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

An Empirical Study on Changes to Logs 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 logs are changed during field debugging. However, little is known about how logs 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 during bug fixes than other code changes. Furthermore, we identify four different types of modifications that developers make to logs during bug fixes, including: (1)modification to logging level, (2)modification to logging text, (3)modification of logged variable and (4)relocation of log. We find that bug fixes that contain changes to logs have larger code churn, but involve fewer developers, require less time and have less discussion during the bug fix. This suggests that given two bugs of similar complexity, the one which leverages logs and has log changes, has likelihood of being resolved faster. We build a regression model to explore the relationship between log churn metrics and the resolution time of bugs. We find that log churn metrics 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 logs and the resolution time of bugs. This

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

Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec
E-mail: shang@encs.concordia.ca

means that bug fixes with log modifications have higher likelihood of being resolved faster.

1 Introduction

Platform software provides an infrastructure to support a variety of applications. To monitor the execution of these applications running on top of the platform software, logs are widely leveraged. Such logs are generated through simple *printf* statements or logging statement in the source code. Logging statements contain a static textual part that provides information about the event, a variable part that contains context of the events, and a logging verbosity level that indicates when the log should be output. An example of a logging statement is shown below where *info* is the logging verbosity 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 [40]. Logs are leveraged for anomaly detection [37,38,32], system monitoring [4], capacity planning [27] and large-scale system testing [33]. The valuable information in logs has created a new market for log analysis platforms such as Splunk [4], XpoLog [36], and Logstash [20], which assist developers in analyzing the 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 due to a system failure in the field. Developers leveraged logs to identify the point of failure. After fixing the bug, the logs were updated to prevent similar bugs from re-occurring. A recent study shows that changes to logs have a strong relationship with code quality and that 16-32% of log changes are made during field debugging [30]. However, there exists no large scale study that investigates how logs are changed during bug fixes. Such knowledge may provide guidance for practitioners who often use or change logs. Moreover, the log analysis platform may be improved to better serve the use of logs in practice.

In this paper, we perform an empirical study on the log changes during bug fixes in three open source platform software, i.e., *Hadoop*, *HBase* and *Qpid*. We consider adding, deleting or modifying a logging statements as log changes.[2] In particular, we sought to answer the following research questions.

RQ1: Are logs changed more often during bug fixes?
We find that logs are changed more often during bug fixes than non-bug-fixing code changes. In particular, we find that adding and modifying logs

[1] https://issues.apache.org/jira/browse/HBASE-3403
[2] In the rest of this paper, we use 'log' to refer to the logging statements in the source code.

occurs more often in bug fixes than non-bug-fixing code changes (statistically significant with non-trivial effect size). We identified four types of modifications to logs, including *modification to logging level*, *modification to logging text*, *modification of logged variable* and *relocation of log*. We find that *modification to logging text*, *modification of logged variable* and *relocation of log* occur more often in bug fixes than non-bug-fixing code changes (statistically significant with medium to high effect sizes).

RQ2: Is there a relationship between log changes and effort spent on bug fixes?
We find that bug fixes with log churn have higher total code churn than bug fixes without log churn. After normalizing the code churn, bug fixes with log churn take less time to get resolved, involve fewer developers and have less discussions during the bug fixing process. This suggests that given two metrics of similar complexity the one with log churn is more likely to be resolved faster and involve fewer developers and less discussion. *[Ian says: TODO]*

RQ3: Can log churn metrics help in explaining the resolution time of bugs?
Using log churn metrics (e.g., new logs added, deleted logs) and traditional metrics (i.e., # comments and # developers) we trained regression models for the resolution time of bug fixes. We find that modifications to logged variables and modifications to log level are statistically significant in the models and have negative impact on resolution time. This suggests that there is a relationship between these metrics and resolution time of bugs. The relationship shows the impact of log churn on resolution time of bugs which should be further studied.

The rest of this paper is organized as follows. Section 2 presents our methodology of extracting data for our study. Section 3 presents our case study and the answers to our three research questions. Section 4 describes the prior research that is related to our work. Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper.

2 Methodology

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

The aim of this paper is to understand how logs are changed during bug fixes and the relationship between bug resolution time and log churn. 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 software.

Hadoop[1]: *Hadoop* is an open source platform for distributed storage and processing of big data on computer clusters. *Hadoop* uses the MapReduce

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

Table 1: An overview of the platform software

Projects	Hadoop		HBase		Qpid	
	Bug fixing	Non-bug fixing	Bug fixing	Non-bug fixing	Bug fixing	Non-bug fixing
Total # of commits	1,808 (49.9 %)	1,809 (50.1 %)	1924 (56.8 %)	1463 (43.2 %)	953 (52.1 %)	875 (47.9 %)
Code churn (LOC)	246 K	1.8 M	653 K	1.5 M	106 k	597 K
Log churn (LOC)	3,536	16,980	4,672	10,335	972	4,953
% of Commits with log churn	24.0 % (433)	46.2 % (656)	36.2 % (648)	42.1 % (616)	22.1 % (211)	32.8 % (287)

data-processing paradigm. The logging characteristics of *Hadoop* have been extensively studied in prior research [21, 30, 38]. We study the changes to logs from *Hadoop* releases 0.16.0 to 2.0.

HBase[2]: *HBase* is a distributed, scalable, big data software, which uses *Hadoop* file-systems. We study the changes to logs in *HBase* from release 0.10 to 0.98.2.RC0. 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 studied system to extract all commits, and identify bug-fixing commits (bug fixes) non-bug-fixing commits (non-bug fixes). (2) We identify log changes in both bug fixes and non-bug fixes. (3) We categorize the log changes into three types, *'New logs'*, *'Modified Logs'* and *'Deleted logs'*. (4) We calculate churn metrics for each type and use a statistical tool R [9], 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 first three steps of our study approach.

2.1.1 Extracting bug fixes and non-bug fixes

The first step in our approach is to extract commits that are associated with bug fixes and the ones that are not associated with bug fixes. First, we extract

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

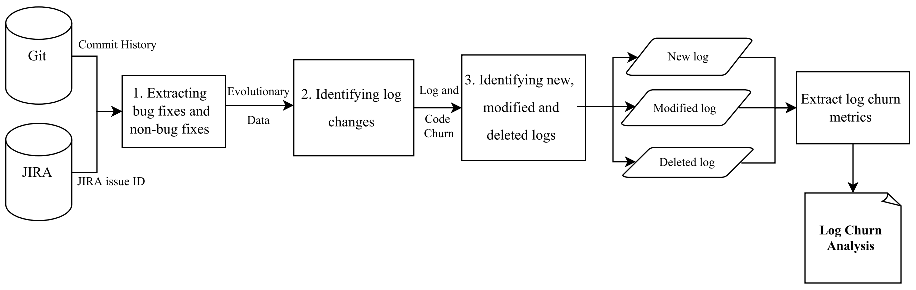

Git
Commit History
JIRA
JIRA issue ID
1. Extracting bug fixes and non-bug fixes
Evolutionary Data
2. Identifying log changes
Log and Code Churn
3. Identifying new, modified and deleted logs
New log
Modified log
Deleted log
Extract log churn metrics
Log Churn Analysis

Fig. 1: Overview of our case study approach

a list of all commits from the git repository of each project. To avoid the branching and merging commits, we enable the 'no-merges' option in the git *log* command. This flattens all the changes that are made to file in different branches and excludes the final merge between branch and trunk. We also filter out the changes to non-Java files or the test files from the commits.

Next, we extract a list of all the 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 fixes. We exclude commits that do not contain any JIRA ID since we cannot identify if the commit is a bug fix or not.

2.1.2 Identifying log changes

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

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

Some logs leverage logging libraries to print logs. For example, *Log4j*[3] is used widely in *Hadoop* and *HBase*. In both projects, logs have a method invocation 'LOG', followed by a logging level. The following log uses *Log4j*:

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

Using regular expressions to match these logs, we automate the process of finding all the logs in the studied projects.

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

2.1.3 Identifying new, modified and deleted logs

Since git *diff* does not track modification to the code, modifications to a log is shown as a deletion followed by an addition. To track these added and deleted logs we used Levenshtein ratio [19]. For every pair of added or deleted logs 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 log similar to prior research [23]. We consider a pair of added and deleted logs as a log modification if they have a Levenshtein ratio of 0.6 or higher. For example, the logs shown below have Levenshtein ratio of 0.86. Hence such a pair of added and deleted logs are categorized as a log modification.

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

If an added log has a Levenshtein ratio higher than 0.6 with more than one deleted log, we consider the pair of added and deleted logs with the highest Levenshtein ratio as a log modification. After identifying all log modifications, we identify three types of log changes in a commit namely 'new logs', 'deleted logs' and 'modified logs'.

3 Study Results

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

RQ1: Are logs changed more often during bug fixes?

Motivation

Prior research finds that up to 32% of the changes to logs are due to field debugging [30]. During debugging, developers may change logs to gain more run-time information about their system. These log changes may also assist developers in resolving future occurrences of a similar bug. However, to the best of our knowledge, there exists no large scale empirical study to show whether logs are changed more often during bug fixes than other development activities. In addition, we want to investigate how logs are changed during the bug fixes. These findings would provide more insight into what developers consider as important knowledge during bug fixes.

Approach

We compare the number of changes to logs between bug fixes and non-bug fixes. In previous section, we identified three categories of logs changes in a commit, i.e., modified, new and deleted logs. Therefore, we compare the number changes to logs within the three types in bug fixes and non-bug fixes. Since commits with higher total code churn may have a higher number of changes to logs, we calculate total code churn for every commit and use it to normalize *# modified, # new* and *# deleted logs*, similar to prior study [40]. The three new metrics are:

$$Modified\ logs\ density = \frac{\#\ of modified\ logs}{code\ churn} \tag{1}$$

$$New\ logs\ density = \frac{\#\ of new\ logs}{code\ churn} \tag{2}$$

$$Deleted\ logs\ density = \frac{\#\ of deleted\ logs}{code\ churn} \tag{3}$$

To future understand how logs are modified during bug fixes, we perform a manual analysis on the modified logs to identify the different types of log modifications. We first collect all the commits that modify logs. We select a random sample of 357 commits. The size of our random sample achieves a 95% confidence level and 5% confidence interval. We follow an iterative process, similar to prior research [24], 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 the normalize by *code churn*, similar to Equation 1 to 3. We call such metrics of log changes, i.e., modified, new, deleted logs density and the densities of each type of log modifications, as log churn metrics.

We would like to find out whether logs are changed more during bug fixes than non-bug fixes and to identify the type of log change that developers favor during bug fixes. We compare the mean densities of log churn metric in bug fixes and non-bug fixes. We further compare the log churn metrics by studying whether there is a statistically significant difference in these metrics, between bug fixes and non-bug fixes. We leverage the *MannWhitney U test* (Wilcoxon rank-sum test) [6], since our metrics are highly skewed. The *MannWhitney U test* is a non-parametric test, such that it does not have any assumptions about the distribution of the sample population. A p-value of ≤ 0.05 means that the difference of the log churn metrics between the bug fixes and non-bug fixes is statistically significant and we may reject the null hypothesis. By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us that there is a statistically significant difference for our metrics between bug fixes and non-bug fixes.

Table 2: Comparing the mean values of each log churn metric between bug fixes and non-bug fixes. The value of 1.9 for Hadoop means logs are modified 1.9 times more during bug fixes compared to non-bug fixes in Hadoop.

Average ratio of $\frac{Bug\ fixes}{Non-bug\ fixes}$	Projects		
	Hadoop	HBase	Qpid
Modified logs density	1.9	3.5	4.1
New logs density	1.4	1.2	1.2
Deleted logs density	1.6	2.5	0.5
Relocation log density	2.0	3.3	4.9
Modified log text density	2.5	6.5	11.1
Modified logged variable density	2.0	1.7	4.4
Modified log level density	4.0	7.5	15.8
Variable addition density	2.7	1.3	25.5
Variable deletion density	2.6	1.6	4.6
Variable modification density	2.9	0.8	4.7

We also calculate the *effect sizes* in order to quantify the differences in log churn metrics between the bug fixes and non-bug fixes. Unlike the *MannWhitney U test*, which only tells us whether the difference between the two distributions is statistically significant, the effect size quantifies the difference between the 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, the p-value are likely to be small even if the difference is trivial). We use *Cohen's d* to quantify the effect size [15,17]. *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* [25]. We use the following thresholds for *Cohen's d* [16]:

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

Results

Developers are more likely to add new logs when fixing bugs. From Table 2, we find that the mean of new log density is 1.2-1.4 times in bug fixes to non-bug fixes. This shows that new logs are 20%-40% more likely to occur in bug fixes that non-bug fixes. Table 3 shows that *new log density* in bug fixes is higher than non-bug fixes in all studied projects (statistically significant with non-trivial effect sizes). This suggest that developers may need additional information during bug fixes and they add new logs to help fix bugs.

Table 3: Comparing log churn metrics between the bug fixes and non-bug fixes. The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size shows that the log changes are more frequent during bug fixes.

Metrics	Hadoop		HBase		Qpid	
	P-values	Effect size	P-values	Effect size	P-values	Effect size
Modified log density	2.0e-12	0.246 (small)	1.9e-15	0.273 (small)	1.6e-11	0.432 (small)
New log density	4.7e-16	0.265 (small)	<2.2e-16	0.215 (small)	2.1e-11	0.474 (small)
Deleted log density	8.1e-07	0.336 (small)	4.9e-07	0.150	0.041	-0.193 (small)

Table 4: Distribution of four types of log modifications.

Projects	Hadoop (%)	HBase (%)	Qpid (%)
Relocation log density	49.8	48.5	62.5
Modified log text density	29.1	32.4	37.5
Modified logged variable density	43.3	41.2	33.6
Modified log level density	9.0	7.2	4.8

Developers are less likely to remove logs during bug fixes. We find that although the difference of *deleted log density* between bug fixes and non-bug fixes is statistically significant in all projects, the effect sizes is trivial for *HBase* and negative for *Qpid* (see Table 3). Moreover, we find that the log density varies between 0.5 and 2.4. This suggests that developers are more likely to remove logs during bug fixes in *Qpid*. Such results confirm the findings from prior research that deleted logs do not have a strong relationship with code quality [30].

Logs are more likely to be modified in bug fixes and non-bug fixes. From Table 2 we find that on average logs are 1.9 to 4.1 times more likely to be modified during bug fixes than non bug fixes. Table 3 shows that *modified log density* in bug fixes is higher than non-bug fixes in all studied projects (statistically significantly with non-trivial effect sizes). Such results show that developers often modify the information provided by logs to assist in bug fixes. Developers may need different information to the information that is provided by the logs to fix bugs. Prior research find that 66 % of the logs are modified when 1) the condition that the log depends on is changed, 2) the logged variable is changed or 3) the function name that is also referred in static text of log is modified [40]. Therefore, we further explore the different types of modifications to logs.

We manually examine the different modifications to logs and categorize the modifications into four types as follows:

Table 5: Comparing metrics about different types of log modification between bug fixes and non-bug fixes. The p-value are calculated from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size shows that the log changes are more frequent during bug fixes. P-values that are smaller than 0.05 are in bold font.

Metrics	Hadoop		HBase		Qpid	
	P-values	Effect sizes	P-values	Effect sizes	P-values	Effect sizes
Relocation log density	**1.1e-10**	0.330 (small)	**3.0e-11**	0.170 (small)	**1.8e-08**	0.700 (medium)
Modified log text density	0.344	-	**0.0075**	0.525 (small)	**4.5e-06**	0.976 (medium)
Modified log variable density	**1.3e-04**	0.351 (small)	**0.0010**	0.420 (small)	**1.2e-04**	1.17 (medium)
Modified log level density	0.108	-	0.503	-	0.399	-

1. **Relocation of log:** The log is not changed but moved to a different place in the file.
2. **Modified log text:** The text that is printed in the logs is modified.
3. **Modified logged variable:** One or more variables in the logs are changed (added, deleted or modified).
4. **Modified logging level:** The verbosity level of logs are changed.

Table 4 shows the distribution of each type in our studied projects. When there is co-occurrence of different types of log modifications, we manually identify which type of modification it falls under. When there are multiple types of modifications to precisely determine which category it belongs to, we exclude such logs and categorize them as new logs being added. This is seen in Table 4 where the total percentage is over 100% as there is co-occurrence of multiple changes in single log.

Developers modify logged variables during bug fixes. From Table 2 we find that variables in logs are 1.7 to 4.4 times more likely to be modified during bug fixes. We find that modifications to the logged variables are more likely to happen in bug fixes than non-bug fixes (statistically significant with medium or small effect sizes, see Table 5). This suggests that developers modify the logged variables in order to provide useful information in the logs. Prior research also shows that 16-32% of log changes are made during field debugging [30]. Therefore, to better understand how developers modify logged variables during bug fixes, we categorize the modifications into three types: a) variable addition, b) variable deletion and c) variable modification.

Table 6 shows that developers modify the logged variables more in bug fixes than non-bug fixes (statistically significant with medium effect sizes). This modification may be because developers need different information rather than the information provided by existing logs. For example, when manually

Table 6: Comparing the different types of changes to variables between the bug fixes and non-bug fixes. A positive effect size shows that the log changes are more frequent during bug fixes. P-values that are smaller than 0.05 are in bold font.

Metrics	Hadoop		HBase		Qpid	
	P-values	Effect sizes	P-values	Effect sizes	P-values	Effect sizes
Variable addition density	0.099	-	**0.00049**	0.659 (medium)	**0.005**	1.40 (large)
Variable deletion density	0.098	-	0.355	-	0.193	-
Variable modification density	**4.11e-05**	1.045 (medium)	0.568	-	**0.0016**	0.949 (medium)

exploring the patch notes for bug QPID-2370[4], we find that developers modify the existing log to capture a newly defined variable. The other reason may be that developers change the name of the existing variable to a more meaningful name. For example, in bug MAPREDUCE-2264[5], we find that developers rename variables and modify the logs accordingly.

From Table 6, we observe that the developers add variables more in bug fixes than non-bug fixes (statistically significant with medium effect size in *HBase* and large effect size in *Qpid*). We find from Table 2 that developers are 1.3 to 25.2 times more likely to add new variables during bug fixes than non-bug fixes. This addition of variables suggests that existing variables may not have all the needed information in the logs and developers have to add new variables during bug fixes. We also find that developers do not delete variables in logs. Deleting variables in logs may change the format of logs. There may be log processing applications that rely on these logs. Deleting variables may impact the correctness of these log processing applications [27]. Therefore, developer may be aware of the such impact and try to avoid deleting variables from logs.

Developers modify logged text more during bug fixes. We find that modification of logged text is more likely to happen in bug fixes than non-bug fixes with non-trivial effect sizes (see Table 5). In some cases, the text description in logs is not clear and developers need to improve the text. For example, to fix bug HBASE-6665[6] developers modify a log to provide a better description of the operation. Prior research shows that up to 39% of modifications to logged text is due to inconsistency between the execution event and the message conveyed by logs [40]. Our results suggest that developers may have

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

faced such challenges during bug fixes and may modify the text in logs for better description.

Relocation of log occurs more in bug fixes. Table 4, shows that there are a large number of log changes that only relocate logs. Table 5 shows that such relocation of logs is statistically significantly more in bug fixes than non-bug fixes (2.0 to 4.9 times more likely in bug fixes as shown in Table 2). 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 logs into the *try/catch* blocks or after condition statements. For example, to fix bug YARN-289 [7] from Hadoop, logs are placed into a proper *if-else* block.

Logging levels are not modified often during bug fixes. We find that logging level modifications are statistically indistinguishable between bug fixes and non-bug fixes in all the studied projects. The reason may be that developers are able to enable all the logs during bug fixes, despite of what level a log has. In addition, prior research shows that developers do not have a good knowledge about how to choose a correct logging level [40].

Developers change logs statistically significantly more in bug fixes than non-bug fixes. In particular, developers modify logs to add or change the variables in logs during bug fixes. This suggests that developers often realize the needed information to be logged as afterthoughts and change the variables in log to assist in fixing bugs.

RQ2: Is there a relationship between log changes and effort spent on bug fixes?

Motivation

In RQ1, we find that logs are more likely to be changed in bug fixes. However, we do not know whether changing logs has a relationship with effort spent on bug fixes. If changing logs during bug fixes has relationship with the effort of bug fixes, developers may consider leveraging and improving logs during bug fixes in order to save effort.

Approach

To study whether there is a relationship between effort spent on bug fixes and changing logs during bug fixes, we collect all JIRA issues with type *bug* from the three studied systems. We obtained the code commits for each of these JIRA issues by searching for the issue id from commit messages. We then separate the JIRA issues into (1) bugs that are fixed with log churn and (2) bugs that are fixed without log churn. We calculate the log churn metrics for each bug. If a issue has multiple commits we calculate the log churn metrics for

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

Table 7: Comparing code and developer metrics between the bug fixes with log churn and bug fixes without log churn.The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size shows that the code changes are more frequent during bug fixes log churn.

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)

each commit individually. We then measure effort of bug fixes using the time taken to resolve a bug, the number of developers involved during the resolution of a bug and the number of discussion posts about the bug on JIRA. Prior research has shown that such metrics are good estimates of developer effort [1].

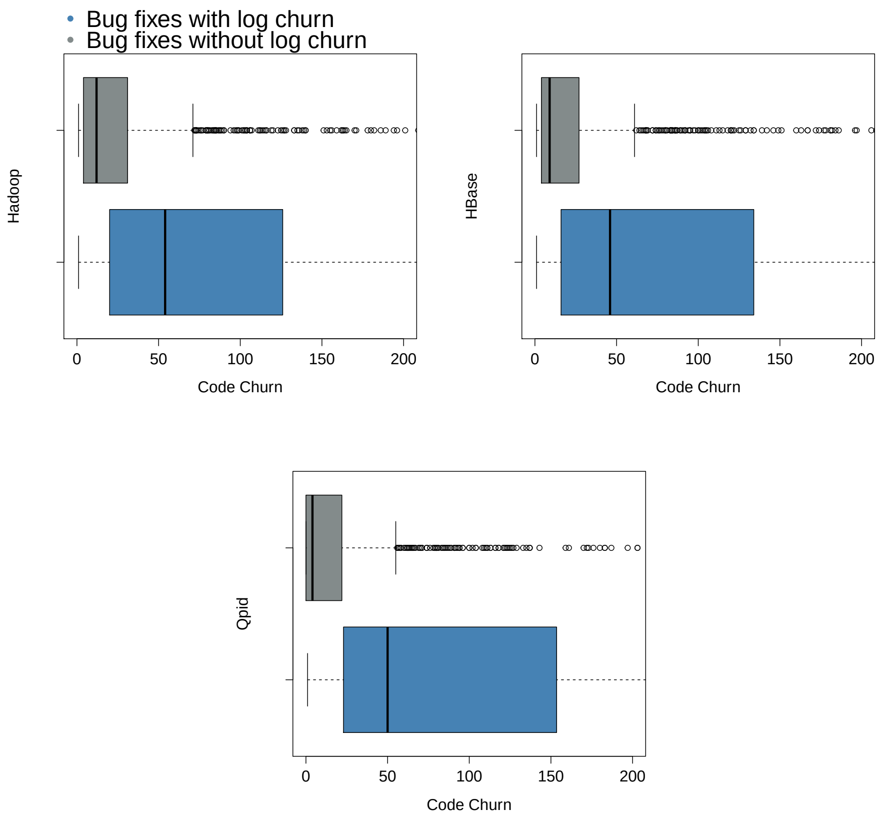
1. **Resolution time:** This metric measures how fast the bug is fixed. This metric is defined as the time taken from when the bug is opened until it is resolved. For example, if a bug was opened on 1st February 2015 and closed on 5th February 2015, the resolution time of the bug is four days.
2. **# of comments:** This metric measures how much discussion is needed to fix a bug. Intuitively, the more discussion in an 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 the number of 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.

Intuitively, a complex bug fix might have more code churn and in turn takes longer time to be resolved, more developers being involved and more discussions on JIRA. Therefore, we use code churn to normalize the resolution time, the number of comments, and the number of developers during bug fixes. We use these normalized effort metrics to find if there is statistically significant difference between bug fixes with log churn and bug fixes without log churn. We use the *MannWhitney U* test to find the $|\rho|$ values and *Cohen's d* test to measure the effect size, similar to RQ1.

Results

We find that the log changes are more likely to occur during more complex bugs fixes. We find that the average code churn for fixing bugs is

Fig. 2: Boxplot of code churn of bug fixes with log churn (shown in blue) against bug fixes without log churn (shown in grey).



significantly higher with log churn than without log churn (see Table 7 and Figure 2). This suggests that complex bug fixes are more likely to have log churn (statically significant with non-trivial effect size).

We find that bugs that are fixed with log churn, 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 log churn than the ones without log churn. This result suggests that given two bugs of the same complexity, the one with log churn is likely to take less time to get resolved and likely to need a fewer number of developers involved with fewer discussions. Changing logs may provide useful information to assist developers in discussing, diagnosing and fixing bugs. For example, when fixing bug HBASE-3074[8], developers suggest providing additional details in the log about where the failure occurs in the JIRA issue report. We find

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

that developers consider this suggestion and add the name of the servers into the logs. Such additional data helps diagnose the cause of the failure and helps fix the bug.

Discussion

To further understand how developers change logs during bug fixes, we conduct a manual analysis. We collected all the bug fixes with log churn for our studied 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 manually examine the code changes and the corresponding JIRA issue reports to find the reasons of changing logs during bug fixes. We follow an iterative process, similar to prior research [24], until we cannot find any new reasons. We find three reasons of changing logs during bug fixes as shown in Table 8. These three reasons may co-occur within a single commit.

Table 8: Log change reasons during bug fix

Projects	Hadoop	HBase	Qpid
Bug diagnosis	157	175	49
Similar bugs detection	156	170	42
Code quality assurance for bug fixes	93	78	18

- **Bug diagnosis.** Developers change logs to print extra or different information into logs during the execution of the system. Such information is printed to ease bug diagnosis. For example, HADOOP-2725[9] is reported when users find that after copying a 100TB file across two clusters, the file sizes has a discrepancy of 6GB. However, the existing logs do not have proper format to show the size of the copied data. To help diagnose the bug, developers modify the logged variable to print the sizes of the copied data into a better format.
- **Similar bugs detection.** After fixing a bug, developers may insert log into the code in order to monitor the execution of the system to detect similar bugs in the future. During the bug fixes, developers identify root-causes of the bug. After fixing the bug, developers change logs to capture the run-time event that may correspond to the root-cause of the bug, in order to identify future occurrence of a similar bug. The changed information in logs is not leveraged to fix the bug, but rather monitoring the systems for detecting a similar bug in the future. We find that logs in this category are mainly new logs being added into the system, unlike bug diagnosis which is mainly log modifications. For example, to fix HADOOP-2890[10],

[9] https://issues.apache.org/jira/browse/HADOOP-2725
[10] https://issues.apache.org/jira/browse/HADOOP-2890

developers identify the reason behind blocks getting corrupted as an runtime exception. In the commit, we observe that the developers fix this bug and add *try catch* block with new logs to catch these exceptions. Such log will notify developers if a similar bug appears.
– **Code quality assurance for bug fixes.** Sometime, developers need to introduce a large amounts of code to fix a bug. The introduction of bug-fixing code, may also introduce new bugs into the system. To ensure the quality of these bug fixes, developers insert new logs into the bug-fixing code. For example, in HBASE-3787[11], developers encounter a non-idempotent operation (i.e., running the operation more than once produces different results) that causes an error in the application. The fix of this bug involves over 13 developers and 112 discussions over the two years. The developers add several new files and functions during the bug fix, and added logs to assure the code quality of the fix.

Logs are more likely to be changed during complex bugs fixes. After normalizing the complexity of bugs using code churn, we find that bug fixes with log churn are resolved faster with fewer people and fewer discussions.

RQ3: Can log churn metrics help in explaining the resolution time of bugs?

Motivation

In RQ2, we find that bug fixes with log churn take shorter time to get resolved than bug fixes without log churn. However, there may be confounding effects between metrics, i.e., resolution time, total churn, # of developers and # of discussions, such that it is difficult to draw unbiased conclusion on the relationship between resolution time and log churn. To further explore this relationship between resolution time and log changes in bug fixes, we build prediction models using metrics from prior research and log churn metrics.

Approach

To better understand the relationship between log changes and the resolution time for fixing bugs, we build a non-linear regression model. Prior research has shown that linear modelling can help in predicting the resolution time of bugs [2]. However, the relationship between resolution time of bug fixes and log changes may be non-monotonic. By building a non-linear regression model we can more appropriately approximate the relationship between resolution time and log churn metrics, during bug fixes.

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

[11] https://issues.apache.org/jira/browse/HBASE-3787

include the log churn metrics (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 an explanatory metric. We use the *rms* package [7] from R, to build non-linear regression models. The overview of our modeling process is shown in Figure 3 and is explained below.

(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 an explanatory metric 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 [8], where x is the number of rows (i.e, # bugs) in each project.

(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 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 can 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 *redun* function that is provided in the *rms* package to perform the redundancy analysis.

(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 the function *spearman* in the *rms* package to calculate the Spearman multiple ρ^2 values for our metrics (metrics with larger ρ^2 values are allocated more degrees of freedom than metrics with smaller ρ^2 values).

(4)Regression modeling and validation

After budgeting degrees of freedom to our metrics we build a non-linear regression model using the function OLS (Ordinary Least Squares) that is

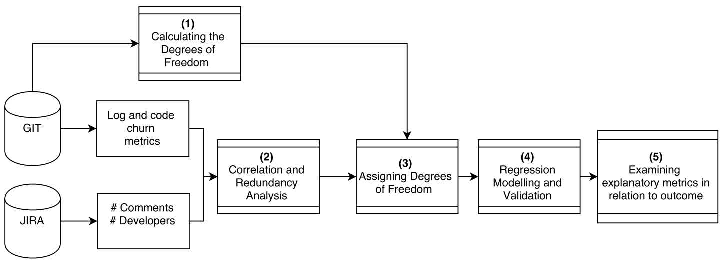

(1) Calculating the Degrees of Freedom
GIT
Log and code churn metrics
JIRA
# Comments
# Developers
(2) Correlation and Redundancy Analysis
(3) Assigning Degrees of Freedom
(4) Regression Modelling and Validation
(5) Examining explanatory metrics in relation to outcome

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

provided by the *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 log churn metrics and the resolution time of bug fixes, we are primarily concerned if the log churn metrics are significant in our models. Therefore, we use chunk test (a.k.a Wald test) to determine the statistically significant metrics to included in our final model. We choose chunk test as some of our explanatory variables are allocated several degrees of freedom and have to be tested jointly, similar to previous research [22]. At each step in the Wald test, we measure the significance of each metric according to its p-value. We consider only those metrics that have p-values lower than 0.05 in the final model. We use *wald.test* function provided by the R package *aod* [18] to perform the chunk test.

(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 knots assigned to that metric. To account for the impact of all knots 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 [7]. The plot follows the relationship as it changes directions at the spline (knot) locations (C-3).

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 log churn metrics and explore their empirical relationship to the resolution time of bugs.

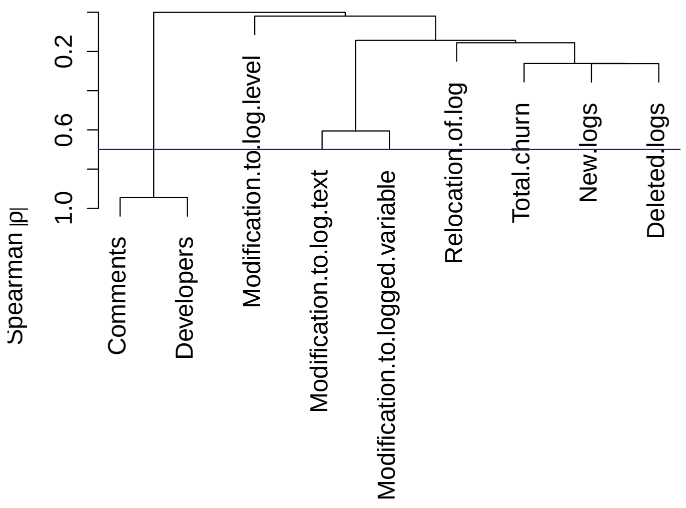


Fig. 4: Correlation between metrics in Qpid. Blue line indicates cut of set to 0.7

Results

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

(1) Calculating degrees of freedom. Our data can support 123 ($\frac{1,925}{15}$ in Hadoop), 63($\frac{953}{15}$ in Qpid) and 183($\frac{2,755}{15}$ in HBase) degrees of freedom for the studied projects. As we have large degrees of freedom in each project, we can be liberal in the allocating splines (knots) to the explanatory variables during model construction.

(2) Correlation and redundancy analysis. Figure 4 shows the hierarchically clustered Spearman ρ values of the three systems. The blue 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 studied projects.

(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

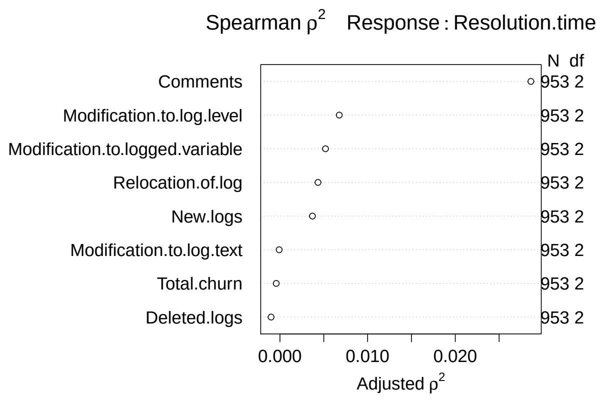


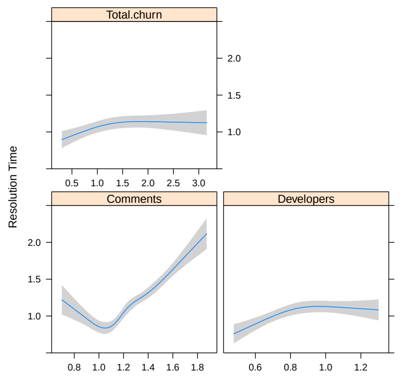
Fig. 5: Spearman multiple ρ^2 of each explanatory metric against Resolution Time of bug fixes with log changes. Larger values indicate more potential for non-linear relationship

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 *#modifications to log level*, *# modifications to log variable*, *#relocation of log* and *#new logs*. The last group consists of 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 [22].

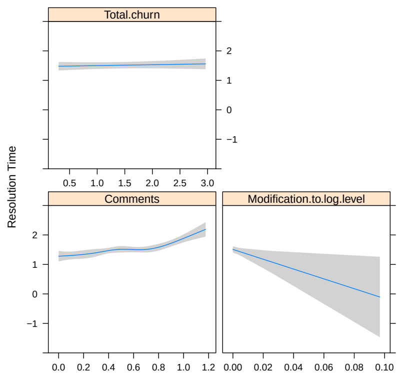
(4) Regression modeling and validation. After allocating the knots to the explanatory metrics, we build the non-linear regression model and use the *validate* function in the *rms* package to find the significant metrics in our studied projects. We find that log churn metrics are significant in Qpid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

(5) Examining explanatory metrics in relation to outcome. Figure 6 shows the direction of impact of log churn metrics on the resolution of bug fixes with log churn in *HBase*. We find that log modifications have a negative impact on the resolution time of bug fixes. Shown in Figure 6, we find that in *HBase* and *Qpid*, modifications to logs, i.e, log level changes and variable changes are significant in the models and have negative correlations with the resolution time of bugs.

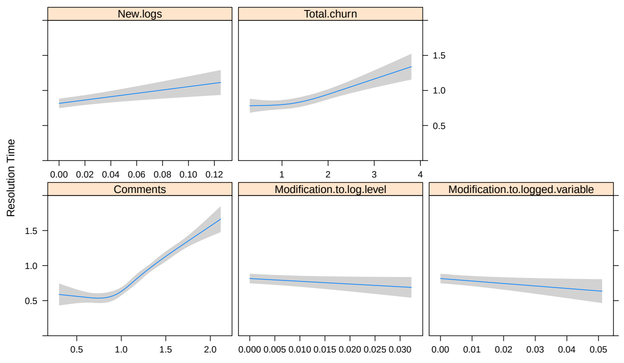
Modifications to log level have a negative relationship with bug resolution time. We find that modifications to log level have statistically significant explanation power to bug resolution time for *Qpid* and *HBase*. Even though developers often do not modify log levels during bug fixes (see RQ1), our results show that changes to log levels may help in faster resolution of bugs. Although prior research finds that developers are confused when estimating the



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

cost and benefit of each verbosity level in a log [40], a properly chosen log level may help developers diagnose filed bugs. On the other hand, a improper log level may result in too little or too much log, which are considered undesirable for diagnosing bugs [42,5]. An example where developers overestimate the log verbosity level is in the issue HBASE-8359[12], where the log is set to *info* and causes confusion among system administrators. We find that developers change the log verbosity level from *info* to *trace*, to reduce bulk logs being generated. On the other hand in HBASE-3401[13], we see that developers underestimate the verbosity level and the log is set to default *debug* and has to be updated to *warn* to give more priority to the logged event.

[12] https://issues.apache.org/jira/browse/HBASE-8359
[13] https://issues.apache.org/jira/browse/HBASE-3401

Modifications to logged variables have a negative relationship with bug resolution time. We find that variable changes have a statistically significant explanation power in the model for *HBase* with negative relationship on the resolution time of bugs. This may be because changing information in the logs may benefit developers during bug fixes. The other reason may be that developers add new functions or keywords and leverage the new information in the logs to ensure the functionality of the new code and assure the bug is fixed [5]. This enables developers to resolve the issue faster. In RQ1, we find that variable changes; especially addition and modification of logged variables is higher in bug fixes than non-bug fixes. These modifications to the logged variables highlight that developers have recognized the importance of information that is printed from logs and act on improving logs in order to assist in bug fixing.

New logs have a positive relationship with bug resolution time. We find that new logs have a statistically significant explanation power in the model for *HBase* with a positive relationship on the resolution time of bug fixes. We find that logs are often added during bug fixes in order to verify the quality of the bug fixing code. From our manual analysis results of RQ2, we find that such bugs are often complex to fix, with a large amounts of code churn and long resolution time. We compare the average code churn of bug fixes with new logs and bug fixes without new log additions. We find that the average code churn of bug fixes with new logs is almost twice that of average code churn of bug fixes without new log additions. For example, in HBASE-7305[14], developers implement a read write lock for table operations to fix a race condition. This issue has total code churn over 2.5 K with over 70 new logs in the code. Because of the addition of new features this issues takes over two months to be resolved with 44 discussion posts on the issue.

Log churn metrics can complement the number of comments, the number of developers and total code churn in modelling the resolution time of bugs. We find that modifications to logged variables and modifications to log level have a negative relationship with resolution time of bug fixes. On the other hand, the new logs have a positive relationship with resolution time of bugs.

4 Related Work

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

4.1 Log Analysis

Prior work leverage logs for detecting anomalies and bugs in large software systems. Lou *et al.* [21] propose an approach to use the variable values printed

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

in logs to detect anomalies in large systems. Based on the variable values in logs, their approach creates invariants (e.g., equations). Any new logs that violates the invariant are considered to be a sign of anomalies. Fu *et al.* [35] built a Finite State Automaton (FSA) using unstructured logs and to detect performance bugs in distributed systems. Xu *et al.* [38] link logs to logs in source code to recover the text and and the variable parts of output logs. They applied Principal Component Analysis (PCA) to detect system anomalies. To assist in fixing bugs using logs, Yuan *et al.* [39] propose an approach to automatically infer the failure scenarios when a log is printed during a failed run of a system. Jiang *et al.* [10] study the leverage of logs in troubleshooting issues from storage systems. They find that logs assist in a faster resolution of issues (i.e., bugs and anomalies) in storage systems. Beschastnikh *et al.* [3] designed automated tools that infers execution models from logs. These models can be used by developers to understand the behaviors of concurrent systems. Moreover, the models also assist in verifying the correctness of the system and fixing bugs.

Logs are also used testing large scale systems. Shang *et al.* [29] propose an approach to leverage logs in verifying the deployment of Big Data Analytic applications. Their approach analyzes logs in order to find differences between running in a small testing environment and a large field environment. Jiang *et al.* [13,12,14,11] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approach automatically abstracts logs into system events. Based on the such events, they identified both functional anomalies [12] and performance degradations [14] in load test results. In addition, they proposed an approach that leverage logs to reduce the load test that are performed in user environment [11].

The extensive prior research of log analysis motivate our paper to study how logs are leveraged during bug fixes. As a first step, we study the changes to log during bug fixes. Our findings show that logs are change more during bug fixes than other types of code changes. The changes to logs have a relationship with the resolution time of bugs.

4.2 Empirical studies on logs

Prior research performs an empirical study on the characteristics of logs. Yuan *et al.* [40] studies the logging characteristics in four open source systems. They find that over 33% of all log changes are after thoughts and logs are changed 1.8 times more than entire code. Fu *et al.* [5] performed an empirical study on where developer put logs. They find that logs 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.* [26] 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 logs and logs outputted during run time [30,28]. They find that logs are co-evolving with the software systems. However, logs are often modified by developers without considering the needs of operators. Furthermore, Shang *et al.* [31] 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.* [41] 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 in the logs thereby improving the output logs. Log Advisor is another tool by Zhu *et al.* [43] which helps in logging by learning where developers log through existing logging instances. Tan *et al.* [34] propose a tool named SALSA, which constructs state-machines from logs. The state-machines are further used to detect anomalies in distributed computing platforms.

The most related prior research by Shang *et al.* [30] empirically study the relationship of logging practice and code quality. Their manual analysis sheds light on the fact that some logs 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 logs are leveraged extensively during bug fixes and have a relationship with the resolution time of bugs.

5 Limitations and Threats to Validity

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

External Validity

Our case study is performed *Hadoop*, *HBase* and *Qpid*. Even though these three studied projects have years of history and large user bases, the three studied projects are all Java based platform software. Systems in other domain may not rely on logs 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 studied 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 logs and bug resolution time cannot claim causal effects, as we are investigating correlations, rather than conducting impact studies. The explanative power of log churn metrics 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 log in the source code. Even though the studied projects leverage logging libraries to generate logs at runtime, there still exist user-defined logs. By manually examining the source code, we believe that we extract most of the logs.

We use Levenshtein ratio and choose a threshold to identify modifications to logs. However, such threshold may not accurately identify modifications to logs. 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 log churn metrics, 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 log churn metrics 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 log churn metrics, 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 logs in bug fixes may vary in different components of the studied projects. More empirical studies on the use of logs in fixing bugs for different components of the systems are needed.

6 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 logs are changed more during bug fixes and how these changes occur. The highlights of our findings are:

– We find that logs are more likely to be changed during bug fixes than non-bug fixes. In particular, we find that logs are modified more likely during bug fixes than non-bug fixes. Variables and textual information in the logs are more likely to be modified during bug fixes than non-bug fixes.
– We find that logs are more likely to be changed during fixing more complex bug fixes. However, bug fixes that change logs are fixed faster, need fewer developers and have less discussion.

– We find that log churn metrics 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 logs are changed more in bug fixes and there is a relationship between changing logs and the resolution time of bugs. Developers should allocate more effort for considering the text, the logged variables and the verbosity levels in the logs the logs are added into the source code. Hence, bugs can be fixed faster without the necessity to change logs during the fix of bugs.

References

1. Amor, J.J., Robles, G., Gonzalez-Barahona, J.M.: Effort estimation by characterizing developer activity. In: Proceedings of the 2006 international workshop on Economics driven software engineering research, pp. 3–6. ACM (2006)
2. Anbalagan, P., Vouk, M.: On predicting the time taken to correct bug reports in open source projects. In: Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, pp. 523–526. IEEE (2009)
3. Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D.: Leveraging existing instrumentation to automatically infer invariant-constrained models. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pp. 267–277. ACM, New York, NY, USA (2011). DOI 10.1145/2025113.2025151
4. Bitincka, L., Ganapathi, A., Sorkin, S., Zhang, S.: 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, pp. 7–7. USENIX Association
5. Fu, Q., Zhu, J., Hu, W., Ding, J.G.L.R., Lin, Q., Zhang, D., Xie, T.: 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
6. Gehan, E.A.: A generalized wilcoxon test for comparing arbitrarily singly-censored samples. Biometrika **52**(1-2), 203–223 (1965)
7. Harrell, F.E.: Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis. Springer Science & Business Media (2013)
8. Harrell, F.E., Lee, K.L., Califf, R.M., Pryor, D.B., Rosati, R.A.: Regression modelling strategies for improved prognostic prediction. Statistics in medicine **3**(2), 143–152 (1984)
9. Ihaka, R., Gentleman, R.: R: a language for data analysis and graphics. Journal of computational and graphical statistics **5**(3), 299–314 (1996)
10. Jiang, W., Hu, C., Pasupathy, S., Kanevsky, A., Li, Z., Zhou, Y.: Understanding customer problem troubleshooting from storage system logs. In: FAST '09: Proceedings of the 7th Conference on File and Storage Technologies, pp. 43–56. USENIX Association, Berkeley, CA, USA (2009)
11. Jiang, Z.M., Avritzer, A., Shihab, E., Hassan, A.E., Flora, P.: 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, pp. 131–140, 2010. IEEE Computer Society, Washington, DC, USA. DOI 10.1109/SSIRI.2010.15
12. Jiang, Z.M., Hassan, A., Hamann, G., Flora, P.: Automatic identification of load testing problems. In: ICSM '08: Proceedings of theIEEE International Conference on Software Maintenance, pp. 307–316 (2008). DOI 10.1109/ICSM.2008.4658079
13. Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: An automated approach for abstracting execution logs to execution events. Journal of Software Maintenance Evolution **20**(4), 249–267 (2008). DOI 10.1002/smr.v20:4

14. Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: Automated performance analysis of load tests. In: ICSM '09: Proceedings of the IEEE International Conference on Software Maintenance, pp. 125–134. IEEE (2009)
15. Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I.: A systematic review of effect size in software engineering experiments. Information and Software Technology **49**(11), 1073–1086 (2007)
16. Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I.K.: A systematic review of effect size in software engineering experiments. Information and Software Technology **49**(11-12), 1073–1086 (2007)
17. Kitchenham, B., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J., et al.: Preliminary guidelines for empirical research in software engineering. Software Engineering, IEEE Transactions on **28**(8), 721–734 (2002)
18. Lesnoff, M., Lancelot, R., Lancelot, M.R.: Package ?aod?
19. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals (1966)
20. logstash: http://logstash.net
21. Lou, J.G., Fu, Q., Yang, S., Xu, Y., Li, J.: Mining invariants from console logs for system problem detection. In: USENIX Annual Technical Conference (2010)
22. McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering p. To appear (2015)
23. Mednis, M., Aurich, M.K.: Application of string similarity ratio and edit distance in automatic metabolite reconciliation comparing reconstructions and models. Biosystems and Information technology **1**(1), 14–18 (2012)
24. Seaman, C.B.: Qualitative methods in empirical studies of software engineering. Software Engineering, IEEE Transactions on **25**(4), 557–572 (1999)
25. Shadish, W.R., Haddock, C.K.: Combining estimates of effect size. The handbook of research synthesis and meta-analysis **2**, 257–277 (2009)
26. Shang, W.: Bridging the divide between software developers and operators using logs. In: ICSE '12 :Proceedings of the 34th International Conference on Software Engineering
27. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P.: An exploratory study of the evolution of communicated information about the execution of large software systems. Journal of Software: Evolution and Process **26**(1), 3–26 (2014). DOI 10.1002/smr.1579
28. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P.: An exploratory study of the evolution of communicated information about the execution of large software systems. Journal of Software: Evolution and Process **26**(1), 3–26 (2014). DOI 10.1002/smr.1579
29. Shang, W., Jiang, Z.M., Hemmati, H., Adams, B., Hassan, A.E., Martin, P.: Assisting developers of big data analytics applications when deploying on hadoop clouds. In: ICSE'13: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 402–411. IEEE Press, Piscataway, NJ, USA (2013)
30. Shang, W., Nagappan, M., Hassan, A.E.: Studying the relationship between logging characteristics and the code quality of platform software. Empirical Software Engineering **20**(1), 1–27 (2015). DOI 10.1007/s10664-013-9274-8
31. Shang, W., Nagappan, M., Hassan, A.E., Jiang, Z.M.: Understanding log lines using development knowledge. In: ICSME '14: Proceedings of the International Conference on Software Maintenance and Evolution,, pp. 21–30. IEEE (2014)
32. Syer, M., Jiang, Z.M., Nagappan, M., Hassan, A., Nasser, M., Flora, P.: Leveraging performance counters and execution logs to diagnose memory-related performance issues. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, pp. 110–119 (2013). DOI 10.1109/ICSM.2013.22
33. Syer, M.D., Jiang, Z.M., Nagappan, M., Hassan, A.E., Nasser, M., Flora, P.: Continuous validation of load test suites. In: Proceedings of the International Conference on Performance Engineering, pp. 259–270 (2014)
34. 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 Association (2008)

35. Wang, Q.J.L.Y., Li., J.: Execution anomaly detection in distributed systems through unstructured log analysis. In: ICDM 09: In Proceedings of 9th IEEE International Conference on Data Mining
36. Xpolog: URL `http://www.xpolog.com/`.
37. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.: 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. DOI 10.1109/ICDM.2009.19
38. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: ACM SIGOPS '09: Proceedings of the 22nd Symposium on Operating Systems Principles, pp. 117–132. New York, NY, USA. DOI 10.1145/1629575.1629587
39. Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., Pasupathy, S.: 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, pp. 143–154. ACM, New York, NY, USA (2010). DOI 10.1145/1736020.1736038
40. Yuan, D., Park, S., Zhou, Y.: Characterizing logging practices in open-source software. In: ICSE '12: Proceedings of the 34th International Conference on Software Engineering, pp. 102–112. IEEE Press (2012)
41. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S.: Improving software diagnosability via log enhancement. ACM Transactions on Computer Systems **30**, 4:1–4:28 (2012). DOI 10.1145/2110356.2110360
42. Zhao, X., Zhang, Y., Lion, D., Faizan, M., Luo, Y., Yuan, D., Stumm, M.: lprof: A nonintrusive request flow profiler for distributed systems. In: Proceedings of the 11th Symposium on Operating Systems Design and Implementation (2014)
43. Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M.R., Zhang, D.: Learning to log: Helping developers make informed logging decisions. In: Proc. of ACM/IEEE ICSE (2015)