is not negligible. Furthermore, the majority of the developers are not satisfied with the existing approaches to control this overhead. In fact, 83% of the participants agreed that many log messages are redundant for diagnosing performance issues. Motivated by the results of their survey, the authors proposed *Log*$^2$, a logging mechanism that allows the system to adjust logging dynamically to achieve the maximum effectiveness, given an overhead threshold. *Log*$^2$ uses two filtering phases to achieve this goal, a *local* filter and a *global* filter. Each local filter is responsible to discard trivial logging requests. The usefulness of each logging request for performance diagnoses is determined by a dynamically-calculated utility score. Local filters discard logging requests with low utility score. Afterwards, global filter writes the top-ranked logging requests to the disk according to the global overhead threshold. The global threshold will be adjusted dynamically based on the environment dynamics.

To evaluate *Log*$^2$, the authors performed a four-stage experiment on a popular open source blogging platform, namely BlogEngine. In the first step, they marked code regions with high potential to cause performance issues and instrument it with *Log*$^2$. In the second step, the authors deployed BlogEngine and two other machines as clients. They used a tool name WebTest to simulate user behavior on BlogEngine. Using WebTest, they generated five typical usage scenarios, namely read blogs, write comments, search, download files and upload files. Then, in the third step, the authors injected three performance issues, namely *upload an extremely large file*, *search a strange term*, *and exhaust CPU by another process*. Finally, in the last step, the authors monitored the I/O throughput by counting the number of times that $Log^2$ flushes the logs to disk in each time interval. They also measure other performance metrics, like CPU and memory overhead of the $Log^2$. In this experiment, they compared $Log^2$ with a baseline that included outputting all the logs executed. They found that, on average, $Log^2$ writes 104 logs per interval, while 3,800 logs are outputted in the baseline approach. Moreover, although $Log^2$ used slightly more memory comparing to the baseline approach, it offered less CPU usage since a large number of logs are excluded in the process. The authors also showed that with a budget size of 120 logs per interval, one can cover all the manually-identified useful logs form the baseline approach.

$Log^2$ proposes a filtering mechanism to reduce the I/O consumption of the logs at runtime. It performs the filtering with having performance problem diagnoses as the main usage of the logs in mind. The approach is deeply integrated into the source code as an API to reduce the number of logs saved during the runtime. However, in this work, we aim to provide a *recommender* to help developers improve logging statements in their source code rather than introducing a new logging approach.

### 2.2.7 Log20 (Zhao *et al.* [ZRL+17])

Zhao *et al.* introduced a tool called Log20. Log20 is a performance-aware tool to inject new logging statements to the source code. It is also capable of adding variables to the injected logging statements without any domain knowledge. The authors introduced an approach to measure the informativeness of each added logging statement. They used informativeness to compute a near-optimal placement of logging statement given a specific performance overhead. The authors measured informativeness of each logging statement placement by its ability to differentiate different execution paths during runtime. Furthermore, they used Shannon's information theory to measure the entropy of a program by considering all possible execution paths and their probability during runtime. Since each logging statement placement can reduce the entropy by disambiguating these execution paths, they calculated informativeness of logging statements by the reduction in the entropy of the program, given its placement. The authors used this setup to compute near-optimal placement of the logging statements given an overhead threshold. Their approach is rather dynamic and can react to different workloads during runtime.

Zhao *et al.* evaluated their approach on four widely used distributed systems namely HBase, HDFS, YARN, and ZooKeeper. They found that Log20 is considerably more efficient in disambiguating execution paths compared to the existing standard logging libraries. By outputting 0.08 log enriches per request, Log20 was as informative as info level logs with 1.58 log entities per request.

They also compared their approach with Ball-Larus path profiling algorithm[BL96]. They found that Log20's instrumentation is at least 3X more efficient than the Ball-Larus algorithm. Furthermore, Zhao *et al.* evaluated the usefulness of Log20 for debugging activities using 41 randomly selected user-reported HDFS failures. The authors found Log20 is helpful in debugging 68% (28/41) of the real-world failures while existing info level logging statements are helpful in 27 cases.

Log20 dynamically inject logging statements with variables to the byte-code of the program. In essence, they introduced a new logging mechanism. We focused on the existing logging practices. Especially, we studied the log-related issues and proposed static analysis tools to help developers improve their existing logging statements.

### 2.2.8 Fu *et al.* [FZH$^+$14]

Fu *et al.* [FZH$^+$14] systematically studied the logging practices of developers in the industry, with a focus on where developers log. The authors investigated two large-scale industrial software systems from Microsoft. They first performed source code analysis to investigate logging characteristics of the subject systems. Based on a manual examination, they categorized logging statements into five groups, namely assertion-check logging, return-value-check logging, exception logging, logic-branch logging, and observing-point logging. The authors used the results of their manual examination to automatically group all the logging statements in the code. Moreover, they confirmed their identified categories by conducting a survey on developers' opinions about their results. The results of their automatic classification showed that about half of the logged snippets belonged to the first three categories, as they logged due to the unexpected situations, while other half belongs to the next two categories where they logged in expected situations to log execution trace information.

Supported by their findings from the characteristic study, the authors trained a classifier based on contextual information and features of training code snippets to predict whether the tested code snippets need to be logged. They performed 10-fold cross-validation on their classifier to evaluate the accuracy of their results. They showed that, by using contextual information, their classifier can achieve 80.8% to 90.4% recall, while having 81.1% to 90.2% precision.

In this work, we also provide suggestions to the developers about whether they need to log exceptions or not, using our catch block checker tool. However, we were not able to compare the results of our tool to the Fu *et al.*'s, since they work on different programming languages. Our catch block checker tool also provides suggestions on whether developers need to log the exception trace as well as whether they need to add logging statements in the catch block. In this study, we focus on finding log-related issues. Thus, we emphasized on preventing developers to write faulty logging statements rather than giving a general recommendation on the code. Therefore, we evaluated our tool on the faulty logging statements in catch blocks rather than all the catch blocks in the source code. Nevertheless, we were able to preserve high precision for our recommendations.

### 2.2.9 LogAdvisor (Zhu *et al.* [ZHF+15])

Zhu *et al.* [ZHF+15] proposed a framework which helps in providing informative guidance on where to insert logging statements. The authors leveraged these guidelines to automatically learn the common logging practices using machine learning algorithms to suggest logging decisions for developers. Focusing on logging for error handling, the authors extracted code snippets from the source code of the subject systems code snippets with exception code, or return-value-check code, and used them as training data to learn logging practices. Then, they extracted code features, such as exception type, from the collected code snippets. If they found too many features, they removed redundant features using frequency and information gain thresholds. Moreover, they used noise detection approach proposed by Kim *et al.* [KZWG11] to eliminate data noises. They used the extracted features to train machine learning algorithms (*e.g.*, Decision Tree) to learn common logging practices. This approach was implemented in a tool called LogAdvisor. LogAdvisor can provide online suggestions based on different features of the code snippet.

Zhu *et al.* evaluated their approach by conducting several experiments. The authors showed that their approach can achieve 84.6% to 93.4% precision when they performed a 10-fold cross-valuation. They also compared different machine learning algorithms while they trained on their data. They found that Decision Tree achieves the best overall accuracy. Furthermore, they evaluated the impact of their noise-handling approach by comparing the prediction results with and without noise handling. They found that most of their data (88%) has a noise degree value close to 0, revealing the quality of the data. Nevertheless, the noise-handling approach improves the prediction accuracy of their approach. Lastly, they performed cross-project validation by training and testing their algorithms on a different project. By utilizing this approach, they were able to achieve 81.5% accuracy, compared to 93.4% in within-project learning.

While LogAdvisor shows promising results, it is only usable for C# projects. Moreover, our checker has very little overhead comparing to their multi-step framework. Our checker only needs the information about the exception types. Although we have a lower recall, we still do not give wrong suggestions in any of the issues. Furthermore, we aim to find multiple patterns rather than focusing on one, as Zhu *et al.* did in their study.

### 2.2.10 Li *et al.* [LSH17]

Li *et al.* [LSH17] proposed an approach which helps developers in choosing the best log level when they are adding a new logging statement. They studied the logging levels among four open source software projects: *Hadoop*, *Directory Server*, *Hama*, and *Qpid*. The authors extracted all the added logging statements of the systems from the historical data and extracted process and product metrics related to them. They started with a preliminary empirical study on the use of log levels in the studied subject systems. They found that although each project has a different distribution of log levels, no single log level dominates other ones in any of the studied projects. Furthermore, different code blocks have different distribution of log levels. For instance, logging statements directly added to catch blocks tend to have less verbose logging levels than the logging statements added to try blocks. They also found that not all the logging statements added to catch blocks have *warn*, *error*, or *fatal* level. These results demonstrate difficulties in suggesting logging levels. Inspired by their empirical study, they introduce a regression model to help developers choose appropriate log level for newly added logging statements. They used five sets of metrics to model the log levels, logging statement metrics, file metrics, change metrics, historical metrics, and containing block metrics. Furthermore, they re-encoded categorical metric to a quantitative variable, combined metrics with strong interaction, and removed strongly correlated metrics. Then, they build an ordinal regression model to suggest logging levels for a given logging statement.

They measured the performance of their model using multi-class Brier score and AUC metrics. Brier score is used to measure the accuracy of probabilistic predictions. AUC (area under the curve) is the area under the Receiver Operating Characteristic curve (i.e., the plot of the true positive rate against the false positive rate). The authors show that, while accounting for optimism, their ordinal regression model achieves a Brier score ranging from 0.44 to 0.66 and an AUC of 0.75 to 0.81. They also show that the characteristics of the containing block, the existing logging statements in the file, and the content of the newly-added log can play an important role in determining the appropriate logging level for the added logging statements.

As mentioned, we also proposed a log-level checker to help developers in identifying inappropriate log levels in their code. We demonstrate the comparison between our approach and the model introduced by Li *et al.* in Section 5.5.

### 2.2.11 Li *et al.* [LSZEH17]

Li *et al.* [LSZEH17] also provided a model to suggest just-in-time suggestions for code changes in logging statements occurring in each commit. The authors performed a manual study to investigate the rationales behind logging statement changes. They studied 32,480 logging statement changes from four open source subject systems. They sampled 380 log changes with 95% confidence level and ±5% confidence interval. Furthermore, they manually investigated the log change, code change, commit message, and the issue report, in order to understand the reason for a log change. They categorized the changed logs into four groups. Logging statements changed because of code changes in the containing block are marked as *bock change*. *Log improvement* category includes logging statements modified to achieve better logging practices. Another category includes logging statements which are changed because they depend on modifications on other code elements (*e.g.*, variables in a log). Finally, logging statements changed because of the issues reported by users and developers constitute the fourth category of log changes.

Driven by this formative study, the authors designed Random Forest classifiers to provide just-in-time suggestions for log changes. For each code change, the classifiers provide a binary respond that shows the likelihood of a log change occurring in a commit. They reported that their Random Forest classifiers can tell whether a log change is required for a commit or not, with an average accuracy of 0.76 to 0.82 even when using commits from other projects as training data. Furthermore, the authors show that their classifier can reach a medium classification power using 192 commits on average.

They also examined the results of their classifier to find the most influential factors that best explain log changes. They found that change measures and product measures are among the most influential factors. In fact, the strong influence of change and product measures indicates that log changes are highly associated with other code changes, and furthermore, the current snapshot of the source code plays an important role in the logging decision for developers. Last but not least, the authors found that different projects follow different log change practices. For instance, catch clause measure is one of the most influential factors in Hadoop, Directory Server, and Qpid projects, while in HttpClient, it ranks the 7th most influential factor.

In this work, we have focused on the *log-related issues* and studied their characteristics, while Li *et al.* looked into the rationale behind log changes. We also proposed a set of tools to help developers in avoiding straightforward log-related issues.

### 2.2.12 Li *et al.* [LCSH18]

Li *et al.* performed a case study on six open source Java systems: *Hadoop*, *Directory-Server*, *Qpid-Java*, *CloudStack*, *Camel* and *Airavata* to investigate the relationship between logging decisions and the functionality of the source code. The authors used topics as a proxy for the functionality of the code. For each system, they first extracted the methods in the source code and filter them base on size. Then, they removed the logging statements from the method before performing topic modeling. Afterwards, they extracted linguistic data such as variable names and code comments and performed LDA to unigrams as well as bigrams in each method.

They found that a small set of the extracted topics are much more likely to be logged. In fact, most of the log heavy topics are related to communication between machines or interaction between threads. Furthermore, the authors found that studied systems share a portion (12% to 62%) of their topics with each other. Meaning that developers can use other systems while making logging decisions or developing logging guidelines. Finally, the authors built regression models to study the relationship between the topics in a method and the likelihood of a method being logged. They found that adding topic-based metrics to baseline model result in improvements form 3% to 13% on AUC and 6% to 16% on Brier score. In fact, five to seven of the top ten important metrics for determining the likelihood of a method being logged were topic-based metrics. They showed that developers can use topically to predict the likelihood of a method being logged.

Li *et al.* suggest that there is a strong relationship between the topics of a code snippet and the likelihood of a code snippet containing logging statements. In essence, they focus on the problem of "where to log". We take a more general approach to improve the quality of the logging statements by providing insights on log-related issues. We also proposed static analysis tools to help developers in different aspects of logging such as "where to log" and "what to log" as well as "how to log". However, our tools focus on specific issues in the logging code rather than providing general solutions to these questions.

### 2.2.13 Kabinna *et al.* [KBSH16]

Kabinna *et al.* [KBSH16] studied the logging library migration in Apache Software Foundation (ASF) projects. The authors identified all the JIRA issues that attempt to migrate logging libraries. Then, they manually analyzed JIRA issues to find the rationales behind logging library migrations. They also collected churn metrics for the related commits as well as developers' metrics from the repositories of the subject systems to identify the effort needed to perform the migrations. The authors found 49 attempts to migrate logging libraries in 33 among the 233 projects under study. They also reported that on median, each logging library migration took 26 days to complete and in the majority of them (66%) at least one top contributor was involved during the migration. The improvements offered by the new library often motivates developers to justify the effort needed to migrate the logging library. The developers mentioned flexibility, performance improvements, code maintenance, functionality, and dependencies as their drivers for migrating logging libraries. Interestingly, 14 attempts for logging library migrations were abandoned by developers. In six projects the developers failed to provide the patches needed for migration. Two of the projects delayed their migration for a better release of the targeted library. Moreover, two projects faced dependency issues during their migration.

Kabinna *et al.* also found that the performance improvement is one of the primary reasons for developers for logging library migrations. Thus, they measured the performance improvements observed after the migrations. They run performance tests before and after each migration in "Debug" and "Info" log levels. Afterwards, they used Mann-Whitney U test to find statistically significant performance differences between pre- and post-migration tests. Furthermore, they also calculated the effect size to quantify the differences in performance. Two subject systems experienced 28-44% performance improvement while in two systems the improvements were negligible.

The authors found that 14% of the ASF projects had at least one logging library migration in their lifetime. They also showed that 70% the migrated projects had at least two issues related to the logging library migration. Although they provided useful insights for logging library migration, they do not propose an approach to automatically aid developers in the process.

Results of our study confirm the findings of Kabinna *et al.*'s. We showed that logging library migrations are one of the root causes of log-related issues. Furthermore, we found that logging library migrations introduce inconsistencies and new issues to the software systems. We also introduced a tool that can help developers in improving the performance of software systems by simply checking for the enabled logging level before executing the logging statement, while using specific logging libraries. Our checker can fix such common issues related to logging library configurations.

### 2.2.14 Kabinna *et al.* [KSBH16]

Kabinna *et al.* [KSBH16] also studied the stability of logging statements. First, they performed a preliminary study to find how often logging statements go under modifications in their lifetime. They used the historical data stored in the repositories of the subject systems to track the changes on logging statements. They performed their analysis on four open source projects namely ActiveMQ, Camel, Cloudstack, and Liferay. They found that 20% to 45% of the logging statements in the studied applications at least went through one modification. Furthermore, they observed that on median, logging statements change 17 times after they were introduced into the system. Motivated by the empirical results, the authors trained a Random Forest classifier based on metrics related to context, the content, and the developers of the logging statements to predict the likelihood of a log change in the future. Furthermore, the authors reported the performance of their classifier using precision, recall, AUC and Brier Score. They also calculated the optimism of their performance measures and reduced it from their evaluation results. The Random Forest classifier achieved 0.83 to 0.91 precision and 0.65 to 0.85 recall, leading to AUC of 0.94 to 0.95. They also achieved a Brier Score of 0.042 to 0.61 among the applications under study. Finally, the authors showed that developer experience, file ownership, SLOC, and log density are the most important metrics for predicting the stability of a logging statement.

Kabinna *et al.* focus on predicting the stability on the logging statement without considering the nature of the changes introduced. We aim to study faulty logging statements and provide useful insights on the characteristics of these logging statements and their surrounding code. Furthermore, we provided a set of tools to help developers to find and fix such issues.

### 2.2.15 Summary

Previous researches mainly focused on suggesting and improving existing logging statements. We, on the other hand, focus on studying log-related issues and aim to improve the quality of the logging statements by automatically detecting these issues. We provide a comparison of related work that aim to improve logging code in Table 1.

Table 1: Comparing our work with prior research on logging suggestions and improvements

| Study | Goal | Code Element Level | Notes |
|---|---|---|---|
| Learning to log: Helping developers make informed logging decisions [ZHF+15] | Adding or removing logging statements to catch blocks | Catch Block level, Return-value check | Providing a tool named Log Advisor to help developers of C# determine whether a log needed or not in a catch block. |
| Improving software diagnosability via log enhancement [YZP+12] | Adding variables to logging statements | Method Level | Injecting all accessible variables to the logging statement in the source code. |
| Characterizing and detecting anti-patterns in the logging code. [YPZ12a] | Finding anti-patterns in logging statement | Statement Level | Studying the anti-patterns in logging code in several Java projects by analyzing log changing commits. |
| Which log level should developers choose for a new logging statement? [LSH17] | Predicting verbosity level | Statement Level | Providing a model to predict appropriate logging level using code metrics. |
| Towards just-in-time suggestions for log changes [LSZEH17] | Predicting the need for log changes | Commit level | Built random forest classifiers using software measures to predicting the need of log change in each commit. |
| Studying and detecting log-related issues (This study) | Detecting log-related issues | Statement Level | Studying characteristics of the log-related issues and providing a checker to detect evident errors in logging code. |

# Chapter 3

# Study Design

In this section, we present our case study setup. In particular, we present the subject systems of our case study and their characteristics as well as our approach to select them. Furthermore, we explain our approach for collecting log-related issues.

## 3.1 Subject systems

Our case study focuses on two large-scale open-source software systems, namely *Hadoop* and *Camel*. To select our subject systems, we picked the top 1,000 most popular Java projects from Github based on the number of stars. Then, we cloned them and counted the number of logging statements in each project using the source code. To count the number of logging statements, we checked the types of the logger variables and whether their corresponding method calls (e.g., trace, debug, info, warn, error, fatal) are standard log librarie's levels. Then, we picked the top two software systems as our subjects. Our approach to select the subject systems is depicted in Figure 1.

*Hadoop* is a well-known parallel computing platform that implements the MapReduce paradigm. *Hadoop* has been widely adopted in practice. *Hadoop* is written in Java with around two million SLOC and nearly 33K issues stored in its issue tracking system for all of its sub-systems. *Camel* is an open-source integration framework based on known Enterprise Integration Patterns with Bean Integration containing more than 1.1 million SLOC and 10K issues in its issue tracking system. Like all other products of Apache, *Hadoop* and *Camel* use JIRA as their issue tracking system. Both subject systems have extensive logging statements in their code and logs are heavily used in their development and operation activities. In particular, *Hadoop* has more than 11K logging statements and *Camel* has more than 6K logging statements in their latest revision of source code.

## 3.2 Collecting log-related issues

In order to conduct the study, we first needed to collect log-related issues from issue tracking system of Camel and Hadoop. There exists no explicit flag in *JIRA* issue reports that label an issue as a log-related issue. Thus, we extracted all available issue reports of our subject systems to performed further investigation and extracted issues that are more likely to be related to logging mechanism of the subject systems. Then, we leveraged a keyword based heuristic to filter such issues, by searching for keywords like *log*, *logging*, or *logger*. Note that we performed our filtering based on the issues' summary rather than issues' description. Issue descriptions usually provide more information about each report. However, because of the nature of logging, many reports include logging related keywords. In fact, logging is one of the most important tools in debugging activities and most of the issue reports include logs and its related keywords in their description. However, most of the issues only use logging as a tool to fix other problems and are not related to logging mechanism of the system. Moreover, we only selected the issues that are labeled as *bug* or *improvement* and that are also *resolved* and *fixed*. We only used fixed and resolved issues since we required the corresponding fix for these issues to understand their characteristics and the root-causes. We included the issues with label *improvement* because, from our preliminary manual exploration of the issue reports, we found that many log-related issues are labeled as *improvement* while they were in fact bugs.

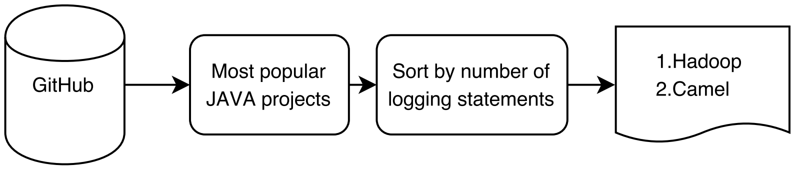

GitHub
Most popular JAVA projects
Sort by number of logging statements
1.Hadoop
2.Camel

Figure 1: An overview of our approach to select the subject systems

Table 2: The Number of Issues in Hadoop and Camel

| **Subject systems** | **# all fixed issues** | **# Issues with log-related keywords** | **# Manually verified** |
|---|---|---|---|
| Hadoop-HDFS | 3,863 | 253 | 178 (4.6%) |
| Hadoop-Common | 5,999 | 221 | 170 (2.8%) |
| Camel | 6,310 | 163 | 85 (1.3%) |
| Hadoop-YARN | 1,542 | 133 | 71 (4.5%) |
| Hadoop-MapReduce | 2,906 | 145 | 61 (2.1%) |

For example, in HADOOP-8075[1], a developer reports that "Lower native-hadoop library log from info to debug". The title clearly shows that the log level in this case is wrong. However, this issue

[1]https://issues.apache.org/jira/browse/HADOOP-8075

is labeled as an improvement in the system. Since we wanted to study the issues that are related to logging, but not the corresponding new logging with new features, we also excluded other issue types like *Task* or *Sub-task* that are usually used to implement new features rather than fixing a bug. Afterwards, we further verified each issue to make sure they are indeed log-related issues. For example, we do not include the issues if developers added new functionality to the code while modifying logging statements since the modification is due to the functionality change instead of an issue related to the logging statement itself. We also exclude the issues that are not fixed or not closed as well as duplicated and invalid issues. Eventually, 563 log-related issues remained, which we manually investigated (Table 2). For each issue, we checked the issues report and their discussion. Furthermore, we investigated their corresponding fix for the issue. We also recorded the information regarding the people interacting with the issue as well as the priority, report date, and fix date.

# Chapter 4

# Empirical Study

In this section, we present our case study results by answering four research questions. For each research question, we show the motivation of the research question, our approach to answer the question and the corresponding results. Figure 2 presents an overview of our approach to answering the research questions.

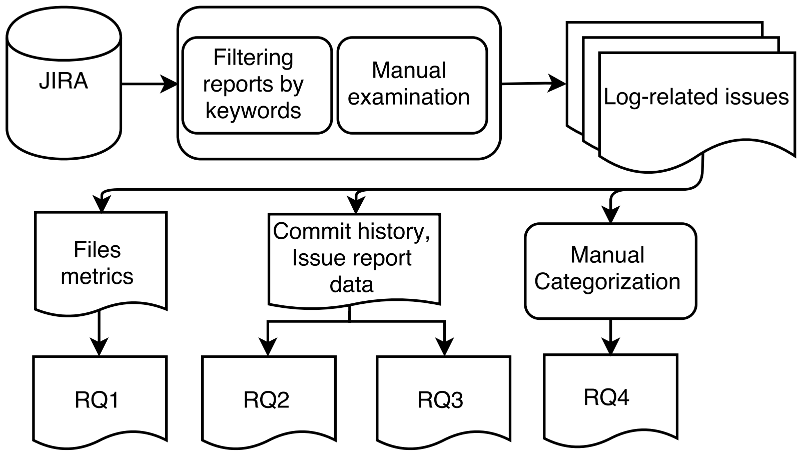


Figure 2: An overview of our approach to answer the research questions

## 4.1 RQ1: What are the characteristics of files with log-related issues?

### 4.1.1 Motivation

The first step towards understanding log-related issues is to find out where they are located. In this research question, we study the characteristics of files that contain log-related issues. Knowing these characteristics might help developers prioritize their efforts when identifying and fixing log-related issues.

### 4.1.2 Approach

To answer this research question, we first extracted the files related to each issue according to its fix. Then, we calculated the following product and process metrics for Java files with and without log-related issues.

- Normalized source lines of code (NSLOC): We use SLOC to measure the size of a file. We do not calculate a complexity metric since, as previous studies have shown before, most of the software complexity metrics are highly correlated with SLOC [HRGB$^{+}$06, HH10, Zha09]. However, larger files tend to contain more logging statements [SNH15]. Having more logging statements increases the probability of having more log-related issues. Thus, we normalized SLOC by the number of logging statements in each file.

- Fan-in: We used fan-in to measure dependency between files. Fan-in measures the number of files that depend on a given file. To calculate fan-in, we first constructed the call graphs of all the methods in each file using an open source tool named "java-callgraph"[Gou17]. Then, we counted the number of methods from other files that call methods from a particular file using the call graph. Files with higher fan-in values have more files in the system depending on them, and thus, have more impact on the system. By calculating the Spearman correlation between Fan-in and number of logging statements in a file, we find that the correlation is low (0.19). Thus, we did not normalize fan-in with the number of logging statements in the files.

- Frequency of prior commits: We used the frequency of prior commits to measure the stability of the files. Operators may need better logs to be aware of the changes on the files that are less stable. We used the total number of prior commits of each file divided by the lifetime length (in number of days) of the file to calculate the frequency of prior commits. The lifetime length of the file is calculated by measuring the time difference between the first commit of the file and the date when we extract data from the Git repository.

- Frequency of prior bugs: We also used the frequency of prior bugs to measure the quality of the files. Developers may depend on logs to ensure the quality of these files. Same as the frequency of prior commits, we use the lifetime length to normalize the total number of prior bugs of a file. We used the JIRA reports for each subject system to collect the number the prior bugs of each file.

Note that we did not include test files since we only wanted to focus on the production code. We used statistical tests to compare metrics between files with log-related bugs and without log-related bugs. More specifically, we used a two-tailed statistical test, namely the Wilcoxon rank-sum test [WW64]. We performed four comparisons on each dataset. To better control for the randomness of our observations, we used Bonferroni correction [DMCSO05]. We adjust our p-value by dividing it by the number of comparisons (four). The results are significant at the significance level alpha $= 0.05/4$ (p-value $< 0.0125$). This shows that the two populations are different. However, studies have shown that when the size of the populations is large, the p-value will be significant even if the difference is very small. Thus, we calculated the effect size using Cliff's delta [KDHS07, CSJ+14] to measure how large the difference between two populations is. The value of Cliff's delta ranges from zero to one. According to Kampenes *et.al.* [KDHS07], Cliff's delta values can be interpreted as shown in Table 3:

Table 3: Cliff's delta effect size interpretation.

| **Effect size** | **Cliff's delta value** |
|---|---|
| Trivial | if $Cliff's\ d \leq 0.147$ |
| Small | if $0.147 < Cliff's\ d \leq 0.33$ |
| Medium | if $0.33 < Cliff's\ d \leq 0.474$ |
| Large | if $0.474 < Cliff's\ d$ |

### 4.1.3 Results

Table 4 presents the median of our studied metrics for files with and without log-related issues. We found that for all subject systems in our case study, files with log-related issues have statistically significantly more prior bugs and prior commits with large effect sizes. However, the difference of our product metrics (NSLOC and fan-in) with and without log-related issues is either statistically indistinguishable or their effect sizes are small or trivial (except for fan-in for *Camel* and *Hadoop-Yarn*). These results imply that files that are more actively under development or bug fixing tend to contain more log-related issues.

However, we found that a large portion of the files does not include any logging statements in them. Thus, they are less likely to have any log-related issues in them. In order to reduce the impact of these files on our results, we also calculated mentioned metrics only for the files with at least one logging statement. Table 5 presents the median of our studied metrics for files with and without log-related issues which at least include one logging statement in them. The ratio of files with logging statements are mentioned in Table 5 subject system. We found that similar to the previous results, files with log-related issues have statistically significantly more prior bugs and prior commits with medium to large effect sizes. However, the difference of our product metrics (NSLOC and fan-in) with and without log-related issues is statistically indistinguishable or their effect sizes are small or trivial (except for fan-in only for *Hadoop-Yarn*). This implies that although removing files without logging statements reduced the effect sizes, the difference is still significant in process metrics.

One possible reason can be that changes and bug fixes in the files make the code inconsistent with the logging statements in the files. Thus, the logging statements become outdated and eventually are reported as issues. In our manual study in RQ4, we found one file called `FSNamesystem.java` with 6K SLOC, 51 contributors and 250 issues, of which 12 are log-related. One of these log-related bugs[1] was specifically reported to clean-up the unnecessary logging statements in the file that became outdated and the corresponding source code no longer existed in the system. In the discussion of another log-related issue in *HDFS*[2], developers mention that "the comments and logs still carry presence of two sets when there is really just one" which specifically shows that the source code and logging statements are inconstant. The results suggest that after finishing development or bug fixing tasks, developers may consider verifying the consistency of the source code and the logging statements to reduce such log-related issues.

**RQ1 Conclusions**: Files with log-related issues have undergone statistically significantly more frequent prior changes, and bug fixes. Developers should prioritize effort of maintaining logging statements on these files.

## 4.2 RQ2: Who reports and fixes log-related issues?

### 4.2.1 Motivation

RQ1 shows that log-related issues often occur in files with less stable source code. Experts of these files may be one of the most important vehicles to ensure the quality of logs. Prior research demonstrates the importance of experts in resolving these log-related issues [SNHJ14]. Furthermore, studies show the importance of developer ownership and its impact on code quality [BNM+11].

[1]https://issues.apache.org/jira/browse/HDFS-9528
[2]https://issues.apache.org/jira/browse/HDFS-2729

Table 4: Medians (*Med.*) and effect sizes (*Eff.*) of the normalized process and product metrics for files with and without log-related issues. Effect sizes are not calculated if the difference between files with and without log-related issues is not statistically significant.

| **Metric** | **Type** | **Camel** | | **Common** | | **Yarn** | | **HDFS** | | **Mapreduce** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Med.** | **Eff.** | **Med.** | **Eff.** | **Med.** | **Eff.** | **Med.** | **Eff.** | **Med.** | **Eff.** |
| Fan-in | With log-related bug | 8.0 | Medium(0.37) | 15.0 | - | 14.0 | Large (0.54) | 27.5 | Small (0.27) | 7.5 | - |
| | Without log-related bug | 3.0 | | 15.0 | | 5.0 | | 12.5 | | 5.0 | |
| NSLOC | With log-related bug | 21.3 | - | 24.1 | - | 21.2 | Small (-0.16) | 24.0 | - | 30.4 | - |
| | Without log-related bug | 20.4 | | 25.0 | | 20.0 | | 27.0 | | 25.0 | |
| Frequency of Prior Commits | With log-related bug | 0.014 | Large (0.49) | 0.008 | Large (0.68) | 0.026 | Large (0.84) | 0.018 | Large (0.63) | 0.017 | Large (0.93) |
| | Without log-related bug | 0.005 | | 0.002 | | 0.003 | | 0.004 | | 0.001 | |
| Frequency of Prior Bugs | With log-related bug | 0.002 | Large ( 0.83) | 0.004 | Large (0.73) | 0.008 | Large (0.88) | 0.007 | Large (0.74) | 0.006 | Large(0.91) |
| | Without log-related bug | 0.000 | | 0.001 | | 0.000 | | 0.001 | | 0.001 | |

Table 5: Medians (*Med.*) and effect sizes (*Eff.*) of the normalized process and product metrics for files with and without log-related issues. Files without logging statements are excluded. Effect sizes are not calculated if the difference between files with and without log-related issues is not statistically significant. The percentage of files with log-related bugs shown in front of each subject system.

| **Metric** | **Type** | **Camel (19%)** | | **Common (22%)** | | **Yarn(18%)** | | **HDFS (27%)** | | **Mapreduce (16%)** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Med.** | **Eff.** | **Med.** | **Eff.** | **Med.** | **Eff.** | **Med.** | **Eff.** | **Med.** | **Eff.** |
| Fan-in | With log-related bug | 5.0 | - | 19.0 | - | 14.0 | Large (0.48) | 28.0 | - | 9.0 | - |
| | Without log-related bug | 4.0 | | 23.00 | | 7.0 | | 16.0 | | 6.0 | |
| NSLOC | With log-related bug | 19.2 | - | 30.3 | Small (-0.29) | 22.3 | - | 24.9 | Small (-0.18) | 35.1 | - |
| | Without log-related bug | 24.0 | | 38.0 | | 27.8 | | 35.7 | | 39.0 | |
| Frequency of Prior Commits | With log-related bug | 0.015 | Medium(0.36) | 0.008 | Medium (0.40) | 0.031 | Large (0.67) | 0.021 | Medium (0.41) | 0.017 | Large (0.75) |
| | Without log-related bug | 0.008 | | 0.004 | | 0.006 | | 0.010 | | 0.004 | |
| Frequency of Prior Bugs | With log-related bug | 0.002 | Large ( 0.63) | 0.005 | Large (0.48) | 0.009 | Large (0.71) | 0.007 | Large (0.55) | 0.007 | Large(0.79) |
| | Without log-related bug | 0.001 | | 0.002 | | 0.001 | | 0.002 | | 0.001 | |

Studies showed that when more people are working on a file, it is more likely to have failures in the feature [BND$^+$09, NMB08]. Therefore, if experts of the log-related issues can be identified, these issues can be fixed with less impact. Therefore, in this research question, we investigate people involved during the lifetime of log-related issues.

### 4.2.2 Approach

To answer this research question, we first needed to know who introduced the logging statement. Thus, for all the log-related issues in Java files, we first searched for JIRA issue IDs in Git commit messages to identify the commit that fixes the issue. Some log-related issues do not have their issue ID mentioned in a commit message. In particular, we can only find commits for 254 of the log-related issues. Then we analyze the history of the files, which contain the logging statements and are changed in the commit, to identify the commit where the logging statement was introduced. We performed our analysis on 1,071 logging statements extracted from these issues fixing commits in our case study.

Furthermore, in our subject systems, the committer of each commit is usually not the actual author of the commit. Instead, the author information is mentioned in the commit message. To extract the author names in the commit message, we looked for names after with terms like *"Thanks to"*, *"Contributed by"*, or *"via"*. Whenever we could not find the names using these heuristics, we tried to find the issue key from the commit message and use the assignee of that issue as the original author of the commit. Finally, if we could not find any links in the message, we use the committer as the actual author of that commit. In total, we only used the committer as the actual author in 12% of the commits. We identified the developers who introduced the logging statements, and we counted the prior number of commits by developers to measure the expertise and the ownership of the code in the repositories. Figure 3 demonstrates the lifetime of an inappropriate logging statement. Based on the Figure 3, we named the author of the commit that added the logging statement to the system (A) as the introducer and the author of the commit that fixes a reported log-related issue by modifying the logging statement (D) as the fixer. Furthermore, we named the top contributor of the file which contains the logging statement the owner of the file [BNM$^+$11].

### 4.2.3 Results

We found that 78% of the time, logging statements are introduced and are fixed by different people. Furthermore, 78% of the log-related issues are fixed by someone other than the owner of the file that contains the logging statement. Moreover, 73% of the fixes to log-related issues are done by the same person who reported the issue (57% of the all the issues). The results show that one may report and fix a logging statement not being an owner of the file nor the person who introduced the logging statement initially. Such findings suggest the lack of systematic ownership of the logging statements. On one hand, the developers who introduce the file realize the importance of placing the particular logging statement in the source code [SNHJ14]. On the other hand, once the logging statements are in the source code, other people would observe the value in the logs and start to depend on these logs in their daily activities. Hence, the users of these logs also have valuable knowledge about what should be included/or not in these logs. Our results show that there are cases when the original author of the logging statement may not understand the needs of other users of the log, leading to the report of log-related issues. However, the users of logs who do not own the file nor initially introduced the logging statement may change the logging statement without notifying the owner of the file or the original developer who introduced the logging statement. Such update may become a log-related issue that causes other people's log analyses to fail [SJA+14, SJA+11].

**RQ2 Conclusions**: Developers contributing to the log-related issues are usually not the developer who introduced the log or the owner of the code containing the logging statements. It is difficult to identify the expert for each logging statement. Thus, the impact of log-related issues cannot be minimized by referring to their experts.

## 4.3 RQ3: How quickly are log-related issues reported and fixed?

### 4.3.1 Motivation

The results of RQ1 and RQ2 illustrate the potential impact of log-related issues and the challenges of mitigating them by experts. Practitioners, such as dev-op engineers, who use the information in logs usually do not have access to the source code. Thus, a simple mistake like wrong verbosity level in a logging statement can hide important information from them. If the logging statements with these issues stay in the software for a long time, they become considerably harmful since they are more likely to impact all the people who depend on them. Whereas, if log-related issues are diagnosed and fixed easily, they might not be as harmful. Therefore, in this research question, we study the time required to report and fix log-related issues.

### 4.3.2 Approach

We aimed to find out how fast log-related issues were reported and fixed. Figure 3 demonstrates the lifetime of an inappropriate logging statement which ended up being reported as a bug. Using the results of our analysis on the history of changes for each logging statement, we estimate how fast log-related issues were reported by calculating the time difference between when the logging

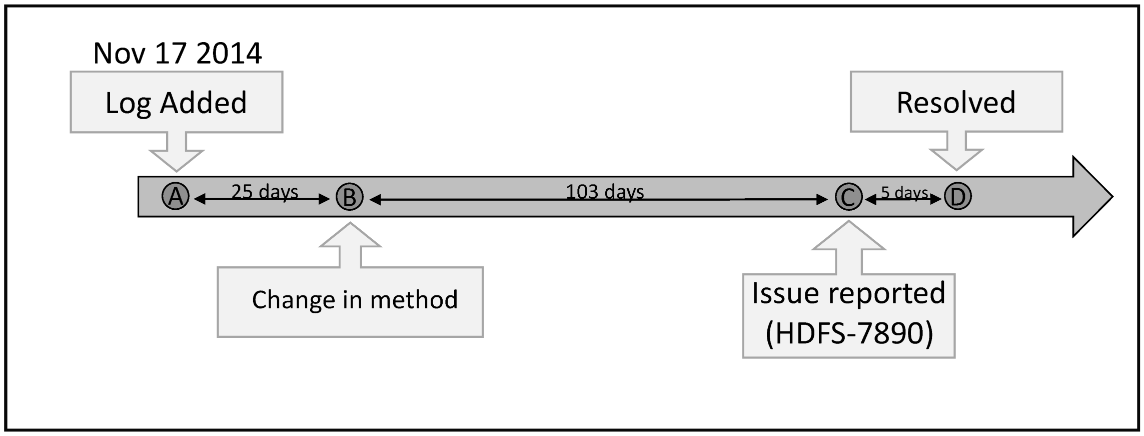


Figure 3: Lifetime of an example inappropriate logging statement

statement is introduced to the time when it is reported in the issue tracking system (Figure 3, A to C). Furthermore, we estimate how fast log-related issues were fixed by calculating the time difference between when the log-related issue is reported and when it is fixed (C to D)

### 4.3.3 Results

The results are depicted in Figure 4 and Figure 5 . We found that more than 80% of the issues were fixed in less than 20 days. In fact, 43% of all issues were fixed within two days of the submission date. Our results suggest that most of these issues are simple and easy to fix once they are found. In our manual analysis in RQ4, we observed that the associated code changes usually include less than ten lines of code, suggesting that the lifetime of these issues mostly involved in code review and tests.

However, inappropriate logging statements exist for a long time in the system before being reported as an issue. Table 7 shows the five-number summary of the number of changes for each logging statement. We can note that the median number of changes is two, where one of them is the commit that fixed the issue. This result suggests that most of the inappropriate logging statements are not the ones that are frequently changed.

Table 6 also shows the long time difference between the introduction of the logging statement and when the issue was reported. Other than *Hadoop-MapReduce*, on median, it takes 229 to 615 days to expose a log-related issue. For example, in HDFS-7890, a developer reported that *"Information on Top users for metrics in RollingWindowsManager should be improved and can be moved to debug. Currently, it is INFO logs at namenode side and does not provide much information."*. We found that this logging statement was added 103 days before the report date and did not change until another developer fixed it and changed the level to debug. Although the fix was small (only one change), it took a long time for developers to figure out that this logging statement is at an inappropriate level. However, it took only five days to fix after it was reported. Figure 3 shows all the changes made to a logging statement during its lifetime which led to issue HDFS-7890.

Table 6: Number of days before an inappropriate logging statement being reported

| **Subject systems** | **Min** | **1st Qu.** | **Median** | **3rd Qu.** | **Max** |
|---|---|---|---|---|---|
| Common (changes) | 0.17 | 159.9 | 459.3 | 482.4 | 1516.0 |
| HDFS (changes) | 0.17 | 41.4 | 229.2 | 431.3 | 1576.0 |
| YARN (changes) | 0.17 | 258.2 | 615.8 | 959.4 | 1357.0 |
| MapReduce (changes) | 0.17 | 41.67 | 41.67 | 91.1 | 1850.0 |
| Camel (changes) | 0.17 | 61.7 | 390.1 | 423.6 | 2689.0 |

Table 7: Number of changes before an inappropriate logging statement get fixed

| **Subject systems** | **Min** | **1st Qu.** | **Median** | **3rd Qu.** | **Max** |
|---|---|---|---|---|---|
| Common (changes) | 1 | 2 | 2 | 2 | 5 |
| HDFS (changes) | 1 | 2 | 2 | 4 | 10 |
| YARN (changes) | 1 | 2 | 2 | 2 | 6 |
| MapReduce (changes) | 1 | 2 | 2 | 2 | 5 |
| Camel (changes) | 1 | 2 | 3 | 2 | 10 |

Furthermore, we analyzed the priority of log-related issues and found that more than 46% of the log-related issues are labeled as Major, Critical, or Blocker. Thus, many of these issues are not likely to be the ones that developers are not interested in reporting and fixing them. The long time needed to expose a log-related issue signifies the potential harm of these issues over such long time periods, people might make decisions based on the inappropriate or incomplete information provided by the logs. These results illustrate the need for **automated tools** that detect such log-related issues in a timely manner.

**RQ3 Conclusions**: It takes a long time for log-related issues to surface and be reported. However, most of the log-related issues took less than two weeks to fix. Automated tools are needed to assist developers in identifying log-related issues in a timely manner.

## 4.4 RQ4: What are the root-causes of log-related issues?

### 4.4.1 Motivation

Previous RQs showed the need for automated tools to assist developers in finding inappropriate logging statements in code. Automatic tools can use the historical data and other information in the system to provide useful suggestions. Thus, we decided to perform a manual investigation on the root causes of the log-related issues, such that we gain a deeper understanding of log-related issues and find repeated patterns that can automatically expose evident log-related issues in the source code.

### 4.4.2 Approach

To answer this research question, we used the issue reports and their code changes we extracted from JIRA. Stol *et al.* [SRF16] suggest that researchers should describe how they analyzed data rather than dressing it up as other well known scientific approaches. To avoid method slurring [BWS92], we explain our approach in details in this section.

We started to examine log-related issues based on their title, description, and other information stored in every issue report. The first two authors independently read all the comments and discussions in each issue report and manually investigated the patches that fix the issue. Then, they grouped log-related issues into categories based on their root causes. More specifically, we manually examined the issue report, discussion and the paths for each log-related issue and added a summary and related key-words to them. Then, issue reports were labeled based on all the information in the related artifacts. Then, we revisited the extracted information and grouped similar labels into categories. Next, based on our observations from previous iterations, similar categories were merged into a new one. This process was repeated iteratively until the categories cannot be merged anymore.

In case of conflict, a proper label is selected after a discussion between the first two authors.

### 4.4.3 Results

The results of our manual study are shown in Table 8. We categorized log-related issues to seven categories based on their root causes namely, inappropriate log message, missing logging statements, inappropriate log level, log library configuration issues, runtime issues, overwhelming logs, and log library changes. We will discuss each category in details. In Table 9, we show the distribution of each of the mentioned types of log-related issues.

**Inappropriate log message.** As shown in Table 8, logging statements with incorrect log messages constitute the majority of log-related issues. We consider every issue regarding log messages (such

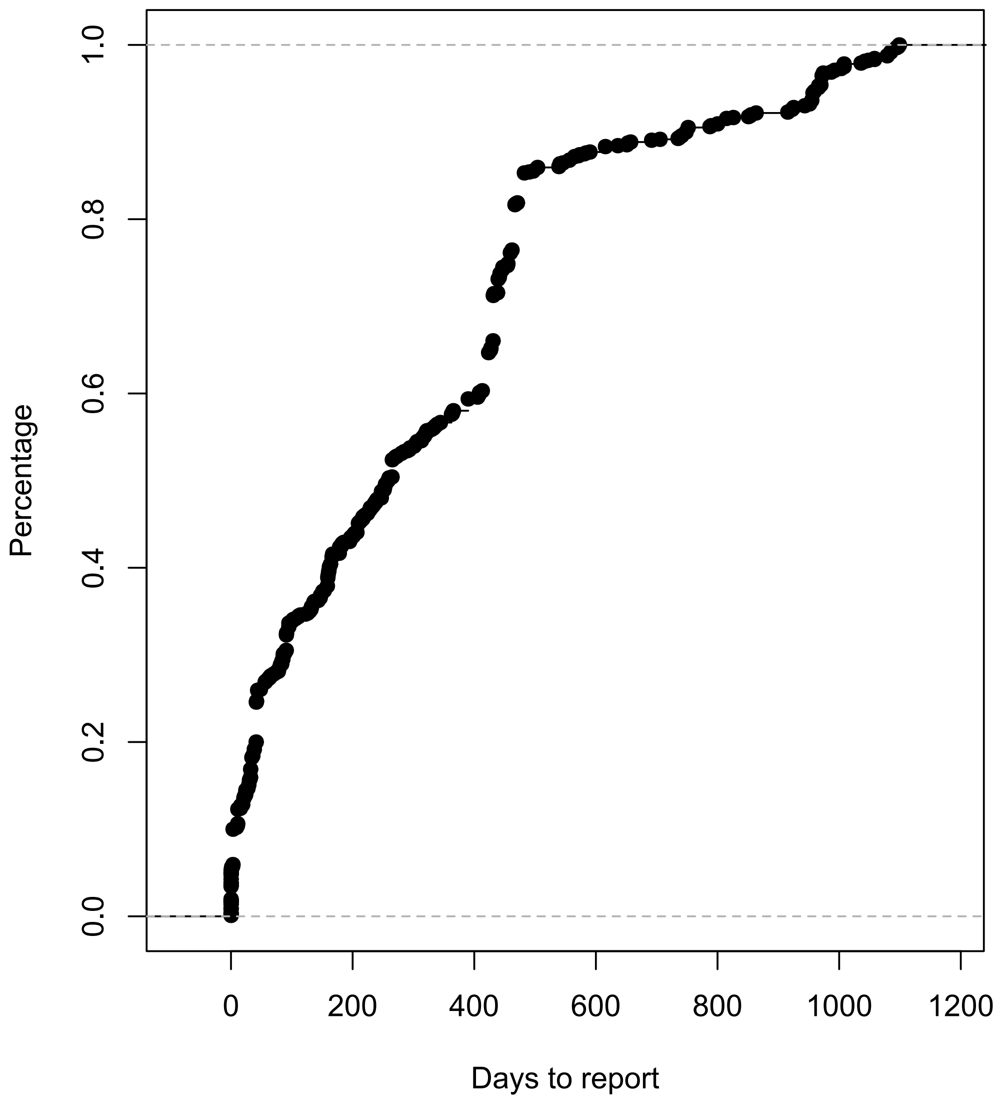


Figure 4: Cumulative distribution of issue report time in days. Outliers that are greater than 1.5 time of the value of the third quartile of the data are not shown in this figure.

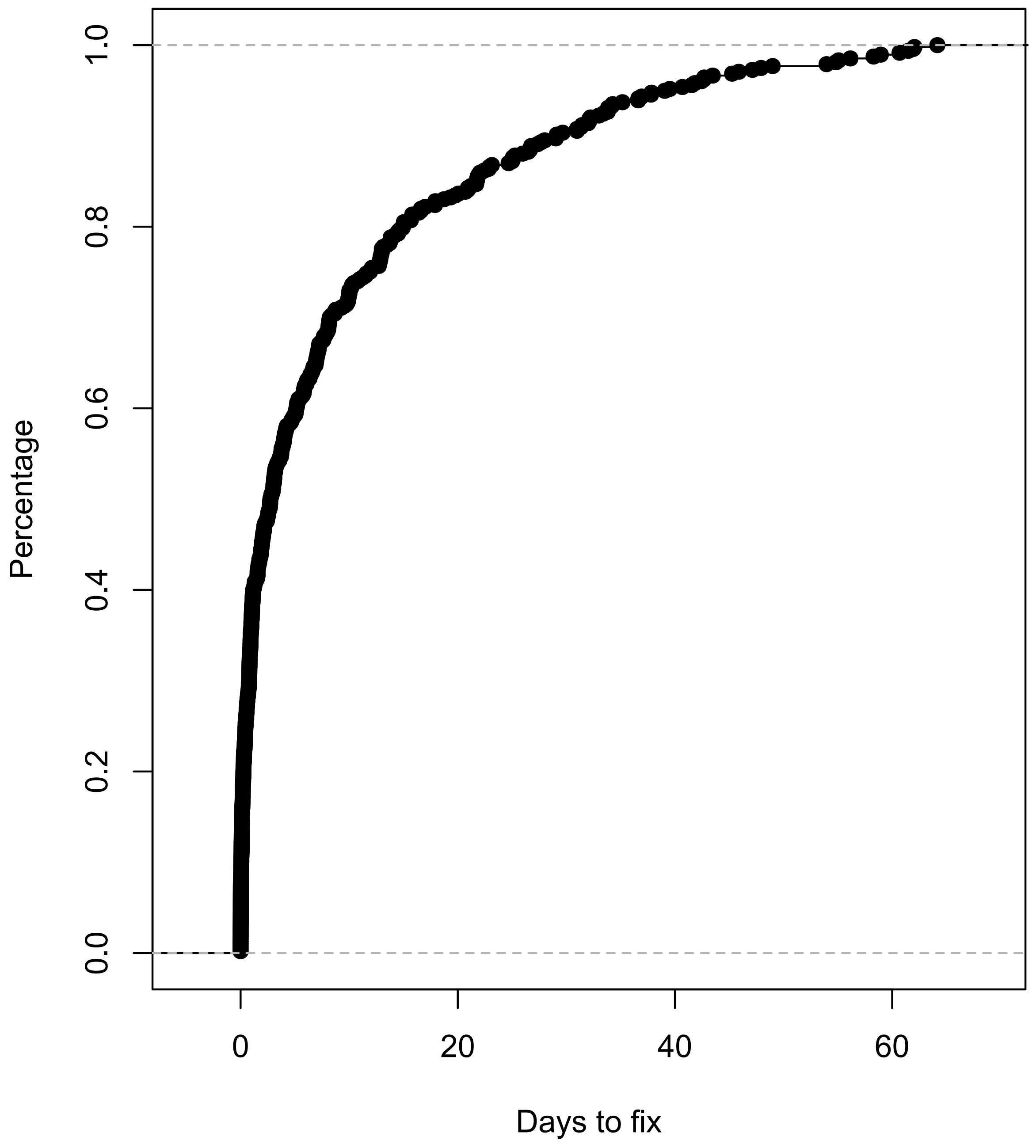


Figure 5: Cumulative distribution of issue fix time in days. Outliers that are greater than 1.5 time of the value of the third quartile of the data are not shown in this figure.

Table 8: Categories of log-related issues

| **Category** | **# of log-related issues** | **Example** |
|---|---|---|
| Inappropriate log messages | 182 | HADOOP-2661,“Replicator log should include block id” |
| Missing logging statements | 110 | HADOOP-5365, “Currently, only failed accesses are logged. Need to log successful accesses as well.” |
| Inappropriate log level | 96 | HADOOP-3399, “A debug message was logged at info level” |
| Log library configuration issues | 70 | HADOOP-276,“The problem is that the property files are not included in the jar file.” |
| Runtime issues | 53 | HADOOP-7695, “RPC.stopProxy can throw an configuration file for unintended exception while logging error” |
| Overwhelming logs | 35 | HADOOP-3168 ,“reduce the amount of logging in Hadoop streaming” |
| Log library changes | 19 | HADOOP-211, “it’s a huge change from older ones to common logging and log4j” |

Table 9: Distribution of the root causes of log-related issues

| **Subject system** | **Inappropriate log messages** | **Missing logging statements** | **Inappropriate log level** | **Log library configuration issues** | **Runtime issues** | **Overwhelming logs** | **Log library changes** |
|---|---|---|---|---|---|---|---|
| Hadoop-Common | 26.2% | 14.9% | 14.9% | 17.9% | 14.3% | 7.1% | 4.8% |
| Hadoop-HDFS | 34.8% | 21.9% | 16.3% | 10.1% | 5.1% | 8.4% | 3.4% |
| Hadoop-YARN | 40.8% | 16.9% | 22.5% | 5.6% | 7.0% | 5.6% | 1.0% |
| Hadoop-Mapreduce | 41.0% | 18.0% | 14.8% | 13.1% | 4.9% | 6.6% | 1.6% |
| Camel | 25.9% | 25.9% | 20.0% | 11.8% | 5.9% | 8.2% | 2.4% |

as missing or incorrect variable, or incorrect string literals) in this category. As an example, in the issue HADOOP-2661, developers mentioned *"Replicator log should include block id"*. Here, the developers asked to add the missing information (i.e., block ID) to the log message.

**Missing logging statements.** There were some cases where developers requested additional logging statements or asked for logging a specific event that had not been logged. We considered all corresponding issues that are fixed by adding logging statements as *Missing logging statements*. HADOOP-5365 is an example of this type of issue, where the developers asked to add new logging statements to capture more information: *"Currently, only failed accesses are logged. Need to log successful accesses as well."*.

**Inappropriate log level.** Another interesting type of issues was problems associated with the *level* of the log. Log messages have levels that show their importance, verbosity, and what should happen after the event is logged. These levels include *fatal* (abort a process after logging), *error* (record error events), *info* (record important, but normal events), *debug* (verbose logging only for debugging), and *trace* (tracing the steps of the execution, the most fine-grained information). Developers use log levels based on the information that they need to print, and considering the overhead that more verbose log messages can impose on the system's execution. These log levels are widely used by analysis tools and operators to filter out unwanted logs and extract relevant information. In some issues, the level of a logging statement was thought to be incorrect and needed to be changed. For example, in the issue HADOOP-3399, the developers clearly mentioned that *"A debug message was logged at info level"*. Setting a lower log level could cause missing important information in the execution log output. In contrast, setting a higher log level will add redundant information to it. In other words, setting an inappropriate log level may lead to confusing log messages.

**Log library configuration issues.** Developers use different APIs in their software system's lifetime to print log messages. Each API uses different configuration files and interfaces to perform logging in the system. We consider any problem in the implementation and configuration of the used APIs as an *Configuration issue*. For instance, in the issue HADOOP-276, developers found out that they needed to add a configuration file for log4j, as the description of the issue reads *"The problem is that the property files are not included in the jar file."*

**Runtime issues.** A considerable number of log-related issues are *runtime issues*. We consider an issue to be on this category if it causes a runtime failure or misbehavior of the system at execution time. For example, in the issue Hadoop-7695, the developers mentioned that *"RPC.stopProxy can throw a configuration file for unintended exception while logging error"*. Here, the developers logged a variable that can throw *Null Pointer Exception*, and the issue was introduced to ask the developers to check the value of the variable against `null`, before logging it.

**Overwhelming logs.** In contrast to *Missing logging statements*, in some issues, the developers requested to remove a logging statement since it was useless, redundant or made the log output noisy. As an example, in HADOOP-3168 one of the developers mentioned that *"reduce the amount of logging in Hadoop streaming"*. In order to fix this specific issue, the developers removed log messages until they reached one log message per 100,000 records since the information of all the records was useless.

**Log library changes.** Eventually, the last category of log-related issues contains the changes that were requested from developers to change or upgrade the logging API in their system (e.g., for upgrading to a newer version of the logging library, *log4j*); these changes fall in the corresponding category.

Based on our experience from the manual investigation, we found repeated patterns in the log-related issues. Some of the patterns are trivial and evident patterns, which raise the opportunity of automatically detecting potential inappropriate logging statements. These patterns can help us develop approaches to automatically expose inappropriate logging statements in the source code. In the next section, we will demonstrate our approach to detect these issues.

**RQ4 Conclusions**: We categorized log-related issues into seven categories based on their root causes. We observed evident root-causes of the log-related issue during our manual investigation. Such evident root-causes show the opportunity of making automated tools to detect log-related issues.

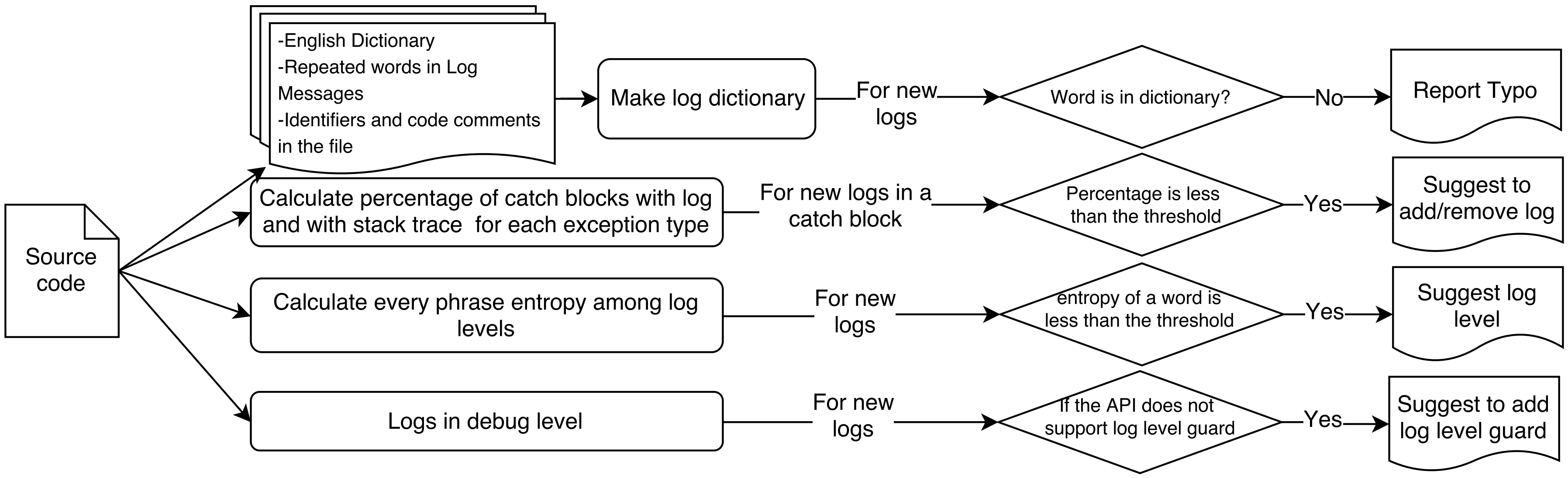
Chapter 5

# Automatic detection of inappropriate logging statements

In our empirical study, we found that although log-related issues are likely to be impactful, they are reported much later than the introduction of the logging statement. Our study results indicate the need for automated tools to help developers detect log-related issues in their code.

Based on our results of the manual study on log-related issues, we found that some of these issues are evident and are due to careless mistakes. We found some patterns and similar suggestions to automatically find defect-prone logging statements and show developers what may cause an issue in such statements. We built four different checkers for four types of log-related issues: Typos, missed exception messages, log level guards, and incorrect log levels. All these four types are evident in root-causes that are identified in RQ4.

- **Incorrect log levels** As explained in RQ4, messages with incorrect log level can make issues, such as providing too little or too much information, to users of the logs.
- **Missed exception message.** Missed exception messages are the catch blocks that do not contain any logging statements, or do not log the exception message inside them. Missed exception message can belong to missing logging statement category or inappropriate log message.
- **Log level guards.** Log level guard issues happen when there is an expensive computation in log messages and developers do not check which level is enabled in the configuration before execution. Log level guard belongs to log library configuration issues.
- **Typos.** As a subset of inappropriate log message category, *typos* are simple mistakes in spelling inside the log strings.



Source code
-English Dictionary
-Repeated words in Log Messages
-Identifiers and code comments in the file
Make log dictionary
For new logs
Word is in dictionary?
No
Report Typo
Calculate percentage of catch blocks with log and with stack trace for each exception type
For new logs in a catch block
Percentage is less than the threshold
Yes
Suggest to add/remove log
Calculate every phrase entropy among log levels
For new logs
entropy of a word is less than the threshold
Yes
Suggest log level
Logs in debug level
For new logs
If the API does not support log level guard
Yes
Suggest to add log level guard

Figure 6: An overview of log-related issue checker.

We will explain how these checkers are designed and the issues that we were able to detect using these checkers. An overall summary of our approach is depicted in Figure 6.

## 5.1 Log level checker

In our empirical study, we found that 76 issues are due to incorrect log level in logging statements, which were fixed by merely changing the log level. Figure 7 demonstrate a patch to such issues. As depicted in Figure 7, they only change needed to fix the issues is a change in verbosity level. To suggest log levels, we focused on the rich dataset of all log messages in the source code of our subject systems. *Hadoop* and *Camel* contain more than 10K and 6K logging statements in their source code, respectively. Thus, we tried to use the text in the log message to suggest the level of the log.

```
-          LOG.info("Loaded the native-Hadoop library");
+         LOG.debug("Loaded the native-Hadoop library");
//…
 //...
-          LOG.info(mode.toString() + "Loaded TrustStore:"
-                     + truststoreLocation);
+         LOG.debug(mode.toString() + "Loaded TrustStore:"
+                    + truststoreLocation);
```

Figure 7: A patch to fix inappropriate logging level

Information Theory deals with assessing and defining the amount of information in a message [Yin13]. The theory seeks to reveal the amount of uncertainty of information. For example, consider that we analyze the words and the combination of words in logging statements. To ease the explanation, we call words and combination of words as phrases. For each new logging statement, we want to guess the log level using the phrases in the logging statement. At first, we were uncertain about our guess. Every time we observe a phrase appearing in a level, our uncertainty decreases. In other words, whenever we observe the phrase "exit" in a *fatal* level logging statement, we are more certain that the logging statement with the phrase "exit" should be in *fatal* level.

Shannon entropy is a metric used to measure the amount of uncertainty (or entropy) in a distribution [Has09]. We used the *Normalized Shanon's Entropy* to calculate the level of the logging statements, based on the probability of appearance of phrases in the log message. We calculated the entropy of the phrases from existing logging statements. We consider the phrases with two or more appearances.

Table 10 shows the five-number summary for the lowest entropy of the phrases in each logging statement. We found that most of the logging statements contain phrases with low entropy. In particular, more than 25% of the logging statements contains a phrase with **zero entropy** (the phrases are only appearing in a unique level). Therefore, we can use phrases with zero entropy to suggest the log level for new logging statements. If a logging statement contains any of these phrases, we can suggest that this particular logging statement is more likely to be at the level that this phrase appeared all the times.

To find inappropriate log levels, we first used the text and variables in existing logging statements in the source code to calculate the entropy for words and combination of the words called phrases. In other words, we make a table of phrases, their corresponding entropy, and the logging level. Then, for each new logging statement, we extracted the phrases and search for them in the table we made from our existing data. If a phrase from the new logging statement existed in the table and its entropy is zero, we compared the logging level of the new statement with the table. Finally, if the log level from the table was different than the log level from the new statement, we suggested that the verbosity level is wrong.

Figure 8 demonstrates an instance of using entropy to suggest appropriate logging level. In this example, the log message contains the phrase "assigned container". This phrase occurred 12 times in different log messages in our data. Furthermore, this phrase always appeared in logging statements in *info* level. Thus, the entropy for this phrase is zero. However, the new logging statement apprised at *debug* level. Given this information, the tool will suggest that the selected logging level *debug* is wrong.

```
LOG.debug("Assigned container in queue:" + getName() + " "
+"container:" + assigned);
```

Token : assigned container
Entropy : 0
Token Level : info
Token occurrence : 12

Figure 8: Using entropy to suggest appropriate logging level

For example, in Figure 8, *"LOG.debug("Assigned container in queue: "+ getName());"* we can see that the log message contains the phrase "assigned container". "assigned container" occurred 12 times in different log messages and always appeared in *info* level. Thus, the entropy for this phrase

Table 10: A five-number summary for the lowest entropy of the phrases in each logging statement

| Subject systems | Min | 1st Qu. | Median | 3rd Qu. | Max |
|---|---|---|---|---|---|
| Hadoop | 0.00 | 0.00 | 0.33 | 0.50 | 0.87 |
| Camel | 0.00 | 0.00 | 0.25 | 0.45 | 0.83 |

of words is zero. However, the new logging statement with this token is in *debug* level. Given this information, the tool will suggest that the current level *debug* is wrong.

To evaluate our approach, we run the tool on existing issues that were fixed by changing levels. For each issue, we trained the checker with the revision before the issue fixing commit. Out of 76 log-related issues containing 209 log level changes we were able to fix 22 logging statements with inappropriate logging level in seven log-related issues with four false positives. A prior study showed that static analysis tools suffer from providing false positive results to practitioners [CSH$^{+}$16]. Therefore, we opt to avoid false positives and to have excellent precision but low recall over lower precision with a higher recall.

## 5.2 Catch block checker

In 21 issues developers simply missed to log the exception inside the catch blocks. Exception messages contain necessary information that is used by the developers while debugging the code. These issues can be fixed simply by adding or removing the exception message in a new logging statement or the end of an existing logging statement. In several issue discussions, the developers mentioned they faced situations in which they needed information related to the exceptions in the log output. In contrast, sometimes they found the information unnecessary and removed them. Issues with these fixes are considered as *inappropriate log messages* issues or *missing logging statement* in our study. Figure 9 demonstrates and example of such issues and its corresponding fix.

We provided a checker to recommend developers to add logging statements or log the exception message inside the catch blocks based on historical information of the source code. We used Eclipse's JDT (Java Development Tools) to parse the Java code and generate its Abstract Syntax Tree (AST). Using the AST, all the catch blocks, and their logging statements are extracted. Afterward, we calculated the percentage of catch blocks that log the exception messages for each exception type. To minimize false positives, we only detected the issue if either all the catch blocks with the same exception type are logged, or none of them are logged (threshold 100%). Using this threshold, we were able to fix 4 logging statements with inappropriate logging level in 2 log-related issues.

HADOOP-9358
Title : "Auth failed" log should include exception string

Fix:

```
-       AUDITLOG.warn(AUTH_FAILED_FOR + clientIP + ":" +
-                       attemptingUser);
+       AUDITLOG.warn(AUTH_FAILED_FOR + clientIP +":"+attemptingUser +
+                     " (" + e.getLocalizedMessage() + ")");
```

Figure 9: An example of faulty logging statements that fixed by adding exception message

## 5.3 Log level guard checker

Logs provide valuable information for developers and operators of the software system. However, each logging statement has some performance overhead to the system. Especially, if the log message contains many variables and method calls in it, the overhead can be costly. Hence, log libraries provide a conditional guard for the logs, such that developers can avoid executing the logging statement if logging in that level is disabled at runtime. When developers feel that creating the log message has a considerable performance overhead, they can use an *if* statement as a log level guard. In some of the libraries like *Self4j* this log level guard is implemented inside the logger method, but for other libraries, developers should add an *if* statement as a log guard manually. We found nine issues that developers forgot to add log level guards before executing logging statements. Thus, logging statements were executed but never shown in the output. Based on these findings, we made a simple checker to find missed log level guards. First, we analyze the logging library of each file. If the logging library does not perform the log level check before executing (i.e. *Log4j*), a log level guard needed for each debug level logging statement. Thus, we checked all the debug level logging statements in the file and if a logging statements have a significant computation in its message (i.e., more than three string concatenations or includes method calls), our tool suggests that developers should add a log level guard to the logging statement or consider migrating to libraries like *SLF4J*. Using this tool we were able to find all nine issues reported on issue trackers of our case studies. We also run this tool on the last revision of our case studies. We found 62 new cases that developers need to add a log level guard. A false positive case would be when developers remove log level guards while our tool suggests keeping the log level guard. We identified two issues (HDFS-8116[1] and HDFS-8971[2]) where developers remove log level guards and our tool did not suggest to keep the guard in either case.

[1]https://issues.apache.org/jira/browse/HDFS-8116
[2]https://issues.apache.org/jira/browse/HDFS-8971

## 5.4 Typo checker

We had 24 issue reports for which the solution was just fixing the typos in log messages. Typos do not have a large impact if the logs are read by operators and developers. However, automated log analysis may be impacted if they depend on these log message. To fix these typos, we need to examine the string literal (i.e., fixed) part of log messages to find the typos in them. Log messages often contain non-English words that might be in-house names or code identifiers. Thus, a simple English spell checker will return many false positives and find actual typos among them can be frustrating for developers. In our tool, we tried to improve the dictionary using the data inside the system. To reduce the number of false positives, we extracted string messages inside all the logs and counted the number of appearances of each word. Then, we added the repeated words inside the log messages to the list of known words. Furthermore, we added identifier names and words in code comments in the file to our dictionary. Using this new dictionary, we check the strings in log messages and report the inappropriate ones as possible typos.

With the typo checker, we were able to find 20 out of 24 reported typos issues in our case study. Among the four issues that are not detected, one of them was due to having extra white space between words, three of them were a typo in the log configuration file that we do not support at the moment. We also run our tool on the last revision of our case studies to find new issues that are not reported yet. In total, we found 25 new typos in log messages. After manual validation, we found seven false positives. One of the false positives was an abbreviation that was not mentioned in the code comments. The other one was a log with a text automatically generated, hence we missed that part and considered that the statement contains a typo. The rest of the false positives were informal words that were meaningful, but not included in our English dictionary.

## 5.5 Results of applying the tool

In order to evaluate our tool, we first tried to detect our manually verified log-related issues. We run our checker on the source code snapshot before each issue fix. The overall results are shown in Table 11. We reported the number of issues and logging statements successfully covered by our tool. Note that the number of issues is different from the number of logging statements since issues can be fixed by changing multiple logging statements. We were able to successfully detect 23% of inappropriate logging statements in 30% of the log-related issues. We also apply our tools to find other possible log-related issues in the latest source code from our subject systems. In total, we identified 226 potential inappropriate logging statements in the latest version of our case studies source code. For each checker, we ranked the suggestions in order to provide the most accurate detection to developers. The suggestions to change the level are ranked based on the entropy and number of occurrence word combination in the last stable version of source code. Catch block logging suggestions are ranked based on the number of occurrences of the exception type and percentage of similar behavior. Eventually, the results of log level guard checker are ranked by the number of method calls and string concatenations in the logging statements. We did not rank the typos since the suggestions were under 20 (i.e. 18 suggestions). Then, we reported the top 20 suggestion of each checker (18 for typos) for all the subject systems. Issues regarding typos in logging statements were accepted immediately and fixed by the author of the thesis. Issues regarding log level guards are also accepted by the developer of the Hadoop and Camel. However, developers of Hadoop mentioned that they plan to move to *SLF4j* in order to fix these issues rather than adding the guards to the mentioned logging statements. In the MapReduce subsystem, developers mentioned the fix is in progress. In the HDFS subsystem of Hadoop, developers have already provided a patch to migrate the logging library to *SLF4j*. Finally, developers of Camel asked us to provide the patch by adding the if statement before the debug level guards. Other reported issues are still under review.

Table 11: The results of our tool on known issues

| **Type** | **# known issues (# of logging statements)** | **# issues successfully detected by the checker (# of logging statements)** |
|---|---|---|
| Typos | 26(40) | 22(34) |
| Missing to log exceptions | 21(65) | 2(4) |
| Inappropriate log level | 76(209) | 7(22) |
| Missing log level guard | 9(15) | 9(15) |

A prior study has proposed an approach that builds a statistical model to predict the appropriate level of a logging statement [LSH17]. Although the goal of this approach is to suggest log level, the approach can be used to detect issues with an inappropriate level in particular. We compared our log level checker with the approach that suggests log levels. We obtained the original data that were used in the prior study [LSH17]. Since *Hadoop* is also a subject system in the prior study, we found 56 logging statements that were manually identified in our study with wrong log levels and were also included in the prior study's data. We examined whether the statistical model that was build based on the prior study could detect these log-related issues. We found that in 32 logging statements, the model failed to find the appropriate level that developers decided on the issues. However, our tool was able to suggest ten correct log levels without any false positive. Note that the threshold of entropy in our log level checker was set to 0.33 in this experiment. These results show that the logging statements that are reported as issues, because of their level, are harder to predict in nature. Studies also show that developers often have difficulties in choosing the appropriate log level and spend much effort on adjusting the log level [YPZ12b].

## 5.6 Impact of the threshold on our checkers

Based on the results of our tools, two of our checkers, i.e., log level checker and catch block checker, have a low number of detected issues. Both checkers are based on thresholds. Our log level checker is based on the entropy of a word or a combination of words existing in different log levels. Our catch block checker is based on the percentage of different behaviors in existing exception's catch blocks. Therefore, we aim to refine these two checkers to achieve better detection results.

### 5.6.1 Log level checker refinement.

The original log level checker only considers the phrases with zero entropy. Because most of the phrases have low entropy, as shown in Table 10. This time, we used the phrase with the lowest entropy in each logging statement to detect inappropriate log level, instead of only considering the phrases with zero entropy. However, we found that also we were able to provide more suggestions (47 instead of 22), our precision becomes lower (73%). Furthermore, we varied the threshold between 0.0 to 0.33 (median entropy, see Table 10), to see its impact on our results. Our precisions are between 84% and 87%, and we can correctly detect 22 to 31 wrong log levels in 7 to 9 log-related issues.

We found that for the older issues, there was limited number of logs in the source code of our subject systems. Limited training data has a significant impact on our checker. Thus, we used the data from other projects to improve the training data. We used all subject systems with at least one logging statement in them from the top 1,000 popular Java projects on Github (354 projects). Then, we excluded the subject system that we were testing against, as well as their forks from the list and trained the checker with source code for remaining projects. However, we found that although this approach let us provide more suggestions (55 to 80 log level change suggestions out of 209), the precision is low (65% - 58% precision) using thresholds between 0.0 to 0.33.

Finally, we decided to add the source code from the revision before the issue fixing commits to the other projects as well. Using this approach we kept the testing and training data separated at all the time. With the extended training set, we were able to suggest a level change for 51 logging statements with eight false positives in 19 log-related issues, resulting into 84% precision using phrases with zero entropy. After changing the threshold to 0.33, we were able to suggest 56 level changes with ten false positives (82% precision) in 20 log-related issues.

### 5.6.2 Catch block checker refinement.

The original catch block checker uses 100% as a threshold, meaning that developers logged the exception in either all or none of the previous catch blocks with the same exception type. We varied the threshold from 100% to 50% (in half of the existing exceptions developers logged the exception).

The results show that when the threshold is set to 80%, the precision decreases from 100% to only 60%, while we are only able to detect three more issues. When the threshold is set to 50%, our precision is only 33%. Such results show that in order to detect more issues, we would need to sacrifice our precision. Therefore, having 100% as our threshold is a better choice.

In this study, we aim to make a recommender tool for developers. Thus, our goal is to have smaller false positive rates rather than higher recall values. In all the checkers, we have very few to no false positives. We evaluated our tool on the log changes extracted from the reported issues. Developers had a hard time to write the appropriate logging statements on the first try. Thus these logging statements are reported as issues. In fact, when we compared the existing works using our dataset, we outperformed them with better precision and recall. We agree that we do not provide many suggestions. However, we try to provide the right suggestions when we do provide them. We plan to improve the recall of our approach in future work.

# Chapter 6

# Threats to Validity

## 6.1 Internal Validity

In this study, we employed different heuristics in our approach that may impact the internal validity of our approach. We only studied the issues with a log-related keyword in their title. However, to see the impact of this filtering, we extracted all the commits in the history of the subject systems where at least one line of code containing a logging statement was modified. We then drew a statistically-random sample with 95% confidence level and ± 5 confidence interval from this pool of commits and investigated them manually. We found that our approach only misses 0.5% of the changes that were done due to a log-related issue. Other commits in the sample are either true positives of our approach (i.e., they are changes due to log-related issues that our technique was able to identify correctly), unrelated fixes, or addition of new functionalities to the system. Moreover, we used text matching to find the corresponding commits for each issue. We ignored issues labeled other than "improvement" or "bug" in our study. However, the majority of all the issues (72%) were labeled either "improvement" or "bug". We wanted to study issues regarding a problem in logs rather than issues that implement a new feature. The results of our second and third research questions impacted by the accuracy of the matching we performed on the issues and commit. Besides, in some cases, the authors of the commits on GitHub may not be the original authors of the code. We mined the commit messages and used issue report data to find the original author of the commits. However, in 12% of the commits, we were not able to find another author mentioned in the commit message for the corresponding issue of the commit. We used Git commands to obtain the log introducing change in the history of the case studies. These commands use Unix diff, which can impact our results. However, in our scripts, we checked the results and removed the commits that we were not able to confirm as log introducing changes by verifying the existence of the logging statement.

## 6.2 Construct Validity

*Construct validity threats* concern the relation between theory and observation [Yin13]. In our study, the main threats to construct validity arise from human judgment used to study and classify the log-related issues. In particular, the categories are based on the classification performed by the first two authors and can be biased by the opinion of the researcher on the issues and the source code. We also used keywords to filter the issues. Thus, we might have missed some log-related issues which do not contain our keywords in their titles. However, the goal of the study is not to exhaustively collect all log-related issues but rather study based on a collection of them. Future studies may consider collecting log-related issues in another approach to complement our findings.

## 6.3 External Validity

We performed our study on *Hadoop* and *Camel*. These systems are large software systems containing millions of lines of code with 11K and 6K logging statements, respectively. However, more case studies on other software systems in different domains are needed to see whether our results are similar to this study. Conducting research on different case studies from other domains will help us to examine the importance of the logging statements in other areas, and also to understand similar and distinct types of logging statements in software systems. However, the results of our study showed that also four subsystems of *Hadoop* are considered as one subject system, they show different behavior in our analysis.

Moreover, we should note that *Hadoop* and *Camel* are open source software. Therefore, the results of our study are based on only open source software systems and may not generalize to commercial systems. To improve our research, we need to replicate it on enterprise software systems to gain a better understanding of their log-related issues. Furthermore, our study focuses on Java-based software systems. Using case studies from different languages can improve our knowledge about logging statements and their problems in other languages. We manually studied 563 log-related issues in seven categories. But, our automated approach can provide suggestions for 132 log-related issues in four categories. In our manual analysis, we found that many of the log-related issues require domain knowledge, as well as, an understanding of the environment being fixed. Hence, we chose to focus on issues that can be detected and fixed automatically. Unfortunately, we were not able to provide a checker for all of the log-related issues we studied in this study. However, we offer characteristic analysis to help developers and users better understand issues regarding logging in their systems.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusion

Logs are one of the most important sources of information for debugging and maintaining software systems. The valuable information in logs motivates the development of log analysis tools. However, issues in logs may highly impact the values of log analysis tools by providing incomplete or inaccurate information to the users of the logs. Therefore, in this research, we empirically studied 563 issues from two open-source software systems, i.e., *Hadoop* and *Camel*. We found that:

- Files with log-related issues have undergone statistically significantly more frequent prior changes, and bug fixes.
- Log-related issues are often fixed by neither the developer who introduced the logging statement nor the owner of the file that contains the logging statement.
- Log-related issues are reported after a long time of the introduction of the logging statement.

Our findings showed the need for automated tools to detect log-related issues. Therefore, we manually investigated seven root-causes of log-related issues. Table 12 summarizes the findings for each research question and its implications. We developed an automated tool that detects four types of evident root-causes of log-related issues. Our tool could detect 40 existing log-related issues and 38 (accepted by developers) previously unknown issues in the latest release of the subject systems. Our work suggests the need for more systematic logging practices in order to ensure the quality of logs.

Table 12: Our findings on log-related issues and their implication.

| **Findings** | **Implications** |
|---|---|
| **Location of the log-related issues** | **Implication** |
| Files with log-related issues have larger number of prior commits and more prior bugs. | This implies that developers often prioritized their efforts on these files. |
| **People involved in log-related issues** | **Implication** |
| 78% of the buggy logging statements are fixed by someone other than original committer.<br>78% of the buggy logging statements are not fixed by owner.<br>24% of the buggy logging statements are introduced by the owner.<br>73% of the buggy logging statements are fixed by the person who reported the issue. | Various people are responsible for adding and fixing the logging statements.<br>Thus, its hard to find experts of the logs. This shows the need for automated tools to aid developers in diagnosing and fixing log-related issues. |
| **Time takes to fix log-related issues** | **Implication** |
| It takes a long time (median 320 days) for a logging statement to be reported buggy. However, 80% of log issues were fixed within ten days of their report. | Log-related issues are fixed fast after their exposure but it takes long time for them to report as an issue. |
| **Root causes of log-related issues** | **Implication** |
| Based on our manual study, we categorized log-related issues into seven categories. | Log-related issues have different root causes. There exist evident patterns which can automatically detected. |

## 7.2 Future Work

We conducted an empirical study on log-related issues and proposed a tool based on the results to detect four types of log-related issues. The set of checkers introduced in this paper can be integrated into continuous integration systems to provide just-in-time suggestions to developers and prevent faulty logging statements to be accepted into the main branch of the software systems. Moreover, we found seven root-causes for log-related issues. More studies are needed to find new patterns and improve the recall of our existing checkers.

As an improvement to this study, we are interested in investigating the impact of log-related issues. We will use the manually-verified log-related issues in this work to measured the impact of these issues in three different ways. First, we aim to look into the test coverage of the logging statements which have issues (*i.e.,* faulty logging statements) before they were fixed, by running the existing unit tests in the source code. Second, we would like to study the benchmarks that exercise our case studies, and investigate the logs generated during the execution of the benchmarks to measure how many of faulty logging statements impact the benchmark logs. Finally, we propose to search through StackOverflow to find the posts of which the contents include faulty logging statements explored in this study, and investigate the impact of log-related issues on the users of StackOverflow in solving their problems.

# Bibliography

[BDDF16] T. Barik, R. DeLine, S. Drucker, and D. Fisher. The bones of the system: A case study of logging and telemetry at microsoft. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 92–101, May 2016.

[BKQ+08] Jerome Boulon, Andy Konwinski, Runping Qi, Ariel Rabkin, Eric Yang, and Mac Yang. Chukwa, a large-scale monitoring system. In *Proceedings of CCA*, volume 8, pages 1–5, 2008.

[BL96] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[BND+09] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality?: an empirical case study of windows vista. *Communications of the ACM*, 52(8):85–93, 2009.

[BNM+11] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[BWS92] Cynthia Baker, Judith Wuest, and Phyllis Noerager Stern. Method slurring: the grounded theory/phenomenology example. *Journal of advanced nursing*, 17(11):1355–1360, 1992.

[Car12] David Carasso. *Exploring Splunk*. CIT, Zagreb, Croatia, Croatia, 2012.

[CJ17] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing and detecting anti-patterns in the logging code. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th IEEE International Conference on*. IEEE, 2017.

[CSH$^+$16] T. H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora. Detecting problems in the database access code of large scale systems - an industrial experience report. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 71–80, May 2016.

[CSJ$^+$14] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1001–1012. ACM, 2014.

[DMCSO05] Alex Dmitrienko, Geert Molenberghs, Christy Chuang-Stein, and Walter W Offen. *Analysis of clinical trials using SAS: A practical guide.* SAS Institute, 2005.

[DZL$^+$15] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 139–150, Berkeley, CA, USA, 2015. USENIX Association.

[elk17] The Open Source Elastic Stack, November 2017. https://www.elastic.co/products/.

[FB99] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[FRZ$^+$12] Xiaoyu Fu, Rui Ren, Jianfeng Zhan, Wei Zhou, Zhen Jia, and Gang Lu. Logmaster: Mining event correlations in logs of large-scale cluster systems. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, pages 71–80, Washington, DC, USA, 2012. IEEE Computer Society.

[FZH$^+$14] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33. ACM, 2014.

[Gou17] Georgios Gousios. java-callgraph: Java Call Graph Utilities, November 2017. https://github.com/gousiosg/java-callgraph.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[Hen17] Idan Hen. GitHub Research:Over 50% of Java Logging Statements Are Written Wrong, February 2017. https://goo.gl/4Tp1nr/.

[HH10] Israel Herraiz and Ahmed E Hassan. Beyond lines of code: Do we need more complexity metrics? *Making software: what really works, and why we believe it*, pages 125–141, 2010.

[HRGB$^+$06] I. Herraiz, G. Robles, J. M. Gonzalez-Barahona, A. Capiluppi, and J. F. Ramil. Comparison between slocs and number of files as size metrics for software evolution analysis. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 8 pp.–213, March 2006.

[KBSH16] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Logging library migrations: A case study for the apache software foundation projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 154–164, New York, NY, USA, 2016. ACM.

[KDHS07] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. Systematic review: A systematic review of effect size in software engineering experiments. *Inf. Softw. Technol.*, 49(11-12):1073–1086, November 2007.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[KSBH16] S. Kabinna, W. Shang, C. P. Bezemer, and A. E. Hassan. Examining the stability of logging statements. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 326–337, March 2016.

[KY18] Weiyi Shang Steve Sporea Andrei Toma Sarah Sajedi Kundi Yao, Guilherme B. de Pádua. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 9th ACM/SPEC on International Conference on Performance Engineering*, ICPE '18, New York, NY, USA, 2018. ACM.

[KZWG11] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 481–490. IEEE, 2011.

[LCSH18] Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Ahmed E Hassan. Studying software logging using topic models. *Empirical Software Engineering*, pages 1–40, 2018.

[LSH17] Heng Li, Weiyi Shang, and Ahmed E Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716, 2017.

[LSZEH17] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Softw. Engg.*, 22(4):1831–1865, August 2017.

[MHH13] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1012–1021, Piscataway, NJ, USA, 2013. IEEE Press.

[NKN12] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[NMB08] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 521–530. IEEE, 2008.

[OA11] Adam J. Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 49–60, Washington, DC, USA, 2011. IEEE Computer Society.

[OGX12] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, February 2012.

[SJA$^+$11] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. In *Proceedings of the 18th Working Conference on Reverse Engineering*, WCRE '11, pages 335–344, 2011.

[SJA$^+$14] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.

[SNH15] Weiyi Shang, Meiyappan Nagappan, and Ahmed E Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.

[SNHJ14] Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan, and Zhen Ming Jiang. Understanding log lines using development knowledge. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 21–30, Washington, DC, USA, 2014. IEEE Computer Society.

[SRF16] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 120–131, New York, NY, USA, 2016. ACM.

[TPK+08] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs*, WASL'08, pages 6–6, Berkeley, CA, USA, 2008. USENIX Association.

[WW64] Frank Wilcoxon and Roberta A Wilcox. *Some rapid approximate statistical procedures.* Lederle Laboratories, 1964.

[XHF+09] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[Yin13] Robert K Yin. *Case study research: Design and methods.* Sage publications, 2013.

[YMX+10] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 143–154, New York, NY, USA, 2010. ACM.

[YPH+12] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 293–306, Berkeley, CA, USA, 2012. USENIX Association.

[YPZ12a] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.

[YPZ12b] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.

[YZP+12] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Trans. Comput. Syst.*, 30(1):4:1–4:28, February 2012.

[Zha09] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 274–283. IEEE, 2009.

[ZHF+15] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 415–425, Piscataway, NJ, USA, 2015. IEEE Press.

[ZRL+17] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 565–581. ACM, 2017.