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

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 often relies on output logs to monitor the applications that run over it. Such output logs are generated through simple *printf* statements or through logs in the source code. Such logs in the source code 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. We use the term 'log' to refer to logs in the source code in our paper. An example of a log 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 in source code are used by developers extensively during the development of software systems [1]. Output 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 such as Splunk [5], XpoLog [8], and Logstash [9], which assist developers in analyzing both output logs and logs in source code.

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 because of server crash. 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 logs changes are made during field debugging [10]. However, there exists no large scale study that investigates how logs are changed during bug fixes. By bridging this gap in knowledge, we can better understand what type of development knowledge is leveraged by developers in debugging, if the leveraged information is useful for debugging and if the information speeds up the debugging process. This knowledge can be used by tools like 'Log Enhancer' [11], 'Salsa' [12] and also developers to add better logs in the source code.

In this paper, we perform an empirical study on the changes that occur to logs 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 logs changed more often during bug fixes?**

We find that logs are changed more often during bug fixes than non-bug fixes. In particular, we find that adding and modifying logs occurs more often in bug fixes than non-bug fixes (statistically significant with non-trivial

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

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 fixes (statistically significant with medium to high effect sizes).

**RQ2: Is there relationship between log change and resolution time of 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.

**RQ3: Can log churn metrics explain 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 log churn metrics are statistically significant in the models and have negative impact on resolution time. This suggests that there is a relationship between log churn 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 for 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 software framework for distributed storage and processing of big data on computer clusters. *Hadoop* uses the MapReduce data-processing paradigm. The logging characteristics of *Hadoop* have been extensively studied in prior research [13, 10, 3]. We study the changes to logs from *Hadoop* releases 0.16.0 to 2.0.

[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) |

**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.*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 studied system to extract all commits and identify the bug fixes and non-bug fixes. (2) We identify log changes in both bug fixes and non-bug fixes. (3) We categorize the log changes into *'New logs'*, *'Modified Logs'* and *'Deleted logs'*. (4) We calculate churn metrics for each category and use a statistical tool R [14], 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 fixes and non-bug fixes*

The first step in our approach is to extract commits associated with bug fixes and non-bug fixes. First, we extract 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 made to file in different branches but only exclude the merge between branch and trunk. We also filter out the non-Java and 'test' files from the commits.

[2] http://hbase.apache.org/

[3] https://qpid.apache.org

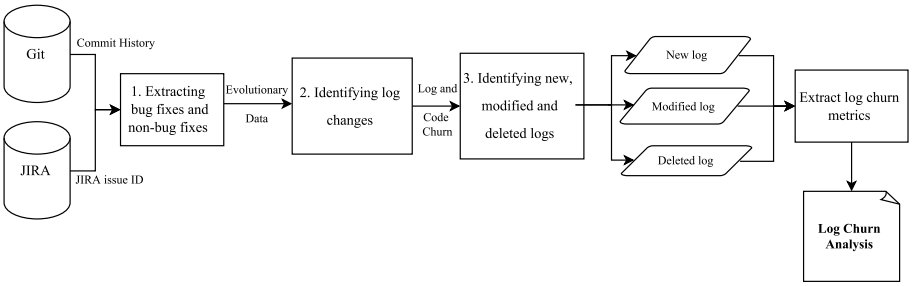



Fig. 1: Overview of our case study approach

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 which do not have JIRA IDs as 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 exception): " + e.what());
```

Some logs leverage logging libraries to print logs. For example, *Log4j*[2] 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.

### *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 [15]. For every pair of added or deleted logs in a

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

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 [16]. We consider a pair of added and deleted log as a 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 log churn is 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 categories of log changes in a commit namely 'newly added 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 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, is due to field debugging [10]. During debugging, developers 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 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*. The three new metrics are:

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

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

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

We compare the density of each type of log change (i.e., modified, new and deleted logs) in bug fixes and non-bug fixes, as we are trying to find how much more logs are changed during bug fixes. Log density is defined as the ratio of total log churn over total code churn across all commits, as used in prior research [1]. We follow the same approach and find log density of each type of log change for bug fixes and non-bug fixes.

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 [17], 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.

To identify the type of log change developers favor during bug fixes, we find whether there is a statistically significant difference in the log churn metrics, between bug fixes and non-bug fixes. To do this we use the *MannWhitney U test* (Wilcoxon rank-sum test) [18], as our metrics are highly skewed. Since the *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 $\leq$ 0.05 means that the difference of the 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.

We also calculate the *effect sizes* in order to quantify the differences in our 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,

Table 2: Comparing the mean log density 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).

| Projects | Modified log density ratio | New log density ratio | Deleted log density |
|---|---|---|---|
| Hadoop | 1.9 | 1.4 | 1.6 |
| HBase | 3.5 | 1.2 | 2.4 |
| Qpid | 4.1 | 1.2 | 0.5 |

the p-value are likely to be small even if the difference is trivial). We use *Cohen's d* to quantify the effect size [19,20]. *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* [21]. We use the following thresholds for *Cohen's d* [22]:

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

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 and P-values are smaller than 0.05.

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

***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 logs ratio* in bug

Table 4: 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 |

fixes is higher than non-bug fixes in all studied systems (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.

**Developers are less likely to remove logs during bug fixes.** We find that although the difference of *deleted logging statements ratio* 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 0.5-2.4. This suggests 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 [6].

**Logging statements are more likely to be modified in bug fixes and non-bug fixes**. From Table 2 we find that logs are 1.9-4.1 times more likely to be modified during bug fixes than non bug fixes. Table 3 shows that $modified\ log-\ ratio$ in bug fixes is higher than non-bug fixes in all studied systems (statistically significantly with non-trivial effect sizes). Such results show that developers often change 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 the bugs. Prior research find that 66 % of the logs are modified when 1) the condition the log code depends on is changed, 2) the logged variable is changed or 3) the function name which is also referred in static text of log is modified [1]. Therefore, we further explore the different types of modifications to logs.

We manually look at the different modifications to logs and categorize the modifications into four types. They are described below and Table 4 shows the distribution of each type in our studied systems. When there is co-occurrence of different types of log modifications in a single log, we treat that as new log because it is difficult to categorize all the different types of co-occurrences.

1. **Logging statement relocation:** The log is not changed but moved to a different place in the file.
2. **Text modification:** The text that is printed in the logs is modified.
3. **Variable change:** One or more variables in the logs are changed (added, deleted or modified).
4. **Logging level change:** The verbosity level of logs are changed.

Table 5: Comparing the mean log density between bug fixes and non-bug fixes.

| Projects | Log relocation density ratio | Text modification density ratio | Variable modification density ratio | Logging level density ratio |
|---|---|---|---|---|
| Hadoop | 2.0 | 2.5 | 2.0 | 4.0 |
| HBase | 3.3 | 6.5 | 1.7 | 7.5 |
| Qpid | 4.9 | 11.1 | 4.4 | 15.8 |

Table 6: Comparing logging modification 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 and p-values are smaller than 0.05

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

**Developers modify logged variables during bug fixes**. From Table 2 we find that variables in logs are 1.7-4.4 times more likely to be modified during bug fixes. We find that changes to the logged variables is higher in bug fixes and non-bug fixes as seen in Table 6 (statistically significant with medium or small effect sizes). This suggests that developers change the logged variables in order to provide useful information in the log outputs. Prior research also shows that 16-32% of log changes are made during field debugging [10]. Therefore, to better understand how developers change logged variables during bug fixes, we categorize the changes into three types: a) variable addition, b) variable deletion and c) variable modification.

Table 7 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 in the output logs, than provided by existing logs. For example, when manually exploring the patch notes for bug QPID-2370[3], we find that developers modify the existing log to capture the newly defined variable. The other reason may be that developers change the name of the existing variable to a more meaning-

[3] https://issues.apache.org/jira/browse/QPID-2370

Table 7: Comparing the different types of changes to variables 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 and p-values are smaller than 0.05

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

Table 8: Comparing the log density between bug fixes and non-bug fixes.

| Projects | Variable addition density ratio | Variable deletion density ratio | Variable modification density ratio |
|---|---|---|---|
| Hadoop | 2.7 | 2.6 | 2.9 |
| HBase | 1.3 | 1.6 | 0.8 |
| Qpid | 25.2 | 4.6 | 4.7 |

ful name. For example, in bug MAPREDUCE-2264[4], we find that developers rename variables and modify the logs accordingly.

From Table 7, 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 1.3-25.2 times more likely to add new variables during bug fixes. This addition of variables suggests, existing variables may not have all the needed information in the log output 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 the output logs. There may be log processing tools that rely on these output logs. 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 logs.

**Developers modify logged text more during bug fixes.** We find that modification of logged text is higher in bug fixes and non-bug fixes with non-trivial effect sizes (see Table 6). In some cases, the text description in logs is not clear and developers need to improve the text to provide more clarity. For example, in *HBase* HBASE-6665[5] developers modify the log to provide more information about which region (i.e., Root or Meta) is being used in the operation. Prior research shows that up-to 39% of modifications

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

to logged text is due to inconsistency between the execution event and the message conveyed by log output [1]. Our results suggest that developers may have faced such challenges during bug fixes and may have modified the text in logs for clarification.

**Logging statement relocation occurs more in bug fixes.** Table 4, shows that there are a large number of logging changes that only relocate logs. Table 6 shows that such relocation of logs is statistically significantly more in bug fixes and non-bug fixes (2.0-4.9 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, in the YARN-289 [6] of Hadoop, logs 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 betweenbug fixes and non-bug fixes in all studied systems. 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 [1].

*Developers change logs statistically significantly more in bug fixes than non-bug fixes in a given file. 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 after-thoughts and change the variables in log to assist in fixing bugs.*

### RQ2: Is there relationship between log change and resolution time of bug fixes?

#### *Motivation*

In RQ1, we find that logs are changed more frequently in bug fixes. However, we do not know if leveraging and changing logs is beneficial during bug fixes. To answer this, we look at the effort spent to fix the bug. We measure effort spent based on the time taken to resolve a bug, the number of developers involved during the resolution of a bug and the number of discussions posts on JIRA. We try to find whether there is a relationship between resolution time, developer involvement and log churn during bug fixes.

#### *Approach*

To find out whether there is a relationship between log churn and resolution time of bug fixes, we collect all JIRA issues with type 'bug' from the three

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

studied systems. We obtained the code commits for each of these JIRA issues by searching for the issue id from the commit messages. We identify the log churn, and the code churn for fixing each issue. We then split the JIRA issues into (1) bugs that are fixed with log churn and (2) bugs that are fixed without log churn. 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 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.

Prior research has shown that complexity of software can be measured using different metrics, including source lines of code change [23]. 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 logs changes are more likely to occur during complex bugs fixes**. We find that the average code churn for fixing bugs is significantly higher with log churn than without log churn (see Table 9 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 usually take less time to get resolved and needs a fewer number of developers involved

Table 9: 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. P-values are smaller than 0.05 and negative effect size implies that metrics for bug fixes without log churn have higher values.

| 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) |

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[7], developers left the first comment to provide additional details in the log about where the failure occurs. In the source code, developers add the name of the servers into the the logs. Such additional data helps diagnose the cause of the failure and helps fix the bug.

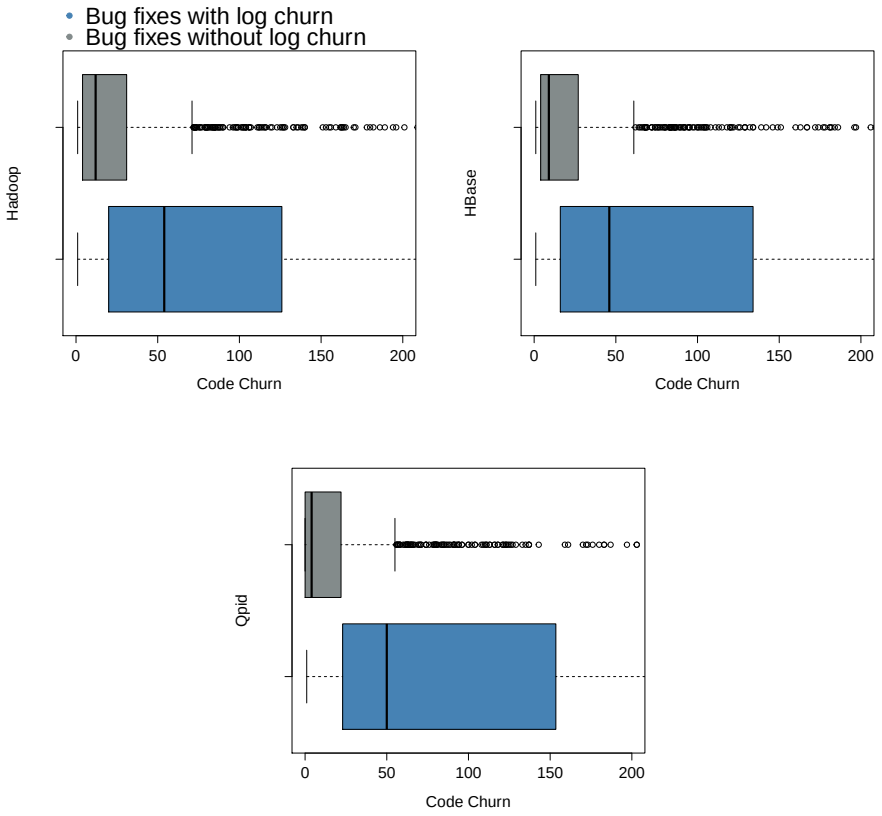
To further understand how developers change logs in bug fixes, we finally conduct a qualitative analysis. We collected all the bug fixes with log churn for our studied systems. We selected a 5% random sample (266 for *HBase*, 268 for *Hadoop* and 83 for *Qpid*) from all the commits. For the sampled commits, we 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 [17], until we cannot find any new types of patterns. We find three reasons of changing logs during bug fixes as shown in Table 10. We find that these three reasons can co-occur within a single commit.

Table 10: Log change reasons during bug fix

| Projects | Hadoop | HBase | Qpid |
|---|---|---|---|
| Bug diagnosis | 157 | 175 | 49 |
| Future bugs prevention | 156 | 170 | 42 |
| Code quality assurance for bug fixes | 93 | 78 | 18 |

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

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




– *Bug diagnosis*

Developers use logs to detect runtime defects. Developers change log to print extra or different information into logs during the execution of the system. The log changes in the category have added and deleted code prior to log changes (i.e., code block is changed). For example, to fix HADOOP-2725 [8] 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 logged variable which outputs the sizes of the files into human readable format instead of bytes. These log changes are committed along with bug fix, as it clarifies the log output and helps in understanding the logging context.

[8] https://issues.apache.org/jira/browse/HADOOP-2725

– *Future bugs prevention*

After fixing a bug, developers may insert log into the code. Such logs monitor the execution of the system to track for similar bugs in the future. Log changes in this category have addition of new blocks (i.e., if, if-else, try-catch and exception) with higher code addition than deletion. For example in HADOOP-2890[9], 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 conditionals (i.e., if-else, try-catch, throw) to verify where the block gets corrupted and they add *try catch* block with new logs to catch these exceptions. The log will notify developers with useful information to diagnose the bug 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 with no code deletion. The introduction of new code, may introduce new bugs into the system. To ensure the quality of these bug fixes, developers insert logs into the bug-fixing code. For example in HBASE-3787[10], where developers encounter a non-idempotent operation (i.e., running the operation more than produces different results) 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 logs to assure the code quality of the fix.

*Logging statements are changed during fixing more complex bugs. 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***

Prior research has shown that resolution time is correlated to the number of developers and the number of comments in an issue report [24]. However, prior research does not look at the type of changes made by developers.

In RQ2, we find that bug fixes with log churn take shorter time to get resolved than bug fixes without log churn. However, resolution time, total churn, # of developers and # of discussions can be multi-correlated and it is difficult to draw unbiased conclusion on the relationship between resolution

[9] https://issues.apache.org/jira/browse/HADOOP-2890
[10] https://issues.apache.org/jira/browse/3787

time and log churn. To further explore this relationship between resolution time and log churn in bug fixes, we build prediction models using metrics from prior research and log churn metrics. We want to understand the effect of log changes during bug fixes and also identify which types of log changes can be beneficial during bug fixes.

***Approach***

To better understand the relationship between log churn metrics 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 bug [25]. However, the relation between resolution time of bug fixes and log churn metrics 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 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 explanatory metrics. We use the *rms* package [26] 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 [27]. 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 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

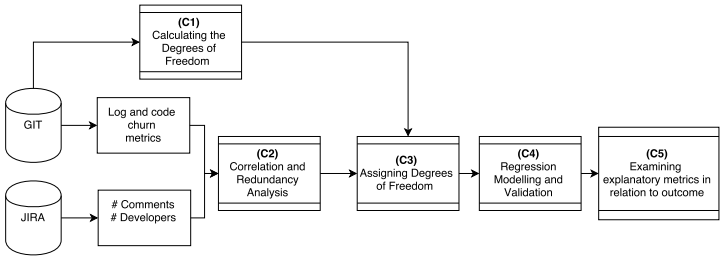



Fig. 3: Overview of our non-linear OLS model construction(C) for the resolution time of bugs

*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.

*(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 $\rho^2$ 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 $\rho^2$ values for our metrics (metrics with larger $\rho^2$ values are allocated more degrees of freedom than metrics with smaller $\rho^2$ values).

*(C-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 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 [28]. At each step in the Wald test, we measure the significance of each metric according to its p-value. We consider only those metrics which have p-value lower than 0.05 for the final model. We use 'wald.test' function provided by the R package *aod* [29] to perform the chunk test.

*(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 [26]. 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 log churn metrics 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 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 systems. 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.

**(C-2) Correlation and redundancy analysis**. Figure 4 shows the hierarchically clustered Spearman $\rho$ 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 systems.

**(C-3)Assigning degrees of freedom**. Figure 5 shows the Spearman multiple $\rho^2$ of the resolution time against each explanatory metric. Metrics that have higher Spearman multiple $\rho^2$ have higher chance of benefiting from

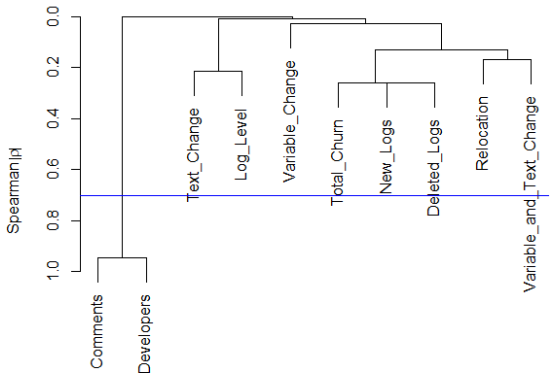



Fig. 4: Correlation between metrics in Qpid. Dashed line indicates cut of set to 0.7

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*, *#kig relocation* and *#new logs*. 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 [28].

**(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 studied systems. We find that log churn metrics 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 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.

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 trend of not changing log levels may be because, developers are confused when estimating the cost and benefit of each verbosity

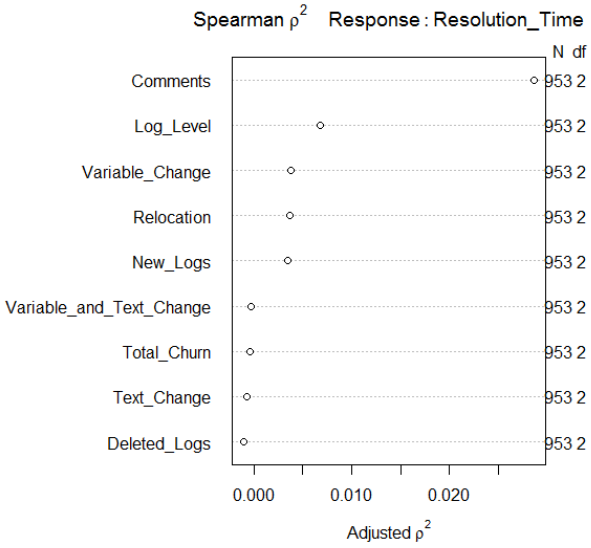


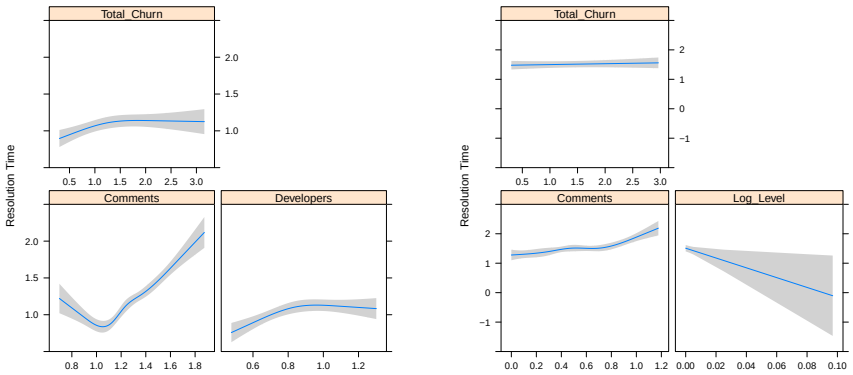

Fig. 5: Spearman multiple $\rho^2$ of each explanatory metric against Resolution Time of bug fixes with log churn. Larger values indicate more potential for non-linear relationship

level in a log as shown by prior research [1]. An example where developers overestimate the log verbosity level is in the issue HBASE-8359[11], 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[12], 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.

We find that variable changes is significant in the model for *HBase* with negative relationship on the resolution time of bugs. This may be because having different information in the output 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. In RQ1, we find that variable changes; especially addition and modification of logged variables is higher in bug fixes than non-bug fixes. These changes to the logged variables highlight the fact that developers recognize the significance of correct information in logged output.

We find that new logs have a positive impact on the resolution time of bug fixes in *HBase* project. We find that the average code churn of bug fixes

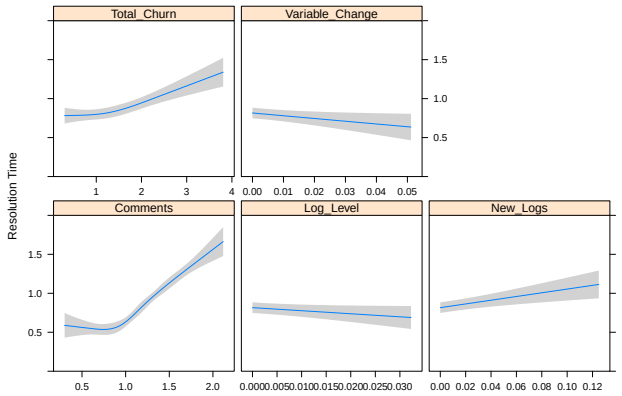
[11] https://issues.apache.org/jira/browse/HBASE-8359
[12] https://issues.apache.org/jira/browse/HBASE-3401




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

with new logs is almost twice that off average code churn of all commits. We also find that the average resolution time is higher for bug fixes with new logs. 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 logs to ensure the functionality of the added code. For example in HBASE-7305[13], developers implement a read write lock for table operations, which previously causes race conditions. We find that this issue has total code churn over 2.5 K and has addition of over 70 new log lines in the code. Because

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

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 logging modifications have a negative effect on resolution time of bug fixes, meaning increase in logging modifications decreases the resolution time of bug fixes. Such results imply that there is a relationship between log churn and the 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 testing and detecting anomalies in large scale systems. Shang *et al.* [30] 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. Lou *et al.* [13] propose an approach to use the variable values printed 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.* [31] built a Finite State Automaton (FSA) using unstructured logs and to detect performance bugs in distributed systems.

Xu *et al.* [3] 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. Tan *et al.* [12] 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.* [32] 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.* [33] 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.* [34] propose an approach to automatically infer the failure scenarios when a log is printed during a failed run of a system.

Jiang *et al.* [35–38] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [35]. Based on the such events, they

identified both functional anomalies [36] and performance degradations [37] in load test results. In addition, they proposed an approach that leverage logs to reduce the load test that are performed in user environment [38].

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 a faster resolution of bugs with fewer people and less discussion.

### 4.2 Empirical studies on logs

Prior research performs an empirical study on the characteristics of logs. Yuan *et al.* [1] 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.* [39] 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.* [40] 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 [10, 41]. 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.* [42] 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.* [11] 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.

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 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 may assist in the resolution of bugs.

## 5 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 studied systems have years of history and large user bases, the three studied systems 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 systems 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. Evaluation on the coverage of our extracted logs can address this threat.

We use keywords to identify bug fixes 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 fixes or branching and merging commits.

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 systems. 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 changed more during bug fixes than non-bug fixes. In particular, we find that logs are modified more frequently during bug fixes. Variables and textual information in the logs are more frequently modified during bug fixes.
- We find that logs are changed more during 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 by developers in bug fixes and there is a relationship between changing logs and the resolution time of bugs. We find that developers modify the text or variables in logs 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 logs when developers first add logs to the source code. Hence, bugs can be fixed faster without the necessity to change logs 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. 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.
12. 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.
13. 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.
14. 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.
15. V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, 1966.
16. 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.
17. C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *Software Engineering, IEEE Transactions on*, vol. 25, no. 4, pp. 557–572, 1999.
18. E. A. Gehan, "A generalized wilcoxon test for comparing arbitrarily singly-censored samples," *Biometrika*, vol. 52, no. 1-2, pp. 203–223, 1965.
19. 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.
20. 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.
21. 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.
22. 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.
23. E. Weyuker, "Evaluating software complexity measures," *Software Engineering, IEEE Transactions on*, vol. 14, no. 9, pp. 1357–1365, Sep 1988.
24. 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.
25. ——, "On predicting the time taken to correct bug reports in open source projects," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 523–526.

26. F. E. Harrell, *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. Springer Science & Business Media, 2013.
27. 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.
28. 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.
29. M. Lesnoff, R. Lancelot, and M. R. Lancelot, "Package ?aod?"
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. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.
38. 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.
39. 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.
40. W. Shang, "Bridging the divide between software developers and operators using logs," in *ICSE '12 :Proceedings of the 34th International Conference on Software Engineering*.
41. 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.
42. 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.
43. 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.