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

An Empirical Study on Logging Statement Changes During Bug Fixes

Suhas Kabinna · Weiyi Shang · Ahmed E. Hassan

Received: date / Accepted: date

Abstract Logs are leveraged by software developers to record and convey important information during the execution of a system. These logs are a valuable source of information for developers to debug large software systems. Prior research has shown that logging statements are changed during field debugging. However, little is known about how logging statements are changed during bug fixes. In this paper, we perform a case study on three large open source platform software namely *Hadoop, HBase* and *Qpid*. We find that logs are added, deleted and modified statistically significantly more in bug fixes than other code changes. Furthermore, we find four different types of modifications that developers make to logging statements during bug fixes, including: (1)logging level change, (2)text modification, (3)variable change and (4)logging statement relocation. We find that bugs that are fixed with logging statement changes have larger code churn, but involve fewer developers, require less time and have less discussion during the bug fix. We build a regression model to explore the relationship between metrics from logging statement changes and the resolution time of bugs. We find that these metrics from logging statement changes can complement traditional metrics, i.e., # of developers and # of comments, in explaining the bug resolution . In particular, we find a negative relationship between modifying logging statements and the resolution time of bugs. Our results suggests that there is a relationship between changing logging statements and the resolution time of bugs.

Suhas. Kabinna, Weiyi. Shang and Ahmed E.Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, Ontario
E-mail: {kabinna,swy,ahmed}@cs.queensu.ca

1 Introduction

Platform software provides an infrastructure for a variety of applications that run on top of it. Platform software relies on logs to monitor the applications that run over them. Such logs are generated through simple *printf* statements or through the use of logging libraries such as 'Log4j', 'Slf4j', and 'JCL'. Each logging statement contains a textual part that gives information about the context, a variable part that contains knowledge about the events, and a logging level that shows the verbosity of the logs. An example of a logging statement is shown below where *info* is the logging level, *Connected to* is the event and the variable *host* contains the information about the logged event.

```
LOG.info( "Connected to " + host);
```

Research has shown that logs are used by developers extensively during the development of software systems [1]. Logs are leveraged for anomaly detection [2–4], system monitoring [5], capacity planning [6] and large-scale system testing [7]. The valuable information in logs has created a new market for log maintenance platforms like Splunk [5], XpoLog [8], and Logstash [9], which assist developers in analyzing logs.

Logs are extensively used to help developers fix bugs in platform software. For example, in the JIRA issue HBASE-3403 [1], a bug was reported when a region is orphaned when there is system failure during a split. Developers leveraged logs to identify the point of failure. After fixing the bug, logs were updated to prevent similar bugs in the system. Prior study shows that the changes to logging statements have a strong relationship with code quality [10]. However, there exists no large scale study that investigate how logging statements are changed during bug fixes.

In this paper, we perform an empirical study on the changes to logging statements during bug fixes in three open source platform software, i.e., *Hadoop*, *HBase* and *Qpid*. In particular, we sought to answer the following research questions.

RQ1: Are logging statements changed more during bug fixes?

We find that logs are changed more in bug fixing commits than non-bug fixing commits. In particular, we find that adding and modifying logging statements appear statistically significantly more in bug fixing commits than non-bug fixing commits with non-trivial effect size. We identified four types of modifications to logging statements, including *Logging level change*, *Text modification*, *Variable change* and *Logging statement relocation*. We find that *Text modification*, *Variable change* and *Logging statement relocation* exist statistically significantly more with medium to high effect sizes in bug fixing commits than non-bug fixing commits.

RQ2: Are bugs fixed faster with logging statement changes?

[1] https://issues.apache.org/jira/browse/HBASE-3403

Table 1: An overview of the platform softwares

Projects	Hadoop		HBase		Qpid	
	Bug fixing	Non-Bug Fixing	Bug fixing	Non-Bug Fixing	Bug fixing	Non-Bug Fixing
# of Revisions	7,366	12,300	5,149	7,784	1,824	5,684
Code Churn	4,09K	3.2M	1.4M	2.18M	175k	2.3M
Log Churn	4,311	23,838	4,566	12,005	597	10,238

We find that bug fixing commits with logging statement changes have higher code churn. After normalizing the code churn, the bugs with logging statement changes take less time to get resolved, involve fewer developers and have less discussions during the bug fixing process.

RQ3: Can metrics from logging statement changes help in explaining the resolution time of bugs?
Using metrics from logging statement changes and the traditional metrics (i.e., # comments and # developers) we trained regression models for the resolution time of bug fixes. We find that metrics from logging statement changes are statistically significant in the models. This suggests that there is a relationship between logging statement changes and the resolution time of bugs.

The rest of this paper is organized as follows. Section 2 presents the methodology for extracting data for our study. Section 3 presents the case studies and the results to answer the three research questions. Section 4 discuss the change to logs through a manual study. Section 5 describes the prior research that is related to our work. Section 6 discusses the threats to validity. Finally, Section 7 concludes the paper.

2 Methodology

In this section, we describe our method for preparing the data to answer our research questions.

The aim of this paper is to understand logging statement changes during bug fixes. We conduct a case study on three open source platform software, i.e., *Hadoop*, *HBase* and *Qpid*. All three platform softwares have extensive logging in their source code. Table 1 highlights the overview of the three platform softwares.

Hadoop[1]: *Hadoop* is an open source software framework for distributed storage and processing of big data on clusters. *Hadoop* uses the MapReduce data-processing paradigm. The logging characteristics of *Hadoop* have been extensively studied in prior research [10, 11, 3]. We study the changes to logging statements from *Hadoop* releases 0.16.0 to 2.0.

[1] http://hadoop.apache.org/

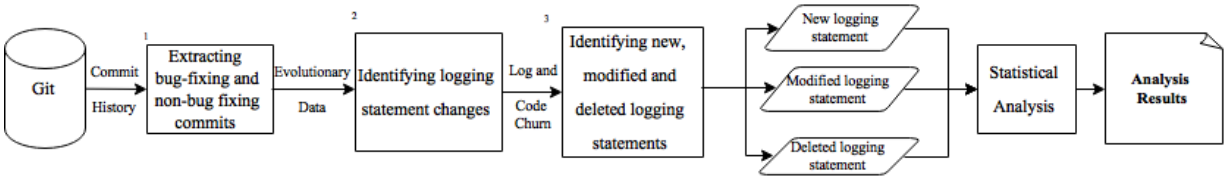


Fig. 1: Overview of our cast study approach

HBase[2]: *HBase* is a distributed, scalable, big data software, using *Hadoop* file-systems. We study the changes to logging statements in *HBase* from release 0.10 to 0.98.2.*RC*0. This covers more than four years of development in *HBase* from 2010 to 2014.

Qpid[3]: *Qpid* is an open source messaging platform that implements an Advanced Message Queuing Protocol (AMQP). We study *Qpid* release 0.10 to release 0.30 that are from 2011 till 2014.

Figure 1 shows a general overview of our approach, which consists of four steps: (1) We mine the Git repository of each subject system to extract all commits and identify bug fixing and non-bug fixing commits. (2) We identify logging statement changes in both bug fixing and non-bug fixing commits. (3) We categorize the logging statement changes into *'New logs'*, *'Modified Logs'* and *'Deleted logs'*. (4) We calculate churn metrics for each category and use statistical tool R [12], to perform experiments on the data to answer our research questions. In the reminder of this section we describe the first three steps.

2.1 Study Approach

In this section we present the approach of our case study.

2.1.1 Extracting bug-fixing and non-bug fixing commits

The first step in our approach is to extract bug fixing and non-bug fixing commits. First, we extract a list of all commits from Git. To avoid the branching and merging commits, we enable the 'no-merges' option in the Git *log* command to exclude all the merging operations in the systems. We also filter the non-Java and 'test' files present in the commits.

Next, we extract a list of all JIRA issues that have the type 'bug'. Developers often mention the JIRA issue ID's in the commit messages. We search JIRA issue IDs in the commit messages to identify all the bug fixing commits. If a commit message does not contain a JIRA issue ID, we search for bug fixing keywords like 'fix' or 'bug'. Prior research has shown that such heuristics can identify bug fixing commits with a high accuracy [10].

[2] http://hbase.apache.org/
[3] https://qpid.apache.org

2.1.2 Identifying logging statement changes

To identify the logging statement changes in the datasets, we first manually explore logging statements in the source code. Some logging statements are specific to a particular project. For example, a logging statement from Qpid invokes 'QPID_LOG' to print logs as follows:

```
QPID_LOG(error, "Rdma: Cannot accept new connection (Rdma exception): " + e.what());
```

Some logging statements leverage logging libraries to print logs. For example, *Log4j*[2] is used widely in *Hadoop* and *HBase*. In both projects, logging statements have a method invocation 'LOG', followed by logging-level. The following logging statement that uses *Log4j* to print logs:

```
LOG.debug(" public AsymptoticTestCase(String"+ name +") called")
```

Using regular expressions to match these logging statements, we automate the process of finding all the logging statements in our datasets.

2.1.3 Identifying new, modified and deleted logging statements

Since, Git *diff* does not provide a feature to track modification to a file, modifications to logging statements are shown as added and deleted logging statements. We used Levenshtein ratio [13] to identify modifications to logging statements. For every pair of added or deleted logging statement in a commit, we compare the text in parenthesis after removing the logging method (e.g, LOG) and the log level (e.g, info). We calculate the Levenshtein ratio between the added and deleted logging statement similar to prior research [14]. We consider a pair of added and deleted logging statements as log modification if they have a Levenshtein ratio of 0.6 or higher. For example, the logging statements shown below have Levenshtein ratio of 0.86. Hence this logging statement change is categorized as a log modification.

```
+ LOG.debug("Call: " +method.getName()+" took "+ callTime + "ms");
- LOG.debug("Call: " +method.getName()+ " " + callTime);
```

If an added or deleted logging statement matches with more than one deleted or added logging statements with over 0.6 Levenshtein ratio, we consider the pair of added and deleted logging statements with the highest Levenshtein ratio as modifications to logging statements. After identifying log modifications to logging statements, we identify modified, new and deleted logging statements.

[2] http://logging.apache.org/log4j/1.2/

3 Study Results

In this section, we present our case study results by answering our three research questions. For each question, we discuss the motivation behind it, the approach to answering the research question and finally the results.

RQ1: Are logging statements changed more during bug fixes?

Motivation

Prior research has shown that up to 32% of the logging statements changes are due to field debugging [10]. During debugging, developers change logging statements to gain more run-time information about the systems. Therefore, future occurrences of a similar bug may be resolved easily with the updated logging statements. However, to the best of our knowledge, there exists no large scale empirical study to show whether logging statements are changed more during bug fixes than other activities during development. In addition, we want to investigate how logging statements are changed during the bug fixing.

Approach

We compare the number of changes to logging statements between bug-fixing and non-bug-fixing commits. In previous section, we identified three types of changes to logging statements, i.e., modified, new and deleted logging statements. Therefore, we compare the number of each type of changes to logging statements between bug-fixing and non-bug-fixing commits. Since commits with higher total code churn may have number of changes to logging statements. Therefore, we calculate total code churn for every commit and use it to normalize *# modified, # new* and *# deleted logging statements*. The three new metrics are:

$$Modified\ logging\ statements\ ratio = \frac{\#\ modified\ logging\ statements}{code\ churn} \quad (1)$$

$$New\ logging\ statements\ ratio = \frac{\#\ new\ logging\ statements}{code\ churn} \quad (2)$$

$$Deleted\ logging\ statements\ ratio = \frac{\#\ deleted\ logging\ statements}{code\ churn} \quad (3)$$

To future understand how logging statements are modified during bug fixes, we perform a manual analysis on the modified logging statements to identify the different types of log modifications. We first collect all the commits that modify logging statement. We select a random sample of 357 commits. The size of our random sample achieves 95% confidence level and 5% confidence interval. We follow an iterative process, similar to prior research [15], to identify

the different types of log modifications, until we cannot find any new types of modifications.

After we identify the types of log modifications, we create an automated tool to label log modifications into the identified types. We calculate the number of log modifications of every type in each commit and normalize for *code churn*, similar to Equation 1 to 3.

To determine whether there is a statistically significant difference of these metrics, in bug-fixing and non-bug-fixing commits, we perform the *MannWhitney U test* (Wilcoxon rank-sum test) [16]. We choose *MannWhitney U test* because our metrics are highly skewed. Since *MannWhitney U test* is a non-parametric test, it does not have any assumptions about the distribution of the sample population. A p-value of ≤ 0.05 means that the difference between the two data sets is statistically significant and we may reject the null hypothesis (i.e., there is no statistically significant difference of our metrics between bug-fixing and non-bug-fixing commits). By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us that there is a statistically significantly difference of our metrics between bug-fixing and non-bug-fixing commits.

We also use *effect sizes* to measure how big is the difference of our metrics between the bug-fixing and non-bug-fixing commits. Unlike *MannWhitney U test*, which only tells us whether the difference between the two distributions are statistically significant, effect sizes quantify the difference between two distributions. Researchers have shown that reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, p-value can be small even if the difference is trivial). We use *Cohen's d* to quantify the effects. *Cohen's d* measures the effect size statistically, and has been used in prior engineering studies [17,18]. *Cohen's d* is defined as:

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s}, \tag{4}$$

where $\bar{x}_1$ and $\bar{x}_2$ are the mean of two populations, s is the pooled standard deviation and d is *Cohen's d* [19]. As software engineering has different thresholds for *Cohen's d* [20], the new scale is shown below.

$$\begin{cases} \text{trivial} & \text{for } d \leq 0.17 \\ \text{small} & \text{for } 0.17 < d \leq 0.6 \\ \text{medium} & \text{for } 0.6 < d \leq 01.4 \\ \text{large} & \text{for } d > 1.4 \end{cases} \tag{5}$$

Results

Developers add new logging statements more during bug fixes. Table 2 shows that *new logging statements ratio* in bug-fixing commits is statistically significantly higher than non-bug-fixing commits in all subject systems with non-trivial effect sizes. This suggests that developers add new logging

Table 2: Comparing logging statement changes metrics between the bug-fixing and non-bug-fixing commits. The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size means that bug-fixing commits have larger metric values and p-values are bold if they are smaller than 0.05.

Metrics	Hadoop		HBase		Qpid	
	P-Values	Effect Size	P-Values	Effect Size	P-Values	Effect Size
Modified logging statements ratio	**2.0e-12**	0.246 (small)	**1.9e-15**	0.273 (small)	**1.6e-11**	0.432 (small)
New logging statements ratio	**4.7e-16**	0.265 (small)	**<2.2e-16**	0.215 (small)	**2.1e-11**	0.474 (small)
Deleted logging statements ratio	**8.1e-07**	0.336 (small)	**4.9e-07**	0.150	**0.041**	-0.193 (small)

Table 3: Distribution of four types of log modifications.

Projects	Hadoop (%)	HBase (%)	Qpid (%)
Logging statement relocation	73.1	70.7	47.4
Text Modification	10.5	13.4	16.8
Variable Modification	9.9	10.1	18.9
Logging Level Change	6.5	5.8	16.8

statements during bug fixes more than non-bug-fixing commits. We find that effect size of new logging statement is higher in *Qpid* than *Hadoop* and *HBase*. This may be because that *Qpid* is a relatively newer system. Some important source code may not be well logged. Therefore, developers may have to add additional logging statements to assist in bug-fixing.

Developers may not delete logging statements during bug fixes. We find that although the difference of *deleted logging statements ratio* between bug-fixing commits and non-bug-fixing commits is statistically significant in all projects, the effect sizes is trivial for *HBase* and negative for *Qpid* (see Table 2). This result shows that developers of Qpid delete logging statements more during non-bug-fixing commits than bug-fixing commits. Such results confirm the findings from prior research that deleted logging statements do not have a strong relationship with code quality [6].

Logging statements are modified more in bug-fixing commits than non-bug-fixing commits. Table 2 shows that $modified\ logging-$

Table 4: Comparing logging modification metrics between the bug-fixing and non-bug-fixing commits. The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size means that bug-fixing commits have larger metric values and p-values are bold if they are smaller than 0.05

Metrics	Hadoop		HBase		Qpid	
	P-values	Effect Size	P-values	Effect Size	P-values	Effect Size
Logging statement relocation	**1.1e-10**	0.330 (small)	**3.0e-11**	0.170 (small)	**1.8e-08**	0.700 (medium)
Text modification	-	-	**0.0075**	0.525 (small)	**4.5e-06**	0.976 (medium)
Variable change	**1.3e-04**	0.351 (small)	**0.0010**	0.420 (small)	**1.2e-04**	1.17 (medium)
Logging level change	-	-	-	-	-	-

Table 5: Comparing variable change metrics between the bug-fixing and non-bug-fixing commits. The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size means that bug-fixing commits have larger metric values and p-values are bold if they are smaller than 0.05

Metrics	Hadoop		HBase		Qpid	
	P-values	Effect Size	P-values	Effect Size	P-values	Effect Size
Variable addition	-	-	**0.00049**	0.659 (medium)	**0.005**	1.40 (large)
Variable deletion	-	-	-	-	-	-
Variable modification	**4.11e-05**	1.045 (medium)	-	-	**0.0016**	0.949 (medium)

statements ratio is statistically significantly higher n bug-fixing commits than non-bug-fixing commits for all subject systems and the effect sizes are non-trivial in all these systems. Such results show that developers often change the information provided by logging statements to assist in bug-fixing. Developers may need different information to the information that is provided by the logging statements to fix the bugs. Prior research also finds that 33% of logging statements are modified at-least once as after-thoughts [1]. Therefore, we further explore how developers modify logging statements.

We manually identify four types of modifications to logging statements. Table 3 shows their distributions.

1. **Logging statement relocation:** The logging statement is not changed but moved to a different place in the file.

2. **Text modification:** The text that is printed in the logging statements is modified.
3. **Variable change:** One or more variables in the logging statements are changed (added, deleted or modified).
4. **Logging level change:** The verbosity level of logging statements are changed.

Developers modify variables more in bug-fixing commits. We find that variable changes are statistically significantly more in bug-fixing commits than non-bug-fixing commits in all the subject systems with small or medium effect sizes (see Table 4). Developers modify the variables that are printed in their logging statements in order to provide useful information about the system to assist in bug-fixing. To better understand how developers change variables in logging statements during bug-fixing, we categorize the variable changes into three types: a) variable addition, b) variable deletion and c) variable modification.

Table 5 shows that developers modify variables statistically significantly more in *Hadoop* and *Qpid* with medium effect sizes. This may be because developers need different information than provided by existing logging statements. For example, when we observe the patch notes for bug QPID-2370 [3], we find that developers modify the existing logging statement to capture the newly defined token. The reason may be that developers change the name of the existing variable to a more meaningful name. For example, in bug MAPREDUCE-2264 [4], we find that developers rename variables and modify the logging statements accordingly.

From Table 5, we observe that the developers add variables into logging statements statistically significantly more in bug-fixing commits than non-bug-fixing commits in *HBase* (medium effect size) and *Qpid* (large effect size). This suggests that the existing variables may not have all the needed information for fixing bugs. Developers have to add new variables into logging statements during bug-fixes. We also find that developers do not delete variables in logging statements. Deleting variables in logging statements may change the format of the logging statements. There may be log processing tools that rely on these logging statements. Deleting variables may impact the correctness of these log processing applications [6]. Therefore, developer may be award of this and try to avoid deleting variables from logging statements.

Developers modify logged text more during bug fixes. We find that text modification is statistically significantly more in bug-fixing commits than non-bug-fixing commits with non-trivial effect sizes (see Table 4). In some cases, the text description in logging statements is not clear and developers need improve the text to help fix bugs. For example, in *HBase* HBASE-6665 [5] developers modify the logging statement to provide more information region splits. Prior research shows that there is a challenge to understand logging

[3] https://issues.apache.org/jira/browse/QPID-2370
[4] https://issues.apache.org/jira/browse/MAPREDUCE-2264
[5] https://issues.apache.org/jira/browse/HBASE-6665

statements in practice [21]. Our results show that developers may have faced such challenges and may need to improve the text in logging statements for better bug-fixing.

Logging statement relocation occurs more in bug fixes. Table 3, shows that there are a large number of logging changes that only relocate logging statements. Table 4 shows that such relocation of logging statements is statistically significantly more in bug-fixing commits than non-bug-fixing commits. We manually examine such commits and find that developers often forget to leverage exception handling or using proper condition statements in the code. After fixing the bugs, developers often move existing logging statements into the *try/catch* blocks or after condition statements. For example, in the YARN-289 [6] of Hadoop, logging statements are placed into the proper *if-else* block.

Logging levels are not modified often during bug fixes. We find that logging level changes are statistically indistinguishable between bug-fixing and non-bug-fixing commits in all subject systems. The reason may be that developers are able to enable all the logging statements during bug-fixing, despite of what level a logging statement has. In addition, prior research shows that developers do not have a good knowledge about how to choose a correct logging level [1].

Developers change logging statements statistically significantly more in bug-fixing commits than non-bug-fixing commits. In particular, developers modify logging statements to add or change the variables in logging statements during bug fixes. Such results show that developers often realize the needed information to be logged as after-thoughts and change the variables in logging statement to assist in fixing bugs.

RQ2: Are bugs fixed faster with logging statement changes?

Motivation

In RQ1, we find that logging statements are changed more frequently in bug fixes. However, there is no study to show if logging statements are useful in debugging process. As a first step of exploring the usefulness of logging statement changes during bug fixes, we try to find out whether bug fixes with logging statement changes are fixed faster than bug fixes without logging statement changes.

Approach

To find out whether bugs are fixed faster with logging statement changes, we collect all JIRA issues with type 'bug' from the three subject systems. We

[6] https://issues.apache.org/jira/browse/YARN-289

Table 6: Comparing code and developer metrics between the bug-fixing commits with log changes and bug-fixing commits without log changes. The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size means that bug-fixing commits with log changes have larger metric values and p-values are bold if they are smaller than 0.05

Metrics	Hadoop		HBase		Qpid	
	p-values	Effect size	p-values	Effect Size	p-values	Effect size
Code churn	**<2.2e-16**	0.178 (small)	**<2.2e-16**	0.023	**<2.2e-16**	0.155
Resolution time	**4.7e-14**	-0.095	**<2.2e-16**	-0.188 (small)	**7.7e-08**	-0.276 (small)
# of comments	**2.2e-16**	-0.573 (small)	**<2.2e-16**	-0.436 (small)	**<2.2e-16**	-0.304 (small)
# of developers	**<2.2e-16**	-0.539 (small)	**<2.2e-16**	-0.617 (medium)	**<2.2e-16**	-0.440 (small)

obtained the code commits for each of these JIRA issues by searching for the issue id from the commit messages. We identify the logging statement changes, and the code churn for fixing each issue. We then split the JIRA issues into (1) bugs that are fixed with logging statement changes and (2) bugs that are fixed without logging statement changes. We use the code churn to measure the complexity of the issue. We then extracted three metrics from JIRA issues to measure the effort of fixing a bug:

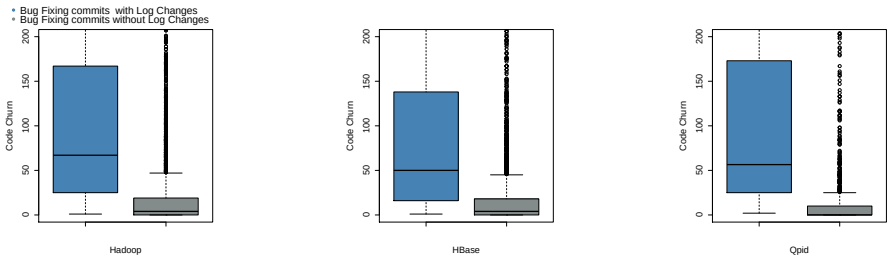
1. **Resolution time:** This metric measures how fast the bug is fixed. This metric is defined as the time taken from when the bug is opened until it is resolved. For example, if a bug was opened on 1^{st} February 2015 and closed on 5^{th} February 2015, the resolution time of the bug is four days.
2. **# of comments:** This metric measures how much discussion is needed to fix a bug. Intuitively, the more discussion in the issue report, the more effort is spent on fixing the bug. We count the total number of comments in the discussion of each issue report.
3. **# of developers:** This metric measures how many developers who participate in the discussion of fixing the bug. Intuitively, more developers who discuss the bug, more effort is spent on fixing the bug. We count the number of unique developers who comment on the issue report. We use the user names in the JIRA discussion to identify the developers.

We first compare the code churn of bug fixes with and without logging statement changes. Similar to RQ1, we use *MannWhitney U test* to study whether the difference is statistically significant and we use *Cohen's d* to measure the size of the difference between code churn of bug fixes with and without logging statement changes. Intuitively, fixing a more complex bug requires longer time, more people and more discussion. Therefore, we use code churn to normalize the resolution time, the number of comments, and the number of developers when compare them.

Results

We find that the logging statements are used to fix more complex bugs. We find that the average code churn for fixing bugs is significantly higher with logging statement changes than without logging statement changes (see Table 6 and Figure 2). This implies that developers may change logging statements to fix more complex bugs.

Fig. 2: Boxplot of code churn of bug fixing commits with logging statement change (shown in blue) against bug fixing commits without logging statement change (shown in grey).



We find that bugs that are fixed with logging statement changes, take shorter time with fewer comments and fewer people. After normalizing the code churn, we find that the resolution time, the number of comments and the number of developers are all statistically significantly smaller in the bug fixes with logging statement changes than the ones without logging statement changes. This result suggests that given two bugs of same complexity, the one with logging statement changes usually take less time to get resolved and needs fewer number of developers involved with fewer discussions. Logging statements may provide useful information to assist developers in discussing, diagnosing and fixing bugs. For example, when fixing bug HBASE-3074 [7], developers left the first comment to provide additional details in the logging statement about where the failure occurs. In the source code, developers add the name of the servers into the the logging statements. Such additional data helps diagnose the cause of the failure and helps fix the bug.

Logging statements are changed during fixing more complex bugs. After normalizing the complexity of bugs using code churn, we find that bug fixes with logging statement changes are resolved faster with fewer people and fewer discussions.

[7] https://issues.apache.org/jira/browse/HBASE-30741

RQ3: Can metrics from logging statement changes help in explaining the resolution time of bugs?

Motivation

In RQ2, we find that bugs that are fixed with logging statement changes take shorter time to get resolved than bugs fixed without logging statement changes. Prior research has shown that resolution time is correlated to the number of developers and the number of comments in an issue report [22]. We want to see whether the metrics from logging statement changes (as shown in RQ1) can complement the number of developers and the number of comments in modelling the resolution time of bugs.

Approach

To better understand the usefulness of logging statement changes on the resolution time for fixing bugs, we build a non-linear regression model for the resolution time of bugs. A non-linear regression model fits the curve of the form $y = \alpha + \beta_1 x_1 + \beta_2 x_x + .. + \beta_n x_n$ to the data, where y is the dependent variable (i.e., resolution time of bugs) and every x_i is an explanatory metrics. The explanatory metrics include the metrics from logging statement changes (as shown in RQ1). Since prior research finds that the number of developers and the number of comments in an issue report are correlated to the resolution time of the issue report, we include the number of developers and the number of comments as explanatory metrics. We find that bugs with more complex fixes may take longer time to resolve (see RQ2). Therefore, we also include code churn as explanatory metrics. We use the *rms* package [23] from R, to build the non-linear regression model. The overview of the modeling process is shown in Figure 3 and is explained below.

(C-1) Calculating the degrees of freedom

During predictive modeling, a major concern is over-fitting. An over-fit model is biased towards the dataset from which it is built and will not well fit other datasets. In non-linear regression models, over-fitting may creep in when a explanatory metrics is assigned more degrees of freedom than the data can support. Hence, it is necessary to calculate a budget of degrees of freedom that a dataset can support before fitting a model. We budget $\frac{x}{15}$ degrees of freedom for our model as suggested by prior research [24]. Here x is the number of rows (i.e, # bugs) in each project.

(C-2) Correlation and redundancy analysis

Correlation analysis is necessary to remove the highly correlated metrics from our dataset. We use Spearman rank correlation to assess the correlation between the metrics in our dataset. We use Spearman rank instead of Pearson correlation because Spearman rank correlation is resilient to data that is

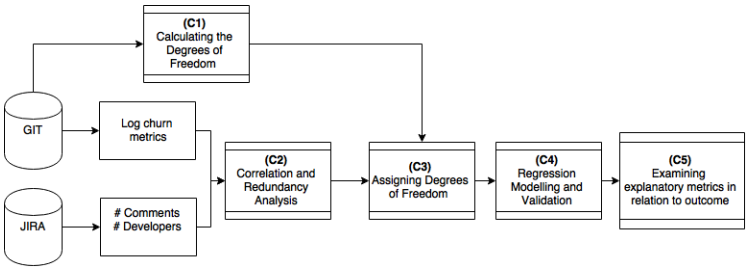


Fig. 3: Overview of our non-linear OLS model construction(C) for the resolution time of bugs

not normally distributed. We use the function *varclus* in R to perform the correlation analysis. From the hierarchical overview of explanatory metrics constructed by the *varclus* function, we exclude one metric from the sub-hierarchies which have correlation $|\rho| > 0.7$.

Correlation analysis does not indicate redundant metrics, i.e, metrics that can be explained by other explanatory metrics. The redundant metrics interfere with the one another and the relation between the explanatory and dependent metrics is distorted. We perform redundancy analysis to remove such metrics. We use the function *redun* provided in *rms* package to perform the redundancy analysis.

(C-3)Assigning degrees of freedom

After removing the correlated and redundant metrics from our datasets, we spend the budgeted degrees of freedom efficiently. We identify the metrics which can use the benefit from the additional degrees of freedom (knots) in our models. To identify these metrics we use the Spearman multiple ρ^2 between the explanatory and dependent metrics. A strong relation between explanatory metrics x_i and the dependent metric y indicates that, x_i will benefit from the additional knots and improve the model. We use *spearman2* function in the *rms* package to calculate the Spearman multiple ρ^2 values for our metrics.

(C-4)Regression modeling and validation

After budgeting degrees of freedom to our metrics we build a non-linear regression model using OLS (Ordinary Least Squares) command that is provided by *rms* package. We use the *restricted cubic splines* to assign the knots to the explanatory metrics in our model. As we are trying to identify the relationship between metrics from logging statement changes and the resolution time of bug fixes, we are primarily concerned if the metrics from logging statement changes are significant in our models. Therefore, we use the backward (step-down) metric selection method [25] to determine the statistically significant metrics that are included in our final models. We choose the backward

selection method since prior research shows that backward selection method usually performs better than the forward selection approach [26]. The backward selection process starts with using all the metrics as predictors in the model. At each step, we remove the metrics that is the least significant in the model. This process continues until all the remaining metrics are significant. We use the fastbw (Fast Backward Variable Selection) function provided by the R package rms [23] to perform the backward metric selection process.

(C-5) Examining explanatory metrics in relation to outcome

After identifying the significant metrics in our datasets we find the relation between each explanatory metric and the resolution time of bugs. In our regression models, each explanatory metric can be explained by several model terms. To account for the impact of all model terms associated with an explanatory metric, we plot the changes to resolution time against each metric, while holding the other metrics at their median value using the *Predict* function in the *rms* package [23]. The plot follows the relationship as it changes directions at knot locations(C-3).

However, to quantify the effects of the significant metrics on resolution time we adopt method suggested in prior research [27,28]. We first set all the significant metrics to their means and then increase one metric by one standard deviation. We use the *predict* function to calculate the variable Y_2. The difference $\Delta Y = Y_2 - Y$ describes the effect of each metrics from logging statement changes on the resolution time of bug fixing commits with logging statement changes. A positive effect means a higher value of the metrics from logging statement changes increases the resolution time of the bug, whereas a negative effect decreases the resolution time of the bug.

We would like to point out that although non-linear regression models can be used to build accurate models for the resolution time of bugs, our purpose of using the non-linear regression models in this paper is not for predicting the resolution time of bugs. Our purpose is to study the explanatory power of metrics from logging statement changes and explore their empirical relationship to the resolution time of bugs.

Results

In this subsection, we describe the outcome of the model construction and analysis outlined in our approach and Figure 3

(C-1) Calculating degrees of freedom. Our data can support between 123 ($\frac{1,925}{15}$ in Hadoop), 63($\frac{953}{15}$ in Qpid) and 183($\frac{2,755}{15}$ in HBase) degrees of freedom. As we have higher number of knots we can be liberal in their allocation during model construction.

(C-2) Correlation and redundancy analysis Figure 4 shows the hierarchically clustered Spearman ρ values of the three systems. The grey dashed

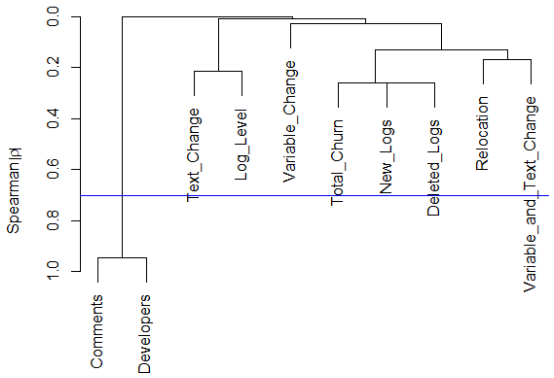


Fig. 4: Correlation between metrics in Qpid. Dashed line indicates cut of set to 0.7

line indicates our cut-off value ($|\rho| = 0.7$). Our analysis reveals that # *Comments* and # *developers* are highly correlated in *Qpid* and *HBase*. We chose to remove *#developers* from our model since *#comments* is a simpler metric than *#developers*. We find that there are no redundant metrics in our metrics in all the subject systems.

(C-3)Assigning degrees of freedom Figure 5 shows the Spearman multiple ρ^2 of the resolution time against each explanatory metric. Metrics that have higher Spearman multiple ρ^2 have higher chance of benefiting from the additional degrees of freedom to better explain resolution time. Based on Figure 5, we split the explanatory metrics into three groups. The first group consists of *#comments*, the second group consists of *#log level changes*, *#log variable changes*, *#logging statement relocation* and *#new logging statements*. The last group consists the remaining metrics. We allocate five degrees of freedom i.e, knots, to the metrics in the first group, three to metrics in second group and no knots to metrics in last group similar to prior research [29].

(C-4)Regression modeling and validation. After allocating the knots to the explanatory metrics, we build the non-linear regression model and use the *validate* function in the *rms* package to find the significant metrics in our subject systems. We find that metrics from logging statement changes are significant in Qpid and HBase systems for predicting resolution time of bug fixes.

(C-5) Examining explanatory metrics in relation to outcome Figure 6 shows the direction of impact of metrics from logging statement changes on the resolution of bug fixing commits with logging statement changes in

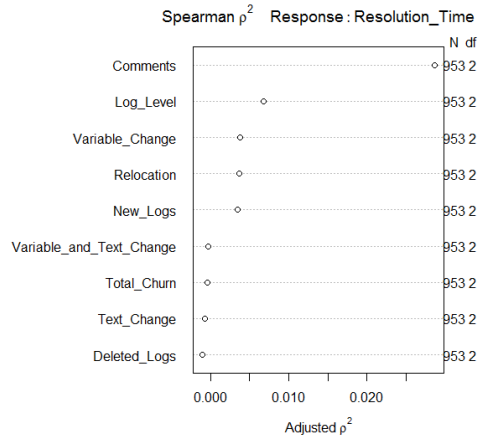


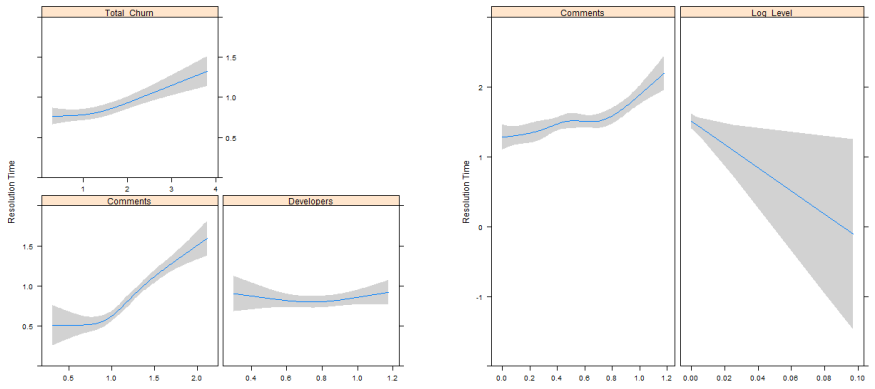
Fig. 5: Spearman multiple ρ^2 of each explanatory metric against Resolution Time of bug fixing commits with logging statement changes. Larger values indicate more potential for non-linear relationship

HBase. We find that log modifications have a negative impact on the resolution time of bug fixes. Shown in Table 7, we find that in *HBase* and *Qpid*, modifications to logging statements, i.e, Log level changes and variable changes are significant in the models and have negative correlations with the resolution time of bugs.

We find that *log level changes* are statistically significant in *Qpid* and *HBase*. Even though developers often do not change log levels during bug fixes as seen in RQ1, our model shows that changes to log levels can help in faster resolution of bugs. This may be because, prior research has shown that developers are confused when estimating the cost and benefit of each verbosity level in a logging statement [1]. This suggests that developers should focus more in picking the right verbosity of logging statement. For example, in HADOOP-9593 [8] the logging statement is set to 'error' and causes confusion to system administrators. On the other hand, in HDFS-1955 [9] we see that logging statements are set to default 'info' and have to be updated to 'error' and 'warn' to remove unnecessary logs being generated.

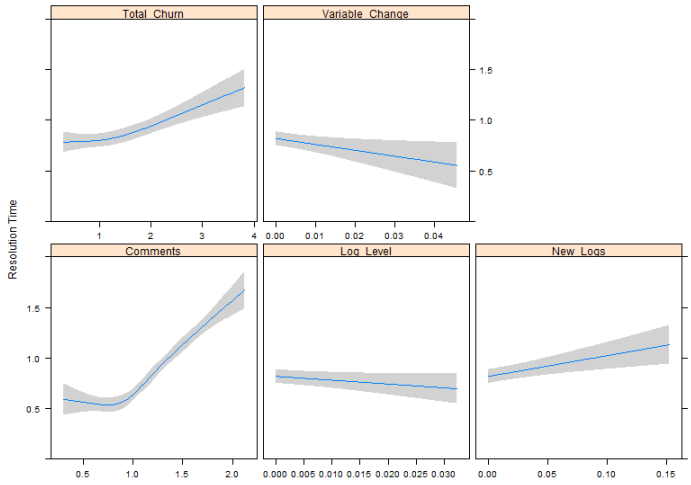
We find that *variable changes* is significant in the model for *HBase* with negative effect on the resolution time of bugs. This suggests that changing logged variables may help in the faster resolution of bugs. We find in RQ1 that developers change variables in the logging statements more in bug-fixing commits than non-bug-fixing commits. We find that such changes to variables may assist in fixing bugs faster in practice.

[8] https://issues.apache.org/jira/browse/HADOOP-9593
[9] https://issues.apache.org/jira/browse/HBASE-1955



a: Hadoop

b: Qpid



c: HBase

Fig. 6: Relation between the explanatory metrics and resolution time of bug fixing commits with log changes. Increasing graph shows increase in explanatory metrics increases the resolution time and decrease reduces the resolution time

We find that New logging statements have a positive impact on the resolution time of bug fixes in *HBase* project. We find that the average code churn of bug-fixing commits with new logging statements is almost twice that off average code churn of all commits. We also find that the average resolution time is higher for bug-fixing commits with new logging statements. These results suggest that during very complex bug fixes, developers might not know the exact cause of the bug and have to add large amount of extra code and new

Table 7: Effect of metrics from logging statement changes on resolution time of bugs. Effect is measured by adding one standard deviation to its mean value, while the other metrics are kept at their mean values.

Delta Y	Variable	Delta Y	Variable	Delta Y	Variable
Qpid		**Hadoop**		**HBase**	
0.0996	# Comments	0.2741	# Comments	0.3562	# Comments
-0.0691	Log Level Changes	0.0258	Total Churn	0.1157	Total Churn
		0.0191	Developers	0.0517	New Logs
				-0.0281	Log Level Changes
				-0.0367	Variable Changes

logging statements to ensure the functionality of the added code. For example to fix HBASE-7305 [10] , which is resolved almost a full year after being open, we observe that developers add over 120 new logging statements and many new feature into the code. Because of the addition of new feature into the code, this takes longer time to resolve than bugs with only log modifications.

Metrics from logging statement changes can complement the number of comments, the number of developers and bug-fixing code churn in modelling the resolution time of bugs. We find that logging modifications have a negative effect on resolution time of bug fixes and help in increasing the resolution time of bugs. Such results imply that there is a relationship between changing logging statements and the resolution time of bugs.

4 Discussion of results

To further understand how developers change logging statements in bug fixes, we finally conduct a qualitative analysis. We collected all the bug fixing commits with logging statement changes for our subject systems. We selected a 5% random sample (266 for *HBase*, 268 for *Hadoop* and 83 for *Qpid*) from all the commits. For the sampled commits, we analyze the code changes made in Git and the corresponding JIRA issue reports to find different patterns of log use in bug fixes. We follow an iterative process, similar to prior research [15], until we cannot find any new types of patterns. We find three reasons of changing logging statements during bug fix as shown in Table 8. Each logging statement changes may have more than one reasons.

– *Bug diagnosis*

Developers use logs to detect runtime defects. Developers change logging statement to print extra or different information into logs during the ex-

[10] https://issues.apache.org/jira/browse/HBASE-7305

Table 8: Log change reasons during bug fix

Projects	Hadoop	HBase	Qpid
Bug diagnosis	157	175	49
Future bugs prevention	156	170	42
Bug-fixing code quality assurance	93	78	18

ecution of the system. For example, to fix HADOOP-2725 [11] we observe that developers notice a discrepancy when a 100TB file is copied across two clusters. To help in debugging we observe that developers modify the log variable which outputs the sizes of the files into human readable format instead of bytes. These logging statement changes are committed along with bug fix, as it clarifies the logging statement and helps in easy understanding.

– *Future bugs prevention*

After fixing a bug, developers may insert logging statement into the code. Such logging statements monitor the execution of the system to track for similar bugs in the future. For example in HADOOP-2890 [12] we see that developers leverage logs to identify the reason behind blocks getting corrupted. In the commit, we observe that the developers fix this bug by adding new checks to verify where the block gets corrupted and they add *try catch* block with new logging statements to catch these exceptions. The logging statement would notify developers with useful information to diagnose the bug if a similar bug appears.

– *Bug-fixing code quality assurance*

Sometime, developers need to introduce a large amounts of code to fix a bug. However, developers may introduce extra bugs with these bug-fixing code. To ensure the quality of these bug-fixing code, developers insert logging statements into the bug-fixing code. For example in HBASE-3787 [13], where developers encounter a non-idempotent increment which causes an error in the application. This fix involves over 13 developers and 112 discussions over the two years. The developers add several new files and functions during the bug fix and new logging statements to assure the code quality of the fix.

Our manual analysis results confirms that developers often change logging statements during bug fixes.

[11] https://issues.apache.org/jira/browse/HADOOP-2725
[12] https://issues.apache.org/jira/browse/HADOOP-2890
[13] https://issues.apache.org/jira/browse/3787

5 Related Work

In this section, we present the prior research that performs log analysis on large software systems and empirical studies on logging statements.

5.1 Log Analysis

Prior work leverage logging statements for testing and detecting anomalies in large scale systems. Shang *et al.* [30] propose an approach to leverage logging statements in verifying the deployment of Big Data Analytic applications. Their approach analyzes logging statements in order to find differences between running in a small testing environment and a large field environment. Lou *et al.* [11] propose an approach to use the variable values printed in logging statements to detect anomalies in large systems. Based on the variable values in logging statements, their approach creates invariants (e.g., equations). Any new logging statements that violates the invariant are considered to be a sign of anomalies. Fu *et al.* [31] built a Finite State Automaton (FSA) using unstructured logging statements and to detect performance bugs in distributed systems.

Xu *et al.* [3] link logs to logging statements in source code to recover the text and and the variable parts of logging statements. They applied Principal Component Analysis (PCA) to detect system anomalies. Tan *et al.* [32] propose a tool named SALSA, which constructs state-machines from logs. The state-machines are further used to detect anomalies in distributed computing platforms. Jiang *et al.* [33] study the leverage of logs in troubleshooting issues from storage systems. They find that logs assist in a faster resolution of issues in storage systems. Beschastnikh *et al.* [34] designed automated tools that infers execution models from logs. These models can be used by developers to understand the behaviours of concurrent systems. Moreover, the models also assist in verifying the correctness of the system and fixing bugs.

To assist in fixing bugs using logs, Yuan *et al.* [35] propose an approach to automatically infer the failure scenarios when a log is printed during a failed run of a system.

Jiang *et al.* [36–39] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [36]. Based on the such events, they identified both functional anomalies [37] and performance degradations [38] in load test results. In addition, they proposed an approach that leverage logs to reduce the load test that are performed in user environment [39].

The extensive prior research of log analysis motivate our paper to study how logging statements are leveraged during bug fixes. As a first step, we study the changes to logging statement during bug fixes. Our findings show that logging statements are change more during bug fixes than other types of code changes. The changes to logging statements have a relationship with a faster resolution of bugs with fewer people and less discussion.

5.2 Empirical studies on logging statements

Prior research performs an empirical study on the characteristics of logging statements. Yuan *et al.* [1] studies the logging characteristics in four open source systems. They find that over 33% of all logging statement changes are after thoughts and logging statements are changed 1.8 times more than entire code. Fu *et al.* [40] performed an empirical study on where developer put logging statements. They find that logging statements are used for assertion checks, return value checks, exceptions, logic-branching and observing key points. The results of the analysis were evaluated by professionals from the industry and F-score of over 95% was achieved.

Shang *et al.* [41] signify the fact that there is gap between operators and developers of software systems, especially in the leverage of logs. They performed an empirical study on the evolution both static logging statements and logs outputted during run time [10, 42]. They find that logging statements are co-evolving with the software systems. However, logging statements are often modified by developers without considering the needs of operators. Furthermore, Shang *et al.* [21] find that understanding logs is challenging. They examine user mailing lists from three large open-source projects and find that users of these systems have various issues in understanding logs outputted by the system. Shang *et al.* propose to leverage different types of development knowledge, such as issue reports, to assist in understanding logs.

Prior research by Yuan *et al.* [43] shows that logs need to be improved by providing additional information. Their tool named *Log Enhancer* can automatically provide additional control and data flow parameters into the logging statements thereby improving the logs. Log Advisor is another tool by Zhu *et al.* [44] which helps in logging by learning where developers log through existing logging instances.

The most related prior research by Shang *et al.* [10] empirically study the relationship of logging practice and code quality. Their manual analysis sheds light on the fact that some logging statements are changed due to field debugging. They also show that there is a strong relationship between logging practice and code quality. Our paper focused on understanding how logs are changed during bug fixes. Our results show that logging statements are leveraged extensively during bug fixes and may assist in the resolution of bugs.

6 Limitations and Threats to Validity

In this section, we present the threats to the validity to our findings.

External Validity

Our case study is performed *Hadoop, HBase* and *Qpid*. Even though these three subject systems have years of history and large user bases, the three

subject systems are all Java based platform software. Systems in other domain may not rely on logging statements in bug fixes. More case studies on other software in other domains with other programming languages are needed to see whether our findings can be generalized.

Internal Validity

Our study is based on the data obtained from Git and JIRA for all the subject systems. The quality of the data contained in the repositories can impact the internal validity of our study.

Our analysis of the relationship between changes to logging statements and bug resolution time cannot claim causal effects, as we are investigating correlations, rather than conducting impact studies. The explanative power of metrics from logging statement changes on the resolution time of bugs does not indicate that logs cause faster resolution of bugs. Instead, it indicates the possibility of a relation that should be studied in depth through user studies.

Construct Validity

The heuristics to extract logging source code may not be able to extract every logging statement in the source code. Even though the subject systems leverage logging libraries to generate logs at runtime, there still exist user-defined logging statements. By manually examining the source code, we believe that we extract most of the logging statements. Evaluation on the coverage of our extracted logging statements can address this threat.

We use keywords to identify bug fixing commits when the JIRA issue ID is not included in the commit messages. Although such keywords are used extensively in prior research [10], we may still miss identify bug fixing commits or branching and merging commits.

We use Levenshtein ratio and choose a threshold to identify modifications to logging statements. However, such threshold may not accurately identify modifications to logging statements. Further sensitivity analysis on such threshold is needed to better understand the impact of the threshold to our findings.

We build non-liner regression models using metrics from logging statement changes, to model the resolution time of bugs. However, the resolution time of bugs can be correlated to many factors other than just logs, such as the complexity of code fixes. To reduce such a possibility, we normalize the metrics from logging statement changes by code churn. However, other factors may also have an impact on the resolution time of bugs. Furthermore, as this is the first exploration (to our best knowledge) in modeling resolution time of bugs using metrics from logging statement changes, we are only interested in understanding the correlation between the two. Future studies should build more complex models, that consider other factors to study if there is any causation.

Source code from different components of a system may have various characteristics. The importance of logging statements in bug fixes may vary in different components of the subject systems. More empirical studies on the use of logging statements in fixing bugs for different components of the systems are needed.

7 Conclusion and Future Work

Logs are used by developers for fixing bugs. This paper is a first attempt (to our best knowledge) to understand whether logging statements are changed more during bug fixes and how these changes occur. The highlights of our findings are:

- We find that logging statements are changed more during bug fixing commits than non-bug-fixing commits. In particular, we find that logging statements are modified more frequently during bug fixes. Variables and textual information in the logging statements are more frequently modified during bug fixes.
- We find that logging statements are changed more during complex bug fixes. However, bug fixes that change logging statements are fixed faster, need fewer developers and have less discussion.
- We find that metrics from logging statement changes can complement the traditional metrics such as the number of comments and the number of developers in modeling the resolution time of bugs.

Our findings show that logging statements are changed by developers in bug fixes and there is a relationship between changing logging statements and the resolution time of bugs. We find that developers modify the text or variables in logging statements frequently as after-thoughts during bug fixes. This suggests that software developers should allocate more effort for considering the text, the printed variables in the logging statements when developers first add logging statements to the source code. Hence, bugs can be fixed faster without the necessity to change logging statements during the fix of bugs.

References

1. D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 102–112.
2. W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *ICDM '09: Proceedings of the 9th IEEE International Conference on Data Mining*, pp. 588–597.
3. W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *ACM SIGOPS '09: Proceedings of the 22nd Symposium on Operating Systems Principles*, New York, NY, USA, pp. 117–132.
4. M. Syer, Z. M. Jiang, M. Nagappan, A. Hassan, M. Nasser, and P. Flora, "Leveraging performance counters and execution logs to diagnose memory-related performance issues," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 110–119.

5. L. Bitincka, A. Ganapathi, S. Sorkin, and S. Zhang, "Optimizing data analysis with a semi-structured time series database," in *SLAML'10: Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*. USENIX Association, pp. 7–7.
6. W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 3–26, 2014.
7. M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "Continuous validation of load test suites," in *Proceedings of the International Conference on Performance Engineering*, Mar 2014, pp. 259–270.
8. Xpolog. [Online]. Available: http://www.xpolog.com/.
9. logstash, "http://logstash.net."
10. W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, vol. 20, no. 1, pp. 1–27, 2015.
11. J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection." in *USENIX Annual Technical Conference*, 2010.
12. R. Ihaka and R. Gentleman, "R: a language for data analysis and graphics," *Journal of computational and graphical statistics*, vol. 5, no. 3, pp. 299–314, 1996.
13. V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, 1966.
14. M. Mednis and M. K. Aurich, "Application of string similarity ratio and edit distance in automatic metabolite reconciliation comparing reconstructions and models," *Biosystems and Information technology*, vol. 1, no. 1, pp. 14–18, 2012.
15. C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *Software Engineering, IEEE Transactions on*, vol. 25, no. 4, pp. 557–572, 1999.
16. E. A. Gehan, "A generalized wilcoxon test for comparing arbitrarily singly-censored samples," *Biometrika*, vol. 52, no. 1-2, pp. 203–223, 1965.
17. V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11, pp. 1073–1086, 2007.
18. B. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, J. Rosenberg *et al.*, "Preliminary guidelines for empirical research in software engineering," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 721–734, 2002.
19. W. R. Shadish and C. K. Haddock, "Combining estimates of effect size," *The handbook of research synthesis and meta-analysis*, vol. 2, pp. 257–277, 2009.
20. V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11-12, pp. 1073–1086, Nov 2007.
21. W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in *ICSME '14: Proceedings of the International Conference on Software Maintenance and Evolution,*. IEEE, 2014, pp. 21–30.
22. P. Anbalagan and M. Vouk, "On predicting the time taken to correct bug reports in open source projects," in *ICSM 2009: Proceedings of IEEE International Conference on the Software Maintenance,*. IEEE, 2009, pp. 523–526.
23. F. E. Harrell, *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. Springer Science & Business Media, 2013.
24. F. E. Harrell, K. L. Lee, R. M. Califf, D. B. Pryor, and R. A. Rosati, "Regression modelling strategies for improved prognostic prediction," *Statistics in medicine*, vol. 3, no. 2, pp. 143–152, 1984.
25. J. F. Lawless and K. Singhal, "Efficient screening of nonnormal regression models," *Biometrics*, vol. 34, no. 2, pp. pp. 318–327, 1978. [Online]. Available: http://www.jstor.org/stable/2530022
26. N. Mantel, "Why stepdown procedures in variable selection," *Technometrics*, vol. 12, no. 3, pp. pp. 621–625, 1970. [Online]. Available: http://www.jstor.org/stable/1267207
27. E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *ACM SIGSOFT '11 : Proceedings of the 19th*

symposium and the 13th European conference on Foundations of software engineering. ACM, 2011, pp. 300–310.
28. A. Mockus, "Organizational volatility and its effects on software defects," in *ACM SIGSOFT '10 : Proceedings of the 18th international symposium on Foundations of software engineering*. ACM, 2010, pp. 117–126.
29. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, p. To appear, 2015.
30. W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *ICSE'13: Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 402–411.
31. Q. F. J. L. Y. Wang and J. Li., "Execution anomaly detection in distributed systems through unstructured log analysis." in *ICDM 09: In Proceedings of 9th IEEE International Conference on Data Mining*.
32. J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Salsa: Analyzing logs as state machines," in *WASL'08: Proceedings of the 1st USENIX Conference on Analysis of System Logs*. USENIX Association, 2008, pp. 6–6.
33. W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou, "Understanding customer problem troubleshooting from storage system logs," in *FAST '09: Proceedings of the 7th Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2009, pp. 43–56.
34. I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 267–277.
35. D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *ASPLOS '10: Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2010, pp. 143–154.
36. Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *Journal of Software Maintenance Evolution*, vol. 20, no. 4, pp. 249–267, Jul. 2008.
37. Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *ICSM '08: Proceedings of theIEEE International Conference on Software Maintenance*, Sept 2008, pp. 307–316.
38. Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *ICSM '09: Proceedings of the IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 125–134.
39. Z. M. Jiang, A. Avritzer, E. Shihab, A. E. Hassan, and P. Flora, "An industrial case study on speeding up user acceptance testing by mining execution logs," in *SSIRI '10: Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement*. Washington, DC, USA: IEEE Computer Society, pp. 131–140, 2010.
40. Q. Fu, J. Zhu, W. Hu, J.-G. L. R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *ICSE Companion '14: Proceedings of the 36th International Conference on Software Engineering,*, pp. Pages 24–33.
41. W. Shang, "Bridging the divide between software developers and operators using logs," in *ICSE '12 :Proceedings of the 34th International Conference on Software Engineering*.
42. W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 3–26, 2014.
43. D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems*, vol. 30, pp. 4:1–4:28, Feb. 2012.

44. J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proc. of ACM/IEEE ICSE*, 2015.