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

Studying the Relationship between Logging Characteristics and the Code Quality of Platform Software

Weiyi Shang · Meiyappan Nagappan · Ahmed E. Hassan

Received: date / Accepted: date

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

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

Our results show that logging characteristics provide strong indicators of defect-prone source code files. However, we note that removing logs is not the

Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, Ontario, Canada
Tel.: +1 613-533-6802
E-mail: {swy, mei, ahmed}@cs.queensu.ca

answer to better code quality. Instead, our results show that it might be the case that developers often relay their concerns about a piece of code through logs. Hence, code quality improvement efforts (e.g., testing and inspection) should focus more on the source code files with large amounts of logs or with large amounts of log churn.

Keywords Mining software repositories · Software logs · Software quality

1 Introduction

Large platform software provides an infrastructure for a large number of applications to run over it. *Hadoop* and *JBoss* are examples of popular open source platform software. Such software relies heavily on logs to monitor the execution of the applications running on top of it. These logs are generated at run-time by logging statements in the source code. Generating logs during the execution plays an essential role in field debugging and support activities. These logs are not only for the convenience of developers and operators, but has already become part of legal requirements. For example, the Sarbanes-Oxley Act of 2002 [5] stipulates that the execution of telecommunication and financial applications must be logged. Although logs are widely used in practice, and their importance has been well-identified in prior software engineering research [18, 53, 63], logs have not yet been fully leveraged by empirical software engineering researchers to study code quality.

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

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

To meet this need for run-time information, developers and operators record note-worthy system events, including domain-level and implementation-level events in the logs. In many cases, logs are expected to be leveraged for fixing issues, with additional analyses and diagnoses. Therefore the inclusion of more logs in a source code file by a developer could be an indicator that this particular piece of source code is more critical. Hence, there could be a direct link between logging characteristics and code quality. However, except for individual experiences and observations, there are no empirical studies that

[1] https://issues.apache.org/jira/browse/HADOOP-2490 last checked on May 2013.
[2] https://issues.apache.org/jira/browse/HADOOP-1034 last checked on May 2013.

attempt to understand the relationship between logs and code quality. In this paper we seek to study the relationship between the characteristic of logs, such as log density and log churn, and code quality, especially for large platform software. We use post-release defects as a measurement of code quality since it is one of the most important and widely studied aspects of it [56]. In order to study this relationship, we perform a case study on four releases of *Hadoop* and four releases of *JBoss*. In particular, we aim to answer the following research questions:

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

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

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

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

This paper is the first work to establish an empirical link between logs and defects. We observe positive correlation between logging characteristics and post-release defects in all studied releases. Therefore, practitioners should allocate more effort on source code files with more logs or log churn. However, such positive correlations do not imply that logs are harmful or that they should be removed. For instance, prior research has shown that files with high churn are more defect prone [41,42]. Such studies do not imply that we should not change files. Instead, our study along with prior studies provide indicators to flag high-risk files that should be carefully examined (tested and/or reviewed) prior to release in order to avoid post-release defects.

The rest of this paper is organized as follows: Section 2 presents a qualitative study to motivate this paper. Section 3 presents the background and related research for this paper. Section 4 presents our new log-related metrics. Section 5 presents the design and data preparation steps for our case study. Section 6 presents the results of our case study and details the answers to our research questions. Section 7 discusses the threats to validity of our study. Finally, Section 8 concludes the paper.

2 Motivating Study

In order to better understand how developers make use of logs, we performed a qualitative study. We first collected all commits that had logging statement changes in *Hadoop* release 0.16.0 to release 0.19.0 and *JBoss* release 3.0 to release 4.2. We then selected a 5% random sample (280 commits for *Hadoop* and 420 commits for *JBoss*) from all the collected commits with logging statement changes. Once we extracted the commit messages from the sample commits, we follow an iterative process similar to the one from Seaman *et al.* [51] to identify the reasons that developers change the logging statements in source code, until we could not find any new reasons. We identified four reasons using this process and their distributions are reported in Table 1. These four reasons are described below:

- **Field debugging:** Developers often use logs to diagnose run-time or field defects. For example, the commit message of revision *954705* of *Hadoop* says: "*Region Server should never abort without an informative log message*". Looking through the source code, we observed that the Region Server would abort without any logs. In this revision, the developer added logging statements to output the reason for aborting. Developers also change logging statements when they need logs to diagnose pre-release defects. For example, the commit message of revision *636972* of *Hadoop* says: "*Improve the log message; last night I saw an instance of this message: i.e. we asked to sleep 3 seconds but we slept <30 seconds*".
- **Change of feature:** Developers add and change logs when they change features. For example, in revision *697068* of *Hadoop*, developer added a new "KILLED" status for the job status of *Hadoop* jobs and adapted the logs for the new job status. Changing logs due the change of feature is the most common reason for log churn.
- **Inaccurate logging level:** Developers sometimes change logging levels because of an inaccurate logging level. For example, developers of *JBoss* changed the logging level at revision *29449* with the commit message "*Resolves (JBAS-1571) Logging of cluster rpc method exceptions at warn level is incorrect.*". The discussion of the issue report "*JBAS-1571*" shows that the developers considered the logged exception as a normal behaviour of the system and the logging level was changed from "warn" to "trace".
- **Logs that are not required:** Developers often think that logs used for debugging are redundant after the defect is fixed and they remove logs after using them for debugging. For example, the commit message of revision *612025* of *Hadoop* says: "*Remove chatty debug logging from 2443 patch*".

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

Table 1 Distribution of log churns reasons

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

3 Background and Related Work

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

3.1 Log Analysis

In the research area of computer systems, logs are extensively used to detect system anomalies and performance issues. Xu *et al.* [62] created features based on the constant and variable parts of log messages and applied Principal Component Analysis (PCA) to detect abnormal behaviours. Tan *et al.* introduced SALSA, an approach to automatically analyze logs from distributed computing platforms for constructing and detecting anomalies in state-machine views of the execution of a system across machines [60].

Yuan *et al.* [64] propose a tool named *Log Enhancer*, which automatically adds more context to log lines. Recent work by Beschastnikh *et al.* [8] designed an automated tool that infers execution models from logs. The models can be used by developers to verify and diagnose bugs. Jiang *et al.* design log analysis techniques to assist in identifying functional anomalies and performance degradations during load tests [31,32]. Jiang *et al.* [30] study the characteristic of customer problem troubleshooting by the use of storage system logs. They observed that the customer problems with attached logs were resolved sooner than those without logs.

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

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

[3] http://sosp2011.gsd.inesc-id.pt/workshops/slaml, last checked on December 2012.

A recent work by Yuan *et al.* [63] study the logging characteristics in 4 open source systems. They quantify the pervasiveness and the benefit of software logging. They also find that developers modify logs because they often cannot get the correct log message on the first attempt. Our previous research [54,55] study the evolution of logs from both static logging statements and log lines outputted during run time. We find that logs are often modified by developers without considering the needs of operators. The findings from these previous studies motivates this work. However, previous studies only empirically study the characteristics of logs, but do not establish an empirical link with code quality. This paper focus on empirically studying the relationship of such logging characteristics and code quality.

3.2 Software defect modeling

Software engineering researchers have built models to understand the rationale behind software defects. Practitioners use such models to improve their processes and to improve the quality of their software systems. Fenton *et al.* [16] provide a critical review on software defect prediction models. They recommend holistic models for software defect prediction, using Bayesian belief networks. Hall *et al.* [21] recently conduct a systematic review on defect prediction models. They find the methodology used to build models seems to be influential to predictive performance. Prior research typically builds models using two families of software metrics:

- **Product metrics:** Product metrics are typically calculated from source code.
  - *Traditional product metrics:* Early work by Ohlsson and Alberg [45] build defect models using complexity metrics. Nagappan *et al.* [42] also performed case studies to understand the relationship between source code complexity and software defects. Several studies found complexity metrics were correlated to software defects although no single set of metrics can explain defects for all projects [42].
  - *Code dependency metrics:* Prior research investigates the relationship between defects and code dependencies. Zimmermann and Nagappan [66] find code complexity metrics have slightly higher correlation to defects than dependency metrics. However, the dependency metrics perform better than complexity metrics in predicting defects. Nguyen *et al.* [43] replicate the prior study and find that a small subset of dependency metrics have a large impact on post-release failure, while other dependency metrics have a very limited impact.
  - *Topic metrics:* A few recent studies have tried to establish a link between topics and defects. Liu *et al.* [33] proposed to model class cohesions by latent topics. They propose a new metric named Maximal Weighted Entropy (MWE) to measure the conceptual aspects of class cohesion. Nguyen *et al.* [44] apply Latent Dirichlet Allocation

(LDA) [12] to the subject systems using K=5 topics, and for each source code entity they multiply the topic memberships by the entity's LOC. They provide evidence that topic-related metrics can assist in explaining defects. Instead of focusing on the cohesiveness of topics in a entity, Chen *et al.* [14] proposed metrics focus on the defect-prone topics in a code entity. They find that some topics are much more defect-prone than others and the more defect-prone topics a code entity has, the higher are the chances that it has defects.

- **Process metrics:** Process metrics leverage historical project knowledge. Examples of process metrics are prior defects and prior commits (code churn).
  - *Traditional process metrics:* Several studies showed that process metrics, such as prior commits and prior defects, better explain software defects than product metrics (i.e., provide better statistical explanatory power) [10, 39–41]. Hassan [25] used the complexity of source code changes to explain defects. He found that the number of prior defects was a better metric to explain software defects than prior changes. A recent study by Rahman and Devanbu [50] analyze the applicability and efficacy of process and product metrics from several different perspectives. They find that process metrics are generally more useful than product metrics.
  - *Social structure metrics:* Studies haven been conducted to investigate the relationship between social structure and software defects. Wolf *et al.* [61] carry out a case study to predict build failures by inter-developer communication. Pingzger *et al.* [46] and Meneely *et al.* [34] use social network metrics to predict software defects. Bacchelli *et al.* [6] investigate the use of code popularity metrics obtained from email communication among developers for defect prediction. Recent work by Bettenburg *et al.* [9, 10] use a variety of measures of social information to study relationships between these measures and code quality.
  - *Ownership metrics:* There is previous defect modeling research focusing on the ownership and developers' expertise of source code. Early work by Mockus and Weiss [38] define a metric to measure developers' experience on a particular modification request. They use this metric to predict defect. Bird *et al.* focus on the low-expertise developers. They find that the contribution from low-expertise developers play a big role in the defect prediction model. Rahman *et al.* [49] find that stronger ownership by a single author is associated with implicated code. Recent work by Posnett *et al.* [47] propose to use module activity focus and developers' attention focus to measure code ownership and developers' expertise. They find that more focused developers are less likely to introduce defects than less focused developers, and files that receive narrowly focused activity are more likely to contain defects than files that receive widely focused activities.

Due to the limitation of version control systems, most research on defect modeling extract the process metrics on a code-commit level. Mylyn [4] is a tool that can record developers' activity in IDE. Using Mylyn enables researchers to investigate finer-level process metrics. For example, Zhang *et al.* [65] leverage data generated by Mylyn to investigate the effect of file editing patterns on code quality.

Studies show that most product metrics are highly correlated to each other, and so are process metrics [57]. Among all the product metrics, lines of code has been shown to typically be the best metric to explain post-release defects [26]. On the other hand, prior commits and pre-release defects are the best metrics among process metrics to explain post-release defects [19]. Prior research rarely considers comments. However, a relatively recent empirical study by Ibrahim *et al.* [27] studied the relationship between code comments and code quality. They find that a code change in which a function and its comment are co-updated inconsistently (i.e., they are not co-updated when they have been frequently co-updated in the past, or vice versa), is a risky change. Hence they have shown an empirical link between commenting characteristics and code quality. Similarly, the goal of this paper is to investigate and establish an empirical link between logging characteristics and code quality (quantified through post-release defects).

4 Log-related Metrics

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

4.1 Log-related product metrics

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

1. **Log density:** We calculate the number of logging statements in each file. We consider each invocation of a logging library method as a logging statement. For example, with *Log4j* [4], a method invocation by "*LOG*" and the method is one of the logging levels, e.g., "*LOG.info()*", is considered a logging statement. To factor out the influence of code size, we calculate the log density (LOGD) of a file as:

$$LOGD = \frac{\#\ of\ logging\ statements\ in\ the\ file}{LOC} \quad (1)$$

[4] http://wiki.eclipse.org/Mylyn last checked on May 2013.

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

2. **Average logging level:** Logging level, such as "INFO" and "DEBUG", are used to filter logs based on their purposes. Intuitively, high-level logs are for system operators and lower-level logs are for development purposes [17]. We transform the logging level of each logging statement into a quantitative measurement. We consider all log levels including "TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL". We consider that the lowest logging level is 1 and the value of each higher logging level increases by 1. For example, the lowest logging level in *Log4j* is "TRACE", therefore we consider the value of the "TRACE" level as 1. One level above "TRACE" is "DEBUG", so the value of a "DEBUG" level logging statement is 2. We calculate the average logging-level (LEVELD) of each source code file as

$$LEVELD = \frac{\sum_{i=1}^{n} logging\ level\ value_i}{n} \tag{2}$$

where n is the total number of logging statements in the source code file and $logging\ level\ value_i$ is the logging level value of the i^{th} logging statement in the source code file. The higher-level logs are used typically by administrators and lower-level logs are used by developers and testers. Hence the log level acts as an indicator of the users of the logs. Our intuition behind this metric is that some log levels are better indicators of defects.

4.2 Log-related process metrics

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

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

$$LOGADD = \frac{\sum_{i=1}^{TPC} \#\ added\ logging\ statements_i}{TPC} \tag{3}$$

$$LOGDEL = \frac{\sum_{i=1}^{TPC} \#\ deleted\ logging\ statements_i}{TPC} \tag{4}$$

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

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

$$FCOC = \frac{N(defect\ fixing\ commits \cap log\ churning\ commits)}{TPC} \quad (5)$$

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

5 Case Study Setup

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

- **Hadoop** [1] is a large distributed data processing platform that implements the MapReduce [15] data-processing paradigm. We use 4 releases (0.16.0 to 0.19.0) of Hadoop in our case study.
- **JBoss Application Server** [2] (referred as "*JBoss*" in the rest of this paper) is one of the most popular Java EE application servers. We use 4 releases (3.0 to 4.2) of *JBoss* in our case study.

The goal of our study is to examine the relationship between our proposed log-related metrics and post-release defects. Previous studies of software defects [10, 57] typically use an *Eclipse* data set provided by Zimmermann *et al.* [68]. We do not use this data set since we are interested in platform software, where logging is more prominent. *Eclipse* does not have substantial logging code, therefore, *Eclipse* is not an optimal subject system for our study. *Hadoop* and *JBoss* are two of the largest and widely used platform software. Both generate large amounts of logs during their execution, and tools have been developed to monitor the status of both systems using their extensive logs [3, 48]. To avoid the noise from the logging statements in the unit testing code in both projects, we exclude all the unit testing folders from our analysis. Table 2 gives an overview of the subject systems.

Figure 1 shows a general overview of our approach. (A) We mine the SVN repository of each subject system using a tool called J-REX [52] to produce high-level source code change information. (B) We then identify the log-related source code changes from the output of J-REX. (C) We calculate our proposed log-related metrics and traditional metrics. (D) Finally, we use statistical tools, such as R [28], to perform experiments on the data to answer our research questions. In the rest of this section we describe the first 2 steps in more detail.

Table 2 Overview of subject systems.

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

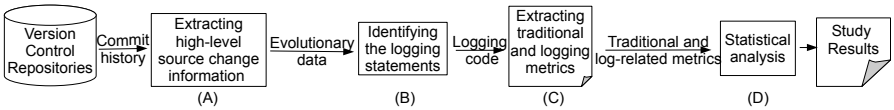

Fig. 1 Overview of our case study approach.

5.1 Extracting high-level source change information

Similar to C-REX [23], J-REX is used to study the evolution of the source code of Java software systems. For each subject system, we use J-REX to extract high-level source change information from their SVN repository.

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

1. **Extraction:** J-REX extracts source code snapshots for each Java file revision from the SVN repository.
2. **Parsing:** Using the Eclipse JDT parser, J-REX outputs an abstract syntax tree for each extracted file in XML.
3. **Differencing:** J-REX compares the XML documents of consecutive file revisions to determine changed code units and generates evolutionary change data. The results are stored in an XML document. There is one XML document for each Java file.

As common practice in the software engineering research community, J-REX uses defect-fixing changes [13, 25, 42] in each source code files to approximate the number of bugs in them. The approximation is widely adopted because (1) only fixed defects can be mapped to specific source code files, (2) some reported defects are not real, (3) not all defects have been reported, and (4) there are duplicate issue reports.

To determine the defect-fixing changes, J-REX uses a heuristic proposed by Mockus *et al.* [37] on all commit messages. For example, a commit message containing the word "fix" is considered a message from a defect-fixing revision. The heuristic can lead to false positives, however, an evaluation of the J-REX heuristics shows that these heuristics identify defect-fixing changes with high accuracy [7].

5.2 Identifying the Logging Statements

Software projects typically leverage logging libraries to generate logs. One of the most widely used logging libraries is *Log4j* [4]. We manually browse a sample of source code from each project and identify that both subject systems use *Log4j* as their logging library.

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

6 Case Study Results

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

6.1 Preliminary Analysis

We start with a preliminary analysis of the log-related metrics presented in Section 5 to illustrate the general properties of the collected metrics.

In the preliminary analysis we calculate seven aggregated metrics for each release of both subject systems: total lines of code, total code churn (total added, deleted and modified lines of code), total number of logging statements, total log churn (total added and deleted logging statements), percentage of source code files with logs, percentage of source code files with pre-release defects, and percentage of source code files with post-release defects. Table 3 shows that around 18% to 28% of the source code files contain logging statements. Since less than 30% of the source code files have logs, we calculate the skew and Kurtosis values for our log-related metrics. We observe that our log-related metrics have positive skew (i.e., all the metric values are on the low scale) and large Kurtosis values (i.e., the curve is too tall). To alleviate the bias caused by these high skew and Kurtosis values, we follow a typical approach used in previous research [57]: to log transform all of the metrics. From this point on, whenever we mention a metric, we actually are referring to its log transformed value.

6.2 Results

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

Table 3 Lines of code, code churn, amounts of logging statements, log churns, percentage of files with logging, percentage of files with pre-release defects, and percentage of files with post-release defects over the releases of Hadoop and JBoss.

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

Motivation: Our qualitative study in Section 2 shows that software developers often add or modify logs to diagnose and fix software defects. We want to first explore the data to study whether source code files with logging statements are more likely to be defect-prone.

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

Results and discussion:

We find that source code files with logging statements are more defect-prone. Table 4 shows the average post-release defect densities of source code files with and without logging statements. The results show that in 7 out of the 8 studied releases, source code files with logging statements have higher average post-release defect densities than source code files without logging statements.

We use independent two-sample unpaired one-tailed T tests to determine whether the average defect density of source code files with logs was statistically greater than those without logs. Our null hypothesis assumes that the average post-release defect densities of source code files with and without logging statements are similar. Our alternate hypothesis was that the average defect density of source code files with logs was statistically greater than those without logs. For 5 of the 7 releases where source code files with logs have higher defect density, the p-values are smaller than 0.05 (see Table 4). We reject the null hypothesis and can conclude that in these 5 releases, the av-

Table 4 Average defect densities of the source code files with and without logging statements in the studied releases. Largest defect densities are shown in bold. The p-value for significance test is 0.05.

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

erage defect density of source code files with logs is greater than those without logs.

We examine in detail the release 0.18.0 of *Hadoop*, which is the exception where the average defect densities of source code files with and without logs are similar. We found that there is a structural change in *Hadoop* before release 0.18.0 and that a large number of defects appear after this release (largest percentage of defect-prone source code files in *Hadoop* as shown in Table 3). This might be the reason that in release 0.18.0 of *Hadoop*, the software source code files with logging statements are not more defect-prone.

Log-related metrics have positive correlation with post-release defects. Table 5 presents the Spearman correlations between our log-related metrics and post-release defects. We find that, in 7 out of 8 releases, the largest correlations between log-related metrics and post-release defects are similar (3 releases with a $+/-$ 0.03 value) or higher than the correlations between pre-release defects and post-release defects. Since the number of pre-release defects is known to have a positive correlation with post-release defects, this observation supports our intuition of studying the relation between logging characteristics and code quality.

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

Density of logging statements added has higher correlation with post-release defects than density of logging statements deleted. We find that the average logging statements added in a commit has the largest correlation with post-release defects in 5 out of 8 releases, while the correlation between the average deleted logging statements in a commit and post-release

Table 5 Spearman correlation between log-related metrics and post-release defects. Largest number in each release is shown in bold.

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

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

defects is much lower than the other log-related metrics (see Table 5). We count the number of added and deleted logging statements in source code files with and without defects separately. We find that in *Hadoop*, the total lines of code ratio between defect-prone source code files and non defect-prone source code files is 1.03; while the number of logging statements added in defect-prone source code files (2,309) is around 3 times that of the number of logging statements added (736) in non defect-prone source code files. This shows that there exists a relation between logs and defect-prone source code files. However, the number of logging statements deleted in defect-prone source code files (268) is only around 2 times that of the number of logging statements deleted in non defect-prone source code files(124). Therefore, even though developers delete more log lines in defect-prone source code files, the ratio with non defect-prone source code files is much lower in comparison to the ratio for log lines added. Hence, this shows that the developers may delete logs when they feel confident with their source code files. Concrete examples of such logging behaviour has been presented in Section 2.

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

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

Table 6 Spearman correlation between the two log-related product metrics: log density (LOGD) and average logging level (LEVELD), and the three log-related process metrics: average logging statements added in a commit (LOGADD), average logging statements deleted in a commit (LOGDEL), and frequency of defect-fixing code changes with log churn (FCOC).

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

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

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

Approach: We use logistic regression models to study the explanatory power of our log-related metrics on post-release defects. However, previous studies show that traditional metrics, such as lines of code (LOC), code churn or the total number of prior commits (TPC), and the number of prior defects (PRE), are effective in explaining post-release software defects [20, 42]. Therefore, we included these metrics as well in the logistic regression models. Note that many other product and process software metrics are highly correlated with each other [57]. To avoid the collinearity between TPC and PRE, we run PCA on TPC and PRE and use the first component as a new metric, which we call TPCPRE:

$$TPCPRE = PCA(TPC, PRE)_{firstcomponent} \quad (6)$$

Before building the logistic regression models, we study the Spearman correlation between the two log-related product metrics and the three log-related process metrics. From the results in Table 6, we find that in some releases, the correlations between the two log-related product metrics and between the three log-related process logging metrics are high.

To address the collinearity as noted in Table 6, we derive two new metrics: a log-related product metric (PRODUCT) and a log-related process metric (PROCESS), to capture the product and process aspects of logging respectively. To compute the two new metrics, we ran Principal Component Analysis (PCA) [29] once on the log-related product metrics (i.e., log density and

average logging level), and another time on the log-related process metrics (average logging statements added in a commit and frequency of defect-fixing code changes with log churn) [22]. Since the previous section shows that average deleted logging statements (LOGDEL) has rather low correlation with post-release defect (see Table 5), we decided not to include LOGDEL in the rest of our analysis and models. From each PCA run, we use the first principal component as our new metric.

$$PRODUCT = PCA(LOGD, LEVELD)_{firstcomponent} \tag{7}$$

$$PROCESS = PCA(LOGADD, FCOC)_{firstcomponent} \tag{8}$$

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

We want to see whether the log-related metrics can complement traditional product and process based software metrics in providing additional explanatory power. The overview of the models is shown in Figure 2. We start with three baseline models that use these best-performing traditional metrics as independent variables.

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

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

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

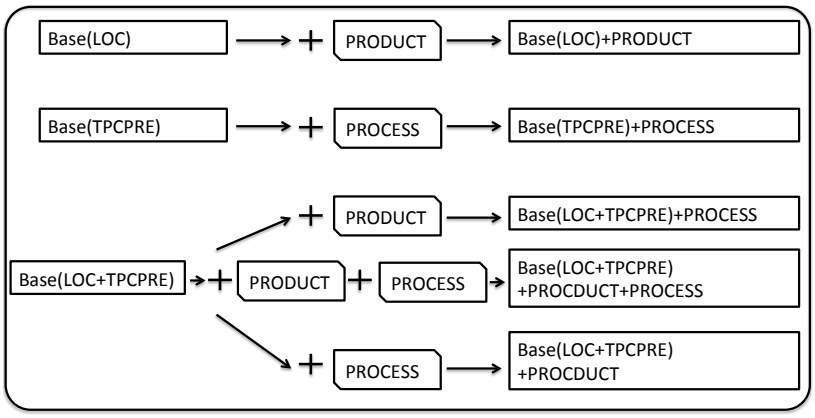

Fig. 2 Overview of the models built to answer RQ2. The results are shown in Table 7, 8 and 9.

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

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

To understand the relationship between logging characteristics and post-release defects, we need to understand the effect of the metrics in the model. We follow a similar approach used in prior research [36, 58]. To quantify this effect, we set all of the metrics in the model to their mean value and record the predicted probabilities. Then, to measure the effect of every log metric, we keep all of the metrics at their mean value, except for the metric whose effect we wish to measure. We increase the value of that metric by 10% off the mean value and re-calculate the predicted probability. We then calculate the percentage of difference caused by increasing the value of that metric by 10%. The effect of a metric can be positive or negative. A positive effect means that a higher value of the factor increases the likelihood, whereas a negative effect means that a higher value of the factor decreases the likelihood of the dependent variable. This approach permits us to study metrics that are of different scales, in contrast to using odds ratios analysis, which is commonly used in prior research [57].

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

Results and discussion:

Table 7 Deviance explained (%) improvement for product software metrics by logistic regression models.

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

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

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

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

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

Table 9 shows the results of using both product and process metrics in the base models. In all studied releases, except for release 0.17.0 of *Hadoop*, at least one log-related metric is statistically significant in enhancing the base model (in bold font). The log-related metrics provide up to 40% of the explanatory power of the traditional metrics.

Release 0.17.0 of *Hadoop* is the release where neither product nor process log-related metrics are significant. In that release, we noted that source code files with logs increased from 18% to 26% (see Table 3). Some logs may be added into defect-free source code files when there is such a large increase in logs. We performed an independent two-sample unpaired two-sided T-test to determine whether the average log-densities of source code files with post-release defects was statistically different to the average log-densities of source code files without post-release defects. The p-value of the test is 0.22. Hence there is no statistical evidence to show that the log-densities of defect-prone and defect-free source code files differ in release 0.17.0 of *Hadoop*. We think this might be the reason that log-released product metrics do not have significant explanation power in *Hadoop* release 0.17.0.

Log-related metrics have a positive effect on the likelihood of post-release defects. In Table 10 we show the effect of the PRODUCT and PROCESS metrics on post-release defects. We measure effect by increasing the value of a metric by 10% from its mean value, while keeping all other metrics at their mean value in a model. We only discuss the effect of log-related metrics that are statistically significant in model Base(LOC+TPCPRE)+PRODUCT +PROCESS (shown in Table 9). The effects of log-related product metric

Table 8 Deviance explained (%) improvement for process software metrics by logistic regression models.

Hadoop				
	0.16.0	0.17.0	0.18.0	0.19.0
Base(TPCPRE)	2.49	8.47	4.53	2.44
Base+PROCESS	11.47(+360%)***	8.55 (+1%)	5.09 (+12%) ⋄	3.69(+51%)***
JBoss				
	3.0	3.2	4.0	4.2
Base(TPCPRE)	10.38	3.75	4.56	2.37
Base+PROCESS	10.55(+2%) ⋄	4.71(+26%)***	4.83(+6%)***	2.73(+15%)***

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

Table 9 Deviance explained (%) improvement using both product and process software metrics by logistic regression models. The values are shown in bold if the model "Base+PRDUCT+PROCESS" has at least one log metric statistically significantly.

Hadoop				
	0.16.0	0.17.0	0.18.0	0.19.0
Base(LOC+TPCPRE)	14.69	13.34	5.3	8.32
Base+PRODUCT	16.56(+13%)**	13.34 (+0%)	6.21 (+17%)*	9.84(+18%)***
Base+PROCESS	19.17(+30%)**	13.4 (+0%)	5.72 (+8%)	8.85(+6%)*
Base+PRODUCT +PROCESS	**20.5(+40%)**	13.42 (+1%)	**6.36 (+20%)**	**9.98(+20%)**
JBoss				
	3.0	3.2	4.0	4.2
Base(LOC+TPCPRE)	12.09	6.46	6.45	3.22
Base+PRODUCT	12.79(+6%)***	6.98 (+8%)***	6.69 (+4%)**	3.34(+4%)*
Base+PROCESS	12.09(+0%)	6.94 (+8%)***	6.51 (+1%)*	3.41(+6%)**
Base+PRODUCT +PROCESS	**12.93(+7%)**	**7.23 (+12%)**	**6.73 (+4%)**	**3.47(+8%)**

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

(PRODUCT) are positive. Since log-related product metric (PRODUCT) is the combination of log density and average logging level, this result implies that more logging statements and/or a higher logging level lead to a higher probability of post-release defects. Table 10 shows that in all 4 releases where the log-related process metric (PROCESS) is statistically significant, the log-related process metric (PROCESS) has a positive effect on defect proneness. The result shows that in some cases, developers change logs to monitor components that might be defect-prone. For example, in revision 226841 of *Hadoop*, developers enhanced the logging statement that tracks nodes in the machine cluster to determine the rationale for field failure of nodes in their cluster. Therefore, in some source code files, the more logs added and/or more defect fixes with log churns, the higher the probability that the source code file is defect-prone.

Summary: Log-related metrics complement traditional product and process metrics in explaining post-release defects. In particular, log-related product metrics contribute to an increased explanatory power in 7 out of 8 studied releases, and log-related process metrics contribute to an increased explanatory

Table 10 Effect of log-related metrics on post-release defects. Effect is measured by setting a metric to 110% of its mean value, while the other metrics are kept at their mean values. The bold font indicates that the metric is statistically significant in the Base(LOC+TPCPRE)+PRODUCT+PROCESS model.

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

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

power in 5 out of 8 studied releases. We also find that both log-related product and process metrics have positive effect on defect proneness.

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

7 Threats to Validity

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

External Validity

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

Internal Validity

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

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

The deviance explained in some of the models may appear low, however this is expected and should not impact the conclusions. One reason for such low deviance is that in a few releases, the percentage of source code files with defects are less than 30% [35, 67]. Moreover, only around 20% to 30% of the

source code files contain logging statements. The deviance explained can be increased by adding more variables to the model in RQ2, however we would need to deal with the interaction between the added variables.

Construct Validity

The heuristics to extract logging source code may not be able to extract every logging statement in the source code. However, since the case study systems use logging libraries to generate logs at runtime, the method in the logging statements are consistent. Hence, this heuristic will capture all the logging statements.

Our software defect data is based on the data produced by J-REX, a software evolution tool that generates high-level evolutionary source code change data. J-REX uses heuristics to identify defect-fixing changes. The results of this paper are dependent on the accuracy of the results from J-REX. We are confident in the results from J-REX as it implements the algorithm used previously by Hassan *et al.* [24] and Mockus *et al.* [37]. However, previous research shows that the mis-classified bug-fixing commits may introduce negative effects on the performance of prediction techniques on post-release defects [11]. We select a random sample of 337 and 366 files for *Hadoop* and *JBoss*, respectively. Only 14% and 2% of the files for *Hadoop* and *JBoss* are misclassified, respectively. Both random sample sizes achieve 95% confidence level with a 5% confidence interval [59]. We will leverage other approaches that identify bug-fixing commits, such as the data in the issue tracking systems, to perform additional case studies in our future work to further understand the relationship between logging characteristics and code quality. J-REX compares the abstract syntax trees between two consecutive code revisions. A modified logging statement is reported by J-REX as an added and a deleted logging statement. Such limitation of J-REX may result in inaccuracy of our metrics. We plan to leverage other techniques to extract the log-related metrics in our future case studies.

In addition, we find that on average, there is a logging statement for every 160 and 130 lines of source code for *Hadoop* and *JBoss*, respectively. A previous study by Yuan *et al.* [64] shows that the ratio between all source code and logging code is 30. We think the reason for such a discrepancy is that the log density metric (LOGD) defined in this paper uses the number of logging statements instead of lines of logging code, and the total lines of code instead of source lines of code. We calculated the ratio between total lines of code and number of logging statements for the four subject systems studied in prior research. We found that the ratios are 155, 146, 137 and 70 for *Httpd*, *Openssh*, *PostgreSQL* and *Squid*, respectively. Such ratios are similar to the ratios in our two studied systems. In this paper, we extract our log-related metrics from AST. We chose to use the AST so that we can accurately extract every invocation to a logging method. Hence due to this choice, we can only get the number of logging statements and not number of lines of logging code. We will compare the metric using the log density metric with the lines of logging code and source lines of code to our log density metric (LOGD) in future case studies.

The possibility of post-release defects can be correlated to many factors other than just logging characteristics, such as the complexity of code and pre-release defects. To reduce such a possibility, we included 3 control metrics (lines of code, pre-release defects, and prior changes) that are well known to be good predictors for post-release defects in our logistic regression model [39,42]. However, other factors may also have an impact on post-release defects. Future studies should build more complex models that consider these other factors.

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

8 Conclusion

Logging is one of the most frequently-employed approaches for diagnosing and fixing software defects. Logs are used to capture the concerns and doubts from developers as well as operators' needs for run-time information about the software. However, the relationship between logging characteristics and software quality has never been empirically studied before. This paper is a first attempt (to the best of our knowledge) to build an empirical link between logging characteristics and software defects. The highlights of our findings are:

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

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

References

1. Hadoop. Http://hadoop.apache.org
2. Jboss application server. Http://www.jboss.org/jbossas
3. Jbossprofiler. Https://community.jboss.org/wiki/JBossProfiler
4. Log4j. Http://logging.apache.org/log4j/1.2/
5. Summary of sarbanes-oxley act of 2002. Http://www.soxlaw.com/

6. Bacchelli, A., D'Ambros, M., Lanza, M.: Are popular classes more defect prone? In: FASE '10: Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering, pp. 59–73. Springer-Verlag, Berlin, Heidelberg (2010)
7. Barbour, L., Khomh, F., Zou, Y.: Late propagation in software clones. In: ICSM 2011: Proceedings of the 27th IEEE International Conference on Software Maintenance, pp. 273 –282 (2011)
8. Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D.: Leveraging existing instrumentation to automatically infer invariant-constrained models. In: ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 267–277. ACM, New York, NY, USA (2011)
9. Bettenburg, N., Hassan, A.: Studying the impact of social interactions on software quality. Empirical Software Engineering **18**(2), 375–431 (2013)
10. Bettenburg, N., Hassan, A.E.: Studying the impact of social structures on software quality. In: ICPC '10: Proceedings of the 18th International Conference on Program Comprehension, pp. 124–133. IEEE Computer Society, Washington, DC, USA (2010)
11. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced?: bias in bug-fix datasets. In: ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 121–130. ACM, New York, NY, USA (2009)
12. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003)
13. Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D.: Software dependencies, work dependencies, and their impact on failures. IEEE Transaction on Software Engineering **35**, 864–878 (2009)
14. Chen, T.H., Thomas, S.W., Nagappan, M., Hassan, A.E.: Explaining software defects using topic models. In: MSR, pp. 189–198. IEEE (2012)
15. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
16. Fenton, N., Neil, M.: A critique of software defect prediction models. Software Engineering, IEEE Transactions on **25**(5), 675–689 (1999)
17. Gilstrap, B.R.: An introduction to the java logging api (2002)
18. Graham, R., Woodall, T., Squyres, J.: The Practice of Programming (2006)
19. Graves, T.L., Karr, A.F., Marron, J., Siy, H.: Predicting fault incidence using software change history. IEEE Transactions on Software Engineering **26**(7), 653–661 (2000)
20. Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting fault incidence using software change history. IEEE Transaction on Software Engineering **26**, 653–661 (2000)
21. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on **38**(6), 1276–1304 (2012)
22. Harrell, F.: Regression Modeling Strategies With Applications to Linear Models, Logistic Regression, and Survival Analysis. Springer (2001)
23. Hassan, A.E.: Mining software repositories to assist developers and support managers. Ph.D. thesis, University of Waterloo (2005)
24. Hassan, A.E.: Automated classification of change messages in open source projects. In: SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, pp. 837–841. ACM, New York, NY, USA (2008)
25. Hassan, A.E.: Predicting faults using the complexity of code changes. In: ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, pp. 78–88. IEEE Computer Society, Washington, DC, USA (2009)
26. Herraiz, I., Hassan, A.: Beyond lines of code: Do we need more complexity metrics? In: A. Oram, G. Wilson (eds.) Making Software: What Really Works, and Why We Believe It? OReilly Media (2010)

27. Ibrahim, W.M., Bettenburg, N., Adams, B., Hassan, A.E.: On the relationship between comment update practices and software bugs. Journal of Systems and Software **85**(10), 2293–2304 (2012)
28. Ihaka, R., Gentleman, R.: R: a language for data analysis and graphics. Journal of computational and graphical statistics pp. 299–314 (1996)
29. Jackson, J., Wiley, J.: A user's guide to principal components, vol. 19. Wiley Online Library (1991)
30. Jiang, W., Hu, C., Pasupathy, S., Kanevsky, A., Li, Z., Zhou, Y.: Understanding customer problem troubleshooting from storage system logs. In: FAST '09: Proccedings of the 7th conference on File and storage technologies, pp. 43–56. USENIX Association, Berkeley, CA, USA (2009)
31. Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: Automatic identification of load testing problems. In: ICSM '08: Proc. of 24th IEEE International Conference on Software Maintenance, pp. 307–316. IEEE, Beijing, China (2008)
32. Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: Automated performance analysis of load tests. In: ICSM '09: Proc. of the 25th IEEE International Conference on Software Maintenance, pp. 125–134. IEEE, Edmonton, Alberta, Canada (2009)
33. Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimothy, T., Chrisochoides, N.: Modeling class cohesion as mixtures of latent topics. In: ICSM 2009: Proceedings of the 2009 IEEE International Conference on Software Maintenance., pp. 233–242 (2009)
34. Meneely, A., Williams, L., Snipes, W., Osborne, J.: Predicting failures with developer networks and social network analysis. In: SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, SIGSOFT '08/FSE-16, pp. 13–23. ACM, New York, NY, USA (2008)
35. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. Software Engineering, IEEE Transactions on **33**(1), 2–13 (2007)
36. Mockus, A.: Organizational volatility and its effects on software defects. In: FSE '10: Proc. of the 18th ACM SIGSOFT International Symp. on Foundations of software engineering, pp. 117–126. ACM, New York, NY, USA (2010)
37. Mockus, A., Votta, L.G.: Identifying reasons for software changes using historic databases. In: ICSM '00: Proceedings of the International Conference on Software Maintenance, pp. 120–. IEEE Computer Society, Washington, DC, USA (2000)
38. Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Technical Journal **5**, 169–180 (2000)
39. Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ICSE 2008: Proceedings of the 30th international conference on Software engineering, pp. 181–190. ACM, New York, NY, USA (2008)
40. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: ICSE '05: Proc. of the 27th international conference on Software engineering, pp. 284–292. ACM, New York, NY, USA (2005)
41. Nagappan, N., Ball, T.: Using software dependencies and churn metrics to predict field failures: An empirical case study. In: ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, pp. 364–373. IEEE Computer Society, Washington, DC, USA (2007)
42. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: ICSE '06: Proceedings of the 28th international conference on Software engineering, pp. 452–461. ACM, New York, NY, USA (2006)
43. Nguyen, T.H.D., Adams, B., Hassan, A.E.: Studying the impact of dependency network measures on software quality. In: ICSM '10: Proceedings of the 2010 IEEE International Conference on Software Maintenance, pp. 1–10. IEEE Computer Society, Washington, DC, USA (2010)
44. Nguyen, T.T., Nguyen, T.N., Phuong, T.M.: Topic-based defect prediction (nier track). In: ICSE '11: Proceedings of the 33rd International Conference on Software Engineering, pp. 932–935. ACM, New York, NY, USA (2011)

45. Ohlsson, N., Alberg, H.: Predicting fault-prone software modules in telephone switches. IEEE Trans. Softw. Eng. **22**, 886–894 (1996)
46. Pinzger, M., Nagappan, N., Murphy, B.: Can developer-module networks predict failures? In: SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp. 2–12. ACM, New York, NY, USA (2008)
47. Posnett, D., D'Souza, R., Devanbu, P., Filkov, V.: Dual ecological measures of focus in software development. In: ICSE '13: Proceedings of the 2013 International Conference on Software Engineering, pp. 452–461. IEEE Press, Piscataway, NJ, USA (2013)
48. Rabkin, A., Katz, R.: Chukwa: a system for reliable large-scale log collection. In: LISA'10: Proc. of the 24th international conference on Large installation system administration, pp. 1–15. USENIX, Berkeley, CA, USA (2010)
49. Rahman, F., Devanbu, P.: Ownership, experience and defects: a fine-grained study of authorship. In: ICSE '11: Proceedings of the 33rd International Conference on Software Engineering, pp. 491–500. ACM, New York, NY, USA (2011)
50. Rahman, F., Devanbu, P.: How, and why, process metrics are better. In: ICSE '13: Proceedings of the 2013 International Conference on Software Engineering, pp. 432–441. IEEE Press, Piscataway, NJ, USA (2013)
51. Seaman, C.B., Shull, F., Regardie, M., Elbert, D., Feldmann, R.L., Guo, Y., Godfrey, S.: Defect categorization: making use of a decade of widely varying historical data. In: ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, pp. 149–157. ACM, New York, NY, USA (2008)
52. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E.: MapReduce as a General Framework to Support Research in Mining Software Repositories (MSR). In: MSR '09: Proceedings of 6th IEEE International Working Conference on Mining Software Repositories, pp. 21–30 (2009)
53. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P.: An exploratory study of the evolution of communicated information about the execution of large software systems. In: WCRE 2011: Proceedings of the 2011 18th Working Conference on Reverse Engineering, pp. 335–344. IEEE Computer Society, Washington, DC, USA (2011)
54. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P.: An exploratory study of the evolution of communicated information about the execution of large software systems. In: WCRE '11: Proceedings of the 2011 18th Working Conference on Reverse Engineering, pp. 335–344. IEEE Computer Society, Washington, DC, USA (2011)
55. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P.: An exploratory study of the evolution of communicated information about the execution of large software systems. Journal of Software: Evolution and Process pp. n/a–n/a (2013)
56. Shihab, E.: An exploration of challenges limiting pragmatic software defect prediction. Ph.D. thesis, Queen's University (2012)
57. Shihab, E., Jiang, Z.M., Ibrahim, W.M., Adams, B., Hassan, A.E.: Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In: ESEM '10: Proc. of the 2010 ACM-IEEE Int. Symposium on Empirical Software Engineering and Measurement, pp. 4:1–4:10. ACM, New York, NY, USA (2010)
58. Shihab, E., Mockus, A., Kamei, Y., Adams, B., Hassan, A.E.: High-impact defects: a study of breakage and surprise defects. In: ESEC/FSE '11: Proc. of the 19th ACM SIGSOFT symp. and the 13th Euro. conf. on Foundations of software engineering, pp. 300–310. ACM, NY, USA (2011)
59. Smithson, M.: Confidence Intervals. Sage Publications, Thousand Oaks, CA, USA (2003)
60. Tan, J., Pan, X., Kavulya, S., Gandhi, R., Narasimhan, P.: Salsa: analyzing logs as state machines. In: WASL'08: Proceedings of the 1st USENIX conference on Analysis

of system logs, pp. 6–6. USENIX, San Diego, California (2008)
61. Wolf, T., Schroter, A., Damian, D., Nguyen, T.: Predicting build failures using social network analysis on developer communication. In: ICSE '09: Proceedings of the 31st International Conference on Software Engineering, pp. 1–11. IEEE Computer Society, Washington, DC, USA (2009)
62. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 117–132. ACM, Big Sky, Montana, USA (2009)
63. Yuan, D., Park, S., Zhou, Y.: Characterizing logging practices in open-source software. In: ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, pp. 102–112. IEEE Press, Piscataway, NJ, USA (2012)
64. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S.: Improving software diagnosability via log enhancement. In: ASPLOS '11: Proceedings of the 16th international conference on Architectural support for programming languages and operating systems, pp. 3–14. ACM, Newport Beach, CA, USA (2011)
65. Zhang, F., Khomh, F., Zou, Y., Hassan, A.E.: An empirical study of the effect of file editing patterns on software quality. In: WCRE '12: Proceedings of the 2012 19th Working Conference on Reverse Engineering, pp. 456–465. IEEE Computer Society, Washington, DC, USA (2012)
66. Zimmermann, T., Nagappan, N.: Predicting defects using network analysis on dependency graphs. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, pp. 531–540. ACM, New York, NY, USA (2008)
67. Zimmermann, T., Nagappan, N., Williams, L.: Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10, pp. 421–428 (2010)
68. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: PROMISE '07: Proc. of the 3rd Int. Workshop on Predictor Models in Software Engineering, pp. 9–15. IEEE, Washington, DC, USA (2007)