Empir Software Eng
DOI 10.1007/s10664-016-9429-5

Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation

Boyuan Chen[1] · Zhen Ming (Jack) Jiang[1]

Abstract Log messages, which are generated by the debug statements that developers insert into the code at runtime, contain rich information about the runtime behavior of software systems. Log messages are used widely for system monitoring, problem diagnoses and legal compliances. Yuan et al. performed the first empirical study on the logging practices in open source software systems. They studied the development history of four C/C++ server-side projects and derived ten interesting findings. In this paper, we have performed a replication study in order to assess whether their findings would be applicable to Java projects in Apache Software Foundations. We examined 21 different Java-based open source projects from three different categories: server-side, client-side and supporting-component. Similar to the original study, our results show that all projects contain logging code, which is actively maintained. However, contrary to the original study, bug reports containing log messages take a longer time to resolve than bug reports without log messages. A significantly higher portion of log updates are for enhancing the quality of logs (e.g., formatting & style changes and spelling/grammar fixes) rather than co-changes with feature implementations (e.g., updating variable names).

Keywords Empirical study · Replication · Log messages · Logging code · Mining software engineering data · MSR

Communicated by: David Lo

✉ Boyuan Chen
chenfsd@gmail.com

✉ Zhen Ming (Jack) Jiang
zmjiang@cse.yorku.ca

[1] Software Construction, AnaLytics and Evaluation (SCALE) Laboratory York University, Toronto, ON, Canada

Published online: 07 May 2016

1 Introduction

Logging code refers to debug statements that developers insert into the source code. Log messages are generated by the logging code at runtime. Log messages, which are generated in many open source and commercial software projects, contain rich information about the runtime behavior of software projects. Compared to program traces, which are generated by profiling tools (e.g., JProfiler or DTrace) and contain low level implementation details (e.g., methodA invoked methodB), the information contained in the log messages is usually higher level, such as workload related (e.g., "Registration completed for user John Smith") or error related (e.g., "Error associated with adding an item into the shopping cart: deadlock encountered"). Log messages are used extensively for monitoring (Shang et al. 2014), remote issue resolution (BlackBerry Enterprise Server Logs Submission 2015), test analysis (Jiang et al. 2008, 2009) and legal compliance (Summary of Sarbanes-Oxley Act of 2002 2015). There are already many tools available for gathering and analyzing the information contained in log messages (e.g., logstash - open source log management (2015), Nagios Log Server - Monitor and Manage Your Log Data (2015), and Splunk (2015)). According to Gartner, tools for managing log messages are estimated to be a 1.5 billion market and have been growing more than 10 % every year (Gartner 2014).

There are three general approaches to instrumenting the projects with log messages (Woodside et al. 2007):

1. *Ad-hoc logging:* developers can instrument the projects with console output statements like "System.out" and "printf". Although ad-hoc logging is the easiest to use, extra care is needed to control the amount of data generated and to ensure that the resulting log messages are not garbled in the case of concurrent logging.
2. *General-purpose logging libraries:* compared to ad-hoc logging, instrumentation through general-purpose logging libraries provides additional programming support like thread-safe logging and multiple verbosity levels. For example, in LOG4J a logging library for Java (2016), developers can set their logging code with different verbosity levels like TRACE, DEBUG, INFO, WARN, ERROR, and FATAL, each of which can be used to support different development tasks.
3. *Specialized logging libraries:* these libraries can be used to facilitate recording particular aspects of the system behavior at runtime. For example, ARM (Application Response Measurement) (Group 2014) is an instrumentation framework, that is specialized at gathering performance information (e.g., response time) from the running projects.

The work done by Yuan et al. (2012) is the first work that empirically studies the logging practices in different open source software projects. They studied the development history of four open source software projects (Apache httpd, OpenSSH, PostgreSQL and Squid) and obtained ten interesting findings on the logging practices. Their findings can provide suggestions for developers to improve their existing logging practices and give useful insights for log management tools. However, it is not clear whether their findings are applicable to other software projects, as the four studied projects are server-side projects written in C/C++. The logging practices may not be the same for projects from other application categories, or projects written in other programming languages. For example, would projects developed in managed programming languages (e.g., Java or C#) log less compared to projects developed in unmanaged programming languages (e.g., C or C++) due to their additional programming constructs (e.g., automated memory management) and enhanced security? As log messages are used extensively in servers for monitoring and remote issue debugging (Hassan et al. 2008), would server-side projects log more than client-side projects?

Replication studies, which are very important in empirical sciences, address one of the main threats to validity (External Validity). Recent replication study in psychology has found that the findings in more than fifty out of the previous published one hundred studies did not hold (Estimating the reproducibility of psychological science 2015). Replication studies are also very important in empirical software engineering, as they can be used to compare the effectiveness of different techniques or to assess the validity of findings across various projects (Basili et al. 1999; Robles 2010). There have been quite a few replication studies done in the area of empirical software engineering (e.g., code ownership (Greiler et al. 2015), software mining techniques (Ghezzi and Gall 2013) and defect predictions (Premraj and Herzig 2011; Syer et al. 2015).

In this paper, we have replicated this study by analyzing the logging practices of 21 Java projects from the Apache Software Foundation (ASF) (2016). The projects in ASF are ideal case study subjects for this paper due to the following two reasons: (1) ASF contains hundreds of software projects, many of which are actively maintained and used by millions of people worldwide; (2) the development process of these ASF projects is well-defined and followed (Mockus et al. 2002). All the source code has been carefully peer-reviewed and discussed (Rigby et al. 2008). The studied 21 Java projects are selected from the following three different categories: server-side, client-side or support-component-based projects. Our goal is to assess whether the findings from the original study would be applicable to our selected projects. The contributions of this paper are as follows:

1. This is the first empirical study (to the best of our knowledge) on characterizing the logging practices in Java-based software projects. Each of the 21 studied projects is carefully selected based on its revision history, code size and category.
2. When comparing our findings against the original study, the results are analyzed in two dimensions: category (e.g., server-side vs. client-side) and programming language (Java vs. C/C++). Our results show that certain aspects of the logging practices (e.g., the pervasiveness of logging and the bug resolution time) are not the same as in the original study. To allow for easier replication and to encourage future research on this subject, we have prepared a replication package (The replication package 2015).
3. To assess the bug resolution time with and without log messages, the authors from the original study manually examined 250 randomly sampled bug reports. In this replication study, we have developed an automated approach that can flag bug reports containing log messages with high accuracy and analyzed all the bug reports. Our new approach is fully automated and avoids sampling bias (Bird et al. 2009; Rahman et al. 2013).
4. We have extended and improved the taxonomy of the evolution of logging code based on our results. For example, we have extended the scenarios of consistent updates to the log printing code from three scenarios in the original study to eight scenarios in our study. This improved taxonomy should be very useful for software engineering researchers who are interested in studying software evolution and recommender systems.

Paper Organization The rest of the paper is organized as follows. Section 2 summarizes the original study and introduces the terminology used in this paper. Section 3 provides an overview of our replication study and proposes five research questions. Section 4 explains the experimental setup. Sections 5, 6, 7, 8 and 9 describe the findings in our replication study and discuss the implications. Section 10 presents the related work. Section 11 discusses the threats to validity. Section 12 concludes this paper.

2 Summary of the Original Study

In this section, we give a brief overview of the original study. First, we introduce the terminologies and metrics used in the original study. These terminologies and metrics are closely followed in this paper. Then we summarize the findings in the original study.

2.1 Terminology

Logging code refers to the source code that developers insert into the software projects to track the runtime information. Logging code includes *log printing code* and *log non-printing code*. Examples of non-log printing code can be *logging object initialization* (e.g., “Logger logger = Logger.getLogger(Log4JMetri-csContext.class)”) and *other code related to logging* such as logging object operation (e.g., “eventLog.shutdown()”). The majority of the source code is not logging code but code related to feature implementations.

Log messages are generated by log printing code, while a project is running. For example, the log printing code “Log.info(‘username’ + userName + ‘ logged in from’ + location.getIP())” can generate the following log message: “username Tom logged in from 127.0.0.1” at runtime. As mentioned in Section 1, there are three approaches to add log printing code into the systems: ad-hoc logging, general-purpose logging libraries and specialized logging libraries.

There are typically four components contained in a piece of log-printing code: a *logging object*, a *verbosity level*, *static texts* and *dynamic contents*. In the above example, the logging object is “Log”; “info” is the verbosity level; “username” and “ logged in from” are the static texts; “userName” and “location.getIP()” are the dynamic contents. Note that “userName” is a variable and “location.getIP()” is a method invocation. Compared to the static texts, which remain the same at runtime, the dynamic contents could vary each time the log-printing code is invoked.

2.1.1 Taxonomy of the Evolution of the Logging Code

Figure 1 illustrates the taxonomy of the evolution of the logging code. The most general concept, the *evolution of logging code*, resides at the top of the hierarchy. It refers to any type of changes on the logging code. The evolution of logging code can be further broken down into four categories: *log insertion*, *log deletion*, *log move* and *log update* as shown in the second level of the diagram. *Log deletion*, *log move* and *log update* are collectively called *log modification*.

The four types of log changes can be applied on log printing code and non-log printing code. For example, log update can be further broken down into *log printing code update* and *log non-printing code update*. Similarly, log move can be broken into *log printing code move* and *log non-printing code move*. Since the focus of the original study is on updates to the log printing code, for the sake of brevity, we do not include further categorizations on log insertion, log deletion, and log move in Fig. 1.

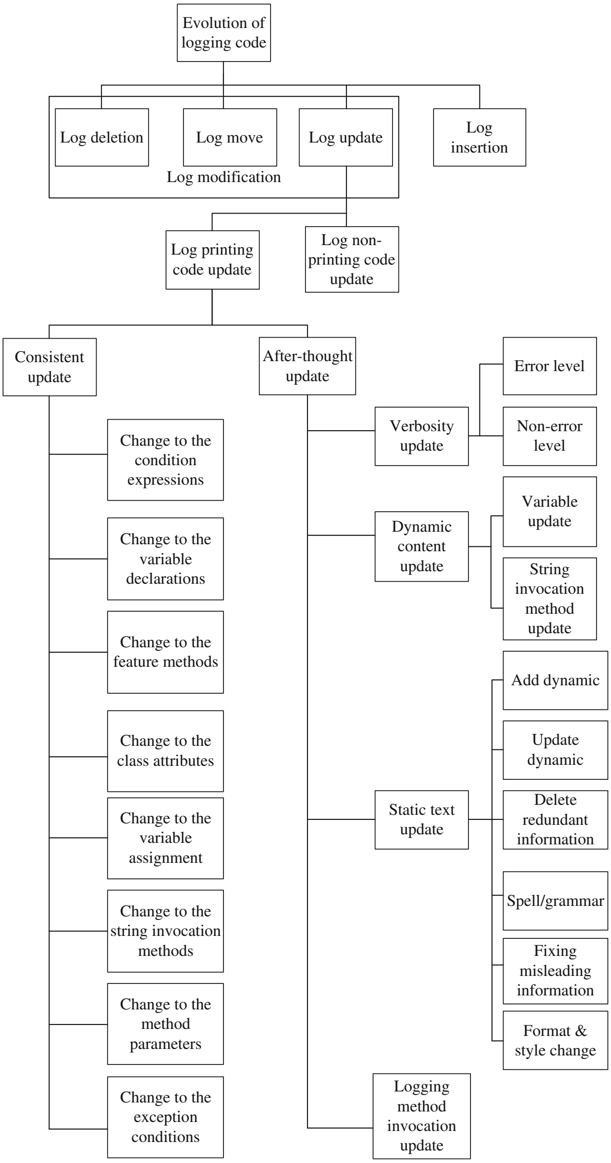
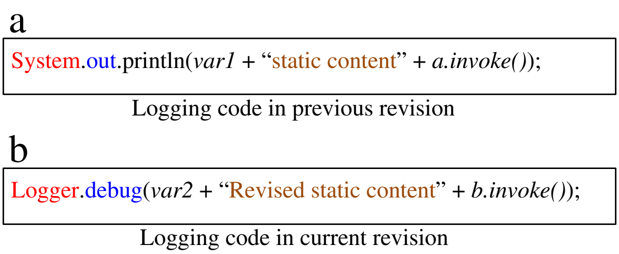

Evolution of logging code
Log deletion
Log move
Log update
Log insertion
Log modification
Log printing code update
Log non-printing code update
Consistent update
After-thought update
Change to the condition expressions
Change to the variable declarations
Change to the feature methods
Change to the class attributes
Change to the variable assignment
Change to the string invocation methods
Change to the method parameters
Change to the exception conditions
Verbosity update
Error level
Non-error level
Dynamic content update
Variable update
String invocation method update
Static text update
Add dynamic
Update dynamic
Delete redundant information
Spell/grammar
Fixing misleading information
Format & style change
Logging method invocation update

Fig. 1 Taxonomy of the evolution of the logging code

There are two types of changes related to updates to the log printing code: *consistent update* and *after-thought update*, as illustrated in the fourth level of Fig. 1. *Consistent updates* refer to changes to the log printing code and changes to the feature implementation code that are done in the same revision. For example, if the variable "userName" referred to in the above logging code is renamed to "customerName", a consistent log update would change the variable name inside log printing code to be like "Log.info('customername' + customerName + 'logged in from' + location.getIP())". We have expanded the scenarios of



a

```
System.out.println(var1 + “static content” + a.invoke());
```

Logging code in previous revision

b

```
Logger.debug(var2 + “Revised static content” + b.invoke());
```

Logging code in current revision

Fig. 2 Log printing code update example

Arten von Changes

consistent updates from three scenarios in the original study to eight scenarios in our study. For details, please refer to Section 8.

After-thought updates refer to updates to the log printing code that are not consistent updates. In other words, after-thought updates are changes to log-printing code that do not depend on other changes. There are four kinds of after-thought updates, corresponding to the four components to the log printing code: *verbosity updates*, *dynamic content updates*, *static text updates* and *logging method invocation updates*. Figure 2 shows an example with different kinds of changes highlighted in different colours: the changes in the logging method invocation are highlighted in red (System vs. Logger), the changes in the verbosity level in blue (out vs. debug), the changes in the dynamic contents in italic (var1 vs. var2 and a.invoke() vs. b.invoke()), the changes in static texts in yellow (“static content” vs. “Revised static content”). A dynamic content update is a generalization of a *variable update* in the original study. In this example, the variable “var1” is changed to “var2”. In the original study, such an update is called variable update. However, there is the case of “a.invoke()” getting updated to “b.invoke()”. This change is not a variable update but a string invocation method update. Hence, we rename these two kinds of updates to be dynamic content updates. There could be various reasons (e.g., fixing grammar/spelling issues or deleting redundant information) behind these after-thought updates. Please refer to Section 9 for details.

2.1.2 Metrics

The following metrics were used in the original study to characterize various aspects of logging:

- *Log density* measures the pervasiveness of software logging. It is calculated using this formula: $\frac{Total\ lines\ of\ source\ code\ (SLOC)}{Total\ lines\ of\ logging\ code(LOLC)}$. When measuring SLOC and LOLC, we only study the source code and exclude comments and empty lines.
- *Code churn* refers to the total number of lines of source code that is added, removed or updated for one revision (Nagappan and Ball 2005). As for log density, we only study the source code and exclude the comments and empty lines.
- *Churn of logging code*, which is defined in a similar way to code churn, measures the total number lines of logging code that is added, deleted or updated for one revision.
- *Average churn rate (of source code)* measures the evolution of the source code. The churn rate for one revision (i) is calculated using this formula: $\frac{Code\ churn\ for\ revision\ i}{SLOC\ for\ revision\ i}$. The average churn rate is calculated by taking the average value of the churn rates across all the revisions.

- *Average churn rate of logging code* measures the evolution of the logging code. The churn rate of the logging code for one revision (i) is calculated using this formula: $\frac{Churn\ of\ logging\ code\ for\ revision\ i}{LOLC\ for\ revision\ i}$. The average churn rate of the logging code is calculated by taking the average value among the churn rate of the logging code across all the revisions.

2.2 Findings from the Original Study

In the original study, the authors analyzed the logging practices of four open-source projects (Apache httpd, OpenSSH, PostgreSQL and Squid). These are server-side projects written in C and C++. The authors of the original study reported ten major findings. These findings, shown in the second column of Table 1 as “F1”, “F2”, ..., “F10”, are summarized below. For the sake of brevity, *F1* corresponds to *Finding 1*, and so on.

First, they studied the pervasiveness of logging by measuring the log density of the aforementioned four projects. They found that, on average, every 30 lines of code contained one line of logging code (*F1*).

Second, they studied whether logging can help diagnose software bugs by analyzing the bug resolution time of the selected bug reports. They randomly sampled 250 bug reports and compared the bug resolution time for bug reports with and without log messages. They found that bug reports containing log messages were resolved 1.4 to 3 times faster than bug reports without log messages (*F2*).

Third, they studied the evolution of the logging code quantitatively. The average churn rate of logging code was higher than the average churn rate of the entire code in three out of the four studied projects (*F3*). Almost one out of five code commits (18 %) contained changes to the logging code (*F4*).

Among the four categories of log evolutionary changes (log update, insertion, move and deletion), very few log changes (2 %) were related to log deletion or move (*F6*).

Fourth, they studied further one type of log changes: the updates to the log-printing code. They found that the majority (67 %) of the updates to the log-printing code were consistent updates (*F5*).

Finally, they studied the after-thought updates. They found that about one third (28 %) of the after-thought updates are verbosity level updates (*F7*), which were mainly related to error-level updates (and *F8*). The majority of the dynamic content updates were about adding new variables (*F9*). More than one third (39 %) of the updates to the static contents were related to clarifications (*F10*).

The authors also implemented a verbosity level checker which detected inconsistent verbosity level updates. The verbosity level checker is not replicated in this paper, because our focus is solely on assessing the applicability of their empirical findings on Java-based projects from the ASF.

3 Overview

This section provides an overview of our replication study. We propose five research questions (RQs) to better structure our replication studies. During the examination of these five RQs, we intend to validate the ten findings from the original study. As shown in Table 1, inside each RQ, one or multiple findings from the original study are checked. We compare our findings (denoted as “NF1”, “NF2”, etc.) against the findings in the original study (denoted as “F1”, “F2”, etc.) and report whether they are similar or different.

Table 1 Comparisons between the original and the current study

Research questions (RQs)	Finding comparison	Implications	Similar or different
(RQ1:) How pervasive is software logging?	F1: On average, every **30** lines of source code contains one line of logging code in server-side projects. NF1: On average, every **51** lines of source code contains one line of logging code in server-side projects. The log density is among server-side, client-side and supporting-component based projects	The pervasiveness of logging varies from project to project. The correlation between SLOC and LLOC is strong, which implies that larger projects tend to have more logging code. However, the correlation between SLOC and log density is weak. It means that the scale of a project is not an indicator different of the pervasiveness of logging. More research like Fu et al. (2014) is needed to study the rationales for software logging	Different
(RQ2:) Are bug reports containing log messages resolved faster than the ones without log messages?	F2: Bug reports containing log messages are resolved 1.4 to 3 times **faster** than bugs reports without. NF2: Bug reports containing log messages are resolved **slower** than bug reports without log messages for server-side and supporting-component based projects	Although there are multiple artifacts (e.g., test cases and stack traces) that are considered useful for developers to replicate issues reported in the bug reports, the factor of logging was not considered in those works. Further research is required to re-visit these studies to investigate the impact of logging on bug resolution time	Different
(RQ3:) How often is the logging code changed?	F3 and NF3: The average churn rate of logging code is almost **two times** (1.8) compared to the entire code.	There are many log analysis applications developed to monitor and debug the health of server-based projects (Oliner et al. 2012). Additional research is required to study the co-evolution of logging code and log monitoring/analysis applications	Similar
	F4 and NF4: Logging code is modified in around **20 %** of all committed revisions		Similar
	F6: Deleting or moving log printing code accounts for only **2 %** of all log modifications	Deleting/moving logging code may hinder the understanding of runtime behavior of these projects.	Different

Table 1 (continued)

Research questions (RQs)	Finding comparison	Implications	Similar or different
	NF6: Deleting and moving log printing code accounts for **26** and **10 %** of all log modifications, respectively	New research is required to assess the risk of deleting/moving logging code for Java-based systems.	
(RQ4:) What are the characteristics of consistent updates to the log printing code?	F5: **67 %** of updates to the log printing code are consistent updates. NF5: **41 %** of updates to the log printing code are consistent updates	There are many fewer consistent updates discovered in our study compared to the original study. We suspect this could be mainly attributed to the introduction of additional program constructs in Java (e.g., exceptions and class attributes). This highlights the need for additional research and tools for recommending changes in the logging code during each code commit	Different
(RQ5:) What are the characteristics of the after-thought updates to the log printing code?	F7: **26 %** of after-thought updates are verbosity level updates, **72 %** of verbosity level updates involve at least one error event. NF7: **21 %** of after-thought updates are verbosity level updates, **20 %** of verbosity level updates involve at least one error event.	Contrary to the original study, which found that developers are confused by verbosity level, we find that developers usually have a better understanding of verbosity levels in Java-based projects in ASF. Further qualitative studies (e.g., developer surveys) are required to understand the rationales behind such differences.	Different
	F8: **57 %** of non-error level updates are changing between two non-default levels. NF8: **15 %** of non-error level updates are changing between two non-default levels		Different
	F9: **27 %** of the after-thought updates are related to **variable logging**. The majority of these updates are adding new variables.	Research on log enhancement should not only focus on suggesting which variables to log (e.g., Yuan et al. 2011; Zhu et al. 2015) but also on suggesting string invocation	Different

Table 1 (continued)

Research questions (RQs)	Finding comparison	Implications	Similar or different
	NF9: Similar to the original study, adding variables into the log printing code is the most common after-thought update related to variables. Different from the original study, we have found a new type of dynamic contents, which is string invocation methods (SIMs)	methods	
	F10 and NF10: **Fixing misleading information** is the most frequent updates to the static text	Log messages are actively used in practice to monitor and diagnose failures. However, out-dated log messages may confuse developers and cause bugs. Additional research is needed to leverage techniques from natural language processing and information retrieval to detect such inconsistencies automatically	Similar

RQ1: How pervasive is software logging? Log messages have been used widely for legal compliance (Summary of Sarbanes-Oxley Act of 2002 2015), monitoring (Splunk 2015; Oliner et al. 2012) and remote issue resolution (Hassan et al. 2008) in server-side projects. It would be beneficial to quantify how pervasive software logging is. In this research question, we intend to study the pervasiveness of logging by calculating the log density of different software projects. The lower the log density is, the more pervasive software logging is.

RQ2: Are bug reports containing log messages resolved faster than the ones without log messages? Previous studies (Bettenburg et al. 2008; Zimmermann et al. 2010) showed that artifacts that help to reproduce failure issues (e.g., test cases, stack traces) are considered useful for developers. As log messages record the runtime behavior of the system when the failure occurs, the goal of this research question is to examine whether bug reports containing log messages are resolved faster.

RQ3: How often is the logging code changed? Software projects are constantly maintained and evolved due to bug fixes and feature enhancement (Rajlich 2014). Hence, the logging code needs to be co-evolved with the feature implementations. This research question aims to quantitatively examine the evolution of the logging code. Afterwards, we will perform a deeper analysis on two types of evolution of the log printing code: consistent updates (RQ4) and after-thought updates (RQ5).

RQ4: What are the characteristics of consistent updates to the log printing code? Similar to out-dated code comments (Tan et al. 2007), out-dated log printing code can confuse and mislead developers and may introduce bugs. In this research question, we study the scenarios of different consistent updates to the log printing code.

RQ5: What are the characteristics of the after-thought updates to the log printing code? Ideally, most of the changes to the log printing code should be consistent updates. However, in reality some changes in the logging printing code are after-thought updates. The goal of this research question is to quantify the amount of after-thought updates and to find out the rationales behind them.

Sections 5, 6, 7, 8, 9 cover the above five RQs, respectively. For each RQ, we first explain the process of data extraction and data analysis. Then we summarize our findings and discuss the implications. As shown in Table 1, each research question aims to replicate one or more of the findings from the original study. Our findings may agree or disagree with the original study, as shown in the last column of Table 1.

4 Experimental Setup

This section describes the experimental setup for our replication study. We first explain our selection of software projects. Then we describe our data gathering and preparation process.

4.1 Subject Projects

In this replication study, 21 different Java-based open source software projects from Apache Software Foundation (2016) are selected. All of the selected software projects are widely used and actively maintained. These projects contain millions of lines of code and three to ten years of development history. Table 2 provides an overview of these projects including a description of the project, the type of bug tracking systems, the start/end code revision

Table 2 Studied Java-based ASF projects

Category	Project	Description	Bug Tracking System	Code History (First, Last)	Bug History (First, Last)
Server	Hadoop	Distributed computing system	Jira	(2008-01-16, 2014-10-20)	(2006-02-02, 2015-02-12)
	Hbase	Hadoop database	Jira	(2008-02-04, 2014-10-27)	(2008-02-01, 2015-03-25)
	Hive	Data warehouse infrastructure	Jira	(2010-10-08, 2014-11-02)	(2008-09-11, 2015-04-21)
	Openmeetings	Web conferencing	Jira	(2011-12-9, 2014-10-31)	(2011-12-05, 2015-04-20)
	Tomcat	Web server	Bugzilla	(2005-08-05, 2014-11-01)	(2009-02-17, 2015-04-14)
Client	Ant	Building tool	Bugzilla	(2005-04-15, 2014-10-29)	(2000-09-16, 2015-03-26)
	Fop	Print formatter	Jira	(2005-06-23, 2014-10-23)	(2001-02-01, 2015-09-17)
	JMeter	Load testing tool	Bugzilla	(2011-11-01, 2014-11-01)	(2001-06-07, 2015-04-16)
	Rat	Release audit tool	Jira	(2008-05-07, 2014-10-18)	(2008-02-03, 2015-09-29)
	Maven	Build manager	Jira	(2004-12-15, 2014-11-01)	(2004-04-13, 2015-04-20)
SC	ActiveMQ	Message broker	Jira	(2005-12-02, 2014-10-09)	(2004-4-20, 2015-3-25)
	Empire-db	Relational database abstraction layer	Jira	(2008-07-31, 2014-10-27)	(2008-08-08, 2015-03-19)
	Karaf	OSGi based runtime	Jira	(2010-06-25, 2014-10-14)	(2009-04-28, 2015-04-08)
	Log4j	Logging library	Jira	(2005-10-09, 2014-08-28)	(2008-04-24, 2015-03-25)
	Lucene	Text search engine library	Jira	(2005-02-02, 2014-11-02)	(2001-10-09, 2015-03-24)
	Mahout	Environment for scalable algorithms	Jira	(2008-01-15, 2014-10-29)	(2008-01-30, 2015-04-16)
	Mina	Network application framework	Jira	(2006-11-18, 2014-10-25)	(2005-02-06, 2015-03-16)
	Pig	Programming tool	Jira	(2010-10-03, 2014-11-01)	(2007-10-10, 2015-03-25)
	Pivot	Platform for building installable Internet applications	Jira	(2009-03-06, 2014-10-13)	(2009-01-26, 2015-04-17)
	Struts	Framework for web applications	Jira	(2004-10-01, 2014-10-27)	(2002-05-10, 2015-04-18)
	Zookeeper	Configuration service	Jira	(2010-11-23, 2014-10-28)	(2008-06-06, 2015-03-24)

date and the first/last creation date for bug reports. We classify these projects into three categories: server-side, client-side, and supporting-component based projects:

1. **Server-side projects:** In the original study, the authors studied four server-side projects. As server-side projects are used by hundreds or millions of users concurrently, they rely heavily on log messages for monitoring, failure diagnosis and workload characterization (Oliner et al. 2012; Shang et al. 2014). Five server-side projects are selected in our study to compare the original results on C/C++ server-side projects. The selected projects cover various application domains (e.g., database, web server and big data).
2. **Client-side projects:** Client-side projects also contains log messages. In this study, five client-based projects, which are from different application domains (e.g., software testing and release management), are selected to assess whether the logging practices are similar to the server-based projects.
3. **Supporting-component based (SC-based) projects:** Both server and client-side projects can be built using third party libraries or frameworks. Collectively, we call them supporting components. For the sake of completeness, 11 different SC-based projects are selected. Similar to the above two categories, these projects are from various applications domains (e.g., networking, database and distributed messaging).

4.2 Data Gathering and Preparation

Five different types of software development datasets are required in our replication study: release-level source code, bug reports, code revision history, logging code revision history and log printing code revision history.

4.2.1 Release-Level Source Code

The release-level source code for each project is downloaded from the specific web page of the project. In this paper, we have downloaded the latest stable version of the source code for each project. The source code is used for the RQ1 to calculate the log density.

4.2.2 Bug Reports

Data Gathering The selected 21 projects use two types of bug tracking systems: BugZilla and Jira, as shown in Table 2. Each bug report from these two systems can be downloaded individually as an XML file. These bug reports are automatically downloaded in a two-step process in our study. In step one, a list of bug report IDs are retrieved from the BugZilla and Jira website for each of the project. Each bug report (in XML format) corresponds to one unique URL in these systems. For example, in the Ant project, bug report 8689 corresponds to https://bz.apache.org/bugzilla/show_bug.cgi?ctype=xml&id=8689. Each URL for the bug reports is similar except for the "id" part. We just need to replace the id number each time. In step two, we automatically downloaded the XML format files of the bug reports based on the re-constructed URLs from the bug IDs. The Hadoop project contains four sub-projects: Hadoop-common, Hdfs, Mapreduce and Yarn, each of which has its own bug tracking website. The bug reports from these sub-projects are downloaded and merged into the Hadoop project.

Data Processing Different bug reports can have different status. A script is developed to filter out bug reports whose status are not "Resolved", "Verified" or "Closed". The sixth column in Table 2 shows the resulting dataset. The earliest bug report in this dataset was opened in 2000 and the latest bug report was opened in 2015.

4.2.3 Fine-Grained Revision History for Source Code

Data Gathering The source code revision history for all the ASF projects is archived in a giant subversion repository. ASF hosts periodic subversion data dumps online (Dumps of the ASF Subversion repository 2015). We downloaded all the svn dumps from the years between 1999 (the earliest) and 2014 (the latest). A local mirror of the software repositories is built for all the ASF projects. The 64 GB of dump files result in more than 200 GB of subversion repository data.

Data Processing We use the following tools to extract the evolutionary information from the subversion repository:

- J-REX (Shang et al. 2009) is an evolutionary extractor, which we use to automatically extract the source code as well as meta information (e.g., committer names, commit logs, etc.) for all the revisions of the 21 projects. Different revisions of the same source code files are recorded as separate files. For example, the source code of the first and the second revisions of Foo.java are recorded as Foo_v1.java, Foo_v2.java, respectively.
- ChangeDistiller (CD) (Fluri et al. 2007) parses two adjacent revisions (e.g., Foo_v1.java and Foo_v2.java) of the source code into Abstract Syntax Trees (ASTs), compares the ASTs using a tree differencing algorithm and outputs a list of fine-grained code changes. Examples of such changes can be updates to a particular method invocation or removing a method declaration.
- We have developed a post-processing script to be used after CD to measure the file-level and method-level code churn for each revision.

The above process is applied to all the revisions of all the Java files from the selected 21 projects. The resulting dataset records the fine-grained evolutionary information. For example, for Hadoop, there are a total of 25,944 revisions. For each revision, the name of the committer, the commit time, commit log, the code churn as well as the detailed list of code changes are recorded. For example, revision *688920* was submitted by *omalley* at 19:33:43 on August 25, 2008 for “HADOOP-3854. Add support for pluggable servlet filters in the HttpServers.”. In this revision, 8 Java files are updated and no Java files are added or deleted. Among the 8 updated files, four methods are updated in “/hadoop/core/trunk/src/core/org/apache/hadoop/http/HttpServer.java”, along with five methods that are inserted. The code churn for this file is 125 lines of code.

4.2.4 Fine-Grained Revision History for the Logging Code

Based on the above fine-grained historical code changes, we applied heuristics to identify the changes of the logging code among all the source code changes. Our approach, which is similar to previous work (Fu et al. 2014; Shang et al. 2015; Yuan et al. 2012), uses regular expressions to match the source code. The regular expressions used in this paper are _“.*?(pointcut|aspect|log|info|debug|error |fatal|warn |trace|(system\.out)|(system\.err)).*?(.*?);”_:

- “(system\.out)|(system\.err))” is included to flag source code that uses standard output (System.out) and standard error (System.err).

- Keywords like “log” and “trace” are included, as the logging code, which uses logging libraries like log4j, often uses logging objects like “log” or “logger” and verbosity levels like “trace” or “debug”.
- Keywords like “pointcut” and “aspect” are also include to flag logging code that uses the AspectJ (The AspectJ project 2015).

After the initial regular expression matching, the resulting dataset is further filtered to removed code snippets that contain wrongly matched words like “login”, “dialog”, etc. We manually sampled 377 pieces of logging code, which corresponds to a 95 % of confidence level with a 5 % confidence interval. The accuracy of our technique is 95 %, which is comparable to the original study (94 % accuracy).

4.2.5 Fine-Grained Revision History for the Log Printing Code

Logging code contains log printing code and non-log printing code. The dataset obtained above (Section 4.2.4) is further filtered to exclude code snippets that contain assignments (“=”) and does not have quoted strings. The resulting dataset is the fine-grained revision history containing only the log printing code. We also manually verified 377 log printing code from different projects. The accuracy of our approach is 95 %.

5 (RQ1:) How Pervasive is Software Logging?

In this section, we studied the pervasiveness of software logging.

5.1 Data Extraction

We downloaded the source code of the recent stable releases of the 21 projects and ran SLOCCOUNT (Wheeler http://www.dwheeler.com/sloccount/) to obtain the SLOC for each project. SLOCCOUNT only counts the actual lines of source code and excludes the comments and the empty lines. A small utility, which uses regular expressions and JDT (JDT Java development tools 2015), is applied to automatically recognize the logging code and count LOLC for this version. Please refer to Section 4.2.4 for the approach to automatically identify logging code.

5.2 Data Analysis

Log density is defined as the ratio between SLOC and LOLC. Smaller log density indicates higher likelihood that developers write logging code in this project. As we can see from Table 3, the log density value from the selected 21 projects varies. For server-side projects, the average log density is bigger in our study compared to the original study (51 vs. 30). In addition, the range of the log density in server-side projects is wider (29 to 83 in our study vs. 17 to 38 in the original study). The log density is generally bigger in client-side projects than server-side projects (63 vs. 51). For SC-based projects, the average log density is the lowest (48) among all three categories. The range of the log density in SC-side project is the widest (6 to 277). Compared to the original study, the average log density across all three categories is higher in our study.

The Spearman rank correlation is calculated for SLOC vs. LOLC, SLOC vs. log density, and LOLC vs. log density among all the project. Our results show that there is a strong

Table 3 Logging code density of all the projects

Category	Project	Total lines of source code (SLOC)	Total lines of logging code (LOLC)	Log density
Server	Hadoop (2.6.0)	891,627	19,057	47
	Hbase (1.0.0)	369,175	9,641	38
	Hive (1.1.0)	450,073	5,423	83
	Openmeetings (3.0.4)	51,289	1,750	29
	Tomcat (8.0.20)	287,499	4,663	62
	Subtotal	2,049,663	40,534	**51**
Client	Ant (1.9.4)	135,715	2,331	58
	Fop (2.0)	203,867	2,122	96
	JMeter (2.13)	111,317	2,982	37
	Maven (2.5.1)	20,077	94	214
	Rat (0.11)	8,628	52	166
	Subtotal	479,604	7,581	**63**
SC	ActiveMQ (5.9.0)	298,208	7,390	40
	Empire-db (2.4.3)	43,892	978	45
	Karaf (4.0.0.M2)	92,490	1,719	54
	Log4j (2.2)	69,678	4,509	15
	Lucene (5.0.0)	492,266	1,779	277
	Mahout (0.9)	115,667	1,670	69
	Mina (3.0.0.M2)	18,770	303	62
	Pig (0.14.0)	242,716	3,152	77
	Pivot (2.0.4)	96,615	408	244
	Struts (2.3.2)	156,290	2,513	62
	Zookeeper (3.4.6)	61,812	10,993	6
	Subtotal	1,688,404	35,414	**48**
	Total	4,217,671	83,529	**50**

correlation between SLOC and LOLC (0.69), indicating that projects with bigger code-base tend to have more logging code. However, the density of logging is not correlated with the size of the system (0.11).

5.3 Summary

NF1: Compared to the original result, the log density for server-side projects is bigger (51 vs. 30). In addition, the average log density of the server-side, client-side and SC-base projects are all different. The range of the log density values varies dramatically among different projects.

Implications: The pervasiveness of logging varies from projects to projects. Although larger projects tend to have more logging code, there is no correlation between SLOC and log density. More research like (Fu et al. 2014) is needed to study the rationales for software logging.

6 (RQ2:) Are Bug Reports Containing Log Messages Resolved Faster than the Ones Without Log Messages?

Bettenburg et al. (2008) (Zimmermann et al. 2010) have found that developers preferred bug reports that contain test cases and stack traces, as these artifacts help reproduce the reported issues. However, they did not look into bug reports that contain log messages. As log messages may provide useful runtime information, the goal of this RQ is to check if bug reports containing log messages are resolved faster than bug reports without.

In the original study, the authors randomly sampled 250 bug reports and categorized them into bug reports containing log messages (BWLs) or bug reports not containing any log messages (BNLs). Then they compared the median of the bug resolution time (BRT) between these two categories. In this RQ, we improved the original technique in two ways. First, rather than manual sampling, we have developed a categorization technique that can automatically flag BWLs with high accuracy. Our technique, which analyzes all the bug reports, can avoid the potential risk of sampling bias (Bird et al. 2009; Rahman et al. 2013). Second, we carried out a more thorough statistical analysis to compare the BRT between BWLs and BNLs.

6.1 Data Extraction

The data extraction process of this RQ consists of two steps: we first categorized the bug reports into BWLs and BNLs. Then we compared the resolution time for bug reports from these two categorizes.

6.1.1 Automated Categorization of Bug Reports

The main objective of our categorization technique is to automatically recognize log messages in the *description* and/or *comments* sections of the bug reports. Figure 3 illustrates the process. We provide a step-by-step description of our technique using real-world examples illustrated in following figures(the texts highlighted in blue are the log messages):

- bug reports that contain neither log messages nor log printing code (Fig. 4a);
- bug reports that contain log messages not coming from this project (Fig. 4b);
- bug reports that contain log messages in the Description section (Fig. 5a);
- bug reports that contain log messages in the Comments section (Fig. 5b);
- bug reports that do not contain log messages but only the log printing code(in red) (Fig. 6a);

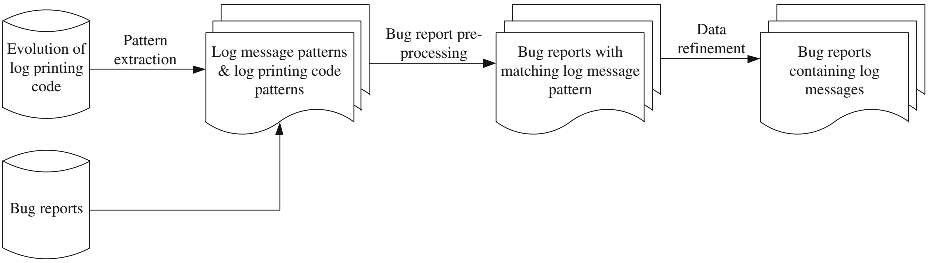
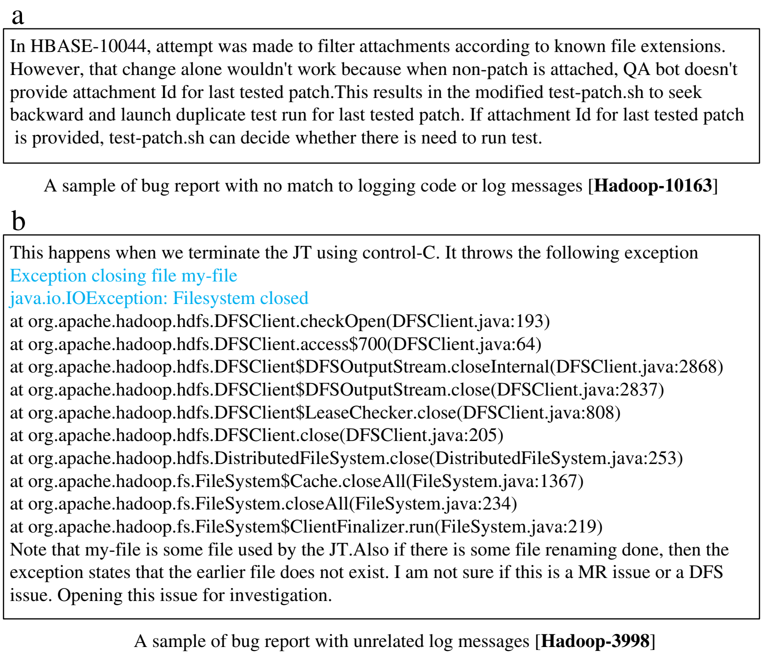

Evolution of log printing code
Pattern extraction
Log message patterns & log printing code patterns
Bug report pre-processing
Bug reports with matching log message pattern
Data refinement
Bug reports containing log messages
Bug reports

Fig. 3 An overview of our automated bug report categorization technique



a

In HBASE-10044, attempt was made to filter attachments according to known file extensions. However, that change alone wouldn't work because when non-patch is attached, QA bot doesn't provide attachment Id for last tested patch.This results in the modified test-patch.sh to seek backward and launch duplicate test run for last tested patch. If attachment Id for last tested patch is provided, test-patch.sh can decide whether there is need to run test.

A sample of bug report with no match to logging code or log messages [**Hadoop-10163**]

b

This happens when we terminate the JT using control-C. It throws the following exception

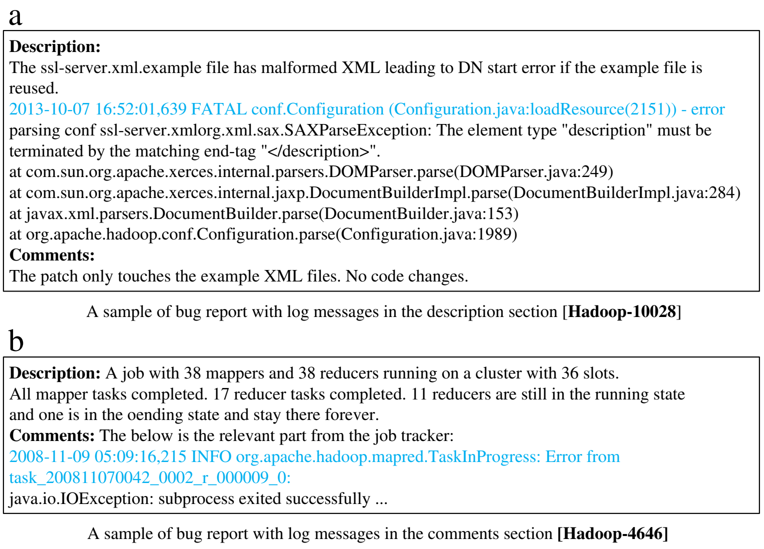
```
Exception closing file my-file
java.io.IOException: Filesystem closed
at org.apache.hadoop.hdfs.DFSClient.checkOpen(DFSClient.java:193)
at org.apache.hadoop.hdfs.DFSClient.access$700(DFSClient.java:64)
at org.apache.hadoop.hdfs.DFSClient$DFSOutputStream.closeInternal(DFSClient.java:2868)
at org.apache.hadoop.hdfs.DFSClient$DFSOutputStream.close(DFSClient.java:2837)
at org.apache.hadoop.hdfs.DFSClient$LeaseChecker.close(DFSClient.java:808)
at org.apache.hadoop.hdfs.DFSClient.close(DFSClient.java:205)
at org.apache.hadoop.hdfs.DistributedFileSystem.close(DistributedFileSystem.java:253)
at org.apache.hadoop.fs.FileSystem$Cache.closeAll(FileSystem.java:1367)
at org.apache.hadoop.fs.FileSystem.closeAll(FileSystem.java:234)
at org.apache.hadoop.fs.FileSystem$ClientFinalizer.run(FileSystem.java:219)
```

Note that my-file is some file used by the JT.Also if there is some file renaming done, then the exception states that the earlier file does not exist. I am not sure if this is a MR issue or a DFS issue. Opening this issue for investigation.

A sample of bug report with unrelated log messages [**Hadoop-3998**]

Fig. 4 Sample bug reports with no related log messages

- bug reports that contain both the log messages and log printing code(in red) (Fig. 6b);
- bug reports that do not contain log messages but contain the keywords(in red) from log messages in the textual contents (Fig. 7).



a

Description:
The ssl-server.xml.example file has malformed XML leading to DN start error if the example file is reused.

```
2013-10-07 16:52:01,639 FATAL conf.Configuration (Configuration.java:loadResource(2151)) - error
parsing conf ssl-server.xmlorg.xml.sax.SAXParseException: The element type "description" must be
terminated by the matching end-tag "</description>".
at com.sun.org.apache.xerces.internal.parsers.DOMParser.parse(DOMParser.java:249)
at com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderImpl.parse(DocumentBuilderImpl.java:284)
at javax.xml.parsers.DocumentBuilder.parse(DocumentBuilder.java:153)
at org.apache.hadoop.conf.Configuration.parse(Configuration.java:1989)
```

Comments:
The patch only touches the example XML files. No code changes.

A sample of bug report with log messages in the description section [**Hadoop-10028**]

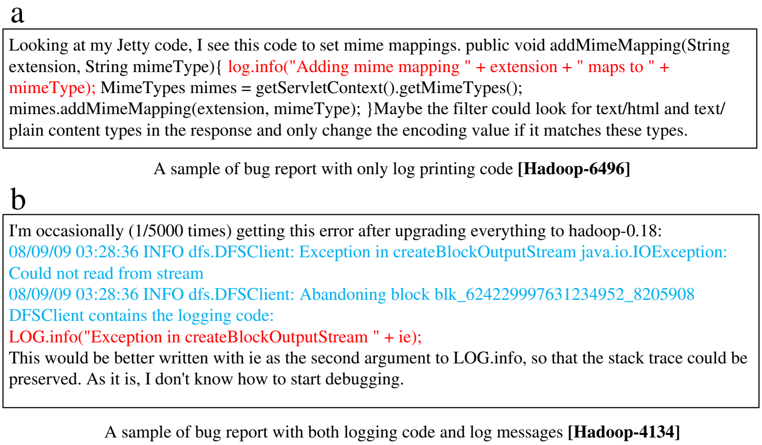
b

Description: A job with 38 mappers and 38 reducers running on a cluster with 36 slots. All mapper tasks completed. 17 reducer tasks completed. 11 reducers are still in the running state and one is in the oending state and stay there forever.
Comments: The below is the relevant part from the job tracker:

```
2008-11-09 05:09:16,215 INFO org.apache.hadoop.mapred.TaskInProgress: Error from
task_200811070042_0002_r_000009_0:
java.io.IOException: subprocess exited successfully ...
```

A sample of bug report with log messages in the comments section **[Hadoop-4646]**

Fig. 5 Sample bug reports with log messages



a

Looking at my Jetty code, I see this code to set mime mappings. public void addMimeMapping(String extension, String mimeType){ log.info("Adding mime mapping " + extension + " maps to " + mimeType); MimeTypes mimes = getServletContext().getMimeTypes(); mimes.addMimeMapping(extension, mimeType); }Maybe the filter could look for text/html and text/plain content types in the response and only change the encoding value if it matches these types.

A sample of bug report with only log printing code **[Hadoop-6496]**

b

I'm occasionally (1/5000 times) getting this error after upgrading everything to hadoop-0.18:
08/09/09 03:28:36 INFO dfs.DFSClient: Exception in createBlockOutputStream java.io.IOException: Could not read from stream
08/09/09 03:28:36 INFO dfs.DFSClient: Abandoning block blk_624229997631234952_8205908
DFSClient contains the logging code:
LOG.info("Exception in createBlockOutputStream " + ie);
This would be better written with ie as the second argument to LOG.info, so that the stack trace could be preserved. As it is, I don't know how to start debugging.

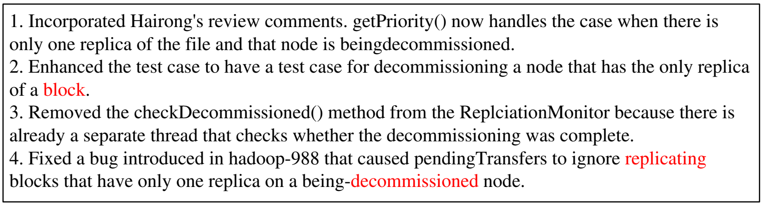
A sample of bug report with both logging code and log messages **[Hadoop-4134]**

Fig. 6 Sample bug reports with logging code

Our technique uses the following two types of datasets:

- **Bug Reports:** The contents of the bug reports, whose status are "Closed", "Resolved" or "Verified", from the 21 projects have been downloaded and stored in the XML file format. Please refer to Section 4.2.2 for a detailed description of this process.
- **Evolution of the Log Printing Code:** A historical dataset, which contains the fine-grained revision history for the log printing code (log update, log insert, log deletion and log move), has been extracted from the code repositories for all the projects. For details, please refer to Section 4.2.5.

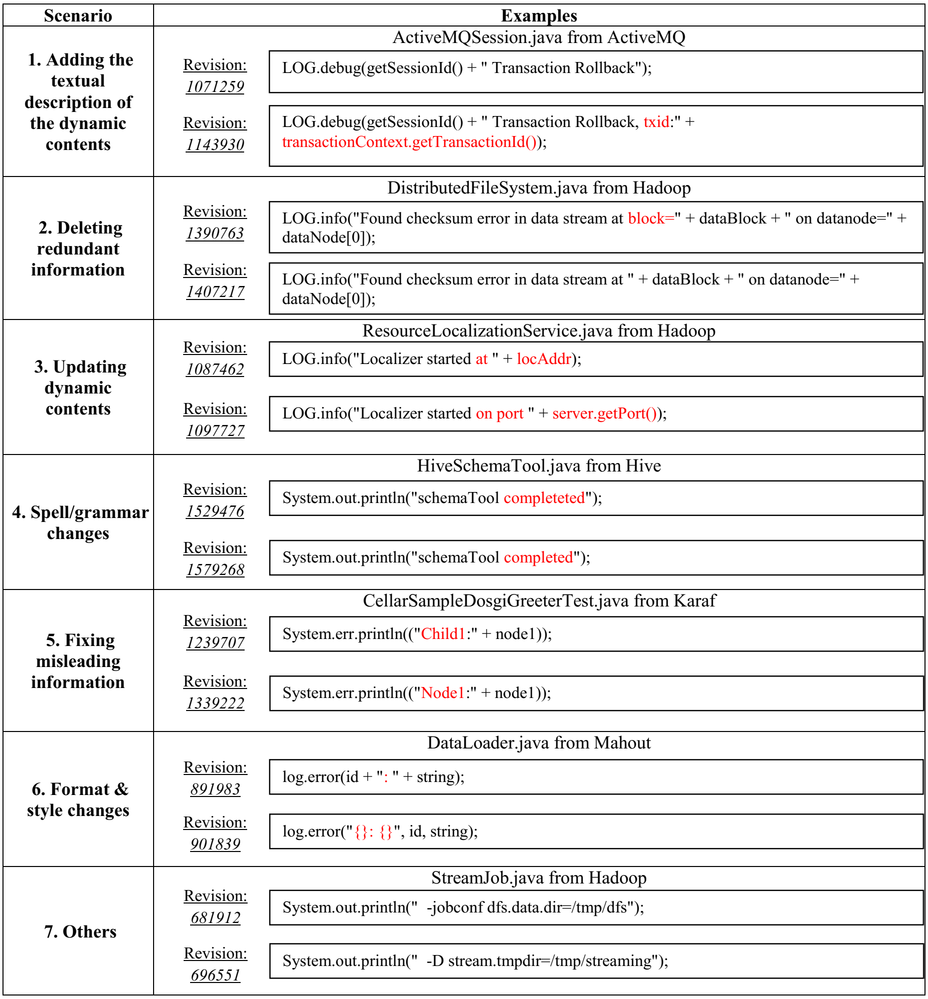
Pattern Extraction For each project, we extract two types of patterns: static log-printing code patterns and log message patterns. *Static log-printing code patterns* refer to all the snippets of log printing code that ever existed throughout the development history. For example, "log.info('Adding mime mapping' + extension + 'maps to' + mimeType')" in Fig. 6a is a static log-printing code pattern. Subsequently, *log message patterns* are derived based on the static log-printing code patterns. The above log printing code pattern would yield the following log message pattern: "Adding mime mapping * maps to *". The static log-printing code patterns are needed to remove the false alarms (a.k.a., all the log printing code) in a bug report, whereas the log message patterns are needed to flag all the log messages in a bug report.



1. Incorporated Hairong's review comments. getPriority() now handles the case when there is only one replica of the file and that node is beingdecommissioned.
2. Enhanced the test case to have a test case for decommissioning a node that has the only replica of a block.
3. Removed the checkDecommissioned() method from the ReplciationMonitor because there is already a separate thread that checks whether the decommissioning was complete.
4. Fixed a bug introduced in hadoop-988 that caused pendingTransfers to ignore replicating blocks that have only one replica on a being-decommissioned node.

Fig. 7 A sample of bug report with textual contents mistakenly matched to logging patterns **[Hadoop-1184]**

Pre-processing Only bug reports containing log messages are relevant for this RQ. Hence, bug reports like the one shown in Fig. 6a should be filtered out. However, the structure and the content of the logging code are very similar to the log messages, as log messages (e.g., "Tom logged in at 10:20") are generated as a result of executing the log printing code ("Log.info(user + 'logged in at' + date.time())"). We cannot directly match the log message patterns with the bug reports, as bug reports containing only the logging code (e.g., Fig. 6a) would be also mistakenly matched. Hence, if the contents of the *description* or the *comments* sections match the static log-printing code patterns, they are replaced with empty strings. Take Hadoop bug report *4134* (shown in Fig. 6b) as an example. The static log-printing code patterns can only match the logging code "LOG.info('Exception in createBlockOutputStream' + ie);", but not the log message "Exception in createBlockOutputStream java.io.IOException ...".



Scenario	Examples	
1. Adding the textual description of the dynamic contents	ActiveMQSession.java from ActiveMQ	
	Revision: *1071259*	LOG.debug(getSessionId() + " Transaction Rollback");
	Revision: *1143930*	LOG.debug(getSessionId() + " Transaction Rollback, txid:" + transactionContext.getTransactionId());
2. Deleting redundant information	DistributedFileSystem.java from Hadoop	
	Revision: *1390763*	LOG.info("Found checksum error in data stream at block=" + dataBlock + " on datanode=" + dataNode[0]);
	Revision: *1407217*	LOG.info("Found checksum error in data stream at " + dataBlock + " on datanode=" + dataNode[0]);
3. Updating dynamic contents	ResourceLocalizationService.java from Hadoop	
	Revision: *1087462*	LOG.info("Localizer started at " + locAddr);
	Revision: *1097727*	LOG.info("Localizer started on port " + server.getPort());
4. Spell/grammar changes	HiveSchemaTool.java from Hive	
	Revision: *1529476*	System.out.println("schemaTool completeted");
	Revision: *1579268*	System.out.println("schemaTool completed");
5. Fixing misleading information	CellarSampleDosgiGreeterTest.java from Karaf	
	Revision: *1239707*	System.err.println(("Child1:" + node1));
	Revision: *1339222*	System.err.println(("Node1:" + node1));
6. Format & style changes	DataLoader.java from Mahout	
	Revision: *891983*	log.error(id + ": " + string);
	Revision: *901839*	log.error("{}: {}", id, string);
7. Others	StreamJob.java from Hadoop	
	Revision: *681912*	System.out.println(" -jobconf dfs.data.dir=/tmp/dfs");
	Revision: *696551*	System.out.println(" -D stream.tmpdir=/tmp/streaming");

Fig. 8 A sample of falsely categorized bug report **[Hadoop-11074]**

Pattern Matching In this step, a bug report is selected, if its textual contents from the description or the comments sections match any of the log message patterns. The selected bug reports are the likely candidates for BWLs. In this step, bug reports like the ones shown in Figs. 4b, 5a, b, and 7 are selected.

Data Refinement However, there could still be false positives in the resulting bug report dataset. One of the main reasons is that some words used in the log messages may overlap with the textual content. For example, although “block replica decommissioned” in Fig. 7 matches one of the log message patterns, it is not a log message but part of the textual contents of this bug report. To further refine the dataset, a new filtering rule is introduced so that bug reports without any timestamps are excluded, as log messages are usually printed with timestamps showing the generation time for the log messages. Various format of timestamps used in the selected projects (e.g. “2000-01-02 19:19:19” or “2010080907”, etc.) are included in this filter rule. In this step, bug reports in Fig. 7 are removed. The remaining bug reports after this step are BWLs. All the other bug reports are BNLs.

To evaluate our technique, 370 out of 9,646 bug reports are randomly sampled from the Hadoop Common project (which is a sub project of Hadoop). The samples correspond to a confidence level of 95 % with a confidence interval of ±5 %. The performance of our categorization technique is: 100 % recall, 96 % precision and 99 % accuracy. Our technique cannot hit 100 % precision as some short log message patterns may frequently appear as the regular textual contents in the bug report. Figure 8 shows one example. Although Hadoop bug report *11074* contains the date string, the textual contents also match the log pattern “adding exclude file”. However, these texts are not log messages but build errors.

6.2 Data Analysis

Table 4 shows the number of different types of bug reports for each project. Overall, among 81,245 bug reports, 4,939 (6 %) bug reports contain log messages. The percentage of bug reports with log messages varies among projects. For example, 16 % of the bug reports in HBase contain log messages but only 1 % of the bug reports in Tomcat contain log messages. None of the bug reports from Pivot and Rat contain log messages.

Figure 9 plots the distribution of BRT for BWLs and BNLs. Each plot is a beanplot (Kampstra 2008), which visually compares the distributions of BRT for bug reports with log messages (the left part of the plot) and the ones without (the right part of the plot). The vertical scale is shown in the natural logarithm of days. The 21 selected projects have very different distributions of BRT for BNLs and BWLs, except a few ones (e.g., Pig and Zookeeper). For example, BRT for BWLs has a much wider distribution than BNLs for EmpireDB. We did not show the plots for Pivot and Rat, as they do not have any bug reports containing log messages.

Table 5 shows the median BRT for both BNLs and BWLs in each project. For example, in ActiveMQ, the median of BRT for BNLs is 12 days and 57 days for BWLs. The median BRTs for BNLs and BWLs are split across the 21 projects: 8 projects have longer median BRTs for BNLs and 10 projects have shorter median BRTs for BNLs. The other two projects (Pivot and Rat) do not contain any BWLs, as none of their bug reports contain log messages. For server-side and SC-based projects, the median of BRT of BNLs is shorter than that of BWLs, whereas the median of BRT of BNLs is longer than that of BWLs for client projects. Our finding is different from that of the original study, which shows the BRT is shorter in BWLs for server-side projects.

To compare the BRT for BWLs and BNLs across all the projects, the original study calculated the average of the median BRT for all the projects. The result is shown in the brackets of the last row of Table 5. In our selected 21 projects, Ant and Fop have very long BRT in general (>1000 days). Taking the average for all the median BRTs from all the projects could result in a long BRTs overall (around 200 days). This number is not representative of all the projects, as most projects have a median BRT smaller than 30 days. Hence, we introduce a new metric in our study, which is the median of the BRT for all the projects. The results of this new metric are shown in the last row of Table 5. The overall median BRT for BNLs (14 days) is shorter than BWLs (17 days) across all the projects.

We performed the non-parametric Wilcoxon rank-sum test (WRS) to compare the BRT for BWLs and BNLs across all the projects. Table 5 shows our results. The two-sided WRS test shows that the BRT for BWLs is significantly different from BRT for BNLs ($p < 0.05$) in nearly half (10/21) of the studied projects. Among three categories, the BRT for BWLs is statistically significant in server-side and SC-based projects. When

Table 4 The number of BNLs and BWLs for each project

Category	Project	# of Bug reports	# of BNLs	# of BWLs
Server	Hadoop	20,608	19,152 (93 %)	1,456 (7 %)
	HBase	11,208	9,368 (84 %)	1,840 (16 %)
	Hive	7,365	6,995 (95 %)	370 (5 %)
	Openmeetings	1,084	1,080 (99 %)	4 (1 %)
	Tomcat	389	388 (99 %)	1 (1 %)
	Subtotal	40,654	36,983 (91 %)	3,671 (9 %)
Client	Ant	5,055	4,955 (98 %)	100 (2 %)
	Fop	2,083	2,068 (99 %)	15 (1 %)
	Jmeter	2,293	2,225 (97 %)	68 (3 %)
	Maven	4,354	4,299 (99 %)	55 (1 %)
	Rat	149	149 (100 %)	0 (0 %)
	Subtotal	13,934	13,696 (98 %)	238 (2 %)
SC	ActiveMQ	5,015	4,687 (93 %)	328 (7 %)
	Empire-db	205	204 (99 %)	1 (1 %)
	Karaf	3,089	3,049 (99 %)	40 (1 %)
	Log4j	749	704 (94 %)	45 (6 %)
	Lucene	5,254	5,241 (99 %)	13 (1 %)
	Mahout	1,633	1,603 (98 %)	30 (2 %)
	Mina	907	901 (99 %)	6 (1 %)
	Pig	3,560	3,188 (90 %)	372 (10 %)
	Pivot	771	771 (100 %)	0 (0 %)
	Struts	4,052	4,007 (99 %)	45 (1 %)
	Zookeeper	1,422	1,272 (89 %)	150 (11 %)
	Subtotal	26,657	25,627 (96 %)	1,030 (4 %)
	Total	81,245	76,306 (94 %)	4,939 (6 %)

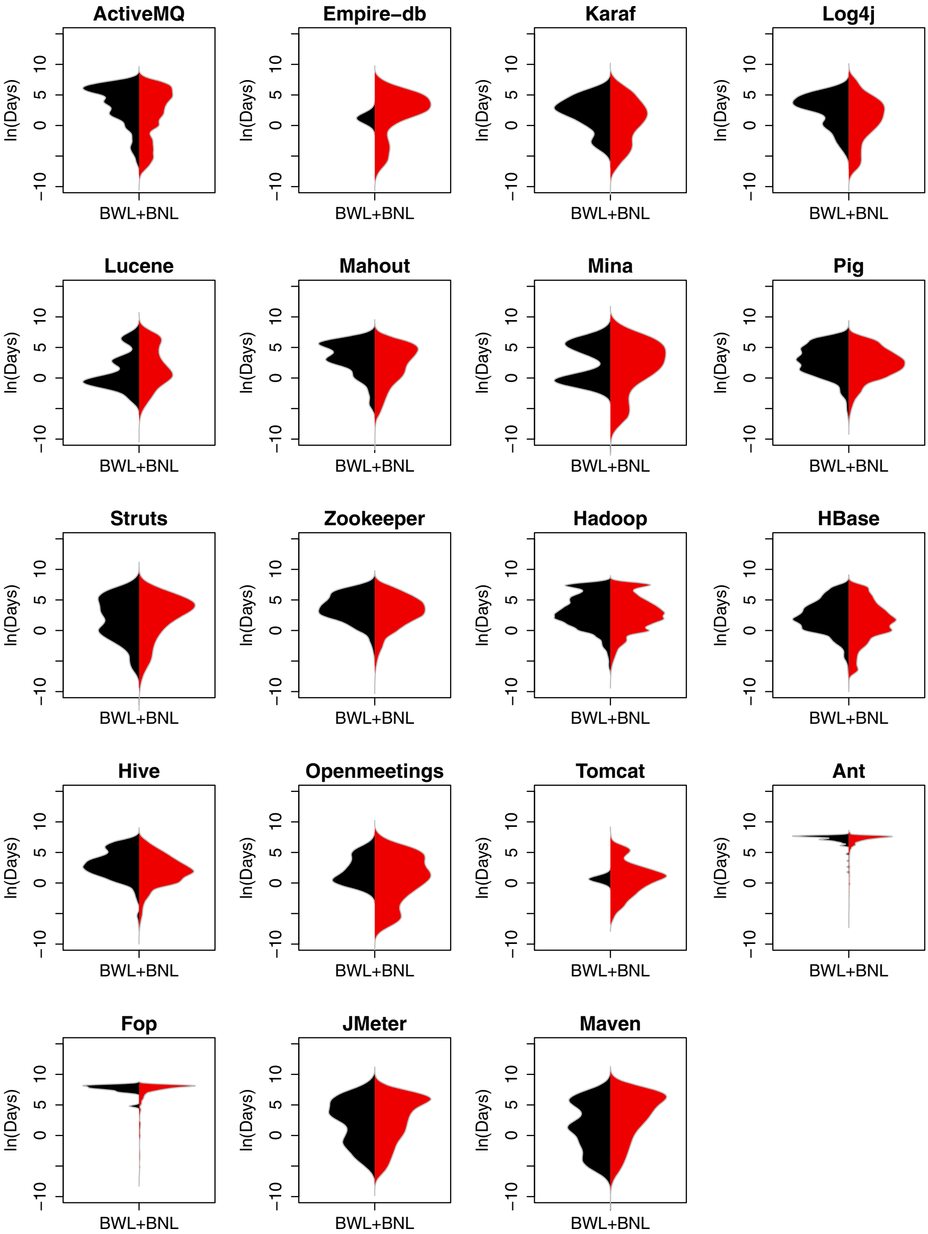


Fig. 9 Comparing the bug resolution time between BWLs and BNLs for each project

we aggregate the data across 21 projects, the BRT between BNLs and BWLs is also different.

To assess the magnitude of the differences between the BRT for BNLs and BWLs, we have also calculated the effect sizes using Cliff's Delta (only for the projects of which the BRT for BWLs and BNLs are significantly different according to WRS result) in Table 5.

The strength of the effects and the corresponding range of *Cliff's Delta (d)* values (Romano et al. 2006) are defined as follows:

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

Our results show that the effect sizes for majority of the projects are small or negligible. Across the three categories and overall, the effect sizes of BRT between BNLs and BWLs are also small and negligible.

Table 5 Comparing the bug resolution time of BWLs and BNLs

Category	Project	BNLs	BWLs	p-values for WRS	Cliff's Delta (d)
Server	Hadoop	16	13	<**0.001**	0.07 (negligible)
	HBase	5	4	<**0.001**	0.12 (negligible)
	Hive	7	7	<**0.001**	0.25 (small)
	Openmeetings	3	8	0.51	0.19 (small)
	Tomcat	3	2	0.86	−0.11 (negligible)
	Subtotal	10	14	<**0.001**	0.08 (negligible)
Client	Ant	1,478	1,665	<**0.05**	0.16 (small)
	Fop	2,313	2,510	0.35	0.13 (negligible)
	Jmeter	24	19	0.50	−0.05 (negligible)
	Maven	46	4	<**0.05**	−0.25 (small)
	Rat	8	N/A	N/A	N/A
	Subtotal	548	499	0.50	−0.03 (negligible)
SC	ActiveMQ	12	57	<**0.001**	0.23 (small)
	Empire-db	13	3	0.50	**−0.39 (medium)**
	Karaf	3	12	<**0.05**	0.22 (small)
	Log4j	4	23	<**0.05**	0.26 (small)
	Lucene	5	1	0.29	−0.16 (small)
	Mahout	15	31	0.05	0.20 (small)
	Mina	12	34	0.84	0.05 (negligible)
	Pig	11	20	<**0.001**	0.13 (negligible)
	Pivot	5	N/A	N/A	N/A
	Struts	20	13	0.6	−0.04 (negligible)
	Zookeeper	24	40	<**0.05**	0.14 (negligible)
	Subtotal	9	28	<**0.001**	0.20 (small)
Overall		14(192)	17(236)	<**0.001**	0.04 (negligible)

The p-values for WRS are bolded if they are smaller than 0.05. The values for the effect sizes are bolded if they are medium or large

6.3 Summary

NF2: Different from the original study, the median BRT for BWLs is longer than the median BRT for BNLs in server-side projects and SC-based projects. The BRT for BNLs is statistically different from the BRT for the BWLs in nearly half of the studied projects (10/21). However, the effect sizes for BRT between the BNLs and BWLs are small.
Implications: As shown in the previous studies (Bettenburg et al. 2008; Zimmermann et al. 2010), multiple factors (e.g., test cases and stack traces) are considered useful for developers to replicate issues reported in the bug reports. However, the factor of software logging was not studied in those works. Further research is required to re-visit these studies to examine the impact of various factors on bug resolution time.

7 (RQ3:) How Often is the Logging Code Changed?

In this section, we quantitatively analyze the evolution of the logging code. We measure the churn rate for both the logging code and the entire source code. We compare the number of revisions with and without log changes. We also categorize and measure the evolution of the logging code (e.g., the amount of insertion and deletion of the logging code).

7.1 Data Extraction

The data extraction step for this RQ consists of four parts: (1) calculating the average churn rate of source code, (2) calculating the average churn rate of the logging code, (3) categorizing code revisions with or without log changes, and (4) categorizing the types of log changes.

7.1.1 Part 1: Calculating the Average Churn Rate of Source Code

The SLOC for each revision can be estimated by measuring the SLOC for the initial version and keeping track of the total number of lines of source code that are added and removed for each revision. For example, the SLOC for the initial version is 2,000. In version 2, two files are changed: file A (3 lines added and 2 lines removed) and file B (10 lines added and 1 lines removed). Hence, the SLOC for version 2 would be $2000 + 3 - 2 + 10 - 1 = 2010$. The churn rate for version 2 is $\frac{3+2+10+1}{2010} = 0.008$. The average churn rate of the source code is calculated by taking the churn rate for all the revisions. The resulting average churn rate of source code for each project is shown in Table 6.

7.1.2 Part 2: Calculating the Average Churn Rate of the Logging Code

The average churn rate of the logging code is calculated in a similar manner as the average churn rate of source code. First, the initial set of logging code is obtained by writing a parser to recognize all the logging code with JDT. Then, the LLOC is calculated by keeping track of lines of logging code added and removed for each revision (please refer to Section 4.2.4 for details). Afterwards, the churn rate of the logging code for each revision is calculated. Finally, the average churn rate of the logging code is obtained by taking the average of the churn rates for all the revisions. The resulting average churn rate of logging code for each project is shown in Table 6.

7.1.3 Part 3: Categorizing Code Revisions with or Without Log Changes

We have already obtained a historical dataset that contains the revision history for all the source code (Section 4.2.3) and another historical dataset that contains all the revision history just for the logging code (Section 4.2.4). We write a script to count the total number of revisions in the above two datasets. Then we calculate the percentage of code revisions that contain changes in the logging code.

7.1.4 Part 4: Categorizing the Types of Log Changes

In this step, we write another script that parses the revision history for the logging code and counts the total number of code changes that have log insertions, deletions, updates and moves. The results are shown in Table 7.

Table 6 Average churn rate of source code vs. average churn rate of logging code for each project

Category	Project	Logging code (%)	Entire source code (%)
Server	Hadoop	8.7	2.4
	HBase	3.2	2.4
	Hive	3.9	2.1
	Openmeetings	3.7	3.0
	Tomcat	2.6	1.7
	Subtotal	4.4	2.3
Client	Ant	5.1	2.4
	Fop	5.5	3.4
	Jmeter	2.6	2.0
	Maven	7.0	4.0
	Rat	7.4	4.1
	Subtotal	5.5	3.2
SC	ActiveMQ	5.4	3.1
	Empire-db	5.0	2.4
	Karaf	11.7	4.7
	Log4j	6.1	2.8
	Lucene	3.4	2.0
	Mahout	10.8	4.0
	Mina	7.0	3.2
	Pig	4.3	2.3
	Pivot	7.0	2.0
	Struts	4.3	2.8
	Zookeeper	5.2	3.4
	Subtotal	6.4	3.0
	Total	5.7	2.9

7.2 Data Analysis

Code Churn Table 6 shows the code churn rate for the logging code and the entire code for all the projects. For server-side projects, the churn rate of the logging code is 1.9 times higher than that of entire code. This result is similar to the original result. The churn rate of logging code in client-side projects and SC based projects is also higher than that of the entire code. The highest churn rate of the logging code is from Karaf (11.7 %) and the lowest from Tomcat and JMeter (2.6 %). Across all the studied projects, the logging code churn rate is higher than the source code churn rate. Similar to the original study, the average churn rate of the logging code for all the projects is 2.3 times higher than the churn rate of source code.

Table 7 Committed revisions with or without logging code

Category	Project	Revisions with changes to logging code	Total revisions	Percentage (%)
Server	Hadoop	8,969	25,944	34.5
	Hbase	4,393	12,245	35.8
	Hive	1,053	4,047	26.0
	Openmeetings	861	2,169	39.6
	Tomcat	4,225	26,921	15.6
	Subtotal	19,501	71,326	**27.3**
Client	Ant	1,771	11,331	15.6
	Fop	1,298	6,941	18.7
	Jmeter	300	2,022	14.8
	Maven	5,736	29,362	19.5
	Rat	24	825	2.9
	Subtotal	9,129	50,481	**18.1**
SC	ActiveMQ	2,115	9,677	21.9
	Empire-db	123	515	23.9
	Karaf	802	2,730	29.3
	Log4j	1,919	6,073	31.5
	Lucene	2,946	28,842	10.2
	Mahout	573	2,249	25.4
	Mina	486	3,251	14.9
	Pig	470	2,080	22.5
	Pivot	280	3,604	7.76
	Struts	712	5,816	12.2
	Zookeeper	499	1,109	44.9
	Subtotal	10,925	65,946	**16.6**
	Total	39,555	187,753	**21.1**

Code Commits with Log Changes Table 7 tabulates the number of revisions that contain changes to the logging code, the total number of revisions, and the percentage of revisions containing log changes for each project and each category. The percentage of code revisions containing log changes varies among different projects and categories. Compared to the original study, the server-side projects in our study have a slightly higher percentage of revisions with changes to logging code (27.3 % vs. 18.1 %). This percentage for client-side (18.1 %) and SC-based (16.6 %) projects is similar to the original study. Overall, 21.1 % of revisions contain changes to the logging code.

Types of Log Changes There are four types of changes on the logging code: log insertion, log deletion, log update and log move. Log deletion, log update and log move are collectively called log modification. Table 8 shows the percentage of each change operation among all the projects and all categories. In general, log insertion and log update are the most frequent log change operations across all the projects (32 % for both operations), followed by log deletion (26 %) and log move (10 %). Our results are different from the

Table 8 Breakdown of different changes to the logging code

Category	Project	Log insertion	Log deletion	Log update	Log move
Server	Hadoop	16,338 (32 %)	13,983 (28 %)	15,324 (30 %)	5,205 (10 %)
	HBase	7,527 (32 %)	6,042 (26 %)	7,681 (33 %)	2,113 (9 %)
	Hive	2,314 (39 %)	1,844 (31 %)	1,331 (21 %)	515 (9 %)
	Openmeetings	1,545 (32 %)	1,854 (38 %)	1,027 (22 %)	429(8 %)
	Tomcat	5,508 (36 %)	4,120 (27 %)	4,215 (28 %)	1,409 (9 %)
	Subtotal	33,232 (33 %)	27,843 (27 %)	29,578 (30 %)	9,671 (10 %)
Client	Ant	2,331 (28 %)	2,158 (26 %)	3,217 (39 %)	588 (7 %)
	Fop	1,707 (29 %)	1,859 (32 %)	1,776 (31 %)	484 (8 %)
	Jmeter	202 (34 %)	115 (19 %)	207 (35 %)	74 (12 %)
	Rat	14 (30 %)	7 (15 %)	21 (45 %)	5 (10 %)
	Maven	6,689 (33 %)	5,810 (29 %)	5,583 (27 %)	2,265 (11 %)
	Subtotal	10,943 (31 %)	9,949 (28 %)	10,804 (31 %)	3,416 (10 %)
SC	ActiveMQ	2,295 (32 %)	1,314 (19 %)	2,978 (42 %)	489 (7 %)
	Empire-db	181 (35 %)	129 (25 %)	161 (31 %)	53 (9 %)
	Karaf	998 (26 %)	817 (21 %)	1,542 (40 %)	521 (13 %)
	Log4j	2,740 (27 %)	2,101 (20 %)	4,698 (46 %)	722 (7 %)
	Lucene	6,119 (36 %)	4,175 (25 %)	4,737 (28 %)	1,801 (11 %)
	Mahout	698 (18 %)	754 (19 %)	2,122 (55 %)	306 (8 %)
	Mina	608 (29 %)	518 (25 %)	759 (36 %)	220 (10 %)
	Pig	394 (32 %)	392 (32 %)	315 (26 %)	127 (10 %)
	Pivot	239 (41 %)	215 (37 %)	116 (20 %)	16 (2 %)
	Struts	718 (27 %)	718 (27 %)	879 (33 %)	345 (13 %)
	Zookeeper	778 (35 %)	575 (26 %)	626 (28 %)	239 (11 %)
	Subtotal	15,768 (31 %)	11,708 (23 %)	18,933 (37 %)	4,839 (9 %)
	Total	59,943 (32 %)	49,500 (26 %)	59,315 (32 %)	17,926 (10 %)

original study, in which there are very few (2 %) log deletions and moves. We manually analyzed a few commits which contain log deletion and move. We found that they are mainly due to code refactorings and to changes in testing code.

7.3 Summary

F3 and F4: Similar to the original study, the logging code churn rate is two times higher than that of the entire code base and around 20 % of the code commits contain log changes.
Implications: Similar to C/C++ projects in the original study, the logging code in Java projects in our study is also actively maintained. The evolution and maintenance of the logging code is a crucial activity in the evolution of software projects. There are many log analysis applications developed to monitor and debug the health of server-based projects (Oliner et al. 2012). The frequency of changes in the logging code bring great challenges in maintaining these log analysis applications. Additional tools and research are required to manage the co-evolution of logging code and log monitoring applications.

NF6: There are much more log deletions and moves (36 % vs. 2 %) across all three categories in our study.
Implications: Deleting and moving logging code may hinder the understanding of runtime behavior of these projects. New research is required to assess the risk of deleting and moving logging code for Java-based systems.

8 (RQ4:) What are the Characteristics of Consistent Updates to the Log Printing Code?

Both our results and the original study show that changes (churn) to the logging code are more frequent than changes to the source code. Among all the changes to the logging code, log update is one of the most frequent operations. As log messages are generated by the log-printing code at runtime, it is important to study the developers' behavior on updates to the log printing code. The updates to the log printing code can be further classified into consistent updates and after-thought updates. An update to the log printing code is a consistent update, if this piece of log printing code is changed along with other non-log related source code. Otherwise, the log update operation is an after-thought update. In this RQ, we study the characteristics of the consistently updated log printing code. In the next section, we will study the after-thought updates.

8.1 Data Extraction

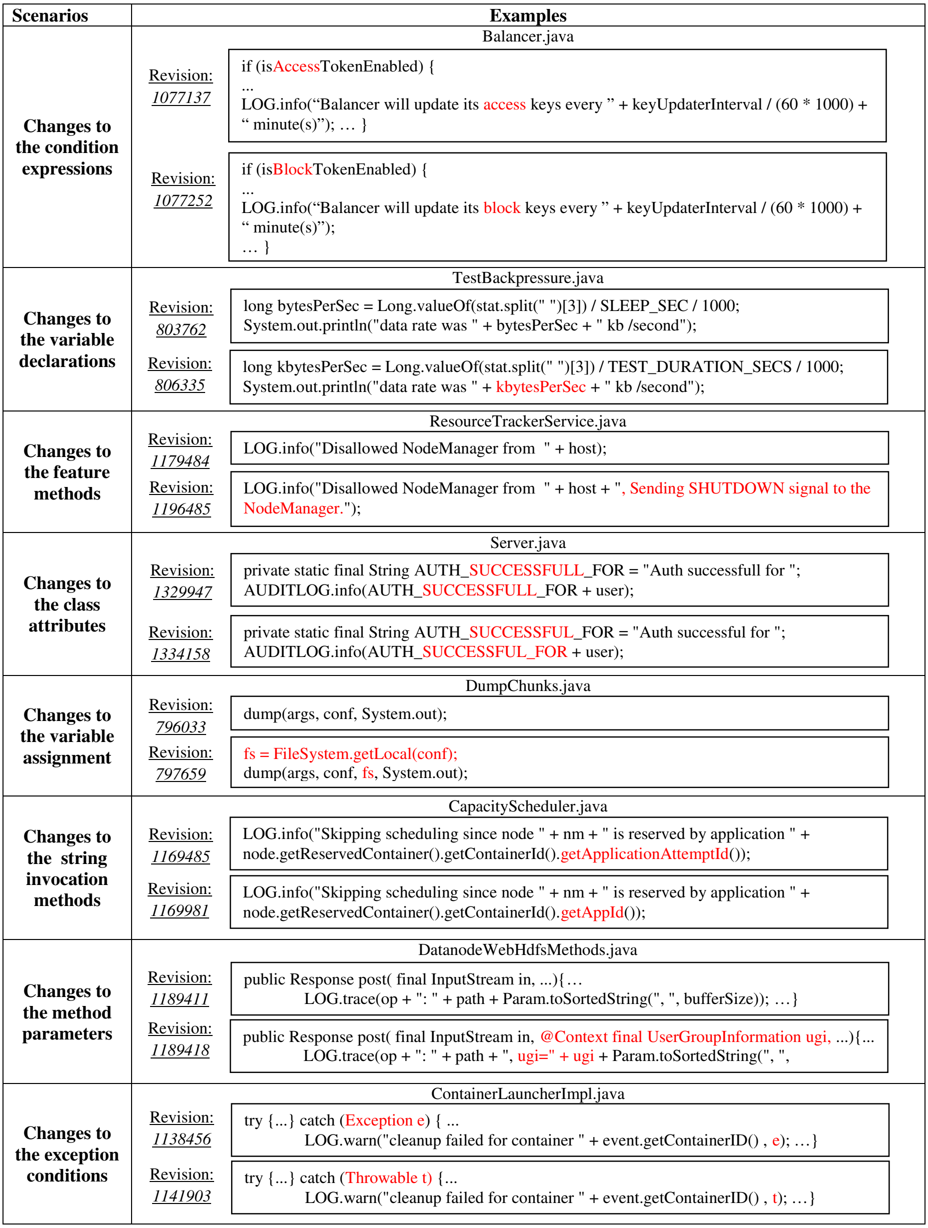
The original study classified consistent updates to the log printing code into three scenarios: *log update along with changes to condition expressions*, *log update along with variable re-declaration*, and *log update along with method renaming*. Based on manual investigation on some code revisions, we have identified a few additional scenarios (e.g., *log update following changes to the method parameters*). This manual investigation was repeated by both authors in this paper, until no new scenarios of consistent updates were found. As a result, we have identified eight in our study. We wrote a Java program that automatically parses each code revision using JDT and categorized the log printing code according to one of the aforementioned eight scenarios.

Below, we explain these eight scenarios of consistent update using real-world examples. For the sake of brevity, we do not include "log update along with" at the beginning of each scenario. The scenario is indicated as "*(new)*" if it is a new scenario identified in our study.

1. *Changes to the condition expressions (CON)* In this scenario, the log printing code is updated along with the conditional expression in a control statement (e.g., if/else/for/while/switch). The second row in Fig. 10 shows an example: the *if* expression is updated from "*isAccessTokenEnabled*" to "*isBlockTokenEnabled*", while the static text of the log printing code is updated from "*Balancer will update its access keys every*" to "*Balancer will update its block keys every*".
2. *Changes to the variable declarations (VD)* is a modified scenario of variable redeclaration in the original study. In Java projects, the variables can be declared or re-declared in each class, method or any code block. For example, the third row of Fig. 10 show that the variable "*bytesPerSec*" is changed to "*kbytesPerSec*". The static text of the log message is updated accordingly.
3. *Changes to the feature methods (FM)* is an expanded scenario of method renaming in the original study. We expand this scenario to include not only method renaming, but also all the methods updated in the same revision. In the example, the static text is added "Sending SHUTDOWN signal to the NodeManager.", and the method "shutdown" is changed in the same revision according to our historical data.
4. *Changes to the class attributes (CA)(new)* In Java classes, the instance variables for each class are called "class attributes". If the value or the name of the class attribute gets updated along with the log printing code, it falls into this scenario. In the example shown in the fourth row of Fig. 10: both the log printing code and the class attributes are changed from "AUTH_SUCCESSFULL_FOR" to "AUTH_SUCCESSFUL_FOR".
5. *Changes to the variable assignments (VA)(new)* In this scenario, the value of a local variable in a method has been changed along with the log printing code. For the example shown in the sixth row of Fig. 10: variable "*fs*" is assigned to a new value in the new revision, while the log printing code adds "*fs*" to its list of output variables.
6. *Changes to the string invocation methods (MI) (new)* In this scenario, the changes are in the string invocations of the logging code. For the example shown in the seventh row of Fig. 10: a method name is updated from "getApplicationAttemptId" to "getAppId", and the change is also made in the log printing code.
7. *Changes to the method parameters (MP)(new)* In this scenario, the changes are in the names of the method parameters. For the example shown in the eighth row of Fig. 10: there is an added variable "*ugi*" in the list of parameters for the "*post*" method. The log printing code also adds "*ugi*" to its list of output variables.
8. *Changes to the exception conditions (EX)(new)* In this scenario, the changes reside in a catch block and record the exception messages. For the example shown in the ninth row of Fig. 10: the variable in the log printing code is also updated due to changes in the catch block from "exception" to "throwable".

8.2 Data Analysis

Table 9 shows the breakdown of different scenarios for consistent updates and the total number of the remaining updates, i.e., after-thought updates, for each project. To conserve space, we use the short names introduced above for each scenario. Within consistent updates, the frequency of each scenario is also shown. Around 50 % of all the updates to the log printing



Scenarios	Examples	
	Balancer.java	

Changes to the condition expressions	Revision: *1077137*	if (isAccessTokenEnabled) {
		...
		LOG.info(“Balancer will update its access keys every ” + keyUpdaterInterval / (60 * 1000) +
		“ minute(s)”); … }

	Revision: *1077252*	if (isBlockTokenEnabled) {
		...
		LOG.info(“Balancer will update its block keys every ” + keyUpdaterInterval / (60 * 1000) +
		“ minute(s)”);
		… }

	TestBackpressure.java	

Changes to the variable declarations	Revision: *803762*	long bytesPerSec = Long.valueOf(stat.split(" ")[3]) / SLEEP_SEC / 1000;
		System.out.println("data rate was " + bytesPerSec + " kb /second");

	Revision: *806335*	long kbytesPerSec = Long.valueOf(stat.split(" ")[3]) / TEST_DURATION_SECS / 1000;
		System.out.println("data rate was " + kbytesPerSec + " kb /second");

	ResourceTrackerService.java	
Changes to the feature methods	Revision: *1179484*	LOG.info("Disallowed NodeManager from " + host);
	Revision: *1196485*	LOG.info("Disallowed NodeManager from " + host + ", Sending SHUTDOWN signal to the NodeManager.");
	Server.java	

Changes to the class attributes	Revision: *1329947*	private static final String AUTH_SUCCESSFULL_FOR = "Auth successfull for ";
		AUDITLOG.info(AUTH_SUCCESSFULL_FOR + user);

	Revision: *1334158*	private static final String AUTH_SUCCESSFUL_FOR = "Auth successful for ";
		AUDITLOG.info(AUTH_SUCCESSFUL_FOR + user);

	DumpChunks.java	
Changes to the variable assignment	Revision: *796033*	dump(args, conf, System.out);

	Revision: *797659*	fs = FileSystem.getLocal(conf);
		dump(args, conf, fs, System.out);

	CapacityScheduler.java	
Changes to the string invocation methods	Revision: *1169485*	LOG.info("Skipping scheduling since node " + nm + " is reserved by application " + node.getReservedContainer().getContainerId().getApplicationAttemptId());
	Revision: *1169981*	LOG.info("Skipping scheduling since node " + nm + " is reserved by application " + node.getReservedContainer().getContainerId().getAppId());
	DatanodeWebHdfsMethods.java	
Changes to the method parameters	Revision: *1189411*	public Response post(final InputStream in, ...){… LOG.trace(op + ": " + path + Param.toSortedString(", ", bufferSize)); …}
	Revision: *1189418*	public Response post(final InputStream in, @Context final UserGroupInformation ugi, ...){... LOG.trace(op + ": " + path + ", ugi=" + ugi + Param.toSortedString(", ",
	ContainerLauncherImpl.java	
Changes to the exception conditions	Revision: *1138456*	try {...} catch (Exception e) { ... LOG.warn("cleanup failed for container " + event.getContainerID() , e); …}
	Revision: *1141903*	try {...} catch (Throwable t) {... LOG.warn("cleanup failed for container " + event.getContainerID() , t); …}

Fig. 10 Examples of the eight scenarios of consistent updates to the log printing code

code for server-side projects are consistent updates. This percentage of consistent updates for server-side projects is much lower in our study compared to the original study. This number is even smaller for client-side (37.8 %) and SC-based (28.5 %) projects. Out of all the updates to the log printing code, 41 % of the updates on the log printing code are consistent updates.

Table 9 Detailed classifications of log printing code updates for each scenario

Category	Project	CON (%)	VD (%)	FM (%)	CA (%)	VA (%)	MI (%)	MP (%)	EX (%)	After-thought (%)
Server	Hadoop	13.1	12.6	3.9	2.8	2.5	8.6	6.3	0.4	49.7
	HBase	10.2	13.3	4.0	4.4	1.9	11.4	4.8	0.2	49.7
	Hive	9.8	8.1	3.8	16.3	1.9	5.5	2.7	0.4	51.5
	Openmeetings	7.9	5.6	18.3	0.1	2.7	3.2	13.9	0.1	48.2
	Tomcat	21.7	7.4	5.4	4.2	1.9	4.0	5.3	1.0	49.1
	Subtotal	13.0	11.6	4.8	3.9	2.3	8.3	6.0	0.4	49.7
Client	Ant	12.9	4.9	34.1	8.2	3.6	5.5	4.1	0.0	26.6
	Fop	19.8	6.6	2.0	2.0	1.5	4.3	5.2	0.1	58.6
	JMeter	13.8	7.7	0.5	11.7	3.1	1.5	4.6	0.0	57.1
	Maven	14.3	5.8	1.6	0.4	1.6	2.8	3.7	0.1	69.6
	Rat	11.1	22.2	0.0	0.0	0.0	0.0	0.0	0.0	66.7
	Subtotal	15.5	6.1	4.0	1.9	1.8	3.3	4.1	0.2	63.2
SC	ActiveMQ	14.4	4.3	1.1	2.0	0.7	1.9	0.8	0.0	74.6
	Empire-db	8.0	7.3	0.0	0.0	0.7	2.7	3.3	0.0	78.0
	Karaf	8.4	6.1	1.3	2.0	0.2	1.2	1.7	0.0	79.0
	Log4j	4.9	3.2	3.6	1.9	0.9	2.7	5.1	0.2	77.6
	Lucene	7.8	9.4	6.3	2.5	2.1	5.5	4.4	1.5	60.4
	Mahout	8.1	1.6	0.5	0.0	0.2	1.7	4.4	0.1	83.4
	Mina	26.1	6.1	0.7	0.3	1.3	2.5	0.7	0.2	62.3
	Pig	15.4	11.1	4.7	1.7	0.0	0.4	7.3	0.0	59.4
	Pivot	4.8	0.0	3.2	0.0	3.2	9.5	4.8	0.0	74.6
	Struts	33.0	3.9	4.5	0.3	0.3	2.2	2.5	0.5	52.7
	Zookeeper	18.7	6.8	1.2	4.4	0.5	6.8	4.9	1.0	55.8
	Subtotal	11.9	5.2	2.6	1.6	0.9	2.8	3.1	0.4	71.5
	Total	13.0	8.7	3.9	2.8	1.7	5.7	4.8	0.3	59.0

When we examine the different scenarios of the consistent updates, changes to the condition expressions are the most frequent scenarios across all three categories. This finding is similar to the original study. However, the portion of this scenario is much lower in our study (13 % vs. 57 %).

Compared to the original study, the amount of after-thought updates is much higher in our study (59 % vs. 33 %). Through manual sampling of a few after-thought updates, we find that many after-thoughts are related to the changes in logging style. For example, the Karaf project contains a very high portion (79 %) of after-thought updates. The static texts are updated in many updates to the log printing code for logging style changes. For example, the log printing code "LOGGER.warn("Could not resolve targets.");" from revision *1171011* of ObrBundleEventHandler.java, is changed to "LOGGER.warn("**CELLAR OBR**: could not resolve targets");" in the next revision. In this same revision, "CELLAR OBR" is added as a prefix in four other updates to the log printing code. These changes are made to reflect the addition of the "CELLAR ORB" component.

We further group the data from each project into their corresponding categories. For server-side projects, the frequency of consistent updates is higher than for the other two categories. This result suggests that developers of server-side projects tend to maintain log printing code more carefully, as log messages play an important role in monitoring and debugging server-side systems. For SC-based projects, the frequency of after-thought updates is the highest (71 %).

We will further investigate the characteristics of after-thought updates in the next section.

8.3 Summary

NF5: We have identified more scenarios of consistent updates (8 vs. 3 scenarios) in our study compared to the original study. However, the percentage of consistent updates of the log printing code is much smaller (50 % vs. 67 %). The percentage of consistent updates is even smaller in client-side (38 %) and SC-based (29 %) projects. Similar to the original study, CON is the most frequent consistent update scenario across all three categories of projects.

Implications: As there are more programming constructs (e.g., exception and class attributes) in Java, there are more scenarios related to consistent updates in our study. More consistent update scenarios bring additional challenges for Java developers to maintain the logging code. This highlights the need for additional research and tools for recommending changes in the logging code during each code commit.

9 (RQ5:) What are the Characteristics of After-Thought Updates on Log Printing Code?

Any log printing code updates that do not belong to consistent updates are after-thought updates. For after-thought updates, there are four scenarios depending on the updated components in the log printing code: *verbosity level updates*, *static text updates*, *dynamic content updates* and *logging method invocation updates*. In this section, we first conduct a high level quantitative study on the scenarios of after-thought updates. Then we perform an in-depth study on the context and rationale for each scenario.

9.1 High Level Data Analysis

We write a small program that automatically compares the differences between two adjacent revisions of the log printing code. For each snippet of the after-thought updates, this program outputs whether there are verbosity level updates, static texts updates, dynamic content updates or logging method invocation updates. Within the dynamic contents updates, we further separate them into whether the differences are changes in variables or changes in string invocation methods.

Table 10 shows the frequency of each scenario of the after-thought updates. The total percentage from each scenarios may exceed 100 %, as a snippet of log printing code may be updated in multiple components (e.g., in both the logging method invocations and the static text). Similar to the original study, we find that the most frequent after-thought scenario for server-side projects is static text changes (53 % vs. 44 %). The dynamic content updates come next with 46 %. In addition, we also study the portion of updates to the invocation of the logging method (e.g., changing from "System.out.println" to "LOG.ERROR"). This is a new scenario introduced in our study. This scenario only accounts for 14.4 %, which is the lowest in all three categories.

Table 10 Scenarios of after-thought updates

Category	Project	Total	Verbosity level	Dynamic contents	Static texts	Logging method invocation
Server	Hadoop	4,821	1,076 (22.3 %)	2,259 (46.9 %)	2,587 (53.7 %)	705 (14.6 %)
	HBase	2,176	312 (14.3 %)	1,155 (53.1 %)	1,391 (63.9 %)	99 (4.5 %)
	Hive	436	178 (40.8 %)	147 (33.7 %)	186 (42.7 %)	42 (9.6 %)
	Openmeetings	423	160 (37.8 %)	125 (29.6 %)	179 (42.3 %)	99 (23.4 %)
	Tomcat	1,056	276 (26.1 %)	423 (40.1 %)	390 (36.9 %)	334 (31.6 %)
	Subtotal	8,912	2,002 (22.5 %)	4,109 (46.1 %)	4,733 (53.1 %)	1,279 (14.4 %)
Client	Ant	97	33 (34.0 %)	22 (22.7 %)	14 (14.4 %)	54 (55.7 %)
	Fop	725	148 (16.1 %)	138 (15.0 %)	179 (19.5 %)	452 (39.3 %)
	JMeter	112	26 (23.2 %)	36 (32.1 %)	58 (51.8 %)	10 (8.9 %)
	Maven	2,203	535 (24.3 %)	444 (20.2 %)	888 (40.3 %)	892 (40.5 %)
	Rat	6	2 (33.3 %)	0 (0.0 %)	2 (33.3 %)	2 (33.3 %)
	Subtotal	3,335	742 (22.2 %)	642 (19.3 %)	1,141 (34.2 %)	1,410 (42.3 %)
SC	ActiveMQ	2,053	423 (20.6 %)	408 (19.9 %)	437 (21.3 %)	1,433 (69.8 %)
	Empiredb	117	40 (34.2 %)	69 (59.0 %)	43 (36.8 %)	22 (18.8 %)
	Karaf	1,118	243 (21.7 %)	132 (11.8 %)	729 (65.2 %)	236 (21.1 %)
	Log4j	1,213	99 (8.2 %)	237 (19.5 %)	300 (24.7 %)	892 (73.5 %)
	Lucene	1,300	357 (27.5 %)	599 (46.1 %)	791 (60.8 %)	317 (24.4 %)
	Mahout	1,459	146 (10.0 %)	183 (12.5 %)	373 (25.6 %)	1,049 (71.9 %)
	Mina	380	77 (20.3 %)	89 (23.4 %)	107 (28.2 %)	196 (51.6 %)
	Pig	139	28 (20.1 %)	24 (17.3 %)	51 (36.7 %)	46 (33.1 %)
	Pivot	47	23 (48.9 %)	24 (51.1 %)	19 (40.4 %)	24 (51.1 %)
	Struts	337	39 (11.6 %)	91 (27.0 %)	141 (41.8 %)	166 (49.3 %)
	Zookeeper	230	70 (30.4 %)	106 (46.1 %)	146 (63.5 %)	10 (4.3 %)
	Subtotal	8,393	1,545 (18.4 %)	1,962 (23.4 %)	3,137 (37.4 %)	4,391 (52.3 %)
	Total	20,640	4,289 (20.8 %)	6,713 (32.5 %)	9,011 (43.7 %)	7,080 (34.3 %)

The results for client-side projects and SC-based projects have a similar trend. But they are quite different from server-side projects. Logging method invocation updates are the most frequent scenario (42 % and 52 %). We manually sampled a few such updates and checked their commit logs. They are all due to switching from ad-hoc logging to the use of general-purpose logging libraries. For example, there are 95 logging method invocation updates in revision *397249* from ActiveMQ. As indicated in the commit log, the purpose was to transform "a bunch of System.out.println() to log.info()". The static text updates are the second most frequent scenario (34 % and 37 %). Dynamic content updates come in third and the verbosity level updates are last.

9.2 Verbosity Level Updates

Similar to the original study, we separate the verbosity level updates into two types: (1) *error-level updates* refer to log updates in which the verbosity levels are updated to/from

Table 11 Scenarios related to verbosity-level updates

Category	Project	Total	Non-default	From/to default	Error
Server	Hadoop	1,076	147 (13.7 %)	717 (66.6 %)	212 (19.7 %)
	HBase	312	50 (16.0 %)	193 (61.9 %)	69 (22.1 %)
	Hive	178	9 (5.1 %)	134 (75.3 %)	35 (19.7 %)
	Openmeetings	160	54 (33.8 %)	12 (7.5 %)	94 (58.8 %)
	Tomcat	276	35 (12.7 %)	179 (64.9 %)	62 (22.5 %)
	Subtotal	2,002	295 (14.7 %)	1,235 (61.7 %)	472 (23.6 %)
Client	Ant	33	1 (3.0 %)	28 (84.8 %)	4 (12.1 %)
	Fop	148	38 (25.7 %)	78 (52.7 %)	32 (21.6 %)
	JMeter	26	2 (7.7 %)	8 (30.8 %)	16 (61.5 %)
	Maven	535	69 (12.9 %)	375 (70.1 %)	91 (17.0 %)
	Rat	0	0	0	0
	Subtotal	742	110 (14.8 %)	489 (65.9 %)	143 (19.3 %)
SC	ActiveMQ	423	67 (15.8 %)	312 (73.8 %)	44 (10.4 %)
	Empire-db	40	1 (2.5 %)	10 (25.0 %)	29 (72.5 %)
	Karaf	243	129 (53.1 %)	83 (34.2 %)	31 (12.8 %)
	Log4j	99	23 (23.2 %)	37 (37.4 %)	39 (39.4 %)
	Lucene	357	13 (3.6 %)	300 (84.0 %)	44 (12.3 %)
	Mahout	146	5 (3.4 %)	140 (95.9 %)	1 (0.7 %)
	Mina	77	3 (3.9 %)	65 (84.4 %)	9 (11.7 %)
	Pig	28	4 (14.3 %)	22 (78.6 %)	2 (7.1 %)
	Pivot	23	0 (0.0 %)	23 (100.0 %)	0 (0.0 %)
	Struts	39	10 (25.6 %)	16 (41.0 %)	13 (33.3 %)
	Zookeeper	70	9 (12.9 %)	29 (41.4 %)	32 (45.7 %)
	Subtotal	1,545	264 (17.1 %)	1,037 (67.1 %)	244 (15.8 %)
	Total	4,289	669 (15.6 %)	2,761 (64.4 %)	859 (20.0 %)

error levels (a.k.a., ERROR and FATAL); and (2) *non-error level updates*, refer to log updates in which the verbosity levels of neither the previous nor the current revision are error levels (e.g., DEBUG to INFO). In non-error level updates, for each project we first manually identify the default logging level, which is set in the configuration file of a project. Then we further break non-error level updates into two categories depending on whether they involve the default verbosity level or not.

The results are shown in Table 11. The majority (76 %) of the verbosity level updates for server-side projects are non-error event updates. Our finding is the opposite of the original study, which reported that only 28 % of verbosity level updates are non-error level updates.

In our results, all three categories have the similar trend. Verbosity level updates containing the default level is the most frequent one (around 65 %). In the original study, developers updating logging levels among non-default levels accounts for 57 % of the verbosity level changes. These changes are called as logging trade-offs, as the authors of the original study suspect the cause is no clear boundary among multiple verbose levels, taking use, benefit, and cost into consideration. In our study, this number drops to only 15 % in general and there are no much differences among the three categories. This finding probably implies that in the Java projects, the logging levels, which often come from common logging libraries like log4j, are better defined compared to the C/C++ projects.

9.2.1 Summary

NF7: Contrary to the original study, the majority (80 %) of the verbosity level modifications are between non-error levels.
NF8: Contrary to the original study, the majority (65 %) of the non-error verbosity level updates involve the default level.
Implications: Contrary to the original study, we find that verbosity levels of Java projects in the ASF are less frequently updated among non-default levels. Further qualitative studies (e.g., developer surveys) are required to understand the rationales behind such differences.

9.3 Dynamic Content Updates

Based on our definition, there are two kinds of dynamic contents in log printing code: variables (Var) and string invocation methods (SIM). Each change can be classified into three types: added, updated or deleted. The details of the variable updates and string invocation method updates are shown in Table 12.

In our study, the percentage of added dynamic contents, updated dynamic contents, and deleted dynamic contents are similar among all three categories. Nearly half (42 %) of the updates are added dynamic content updates, followed by deleted dynamic content updates (33 %) and updated dynamic content updates (23 %).

Similar to the original study, added variables are the most common changes in variable updates. Since we have introduced a new category (SIM), the added variable updates account for 30 % in server-side projects, which is much less than that in the original study (62 %). The percentage of added variable updates in client-side projects is 24 and 33 % in SC-based projects.

Among string invocation method updates, deleted SIM updates are the most common (20 %). The added and updated SIM update account for 14 and 10 % of all dynamic updates, respectively. For server-side and client-side projects, deleted SIM updates are the most common scenario. In SC based projects, the added SIM update is the most common scenario. In addition, among all three categories, the updated SIM update is the least common scenario.

9.3.1 Summary

NF9: Similar to the original study, adding variables into the log printing code is the most common after-thought change related to variables. Different from the original study, SIM is a new type of dynamic content update identified in our study. The majority of the changes to the SIMs (20 %) are deleted SIMs.
Implications: Among all the after-thought updates, there are much more dynamic content updates compared to the original study. This is due to the addition of SIMs for Java-based projects. Research on log enhancement should not only focus on suggesting which variables to log (e.g., Yuan et al. 2011; Zhu et al. 2015) but also on suggesting updates to the string invocation methods.

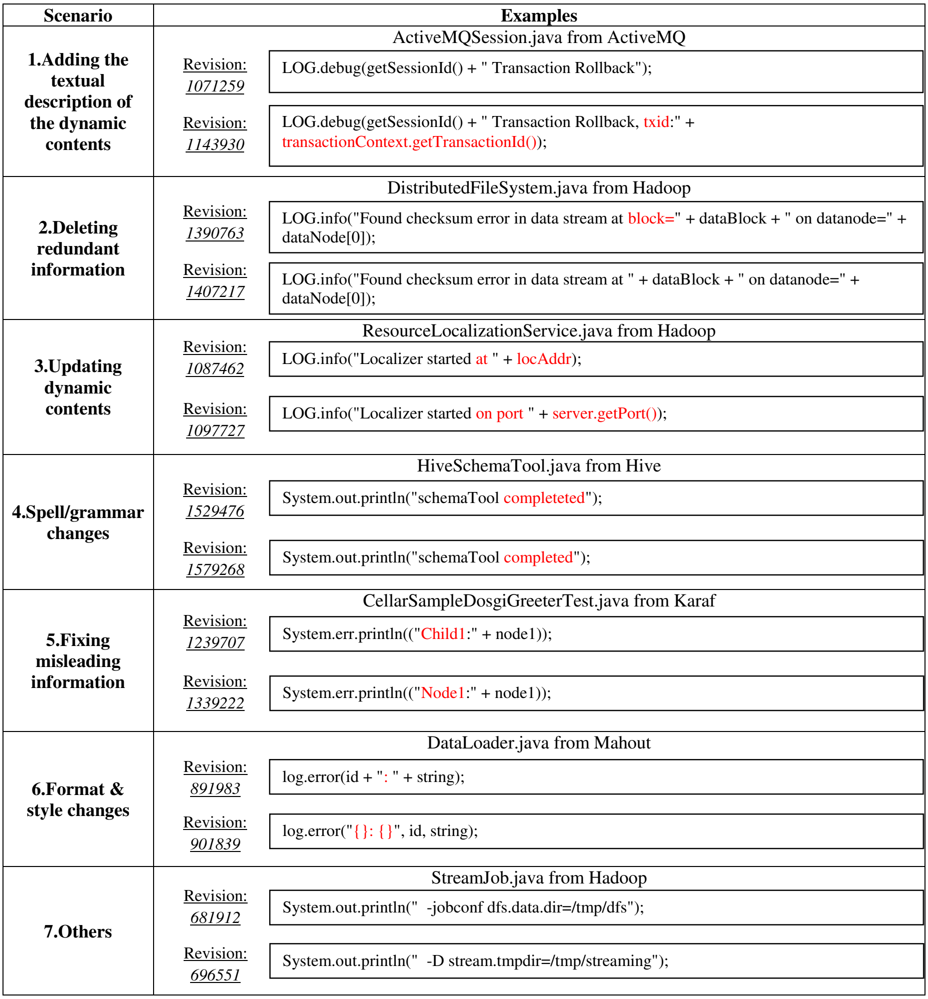
9.4 Static-Text Updates

44 % of the after-thought updates change the static text. Similar to the original study, we manually sample some static text changes to understand the their rationales.

In the original study, the authors manually sampled 200 static text changes. In this paper, we used the stratified sampling technique (Han 2005) to ensure representative samples are selected and studied from each project. Overall, a total of 372 static text modifications are selected from the 21 projects. This corresponds to a confidence level of 95 % with a confidence interval of ± 5 %. The portion of the sampled static text updates from each project is equal to the relative weight of the total number of static text updates for that project. For example, there are 437 static text updates of ActiveMQ out of a total of 9,011 updates from all the projects. Hence, 18 updates from ActiveMQ updates are picked. As a result, six scenarios are identified in our study. Below, we explain each of these scenarios using real world examples.

Table 12 Dynamic content updates

Category	Project	Added dynamic contents		Updated dynamic contents		Deleted dynamic contents	
		Var	SIM	Var	SIM	Var	SIM
Server	Hadoop	745 (33.0 %)	256 (11.3 %)	244 (10.8 %)	280 (12.4 %)	235 (10.4 %)	499 (22.1 %)
	HBase	269 (23.3 %)	178 (15.4 %)	148 (12.8 %)	145 (12.6 %)	149 (12.9 %)	266 (23.0 %)
	Hive	68 (46.3 %)	15 (10.2 %)	2 (1.4 %)	18 (12.2 %)	13 (8.8 %)	31 (21.1 %)
	Openmeetings	36 (28.8 %)	17 (13.6 %)	19 (15.2 %)	16 (12.8 %)	11 (8.8 %)	26 (20.8 %)
	Tomcat	126 (29.8 %)	65 (15.4 %)	43 (10.2 %)	45 (10.6 %)	48 (11.3 %)	96 (22.7 %)
	Subtotal	1,244 (30.3 %)	531 (12.9 %)	456 (11.1 %)	504 (12.3 %)	456 (11.1 %)	918 (22.3 %)
Client	Ant	2 (9.1 %)	2 (9.1 %)	4 (18.2 %)	2 (9.1 %)	4 (18.2 %)	8 (36.4 %)
	Fop	49 (35.5 %)	14 (10.1 %)	24 (17.4 %)	8 (5.8 %)	16 (11.6 %)	27 (19.6 %)
	JMeter	6 (10.0 %)	14 (23.3 %)	2 (3.3 %)	8 (13.3 %)	3 (5.0 %)	27 (45.0 %)
	Maven	97 (21.8 %)	82 (18.5 %)	28 (6.3 %)	76 (17.1 %)	56 (12.6 %)	105 (23.6 %)
	Rat	2 (100.0 %)	0 (0.0 %)	0 (0.0 %)	0 (0.0 %)	0 (0.0 %)	0 (0.0 %)
	Subtotal	156 (24.3 %)	118 (18.4 %)	58 (9.0 %)	91 (14.2 %)	79 (12.3 %)	140 (21.8 %)
SC	ActiveMQ	107 (26.2 %)	120 (29.4 %)	19 (4.7 %)	27 (6.6 %)	88 (21.6 %)	47 (11.5 %)
	Empiredb	31 (44.9 %)	5 (7.2 %)	1 (1.4 %)	1 (1.4 %)	2 (2.9 %)	29 (42.0 %)
	Karaf	70 (53.0 %)	24 (18.2 %)	7 (5.3 %)	5 (3.8 %)	9 (6.8 %)	17 (12.9 %)
	Log4j	80 (33.8 %)	24 (10.1 %)	41 (17.3 %)	11 (4.6 %)	28 (11.8 %)	53 (22.4 %)
	Lucene	276 (46.1 %)	89 (14.9 %)	50 (8.3 %)	28 (4.7 %)	77 (12.9 %)	79 (13.2 %)
	Mahout	25 (13.7 %)	3 (1.6 %)	74 (40.4 %)	12 (6.6 %)	49 (26.8 %)	20 (10.9 %)
	Mina	9 (10.1 %)	19 (21.3 %)	4 (4.5 %)	12 (13.5 %)	23 (25.8 %)	22 (24.7 %)
	Pig	6 (25.0 %)	4 (16.7 %)	8 (33.3 %)	1 (4.2 %)	0 (0.0 %)	5 (20.8 %)
	Pivot	4 (16.7 %)	5 (20.8 %)	8 (33.3 %)	0 (0.0 %)	5 (20.8 %)	2 (8.3 %)
	Struts	22 (24.2 %)	16 (17.6 %)	12 (13.2 %)	2 (2.2 %)	26 (28.6 %)	13 (14.3 %)
	Zookeeper	36 (34.0 %)	11 (10.4 %)	16 (15.1 %)	15 (14.2 %)	13 (12.3 %)	15 (14.2 %)
	Subtotal	666 (33.9 %)	320 (16.3 %)	240 (12.2 %)	114 (5.8 %)	320 (16.3 %)	302 (15.4 %)
	Total	2,066 (30.8 %)	969 (14.4 %)	754 (11.2 %)	709 (10.6 %)	855 (12.7 %)	1,360 (20.3 %)



Scenario	Examples	
1.Adding the textual description of the dynamic contents	ActiveMQSession.java from ActiveMQ	
	Revision: 1071259	LOG.debug(getSessionId() + " Transaction Rollback");
	Revision: 1143930	LOG.debug(getSessionId() + " Transaction Rollback, txid:" + transactionContext.getTransactionId());
2.Deleting redundant information	DistributedFileSystem.java from Hadoop	
	Revision: 1390763	LOG.info("Found checksum error in data stream at block=" + dataBlock + " on datanode=" + dataNode[0]);
	Revision: 1407217	LOG.info("Found checksum error in data stream at " + dataBlock + " on datanode=" + dataNode[0]);
3.Updating dynamic contents	ResourceLocalizationService.java from Hadoop	
	Revision: 1087462	LOG.info("Localizer started at " + locAddr);
	Revision: 1097727	LOG.info("Localizer started on port " + server.getPort());
4.Spell/grammar changes	HiveSchemaTool.java from Hive	
	Revision: 1529476	System.out.println("schemaTool completeted");
	Revision: 1579268	System.out.println("schemaTool completed");
5.Fixing misleading information	CellarSampleDosgiGreeterTest.java from Karaf	
	Revision: 1239707	System.err.println(("Child1:" + node1));
	Revision: 1339222	System.err.println(("Node1:" + node1));
6.Format & style changes	DataLoader.java from Mahout	
	Revision: 891983	log.error(id + ": " + string);
	Revision: 901839	log.error("{}: {}", id, string);
7.Others	StreamJob.java from Hadoop	
	Revision: 681912	System.out.println(" -jobconf dfs.data.dir=/tmp/dfs");
	Revision: 696551	System.out.println(" -D stream.tmpdir=/tmp/streaming");

Fig. 11 Examples of static text changes

1. *Adding textual descriptions of the dynamic contents:* When dynamic contents are added in the logging line, the static texts are also updated to include the textual description of the newly added dynamic contents. The first scenario in Fig. 11 shows an example: a string invocation method called "transactionContext.getTransactionId()" is added in the dynamic contents, since developers need to record more runtime information.
2. *Deleting redundant information* refers to the removal of static text due to redundant information. The second scenario in Fig. 11 shows an example: the text "block=" is deleted, since "at" and "block=" mean the same thing.
3. *Updating dynamic contents* refers to the changing of dynamic content like variables, string invocation methods, etc. The third scenario in Fig. 11 shows an example: the variable "locAddr" is replaced with string invocation method "server.getPort()" and the static text is updated to reflect this change.

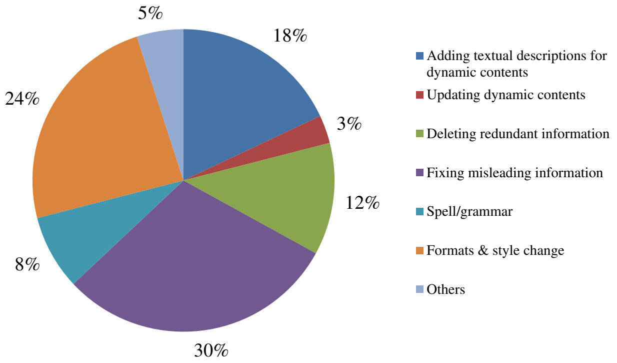


Fig. 12 Breakdown of different types of static content changes

4. *Fixing spelling/grammar issues* refers to the change in the static texts to fix the spelling or grammar mistakes. The fourth scenario in Fig. 11 shows an example: the word "completed" is misspelled and so it is corrected in the revision.
5. *Fixing misleading information* refers to the change in the static texts due to clarifications of this piece of log printing code. This scenario is a combination of the two scenarios (*clarification* and *fixing inconsistency*) proposed in the original study, as we feel both of them are related to fixing misleading information. The fifth scenario in Fig. 11 shows an example: the developer thinks that "Node" instead of "Child" better explains the meaning of the printed variable.
6. *Formatting & style changes* refer to changes to the static texts due to formatting changes (e.g., indentation). The sixth scenario in Fig. 11 shows an example: the code changes from string concatenation to the use of a format string output while the content stays the same.
7. *Others* Any other static text updates that do not belong to the above scenarios are labeled as others. One example shown in the last row Fig. 11 is for updating command line options.

Figure 12 shows the breakdown of different types of static text changes: the most frequent scenario is fixing misleading information (30 %), followed by formatting & style changes (24 %) and adding the textual description of the dynamic contents (18 %).

9.4.1 Summary

F10: Similar to the original study, fixing misleading changes account for nearly one third of the static text updates. There is also a significant portion of textual changes due to the formatting & style changes and adding the textual description of the dynamic contents.
Implications: The static contents of log printing code is actively maintained to properly enhance the execution contexts. Misleading or outdated static contents of log printing code confuse developers and cause bugs. Currently, developers tend to manually update these contents to ensure log messages properly reflect the execution contexts. Additional research is needed to leverage techniques from natural language processing and information retrieval to detect such inconsistencies automatically.

Table 13 Empirical studies on logs

Previous work	(Fu et al. 2014; Zhu et al. 2015)	(Yuan et al. 2012)	(Shang et al. 2015)
Main focus	Categorizing logging code snippets Predicting the location of logging	Characterizing logging practices Predicting inconsistent verbosity levels	Studying the relation between logging and post-release bugs Proposing code metrics related to logging
Projects	Industry and GitHub projects in C#	Open-source projects in C/C++	Open-source projects in Java
Studied log modifications	No	Yes	Yes

10 Related Work

In this section, we discuss two areas of related works on software logging: research done on the logging code and research done on log messages.

10.1 Logging Code

We define several criteria (Table 13) to summarize the differences among previous empirical studies on logs.

- **Main focus** presents the main objectives for each work;
- **Projects** show the programming languages of the subject projects in each work; and
- **Studied log modifications** indicates whether the work studied modifications on logs.

The work done by Yuan et al. (2012) is the first empirical study on characterizing the logging practices. The authors studied four different open-source applications written in C/C++. Fu et al. studied the location of software logging (Fu et al. 2014; Zhu et al. 2015) by systematically analyzing the source code of two large industrial systems from Microsoft and two open source projects from GitHub. All these projects are written in C#. Shang et al. (2015) found that log related metrics (e.g., log density) were strong predictors of post release defects. Ding et al. (2015) tried to estimate the performance overhead of logging.

Two works have proposed techniques to assist developers in adding additional logging code to better debug or monitor the runtime behavior of the systems. Yuan et al. (2011) use program analysis techniques to automatically instrument the application to diagnose failures. Zhu et al. (2015) use machine leaning techniques to derive common logging patterns from the existing code snippets and provide logging suggestions to developers in similar scenarios.

Most of the studies (Fu et al. 2014; Yuan et al. 2012, 2011; Zhu et al. 2015) are done in C/C++/C# projects except the work of Shang et al. (2015). Our paper is a replication study of Yuan et al. (2012). The goal of our study is to check whether their empirical findings can be generalizable to software projects written in Java.

10.2 Log Messages

Log messages are the messages generated by the log printing code at runtime. Log messages have been used and studied extensively to diagnose field failures (Oliner et al. 2012; Yuan et al. 2010), to understand the runtime behavior of a system (Beschastnikh et al. 2014; Beschastnikh et al. 2011), to detect abnormal runtime behavior for big data applications (Shang et al. 2013; Xu et al. 2009), to analyze the results of a load test (Jiang et al. 2008; 2009) and to customize and validate operational profiles (Hassan et al. 2008; Syer et al. 2014). Shang et al. (2014) performed an empirical study on the evolution of log messages and found that log messages change frequently over time. There are also many open source and commercial tools available for gathering and analyzing log messages (e.g., logstash - open source log management (2015), Nagios Log Server - Monitor and Manage Your Log Data (2015), and Splunk (2015)).

11 Threats to Validity

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

11.1 External Validity

11.1.1 Subject Systems

The goal of this paper is to validate whether the findings in the original study can be applicable to other projects or projects written in Java. In this study, we have studied 21 different Java-based projects, which are selected based on different perspectives (e.g. categories, sizes, development history and application domains). Based on our study, we have found that many of our results do not match with some of the findings in the original study, which was done on four C/C++ server-based projects. In addition, the logging practices in server-side projects are also quite different than those in client-side and SC-based projects. However, our results may not be generalizable to all the Java-based projects since we only studied projects from Apache Software Foundation. Additional empirical studies on the logging practices are needed for other Java-based projects (e.g., Eclipse and its ecosystem, Android related systems, etc.) or projects written in other programming languages (e.g., .NET or Python).

11.1.2 Sampling Bias

Some of the findings from the original study are based on random sampling. However, the sizes of the studied samples were not justified. In this paper, we have addressed this issue in several aspects.

- *Analyzing all instances in a dataset:* in the case of RQ2 (bug resolution time with and without log messages), we have studied all the bug reports instead of the selected samples.
- *Data-aware sampling:* Whenever we are doing random sampling, we have always ensured that the results fall under the confidence level of 95 % with a confidence interval of ± 5 %. For sampling across multiple projects (e.g., RQ5), we have used stratified sampling, so that a representative number of subjects is studied from each projects.

11.2 Internal Validity

In our study, we have found that bug reports containing log messages often take a shorter time to be resolved than bug reports without log messages for Java-based projects. Since there are many additional factors (e.g., the severity, the quality of bug descriptions and the types of bugs), which are not assessed in this study, we cannot extend the correlation between log messages and long bug resolution time to causation.

11.3 Construct Validity

In this study, we have used J-REX and CD to extract the code revision history. Both tools are very robust and have been used in quite a few other studies (e.g., Gall et al. 2009; Ghezzi and Gall 2013; Shang et al. 2014, 2015). For most of our developed programs (e.g., for bug categorization or for categorizing consistent updates of log printing code), we have performed thorough testing to ensure our results are correct.

12 Conclusion

Log messages have been used widely for developers, testers and system administers to understand, debug and monitor the behavior of systems at runtime. Yuan et al. reported a series findings regarding the logging practices based on their empirical study of four server-side C/C++ projects. In this paper, we have performed a large-scale replication study to check whether their findings can be applicable to 21 Java project in Apache Software Foundation. In addition to server-side projects, the other projects are client-side projects or support-component-based projects. Similar to the original study, we have found that logging is pervasive in most of the software projects and the logging code is actively maintained. Different from the original study, the median BRT of bug reports containing log messages is longer than bug reports without log messages. In addition, there are more scenarios of consistent updates to log printing code while the portion of after-thought updates is much bigger. Our study shows that certain aspects of the logging practices in Java-based systems are different from C/C++ based systems. Further research study is needed to study the rationales for these differences.

References

ASF Apache Software Foundation (2016) https://www.apache.org/. Accessed 8 April 2016

Basili VR, Shull F, Lanubile F (1999) Building knowledge through families of experiments. IEEE Trans Softw Eng 25(4):456–473

Beschastnikh I, Brun Y, Ernst MD, Krishnamurthy A (2014) Inferring models of concurrent systems from logs of their behavior with csight. In: Proceedings of the 36th International Conference on Software Engineering (ICSE)

Beschastnikh I, Brun Y, Schneider S, Sloan M, Ernst MD (2011) Leveraging existing instrumentation to automatically infer invariant-constrained models. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11

Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)

BlackBerry Enterprise Server Logs Submission (2015). https://www.blackberry.com/beslog/. Accessed 10 May 2015

Ding R, Zhou H, Lou JG, Zhang H, Lin Q, Fu Q, Zhang D, Xie T (2015) Log2: A cost-aware logging mechanism for performance diagnosis. In: USENIX Annual Technical Conference

Dumps of the ASF Subversion repository (2015) Dumps http://svn-dump.apache.org/. Accessed 10 May 2015

Estimating the reproducibility of psychological science (2015) Open Science Collaboration

Fluri B, Wursch M, Pinzger M, Gall H (2007) Change distilling:tree differencing for fine-grained source code change extraction. IEEE Trans Softw Eng 33(11):725–743

Fu Q, Zhu J, Hu W, Lou JG, Ding R, Lin Q, Zhang D, Xie T (2014) Where do developers log? An empirical study on logging practices in industry. In: Companion Proceedings of the 36th International Conference on Software Engineering

Gall HC, Fluri B, Pinzger M (2009) Change analysis with Evolizer and ChangeDistiller. IEEE Softw 26(1)

Gartner (2014) SIEM Magic Quadrant Leadership Report. http://www.gartner.com/document/2780017. Last accessed 05/10/2015

Ghezzi G, Gall HC (2013) Replicating mining studies with SOFAS. In: Proceedings of the 10th working conference on mining software repositories

Greiler M, Herzig K, Czerwonka J (2015) Code ownership and software quality: a replication study. In: Proceedings of the 12th working conference on mining software repositories (MSR), pp 2–12. IEEE Press

Group TO (2014) Application Response Measurement - ARM. https://collaboration.opengroup.org/tech/management/arm/. Last accessed 24 November 2014

Han J (2005) Data mining: concepts and techniques. Morgan Kaufmann Publishers Inc., San Francisco

Hassan AE, Martin DJ, Flora P, Mansfield P, Dietz D (2008) An industrial case study of customizing operational profiles using log compression. In: Proceedings of the 30th International Conference on Software Engineering (ICSE)

JDT Java development tools (2015) https://eclipse.org/jdt/. Accessed 23 October 2015

Jiang ZM, Hassan AE, Hamann G, Flora P (2008) Automatic identification of load testing problems. In: Proceedings of the 24th IEEE international conference on software maintenance (ICSM)

Jiang ZM, Hassan AE, Hamann G, Flora P (2009) Automated performance analysis of load tests. In: Proceedings of the 25th IEEE international conference on software maintenance (ICSM)

Kampstra P (2008) Beanplot: a boxplot alternative for visual comparison of distributions. J Stat Softw Code Snippets 28(1)

logstash - open source log management (2015) http://logstash.net/. Accessed 18 April 2015

LOG4J a logging library for Java (2016) http://logging.apache.org/log4j/1.2/. Accessed 8 April 2016

Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and mozilla. ACM Trans Softw Eng Methodol 11(3):309–346

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. Association for Computing Machinery, Inc.

Nagios Log Server - Monitor and Manage Your Log Data (2015) https://exchange.nagios.org/directory/Plugins/Log-Files. Accessed 10 May 2015

Oliner A, Ganapathi A, Xu W (2012) Advances and challenges in log analysis. Commun ACM 55(2):55–61

Premraj R, Herzig K (2011) Network versus code metrics to predict defects: A replication study. In: Proceedings of the 2011 international symposium on empirical software engineering and measurement (ESEM), pp. 215–224

Rahman F, Posnett D, Herraiz I, Devanbu P (2013) Sample size vs. bias in defect prediction. In: Proceedings of the 9th joint meeting on foundations of software engineering (ESEC/FSE)

Rajlich V (2014) Software Evolution and Maintenance. In: Proceedings of the on future of software engineering (FOSE), pp 133–144. ACM

Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: a case study of the apache server. In: Proceedings of the 30th international conference on software engineering (ICSE), pp 541–550

Robles G (2010) Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR), pp 171–180

Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: Annual meeting of the Florida Association of Institutional Research

Shang W, Jiang ZM, Adams B, Hassan A (2009) MapReduce as a general framework to support research in Mining Software Repositories (MSR). In: Proceedings of the 6th IEEE international working conference on mining software repositories

Shang W, Jiang ZM, Adams B, Hassan AE, Godfrey MW, Nasser M, Flora P (2014) 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

Shang W, Jiang ZM, Hemmati H, Adams B, Hassan AE, Martin P (2013) Assisting developers of big data analytics applications when deploying on hadoop clouds. In: Proceedings of the 35th international conference on software engineering (ICSE)

Shang W, Nagappan M, Hassan AE (2015) Studying the relationship between logging characteristics and the code quality of platform software. Empir Softw Eng 20(1)

Splunk (2015) http://www.splunk.com/. Accessed 18 April 2015

Summary of Sarbanes-Oxley Act of 2002 (2015) http://www.soxlaw.com/. Accessed 10 May 2015

Syer MD, Jiang ZM, Nagappan M, Hassan AE, Nasser M, Flora P (2014) Continuous validation of load test suites. In: Proceedings of the 5th ACM/SPEC international conference on performance engineering (ICPE)

Syer MD, Nagappan M, Adams B, Hassan AE (2015) Replicating and re-evaluating the theory of relative defect-proneness. IEEE Trans Softw Eng 41(2):176–197

Tan L, Yuan D, Krishna G, Zhou Y (2007) /* iComment: Bugs or Bad Comments? */. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)

The AspectJ project (2015) https://eclipse.org/aspectj/. Accessed 10 May 2015

The replication package (2015) https://www.dropbox.com/s/tf5omwtaylffsbs/replication_package_major_revision.zip?dl=0. Accessed 23 October 2015

Wheeler D SLOCCOUNT source lines of code count. http://www.dwheeler.com/sloccount/

Woodside M, Franks G, Petriu DC (2007) The Future of Software Performance Engineering. In: Proceedings of the future of software engineering (FOSE) track, international conference on software engineering (ICSE)

Xu W, Huang L, Fox A, Patterson D, Jordan MI (2009) Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles (SOSP)

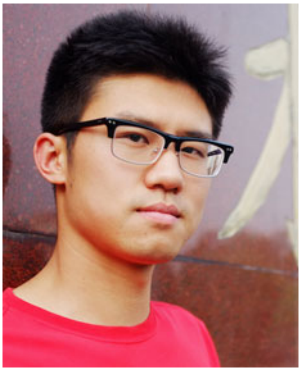
Yuan D, Mai H, Xiong W, Tan L, Zhou Y, Pasupathy S (2010) Sherlog: Error diagnosis by connecting clues from run-time logs. In: Proceedings of the fifteenth edition of ASPLOS on architectural support for programming languages and operating systems (ASPLOS)

Yuan D, Park S, Zhou Y (2012) Characterizing logging practices in open-source software. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Piscataway, pp 102–112

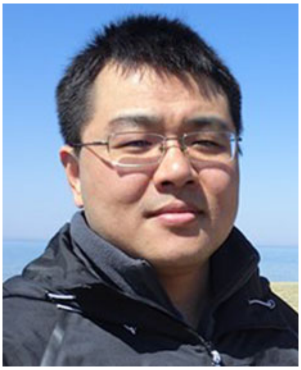
Yuan D, Zheng J, Park S, Zhou Y, Savage S (2011) Improving software diagnosability via log enhancement. In: Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems (ASPLOS)

Zhu J, He P, Fu Q, Zhang H, Lyu MR, Zhang D (2015) Learning to log: Helping developers make informed logging decisions. In: Proceedings of the 37th international conference on software engineering

Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C (2010) What makes a good bug report? Transactions on Software Engineering (TSE)



Boyuan Chen is a graduate student at the Department of Electrical Engineering and Computer Science, York University in Toronto, ON, Canada. He received his Bachelor of Engineering degree from the School of Computer Science at University of Science and Technology of China in Hefei, China. His research interests are mining software repositories, source code analysis and software visualizations.



Zhen Ming (Jack) Jiang He received the BMath and MMath degrees in computer science from the University of Waterloo, and the PhD degree from the School of Computing at the Queen's University. He is an assistant professor in the Department of Electrical Engineering and Computer Science, York University. Prior to joining York, he was at BlackBerry Performance Engineering Team. His research interests lie within software engineering and computer systems, with special interests in software performance engineering, mining software repositories, source code analysis, software architectural recovery, software visualizations and debugging and monitoring of distributed systems. Some of his research results are already adopted and used in practice on a daily basis. He is the cofounder and co-organizer of the annually held International Workshop on Large-Scale Testing (LT). He also received several Best Paper Awards including ICSE 2015 (SEIP track), ICSE 2013, WCRE 2011, and MSR 2009 (challenge track).