Noname manuscript No.
(will be inserted by the editor)

Studying and Detecting Log-Related Issues

Mehran Hassani · Weiyi Shang · Emad Shihab · Nikolaos Tsantalis

Received: date / Accepted: date

Abstract Logs capture valuable information throughout the execution of software systems. The rich knowledge conveyed in logs is highly 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 Log-related issues, such as missing or having outdated information, may have a large impact on the users who depend on these logs.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 find that the files with log-related issues have undergone statistically significantly more frequent prior changes, and bug fixes. We also find 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 clear experts. Finally, we find 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 evident 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.

Keywords Empirical study · Log · Software Bug · Mining software repositories

Mehran Hassani · Weiyi Shang · Emad Shihab · Nikolaos Tsantalis
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
E-mail: {m_assani, shang, shihab, tsantalis}@encs.concordia.ca

1 Introduction

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. During system runtime, the invocation of these logging statements would generate 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 [23]. Logs are used during various software development activities such as bug fixing [39], anomaly detection [37], testing results analyses [27], and system monitoring [6, 45]. 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 [43]. Moreover, log processing infrastructures such as Splunk [7] and ELK stack [1] are developed for the ease of systematic log analyses.

To improve logging statements, similar as fixing bugs, developers would 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 [43] and the places where developers embed logging statements [13]. Also, prior research aims to enhance logging statements by automatically including more information [41, 45], and provide suggestions on where to log [48]. However, these empirical results and the above-mentioned approaches do not aim to help developers write an issue-free logging statement. In fact, there exist limited guidelines that developers can follow to write appropriate logging statements.

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

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

statements may lead to wrong decisions made by system operators; and overwhelming information in logs would prevent practitioners from identifying the truly needed information [25]. Recent research on Github projects claims that over half of the Java logging statements are "wrong" [17]. 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 paper, 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. In particular, we extract 563 log-related issues from the JIRA issue tracking systems of the two subject systems and study these issues 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?*
We found that in 78% the cases, logging statements are added and fixed by different people. In other words, 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.

[6] https://issues.apache.org/jira/browse/HADOOP-4190

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 43 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.

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.

The rest of the paper is organized as follows. Section 2 describes the studied systems and our data collection approach. Section 3 presents the results to answer our research questions. Section 4 demonstrates our proposed tool that automatically detects log-related issues. Section 5 discusses the related works. Section 6 discusses the potential threats to the validity of our study. Finally, Section 7 concludes the paper.

2 Case Study Setup

In this section, we present our case study setup. In particular, we present the subject systems of our case study and our approach for collecting log-related issues.

2.1 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 libraries levels. Then, we picked the top two software systems as our subjects.

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

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

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 actives. In particular, *Hadoop* has more than 11K logging statements and *Camel* has more than 6K logging statements in their latest revision of source code.

2.2 Collecting log-related issues

In order to conduct the study, we first need to collect log-related issues in the subject systems. There exists no explicit flag in *JIRA* issue reports that label an issue as a log-related issue. Thus, we extract all available issue reports of our subject systems. Then, we leverage a keyword based heuristic to filter log-related issues, by searching for keywords like *log*, *logging*, or *logger*. We only select 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 would require the corresponding fix to these issues to understand the characteristics and the root-causes of the issues. We include 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. For example, in HADOOP-8075, 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 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 1).

Table 1: 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%)

3 Case Study Results

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 answering the question and the corresponding results. Figure 1 presents an overview of our approach to answering the research questions.

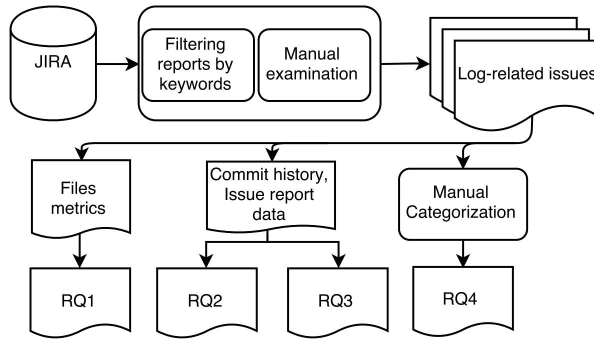

Fig. 1: An overview of our approach to answer the research questions

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

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.

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 [18, 19, 47]. However, larger files tend to contain more logging statements [34]. Having more logging statements increases the probability of having more log-related issues. Thus, we normalize 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” [15]. 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 use 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 use 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 use 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 [38]. We perform four comparisons on each dataset. To better control for the randomness of our observations, we used Bonferroni correction [12]. We adjust our p-value by dividing it by the number of comparisons (four). The results are significant at the significance level $\text{alpha} = 0.05/4$ ($\text{p-value} < 0.0125$). 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 [10, 22] 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.* [22], Cliff's delta values can be interpreted as shown in Table 2:

Table 2: 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 $0.474 < Cliff's\ d$

Results. Table 3 presents the median of our studied metrics for files with and without log-related issues. We find 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 find 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 4 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 4 subject system. We find 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[8] 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*[9], 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.

[8] https://issues.apache.org/jira/browse/HDFS-9528
[9] https://issues.apache.org/jira/browse/HDFS-2729

Table 3: 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 4: Medians (*Med.*) and effect sizes (*Eff.*) of the normalized process and product metrics for files with and without log-related issues. Files without log 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	

RQ2: Who reports and fixes log-related issues?

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 [35]. Furthermore, studies show the importance of developer ownership and its impact on code quality [5]. Studies showed that when more people are working on a file, it is more likely to have failures in the feature [4, 28]. 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.

Approach. To answer this research question, we first need to know who introduced the logging statement. Thus, for all the log-related issues in Java files, we first search 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 identify the developers who introduced the logging statements, and we count the prior number of commits by developers to measure the expertise and the ownership of the code in the repositories. Figure 2 demonstrates the lifetime of an inappropriate logging statement. Based on the Figure 2, 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 [5].

Results. We find 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 without 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 the one hand, the developers who introduce the file realize the importance of placing the particular logging statement in the source code [35]. 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 [32, 33].

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.

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

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 needed to report and fix log-related issues.

Approach. We aim to find out how fast log-related issues were reported and fixed. Figure 2 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 statement is introduced to the time when it is reported in the issue tracking system (Figure 2, 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)

Results. The results are depicted in Figure 3 and Figure 4 . We find that more than 80% of the issues were fixed in less than 20 days. In fact, 43% of all

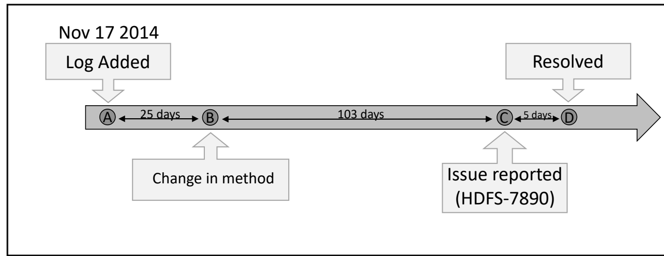

Fig. 2: Lifetime of an example inappropriate logging statement

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 6 shows the five-number summary of the number of changes for each logging statement. We can see 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 5 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 2 shows all the changes made to a logging statement during its lifetime which led to issue HDFS-7890.

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.

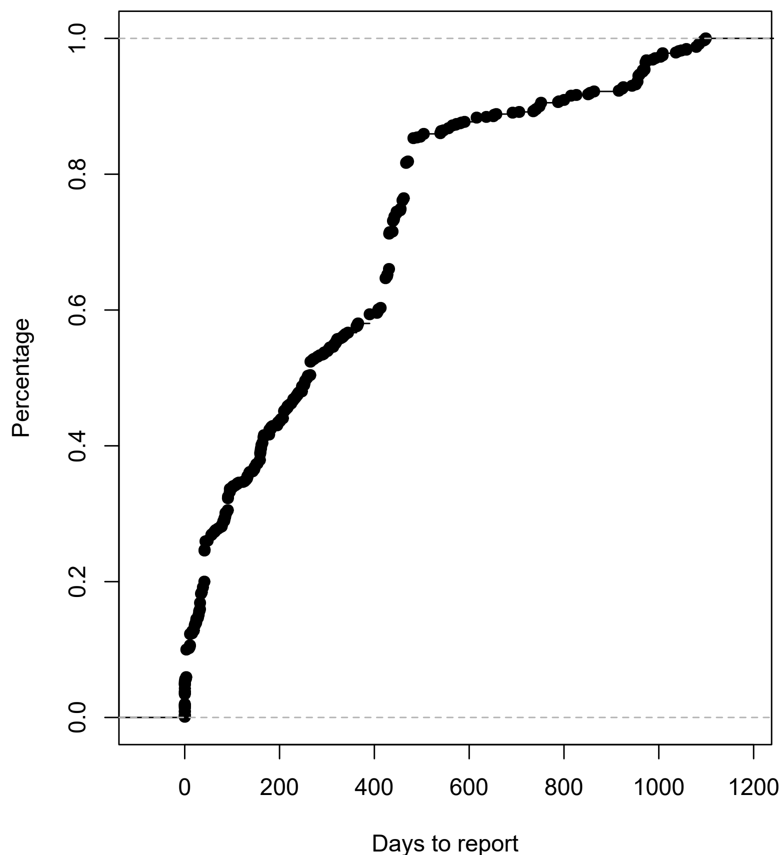

Fig. 3: Cumulative distribution of issue report time in days. Outliers that are greater than 1.5 time of the value of the third quartile of the data are not shown in this figure.

Table 5: 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

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.

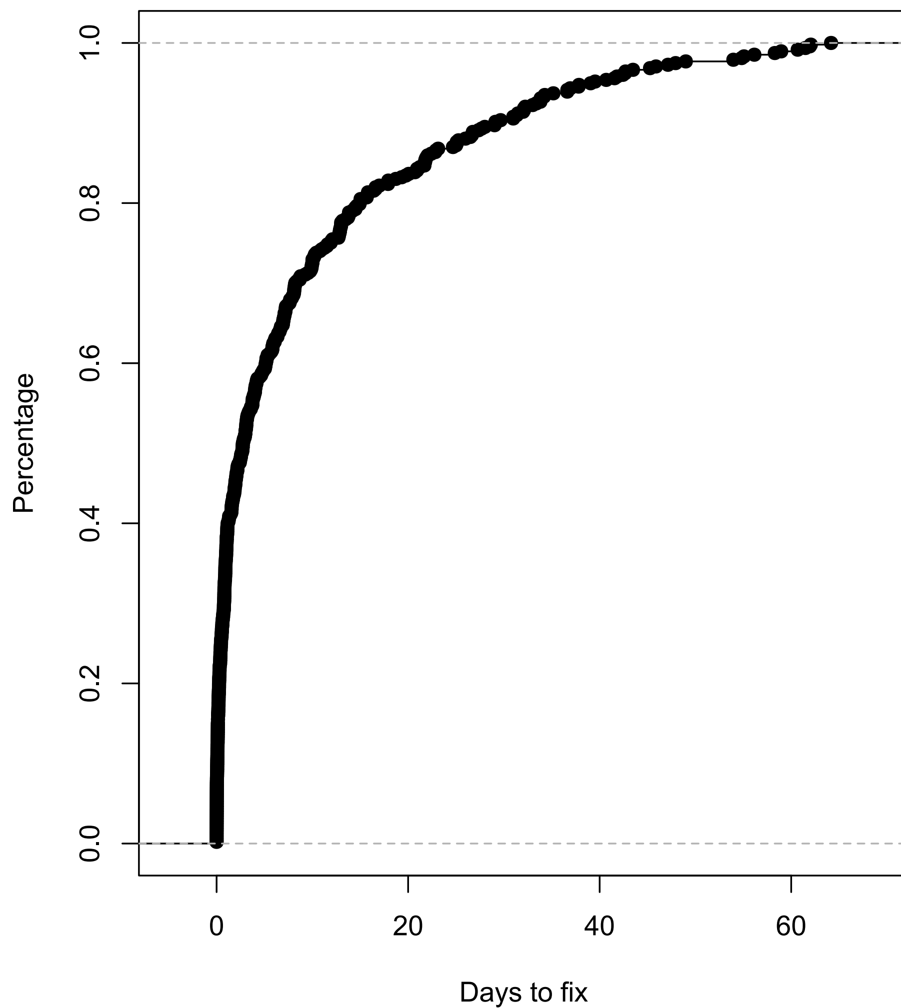

Fig. 4: Cumulative distribution of issue fix time in days. Outliers that are greater than 1.5 time of the value of the third quartile of the data are not shown in this figure.

Table 6: 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

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

Motivation. Previous RQs show 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.

Approach. To answer this research question, we used the issue reports and their code changes we extracted from JIRA. Stol *et al.* [36] suggest that researchers should describe how they analyzed data rather than dressing it up as other well known scientific approaches.To avoid method slurring [2], 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 categorized 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.

Results. The results of our manual study are shown in Table **??**. 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 8, we show the distribution of each of the mentioned types of log-related issues.

Inappropriate log message. As shown in Table 7, logging statements with incorrect log messages constitute the majority of log-related issues. We consider every issue regarding log messages (such 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 consider 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

Table 7: 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 8: 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%

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 steps of the execution, 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, 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 in this category if it causes a runtime failure or misbehavior of the system at execution time. For example, in the issue Hadoop-7695, developers mentioned that *"RPC.stopProxy can throw a configuration file for unintended exception while logging error"*. Here, 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 log*, 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, 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 observe 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.

4 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 guard, 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.

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 5.

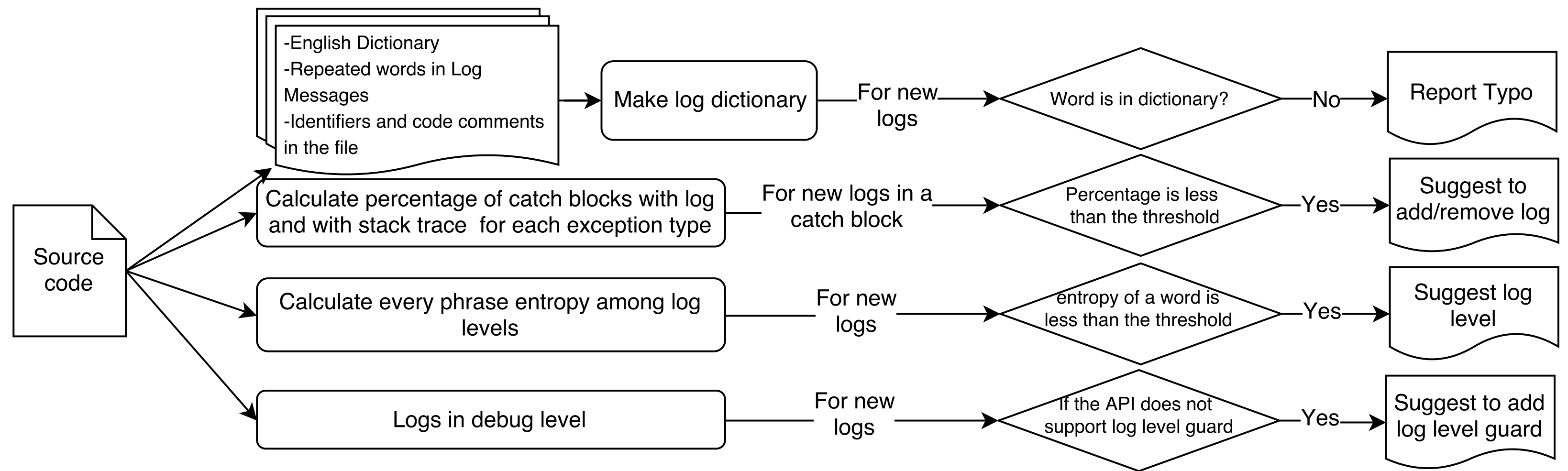

Fig. 5: An overview of log-related issue checker.

4.1 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. 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.

Information Theory deals with assessing and defining the amount of information in a message [40]. 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 [16]. 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 9 shows the five-number summary for the lowest entropy of the phrases in each logging statement. We find 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 use 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 extract the phrases and search for them in the table we made from our training data. If a phrase from the new logging statement exists in the table and its entropy is zero, we compare the logging level of the new statement with the table. Finally, if the log level from the table is different than the log level from the new statement, we suggest that the verbosity level is wrong.

For example, in *"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

Table 9: 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

appeared in *info* level. Thus, the entropy for this phrase 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 [9]. Therefore, we opt to avoid false positives and to have excellent precision but low recall over lower precision with a higher recall.

4.2 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, developers mention 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.

We provide 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 calculate the percentage of catch blocks that log the exception messages for each exception type. To minimize false positives, we only detect 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.

4.3 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 check 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 identify two issues (HDFS-8116[10] and HDFS-8971[11]) where developers remove log level guards and our tool did not suggest to keep the guard in either case.

4.4 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

[10] https://issues.apache.org/jira/browse/HDFS-8116
[11] https://issues.apache.org/jira/browse/HDFS-8971

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.

4.5 Results of applying the tool

In order to evaluate our tool, we first try 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 10. 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 first author of the paper. 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 10: 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 [25]. 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 [25]. 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 our 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 [44].

5 Related Work

In this section, we discuss the prior research that is related to this paper.

5.1 Log analysis

Logs are widely used in software development and operation to ensure the quality of large software systems [3]. Prior research focuses on the analysis of logs [30] to assist different software development and operation activities. Valuable information is extracted from logs, including event correlations [14, 29], resource usages [21], component dependency [31], and causal paths [45]. The extensive usage of logs motivates our paper since quality logs are extremely

important for the effectiveness of prior log analysis research. The outcome of this paper would help reduce log-related issues, hence improve the adoption of advanced log analysis in practice.

5.2 Logging enhancement

The closest recent research is from Chen *et al.* [8] on the detection of anti-patterns in logging code. 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 five anti-patterns and proposed static code checkers to detect these patterns. However, instead of detecting anti-patterns (like code smell [8]) that have the possibility to be an issue, we focus on the event issues that are more certain. In fact, by comparing our study to the research by Chen *et al.*, only one anti-pattern/root-cause (Wrong log level) overlaps in 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, our empirical study results on log-related issues provide more insights on these issues.

The prior research proposes techniques in order to enhance logging statements. Yuan *et al.* have conducted a series of research [41, 42, 43, 45] on how to perform better logging. They found that developers usually change logging statements since they do not write the appropriate logging statement in the first attempt. Furthermore, they give some insights on how to improve the logging practices. They also built a simple checker that can help developers to write better logging statements. Yuan *et al.* [43] start off by performing a characteristic study on log practices by mining the revision histories of four open-source software projects. They propose a tool named LogEnhancer [41, 45] to automatically detect valuable information and add the information to logging statements. LogEnhancer looks for accessible variables for each existing logging statement and adds them to the logging library method call. With our exceptions checker, we suggest developers add exception variable to the logging statement if the thrown exception type always logged the exception variable in the source code of the subject system. Even though we cannot run their approach, by the description of their approach, we can know that their approach is guaranteed to solve the issues that fixed by adding variables to logging statements as well as the issues with missing exception variable (28 log-related issues). However, it can potentially produce noise to the log output. In fact, seven log-related issues were fixed by removing variables from logging statements. Their approach may results in worsening the issues. Zhu *et al.* [48], also mention that adding all the variables to logging statement pollutes the log output and not recommended by developers. Yao *et al.* [24] proposed an approach that recommends 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. Our paper finds that missing logging statements are one of the root-causes of log-related issues. However, their approach is only suitable with the consideration of performance monitoring. Yuan *et al.* [42] investigate 250 real-world failure reports find an additional point to log the possible failures. Ding *et al.* [11] propose a filtering mechanism to reduce the I/O consumption of the logs at runtime. They perform the filtering with having performance problem diagnoses as the main usage of the logs in their mind. Their approach is deeply integrated into the source code as an API to reduce the number of logs saved during the runtime. However, we aim to provide a recommender in our approach to help developers improve logging statements in their source code.

Fu *et al.* [13] systematically studied the logging practices of developers in the industry, with a focus on where developers log and come up with useful lessons for developers regarding where to log. Follow-up work by Zhu *et al.* [48] proposes a framework, which helps provide informative guidance on where to insert logging statements. They implemented LogAdvisor to give usable suggestions based on the different features of the code snippet. They used machine learning algorithms to suggest logging decisions for developers. Their approach shows promising results in their paper. However, 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 of the exception type. 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. Li *et al.* [25] proposed an approach which helps developers choose the best log level when they are adding a new logging statement. In another work [26], they provided a model to suggest the need for log change in commits. Li *et al.* also studied the rationale behind log changes. Li *et al.* found that the reasons behind log changes can be categorized as block change, log improvement, dependence-driven change, and logging issue. Kabinna *et al.* [20], studied the logging library migration in Apache Software Foundation (ASF) projects. The found that 14% of the ASF projects had at least one logging library migration in their lifetime. The also show that 70% the migrated projects had at least two issues related to the logging library migration. Although they provide useful insights for logging library migration, they do not propose an approach to automatically aid developers in the process. In another work [21], Kabinna *et al.* studied the stability of logging statements and proposed prediction techniques to help developer avoid depending on unstable logging statements.

Existing research mainly focuses on suggesting and improving existing logging statements. However, we study the log-related issues and aim to improve the quality of logging statements by automatically detecting these issues. We provide a comparison of related works that aim to improve logging code in Table 11.

Table 11: Comparing our work with prior research on logging suggestions and improvements

Papers	Goal	Code element level	Note
Learning to log: Helping developers make informed logging decisions [48]	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 [46]	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. [44]	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? [25]	Predicting verbosity level	Statement Level	Providing a model to predict appropriate logging level using code metrics.
Towards just-in-time suggestions for log changes [26]	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 paper)	Detecting log-related issues	Statement Level	Studying characteristics of the log-related issues and providing a checker to detect evident errors in logging code.

6 Discussions and Threats to Validity

6.0.1 Impact of the threshold on our checkers

Based on the results of our tool, 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.

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 9. This time, we use 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 find that also we were able to provide more suggestions (47 instead of 26), our precision becomes lower (73%). Furthermore, we varied the threshold between 0.0 to 0.33 (median entropy, see Table 9), 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 find 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 find 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.

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 vary 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.

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 manually 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 [40]. 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 perform 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 find 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 paper. However, we offer characteristic analysis to help developers and users better understand issues regarding logging in their systems.

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. 78% of the buggy logging statements are not fixed by owner. 24% of the buggy logging statements are introduced by the owner. 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. 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 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 paper, we empirically study 563 issues from two open-source software systems, i.e., *Hadoop* and *Camel*. We find that 1) files with log-related issues have undergone statistically significantly more frequent prior changes, and bug fixes, 2) 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, and 3) log-related issues are reported after a long time of the introduction of the logging statement. Our findings show the need for automated tools to detect log-related issues. Therefore, we manually investigate seven root-causes of log-related issues. Table 12 summarizes the findings for each research question and its implications. We develop 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.

References

1. The Open Source Elastic Stack (2017). Https://www.elastic.co/products/
2. Baker, C., Wuest, J., Stern, P.N.: Method slurring: the grounded theory/phenomenology example. Journal of advanced nursing **17**(11), 1355–1360 (1992)
3. Barik, T., DeLine, R., Drucker, S., Fisher, D.: 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), pp. 92–101 (2016)
4. Bird, C., Nagappan, N., Devanbu, P., Gall, H., Murphy, B.: Does distributed development affect software quality?: an empirical case study of windows vista. Communications of the ACM **52**(8), 85–93 (2009)
5. Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P.: 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, pp. 4–14. ACM (2011)
6. Boulon, J., Konwinski, A., Qi, R., Rabkin, A., Yang, E., Yang, M.: Chukwa, a large-scale monitoring system. In: Proceedings of CCA, vol. 8, pp. 1–5 (2008)
7. Carasso, D.: Exploring Splunk. CIT, Zagreb, Croatia, Croatia (2012)
8. Chen, B., Jiang, Z.M.J.: Characterizing and detecting anti-patterns in the logging code. In: Software Engineering (ICSE), 2017 IEEE/ACM 39th IEEE International Conference on. IEEE (2017)
9. Chen, T.H., Shang, W., Hassan, A.E., Nasser, M., Flora, P.: 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), pp. 71–80 (2016)
10. Chen, T.H., Shang, W., Jiang, Z.M., Hassan, A.E., Nasser, M., Flora, P.: Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proceedings of the 36th International Conference on Software Engineering, pp. 1001–1012. ACM (2014)

11. Ding, R., Zhou, H., Lou, J.G., Zhang, H., Lin, Q., Fu, Q., Zhang, D., Xie, T.: Log2: A cost-aware logging mechanism for performance diagnosis. In: Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '15, pp. 139–150. USENIX Association, Berkeley, CA, USA (2015). URL http://dl.acm.org/citation.cfm?id=2813767.2813778
12. Dmitrienko, A., Molenberghs, G., Chuang-Stein, C., Offen, W.W.: Analysis of clinical trials using SAS: A practical guide. SAS Institute (2005)
13. Fu, Q., Zhu, J., Hu, W., Lou, J.G., Ding, R., Lin, Q., Zhang, D., Xie, T.: Where do developers log? an empirical study on logging practices in industry. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 24–33. ACM (2014)
14. Fu, X., Ren, R., Zhan, J., Zhou, W., Jia, Z., Lu, G.: 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, pp. 71–80. IEEE Computer Society, Washington, DC, USA (2012). DOI 10.1109/SRDS.2012.40. URL http://dx.doi.org/10.1109/SRDS.2012.40
15. Gousios, G.: java-callgraph: Java Call Graph Utilities (2017). Https://github.com/gousiosg/java-callgraph
16. Hassan, A.E.: Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 78–88. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/ICSE.2009.5070510. URL http://dx.doi.org/10.1109/ICSE.2009.5070510
17. Hen, I.: GitHub Research:Over 50% of Java Logging Statements Are Written Wrong (2017). Https://goo.gl/4Tp1nr/
18. Herraiz, I., Hassan, A.E.: Beyond lines of code: Do we need more complexity metrics? Making software: what really works, and why we believe it pp. 125–141 (2010)
19. Herraiz, I., Robles, G., Gonzalez-Barahona, J.M., Capiluppi, A., Ramil, J.F.: Comparison between slocs and number of files as size metrics for software evolution analysis. In: Conference on Software Maintenance and Reengineering (CSMR'06), pp. 8 pp.–213 (2006). DOI 10.1109/CSMR.2006.17
20. Kabinna, S., Bezemer, C.P., Shang, W., Hassan, A.E.: 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, pp. 154–164. ACM, New York, NY, USA (2016). DOI 10.1145/2901739.2901769. URL http://doi.acm.org/10.1145/2901739.2901769
21. Kabinna, S., Shang, W., Bezemer, C.P., Hassan, A.E.: Examining the stability of logging statements. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 326–337 (2016). DOI 10.1109/SANER.2016.29
22. Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I.K.: Systematic review: A systematic review of effect size in software engineering experiments. Inf. Softw. Technol. **49**(11-12), 1073–1086 (2007). DOI 10.1016/j.infsof.2007.02.015. URL http://dx.doi.org/10.1016/j.infsof.2007.02.015
23. Kernighan, B.W., Pike, R.: The Practice of Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
24. Kundi Yao Guilherme B. de Pádua, W.S.S.S.A.T.S.S.: 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. ACM, New York, NY, USA (2018)
25. Li, H., Shang, W., Hassan, A.E.: Which log level should developers choose for a new logging statement? Empirical Softw. Engg. **22**(4), 1684–1716 (2017). DOI 10.1007/s10664-016-9456-2. URL https://doi.org/10.1007/s10664-016-9456-2
26. Li, H., Shang, W., Zou, Y., E. Hassan, A.: Towards just-in-time suggestions for log changes. Empirical Softw. Engg. **22**(4), 1831–1865 (2017). DOI 10.1007/s10664-016-9467-z. URL https://doi.org/10.1007/s10664-016-9467-z
27. Malik, H., Hemmati, H., Hassan, A.E.: 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, pp. 1012–1021. IEEE Press, Piscataway, NJ, USA (2013). URL http://dl.acm.org/citation.cfm?id=2486788.2486927

28. Nagappan, N., Murphy, B., Basili, V.: The influence of organizational structure on software quality. In: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, pp. 521–530. IEEE (2008)
29. Nagaraj, K., Killian, C., Neville, J.: 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, pp. 26–26. USENIX Association, Berkeley, CA, USA (2012). URL http://dl.acm.org/citation.cfm?id=2228298.2228334
30. Oliner, A., Ganapathi, A., Xu, W.: Advances and challenges in log analysis. Commun. ACM **55**(2), 55–61 (2012). DOI 10.1145/2076450.2076466. URL http://doi.acm.org/10.1145/2076450.2076466
31. Oliner, A.J., Aiken, A.: 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, pp. 49–60. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/DSN.2011.5958206. URL http://dx.doi.org/10.1109/DSN.2011.5958206
32. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P.: 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, pp. 335–344 (2011)
33. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P.: 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)
34. Shang, W., Nagappan, M., Hassan, A.E.: Studying the relationship between logging characteristics and the code quality of platform software. Empirical Software Engineering **20**(1), 1–27 (2015)
35. Shang, W., Nagappan, M., Hassan, A.E., Jiang, Z.M.: Understanding log lines using development knowledge. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, ICSME '14, pp. 21–30. IEEE Computer Society, Washington, DC, USA (2014). DOI 10.1109/ICSME.2014.24. URL http://dx.doi.org/10.1109/ICSME.2014.24
36. Stol, K.J., Ralph, P., Fitzgerald, B.: Grounded theory in software engineering research: A critical review and guidelines. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp. 120–131. ACM, New York, NY, USA (2016). DOI 10.1145/2884781.2884833. URL http://doi.acm.org/10.1145/2884781.2884833
37. Tan, J., Pan, X., Kavulya, S., Gandhi, R., Narasimhan, P.: Salsa: Analyzing logs as state machines. In: Proceedings of the First USENIX Conference on Analysis of System Logs, WASL'08, pp. 6–6. USENIX Association, Berkeley, CA, USA (2008). URL http://dl.acm.org/citation.cfm?id=1855886.1855892
38. Wilcoxon, F., Wilcox, R.A.: Some rapid approximate statistical procedures. Lederle Laboratories (1964)
39. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, pp. 117–132. ACM, New York, NY, USA (2009). DOI 10.1145/1629575.1629587. URL http://doi.acm.org/10.1145/1629575.1629587
40. Yin, R.K.: Case study research: Design and methods. Sage publications (2013)
41. Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., Pasupathy, S.: 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, pp. 143–154. ACM, New York, NY, USA (2010). DOI 10.1145/1736020.1736038. URL http://doi.acm.org/10.1145/1736020.1736038
42. Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M.M., Tang, X., Zhou, Y., Savage, S.: Be conservative: Enhancing failure diagnosis with proactive logging. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pp. 293–306. USENIX Association, Berkeley, CA, USA (2012). URL http://dl.acm.org/citation.cfm?id=2387880.2387909

43. Yuan, D., Park, S., Zhou, Y.: Characterizing logging practices in open-source software. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 102–112. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2337223.2337236
44. Yuan, D., Park, S., Zhou, Y.: Characterizing logging practices in open-source software. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 102–112. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2337223.2337236
45. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S.: Improving software diagnosability via log enhancement. ACM Trans. Comput. Syst. **30**(1), 4:1–4:28 (2012). DOI 10.1145/2110356.2110360. URL http://doi.acm.org/10.1145/2110356.2110360
46. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S.: Improving software diagnosability via log enhancement. ACM Transactions on Computer Systems (TOCS) **30**(1), 4 (2012)
47. Zhang, H.: An investigation of the relationships between lines of code and defects. In: Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, pp. 274–283. IEEE (2009)
48. Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M.R., Zhang, D.: Learning to log: Helping developers make informed logging decisions. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pp. 415–425. IEEE Press, Piscataway, NJ, USA (2015). URL http://dl.acm.org/citation.cfm?id=2818754.2818807