An Empirical Study On Leveraging Logs During Bug Fixes

Suhas Kabinna, Weiyi Shang, Ahmed E. Hassan
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 2N8
Email: {kabinna, swy, ahmed}@queensu.ca

***Abstract*—Logging is a practice used by software developers to record and convey important information during the execution of a system. Logs can be used to output the behavior of the system when running, to monitor the choke points of a system, and to help in debugging the system. These logs are valuable sources of information for developers in debugging large scale software systems. Prior research has shown that over 41% of log changes are done during bug fixes. However there exists little knowledge about how logs leveraged during bug fixes. In this paper we sought to study the leverage of logs during bug fixes through case studies on three large open source systems namely Hadoop, HBase and Qpid. We found that logs are statistically significantly more changed in bug fixes than other code changes. We found four different types of log modifications, that developers make to logging statements during bug fixes namely - changes to logging level, changes to the textual content of a log, changes to log variables (parameters) and relocation of logs. We found that developers modify variables more during bug fixes. Finally we found that issue reports with log changes have larger code churn, but require less people, less time and less discussion to get fixed. We created a logistic regression model and found that adding and modifying logging statements are statistically significant has negative effect impact on resolution time of bug fixes. These findings show that logs help in bug fixes and motivates developers to log more.**

I. Introduction

Logging is a software practice leveraged by developers to record useful information during the execution of a system. Logging can be done through simple *printf* statements or through the usage of logging libraries such as 'Log4j', 'Slf4j', 'JCL' etc. Each log contains a textual part which gives information about the context, a variable part which contains information about the events, and logging levels which provide the verbosity of the logs. An example of a logging statement is shown below where 'info' is the log level, 'Connected to' is the context information and 'host' is the variable.

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

Research has shown that logs are used by developers extensively during the development of software systems [1]. Logs are leveraged for detecting anomalies [2], [3], [4], monitoring performance of systems [5], maintenance of large scale systems [4], capacity planning of large-scale systems [6] and debugging [7]. The valuable information in logs leads to a new market for log maintenance platforms like Splunk [5], XpoLog [8], Logstash [9] which assist developers in analyzing logs.

Logs are used also extensively to help developers in fixing bugs in large software systems . For example, in the JIRA issue HBASE-3403 (commit 1056484), a bug is reported when a module does not exit upon system failure. In order to fix this bug developers record more information in the existing logs. After the bug is fixed, the added information in the logs are used by developers to prevent similar bugs in the system.

Prior research performs a manual study on bug fixing changes. The results show that 41% of the log changes are used to field debugging [7]. However there exists no large scale empirical study to investigate how logs are leveraged during bug fixing.

In this paper we perform an empirical study on the leverage of logs during bug fixes on three open source systems namely Hadoop, HBase and Qpid. In particular we sought to answer following research questions.

RQ1: Are logs leveraged more during bug fixes?

We found that logs are leveraged more frequently during bug fixes than non-bug fixing changes . In particular we found that log addition and log modification are statistically significant during bug fixing commits than non-bug fixing commits, with non-trivial effect size. We identified 4 types of log modifications namely *'Logging level change', 'Text Modification', 'Variable Change' and* 'Relocating logs'. We found that *'Text Modification', 'Variable Change' and* 'Relocating' are statistically more significant, with medium to high effect sizes in bug fixing commits. This shows that developers need specific information from logs to help fix bugs.

RQ2: Are logs useful in bug fixes?

We found that logs help in a faster resolution of bugs with less developer involvement. We found that bug fixing commits with log changes, have higher total churn. This implies that logs are leveraged more to fix complex bugs. After controlling the code churn, we found that the issues with log changes takes less time to get resolved, need less developer involvement and have less discussions. Using these metrics we built a logistic regression model to find the effect of log churn metrics on resolution time. We found that log churn metrics has negative effect impact on resolution time of bug fixes. In particular we found that *'New Log churn'* and *'Modified Log churn'* have negative effect impact on resolution time and are statistically

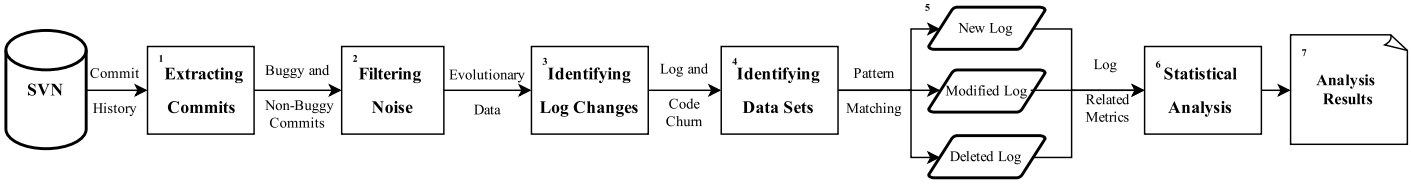

Fig. 1. Overview of our cast study approach

significant. This implies that adding and modifying logging statements during bug fixes, can help in reducing the resolution time for the bug fix.

The rest of this paper is organized as follows. Section 2 contains details of our methodology in gathering and extracting the data for our study. We also present an overview of the statistical techniques we will be using in this paper. Section 3 contains the detailed case studies and the results we obtained. Section 4 contains the related work in field of logging. Section 5 contains the limitations of our study and threats to validity. Section 6 contains further work and concludes the paper.

II. Methodology

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

The aim of this paper is to understand what roles logging statements play in bug fixes. To answer this question, we conduct a case study on three open source projects. Hadoop, Hbase and Qpid have extensive logging and suit our case study. Table I highlights the overview of the three subject systems.

Hadoop[1]: Hadoop is an open source software framework for distributed storage and distributed processing of big data on clusters. It uses the MapReduce data-processing paradigm and the debugging process of Hadoop has been studied in prior research [7], [10], [3]. We used Hadoop releases 0.16.0 to 2.0.

HBase[2]: Apache HBase is a distributed, scalable, big data store using Hadoop file-systems. We used Hbase releases 0.10 till 0.98.2.RC0. Those release have more than 4 years of development in Hbase from 2010 till 2014.

Qpid[3]: Qpid is an open source messaging system that implements a Advanced Message Queuing Protocol (AMQP). We studied from release 0.10 till release 0.30 of QPID. Those releases are from 2011 to 2014.

Figure 1 shows a general overview of our approach. (1) We mine the SVN repository of each subject system to extract all commits. (2) We then identify each commit as bug-fixing or non-bug-fixing commit . (3) We remove the noise from our extracted data sets (4) We identify log changes in both bug-fixing and non-bug-fixing commits. (5) We categorize the log changes into *'New logs'*, *'Modified Logs'* and *'Deleted logs'*. We calculate churn metrics using these categories and use statistical tools, such as R [11], to perform experiments on the data to answer our research questions. In the rest of this section we describe the first 5 steps in more detail.

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

A. Study Approach

We used SVN to study the evolution of Java source code in the three projects. We extract the changes made in each commit and using this data calculate the metrics to answer our research questions. The entire process is broken down into 3 steps and is illustrated in Figure 1.

1) Extracting Commits : The first step in our approach is to extract buggy and non-buggy commits. We extracted a list of all commits from SVN and the commit message from each commit. We extracted a list of all JIRA issues related to bug fixes. As developers mention the JIRA issue numbers in the commit messages, we matched the commit messages against the JIRA issues to identify all the defect fixing changes. If a commit message does not contain a JIRA issue we search search for bug-fixing key words like 'fix' or 'bug'. Prior research has shown that such heuristics can identify bug-fixing commits with a high accuracy.

2) Filtering Noise: After separating our data into buggy and non-buggy commits, we calculated the churn for each commit. As commits contain changes to non-Java files, we filtered out the changes to non-Java files from both the datasets.

We found that some commits have a high code churn because of branch and merge operations. To filter out such commits with high code churn(over 50,000), we consider only those commits which have both code addition and deletion. For example - in the commit 952,410 the total churn is over 100,000 and it has no deletion of code. To filter branching commits with churn less than 50,000 we use the change-list file where branching commits fall under 'BRANCH_SYNC' category.

3) Identifying Log changes : To identify the log changes in the datasets, we manually looked at some of the commits to find common patterns in the logging statements. Some of the patterns were specific to a particular project. For example a logging statement from Qpid invokes 'QPID_LOG', for example -

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

Some patterns are uniform across projects due to the use of same logging libraries. For example -

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

TABLE I
OVERVIEW OF THE DATA

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

Using regular expressions to match these patterns, we automated the process of finding all the logging statements in our data sets. For example, *Log4j* is used widely in Hadoop and HBase. In both projects, logging statements have method invocation like "LOG", followed by log-level and other information. We count every such invocation as a logging statement.

4) Identifying Data Sets.: After identifying the logging statements in each commit, we found two types of log changes.

Added Log: This type includes all log lines added in a commit.

Deleted Log: This type includes all log lines deleted in a commit.

Since SVN *diff* does not provide a built in feature to track modification to a file line by line, modifications to logging statements are shown as added and deleted logging statements. To track these modifications we used levenshtein measures [12]. We remove the logging method and the log level and compare the text in the parenthesis. If the levenshtein distance between the added and deleted logging statement is less than 5 or the ratio greater than 0.5 we consider it as log modification. We used levenshtein distance of 5 to match smaller logging statements and ratio is used to match longer statements. For example, the logging statements show below have levenshtein distance of 16 and ration of 0.86 when we compare both the logging statements entirely. Hence this is categorized as a log modification.

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

After identifying log modifications we obtained three new data sets namely:

1) Modified Logs: This includes all the modified logging statements in a commit.
2) New Logs: This includes all those logs which were newly added in a commit. To obtain this we removed all the added logs from the modified logs.
3) Removed Logs: This includes all those logs which were deleted in a commit. Similar to new logs we removed all the deleted logs from the modified logs.

We use this data to answer the three RQ's in the next section.

III. STUDY RESULTS

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

RQ1: Are logs leveraged more during bug fixes?

Motivation: Prior research has shown that logs are used during bug fixing [7]. During debugging, developers update log statements, to gain more run-time information of the systems and ensure that future occurrences of a similar bug can be resolved easily with the updated information. However, to the best of our knowledge, there exists no large scale empirical study to show how extensively logs are leveraged during bug fixes. Moreover, little is known about how logs are leveraged during bug fixes.

Approach: We try to find if there is a difference between bug fixing and non-bug fixing commits with respect to log churn. To do this, we used the data sets obtained in previous section i.e modified, new and removed logs, and we calculated code churn for each data set. We used the total code churn of a revision to control the other metrics. The 3 new metrics are:

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

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

$$Removed\ log\ churn\ ratio = \frac{\#\ removed\ log\ churn}{total\ code\ churn} \quad (3)$$

To understand the different types of log modifications during bug-fixing-commits, we performed a manual analysis on the modified logging statements to identify the different types of log modifications. We first collected all the commits which had logging statement changes in our projects. We selected a random sample of 357 commits from all the commits with logging statement changes. The size of our random sample achieves 95% confidence level and 5% confidence interval. We followed an iterative process, as prior research [**?**], to identify the different types of logging modifications, until we cannot find any new types of modifications.

After we identify the types of log modifications, we created an automated tool to label log modifications into the four categories. We calculated the number of each type of log modifications in each commit and used *total code churn* as the controlling measure similar to equation 1 to 3.

To determine whether there is a statistically significant difference for these metrics, in bug-fixing and non-bug-fixing commits, we perform the *MannWhitney U test* (Wilcoxon rank-sum test) [13]. We choose *MannWhitney U test* because we our metrics are highly skewed (shown in table **??**) and as it

TABLE II
P VALUES AND EFFECT SIZE OF FOR COMPARISON . A POSITIVE EFFECT SIZE MEANS BUG FIXING COMMITS ARE LARGER. P-VALUES ARE BOLD IF IT IS LESS THAN 0.05.

Projects	Hadoop		Hbase		Qpid	
Metrics	P-Values	Effect Size	P-Values	Effect Size	P-Values	Effect Size
Modified Log Churn	**2.88e-4**	0.167(small)	**0.0353**	0.0886	**0.0281**	0.329(small)
New Log Churn	**0.00202**	0.0078	**0.00353**	0.134	**0.0032**	0.234(small)
Deleted Log Churn	0.087	-0.0455	**0.00489**	0.120	**0.00952**	0.042

is a non-parametric test, which does not have any assumptions about the distribution of the sample population. A p-value of ≤ 0.05 means that the difference between the two data sets is statistically significant and we may reject the null hypothesis (i.e., there is no statistically significant difference of our metrics in in bug-fixing and non-bug-fixing commits). By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us there is statistically significantly difference of our metrics in bug-fixing and non-bug-fixing commits.

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

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

where $\bar{x}_1$ and $\bar{x}_2$ are the mean of two populations, and s is the pooled standard deviation [**?**]. As software re-engineering has different thresholds for *Cohen's d* [14], the new scale is shown below.

$$Effect\ Size = \begin{cases} 0.16\ < & Trivial \\ 0.16 - 0.6 & Small \\ 0.6 - 1.4 & Medium \\ 1.4\ > & Large \end{cases}$$

Results: **Developers may add new logs more during bug fixes.** From table II, $new\ log\ churn\ ratio$ in bug-fixing commits is statistically significantly larger than non-bug-fixing commits in all subject systems but only Qpid has non-trivial effect size. This implies in some cases developers need to add more logging statements in some places in the source code. For new projects like Qpid, some important source code is not well logged. Therefore, developers may find that they need to add logging statements to assist in bug fixing. For mature projects such as Hadoop and Hbase, source code is well logged so, developers may focus more on improving existing logging statements rather than adding new logging statements.

Developers do not delete logs during bug fixes. We find that although $removed\ log\ churn\ ratio$ in bug-fixing commits is statistically significantly larger than non-bug-fixing commits in Hbase and Qpid, the effect sizes are trivial (see table II). Developers do not remove logging statements for fixing bugs. In Hadoop, we find logging statements are even removed more from non-bug fixing commits than bug fixing commits. Such results confirm the findings from prior research that deleted logs do not have a strong relationship with code quality [**?**].

TABLE III
DISTRIBUTION OF FOUR TYPES OF LOG MODIFICATIONS.

Projects	Hadoop (%)	Hbase (%)	Qpid (%)
Relocating	82.6	61.4	55.8
Text Modification	7.85	12.1	18
Variable Modification	7.9	8.4	12.5
Logging Level Change	3.85	5.4	13.6

Logs are modified more in bug fixing commits than non-bug-fixing commits. Table II shows that $modified\ log\ churn\ ratio$ is statistically significantly higher for all subject systems and the effect sizes are non-trivial in Qpid and Hadoop. Such results show that developers often change the information provided by logging statements to assist in bug fixing. Prior research that 36% of log messages are modified at-least once as after-thoughts [1]. Developers may find out that they need different information from logs to help them fix bugs. We find that the significancy and effect size of $modified\ log\ churn\ ratio$ is bigger than $new\ log\ churn\ ratio$, which implies that developers do not tend to provide additional information in logs but rather improve the existing logs. Prior research shows that too much information provided by logs may have become a challenge for developers to fix bugs [15]. Such finding may explain the reason why developer choose modifying logs over adding new logs.

From our manual analysis, we identified four types of log modifications. The distribution of the four types is shown in table III. The four types of changes are described below:

1) **Log relocation.** The logging statement is kept intact with only white space changes but moved to a different place in the file.
2) **Text modification.** The text printed from the logging statements is modified.
3) **Variable change.** One or more variables in the logging statements are changed (added, deleted or modified).

4) **Logging level change.** The verbosity level of logging statements are changed.

Developers modify variables more in bug fixing commits. We find that variable change is statistically significant more in all the subject systems and has small or medium effect sizes (see table IV). This implies that developers may modify the variables printed in their logging statements in order to provide useful information about the system to assist in bug fixing. To better understand how developers change variables in logging statements during bug fixing, we put the variable change into three types: variable addition, variable deletion and variable modification. From table V, we see that developers modify variables statistically significantly more in all projects with has non-trivial effect sizes. This implies that developers may not know what exact information is needed when they add the logging statements into the source code. The developers may realize the need of some information and modify logging statements to print the value of needed variables. Similar findings are presented in prior research that developers often have after-thoughts on logging statements [1]. Similar to the above finding that developers choose modifying logs over adding logs, developers also choose modifying variables in logging statements over adding more variables, due to the massive amounts of logs can be a burden for bug fixing.

Developers modify text in logs more during bug fixes. We find that text modification is statistically significant more in bug-fixing commits than non-bug-fixing commits with non-trivial effect sizes (see table IV). This implies that in some cases, the text description in logs are not clear and developers improve the text to understand the logs better to fix the bugs. For example, in commit 1,405,354 developers modify the logging statement to provide more information about the cause of an exception being raised. Prior research finds that there exists challenge of understating logs in practice [16]. Our results show that developers may have encountered such challenge and try to improve the text in logs for a better bug fixing.

We found log relocation is more in bug fixes. From table III, we see that there are a large number of logging changes that only relocates logging statements the code. Table IV shows that such relocation happens statistically significantly more in bug-fixing changes. We manually examined such commits and find that developers often forget to leverage exception handling or using proper condition statements in the code. After fixing the bug, developers often move existing logging statements into the *try/catch* blocks or after condition statements. For example, in the revision 792,522 of Hadoop, we see that the a logging statements are placed into the proper *try/catch* block.

Logging levels are not modified often during bug fixes. We find that logging level changes only happens statistically significantly more in Hadoop project. This implies developers typically do not change log levels during bug fixes. The reason of logging level not being changed during bug fix may be that developers are able to enable all the logging statements during debugging despite of what level a logging statement has. In addition, prior research shows that developers do not have a good knowledge about how to choose a correct logging level [1].

> *Developers change logs more in bug-fixing commits than non-bug-fixing commits. In particular, developers modified logs to change the variables in logging statements during bug fixes. Such results show that developers often realize the needed information to be logged and change the printed variables in logging statement to assist in fixing bugs.*

RQ2: Are logs useful in bug fixes?

Motivation: In our previous research questions we found that logs are modified more frequently in bug fixes. However, we cannot come to any inference about the usefulness of logs in bug fixes. In this RQ we try to understand the usefulness of leveraging logs during bug fixes.

Approach: To find the usefulness of logs in bug fixes we identified all JIRA issues of type 'bug' from the three subject systems. We obtained the code commits for each of these JIRA issues and identified log churn for each commit. We then split the JIRA issues into (1) bugs fixed with log changes (2) bugs fixed without log changes. We calculate the total code churn for each of the JIRA issues. We use the total code churn to measure the complexity of the issue. We then compare the total code churn of bug fixes with log changes, against bug fixes without log changes. We use *Wilcoxon* test to find the statistical significance and *Cohens.d* to measure the size of the difference.

We then parsed the JIRA issue files for each commit and extracted three metrics namely -

1) **Resolution Time:** This is the time taken from the time a bug is opened till its resolved. For example in JIRA if bug was reported in 1^{st} Feb 2015 and closed in 5^{th} Feb 2015 it means the time taken to fix the bug was 4 days. We used to measure how long it takes for a bug to be fixed.
2) **Number of Comments:** This metric gives the total number of comments are present in a JIRA issue. We obtained this by finding the number of comment id's present in the JIRA XML file. We used this measure how much effort is needed to discuss on fixing a bug.
3) **Number of Developers:** Every task in JIRA is generally assigned to particular developer and there number of viewers who are interested in the issue and help in resolving it. This metric gives the number of such unique developers who commented on the JIRA issue posts and were involved in resolving it. We obtained this by finding all the unique author names in the XML file. We used this as a metric as it shows human effort needed. More number of developers involved and commenting on a issue list, signifies more human effort spent resolving the bug.

TABLE IV
P-VALUES AND EFFECT SIZE FOR LOG MODIFICATIONS. P-VALUES ARE BOLD IF THEY ARE SMALLER THAN 0.05.

Projects	Hadoop		Hbase		Qpid	
Metrics	P-values	Effect Size	P-values	Effect Size	P-values	Effect Size
Log relocation	**1.69e-11**	0.260(small)	**6.33e-03**	0.2092(small)	**9.14e-08**	0.987(med)
Text modification	**7.75e-04**	0.153(small)	**2.94e-05**	0.308(small)	**4.68e-08**	0.531(small)
Variable change	**1.94e-06**	0.447(small)	**3.51e-04**	0.614(med)	**5.19e-05**	1.209(med)
Logging level change	**0.0057**	0.412	0.153	-0.05	0.341	0.396

TABLE V
P-VALUES AND EFFECT SIZE FOR THE DIFFERENT TYPES OF VARIABLE CHANGE. P-VALUES ARE BOLD IF THEY ARE SMALLER THAN 0.05.

Projects	Hadoop		Hbase		Qpid	
Metrics	P-values	Effect Size	P-values	Effect Size	P-values	Effect Size
Variable addition	0.22	-0.069	0.129	0.222	0.486	-0.152
Variable deletion	0.25	-0.114	0.585	0.165	0.22	-0.195
Variable modification	**0.047**	.221(small)	**0.0032**	0.268(small)	**1.61-05**	0.550(small)

TABLE VI
P-VALUES AND EFFECT SIZE FOR BUG FIXES

Projects	Hadoop		Hbase		Qpid	
Metrics	P -values	Effect Size	P -values	Effect Size	P -values	Effect Size
Total Churn	**2.2e-16**	0.563 (small)	**2.2e-16**	**0.168** (small)	**3.15e-08**	0.270 (small)
Resolution Time	**4.26e-03**	-0.145 (small)	**7.44e-14**	-0.167 (small)	0.0865	-0.119
# of Comments	**2.2e-16**	-0.507 (small)	**5.16e-11**	-0.289 (small)	**2.34e-03**	-0.227 (small)
# of Developers	**2.2e-16**	-0.577 (small)	**2.2e-16**	-0.538 (small)	**4.73e-02**	-0.375 (small)

To better understand the effect of log churn on 'Resolution Time', we create logistic regression model. Prior research has shown that resolution time is correlated to the number of developers and the number of comments in a issue report [17]. So we construct the following logistic regression model to predict the resolution time of bug fixes,

$$RT = \alpha.Dev + \beta.Comm + \gamma.New + \theta.Del + \delta.Mod \quad (5)$$

where α, β, γ, θ and δ are the coefficients. *'Dev'* is the number of developers involved in the fix, *'Comm'* is the number of comments, *'New'* is the *'New log churn'*, *'Del'* is the *'Deleted log churn'* and *'Mod'* is the *'Modified logchurn'*.

To measure the effect impact of each metric on the model we follow a similar approach used in prior research [18], [19]. We set all the metrics in the model to their means and find the predicted probabilities. Then we increase the metric of which we want to measure by one standard deviation value, while keeping the other metrics at their means. We then calculate the percentage of difference caused be increasing one of metrics by its standard deviation. A positive effect means a higher value of the factor increases the likelihood, whereas a negative effect means that a higher value of the factor decreases the likelihood of the dependent variable.

We would like to point out that our purpose is not to predict the resolution time of bug fixes but to understand the effect impact of log churn metrics on resolution time of bug fixes.

Results: **We found that the logs are used to fix more complex bugs**. From figure 2 we see that average code churn per commit is significantly higher for commits with log changes and has non-trivial effect sizes. This implies that when dealing with more complex bugs, developers may leverage logs more.

We found that bugs are easier to fix with logs. After controlling the churn, we found that given two bugs of same complexity the one with log changes takes lesser time to get resolved and needs lesser number of developers involved in the fix with less discussion. This implies that logs provide useful information for developers to discuss, diagnose and fix bugs easily. For example, logs are used in fixing issue HBASE-3074 (commit 1,005,714). In this JIRA issue we see the very first comment is to provide additional details in the logging message about where the connection manager fails. When we looked at the commit, we see that the developers add the name of the existing server which has gone stale in the logging statements. This additional data helps trace the cause of the failure and helps in the debugging process.

Using these metrics we built the logistic regression model to understand the impact of log metrics on resolution time. This shows the effect of logging statements on bug fixes.

We find that New Logs help in decreasing the resolution time of bug fixes. From table VII we see that 'New Log churn' are statistically significant in 2 of our subject systems and have negative effect on resolution time. This implies that new logs added during bugg fixing commit can help in reducing its resolution time.

Modifying logs has a negative impact on resolution of bug fix. We observe that from table VII, modified log churn has negative effect has negative effect in two subject systems and is statistically significant in Hadoop. This implies modifying logs helps in fixing bugs faster. We observe that

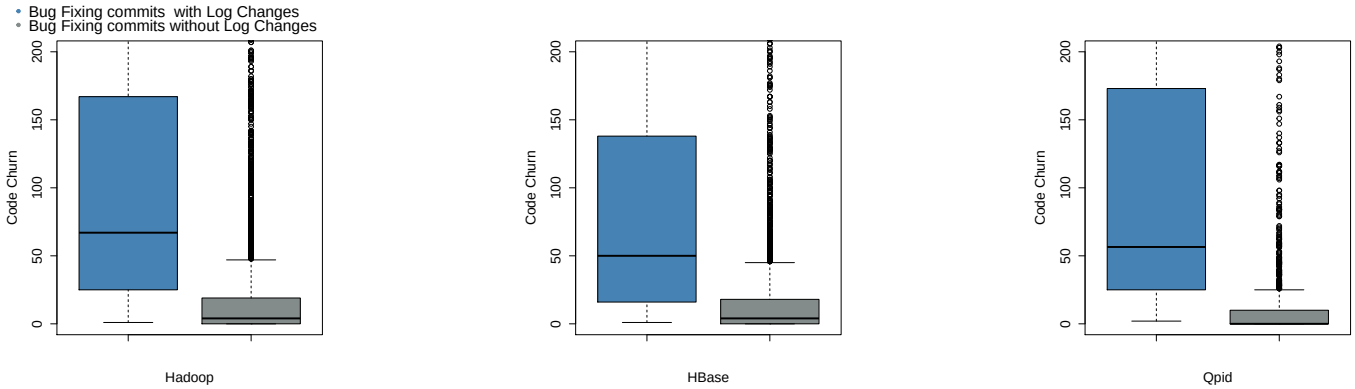

Fig. 2. Boxplot of Code Churn of Bug fixing commits with log changes(shown in blue) against Bug fixing commits without log changes(shown in grey)

TABLE VII
EFFECT OF LOG CHURN METRICS METRICS ON RESOLUTION TIME OF BUG FIXES. EFFECT IS MEASURED BY ADDING ONE STANDARD DEVIATION TO ITS MEAN VALUE, WHILE THE OTHER METRICS ARE KEPT AT THEIR MEAN VALUES. THE BOLD FONT INDICATES THAT THE METRIC IS STATISTICALLY SIGNIFICANT

Projects	Hadoop	Hbase	Qpid
New Log Churn	**-4.55** **	**-5.06** **	**-6.00** ◇
Deleted Log Churn	-3.24	-4.16	-2.37
Modified Log Churn	**-5.31** *	-3.7	-6.31

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, ◇ $p < 0.1$

modified log churn is not significant in Hbase. This maybe because developers do not modify logs in Hbase as seen from table II, where effect sizes are trivial for HBase.

We see that Deleted log churn has negative impact on resolution time of bug fixes but are not statistically significant. This may be because developers do not delete logs frequently as seen in our previous research questions.

Logs are leveraged during complex bugs and help in quicker resolution. Developers leverage logging statements to fix complex bugs. Bug fixes with log changes are resolved quicker with fewer people and less discussions. This implies provide useful information to fix bugs.

IV. MANUAL STUDY ON LEVERAGING LOGS BUG FIXES

From our research question we found that logs help in reducing the time and human effort necessary during the debugging process. We performed a manual analysis to find out where and in what scenarios logs are used. We first collected all the bug fixing commits which had logging statement changes in our projects. We then selected a random sample from all the issue reports (180 samples) with confidence level 95% and interval of 5%.

Q1. What often are logs leveraged in the debugging process ?

We found that 70.7% of the sample used logs directly for fixing the buggy issues. We looked into the JIRA discussion posts and we observer that over 128 of the reports, made use of log dumps, master log records or the developers in their comments mentioned logs to trace the problem. We observed that developers use logging statements to trace the bugs and even use the exceptions generated (example -QPID-2979). In the remaining 30% of the issue reports developers make use of exceptions (example HBASE-3654) or reproduce the issue (example QPID-4312) and fix them.

Q2. What type of information do developers look in logs?

61% of the reports (78) made use of both variable (dynamic objects) and textual part present in the logging statements to resolve the bugs. We observed that developers output the system state, the server/connection name, time, logging level, and even object names in the log lines. We found that these details are generally sought after by developers more during debugging process. For example - In HBASE-4797, developers provide a system generated log dump on JIRA. We observe that developers draw many conclusions using the logs present and this helps in resolving the bug. In this particular example, developers used the time-stamp, region-servers and the sequence-id's present in the logs. We also observed that these commits were associated with higher code churn implying, developers need more information to fix more complex problems.

39 % of the reports used only the log message themselves to diagnose the problem. We observed that majority of these bugs (38 commits), were caused by the logs themselves. These issues were either trivial from typos (example - commit 1333321), haphazard logs (example - commit 1333321) to complex issues degrading performance (example - commit 999046).

***Findings**: 70% of the issues leverage logs during bug fixes, out of which 61% of the issues leverage both textual and variable parameters in a logging statement.*

V. Related Work

In this section, we present the related work of this paper. In particular, we present the prior research that performs log analysis to for large software systems and empirical studies on logs.

A. Log Analysis

Prior work leverage log analysis for testing and detecting anomalies in large scale systems. *Shang et al.* [20] 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.* [10] propose an approach to leverage 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 a sign of anomalies. *Fu et al* [21] built a Finite State Automaton (FSA) using unstructured logs and to detect performance bugs in distributed systems. *Xu et al* [3] link logs to logging statements in source code to recover the text and and the variable parts of log messages. They applied Principal Component Analysis (PCA) to detect system anomalies. Tan et al. [22] 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. [23] 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. [24], [25] 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*. [26] propose an approach to automatically infer the failure scenarios when a log is printed during a failed run of a system.

Jiang et al. [27], [28], [29], [30] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [27]. Based on the such events, they identified both functional anomalies [28] and performance degradations [29] in load test results. In addition, they proposed an approach that leverage logs to reduce the load test that are performed in user environment [30].

The extensive prior research of log analysis motivate our paper to study how logs are leveraged during bug fixes. Our findings confirms that logs are widely leveraged during bug fixes and the use of logs assists developers in a faster resolution of logs with fewer people and less discussion involved.

B. Empirical studies on logs

Prior research has performed empirical studied on logs and logging characteristics. 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.* [31] performed an empirical study on where developer put logging statements. They find that logging statements are used for assertion checks, return value checks, exceptions, logic-branching and observing key points. The results of the analysis was evaluated by professionals from the industry and F-score of over 95% was achieved.

Shang et al. [32] signify the fact that there is gap between operators and developers of software systems, especially in the leverage of logs. They performed an empirical study on the evolution both static logging statements and log lines outputted during run time [7], [33]. 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* [16] 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. *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.* [34] shows that logs need to be improved by providing additional information. Their tool named *Log Enhancer* can automatically provides additional control and data flow parameters into logs.

The most related prior research by Shang *et al*. [7] empirically study the relationship of logging practice and code quality. Their manual analysis sheds some lights 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 leveraged during bug fixes. Our results show that logs are leveraged extensively during bug fixes and assist in a quick resolution of bugs.

VI. Limitations and Threats to Validity

External Validity: In this study we found that logs are leveraged during bug fixes more than any other type of changes. We looked at three projects from the Apache Software Foundation namely Hadoop, HBase and Qpid. This is also limiting factor as these projects are all Java based and we have not looked into other programming languages.

Internal Validity: In our initial research questions, we tried to find if there is relation which proves logs getting changed implies a high probability of bug fix or not. This was shown to be true and we deduced that logs are in-fact changed more often during bug fixes than other types of changes. But there can be instances where logs are used in bug fix, but they are not changed as they provide sufficient information. In such cases its impossible to determine if logs were useful or not.

Although our study shows presence of logs reduces time of resolution, number of developers and number of discussions

we do not claim any causal relationship. There are many other contributing factors and further analysis is necessary.

Construct Validity: When correcting the data, we found that some of the revisions were only related to branching or merging . To eliminate this we looked at all SVN commits and removed branching commits from our data set. We also used keywords to find branching and merging commits. Such keyword based filtering may not be entirely correct. Although prior research uses similar approach, better ways to filter branching and merging commits are needed.

The other major limitation is user defined logs in the Java Code. When searching for log lines we looked at specific patters in the files. These patters were 'Log', 'Logger', 'LOGGER', 'LOG', 'log' and few other variants. But the users can define their own names for these statements and it would make it impossible to search for such user defined functions in the change commits.

VII. Conclusion and Future Work

Logs are used by developers for finding anomalies, monitor performance, software maintenance, capacity planning and also in debugging errors. Our paper presents a empirical study of log leverage during bug fixes. Using three subject systems (Hadoop, Hbase and Qpid) we understand how logs are leveraged, the different types of log changes developers make and if logs are useful in the debugging process. We find that logs are leveraged more during bug fixing commits by developers. In particular we find logs are modified and new logs are added during bug fixing commits. This means existing logs do not convey all the information for fixing the bug and have to be modified or new logs added. We find four different types of log modifications namely change to 'Text Modification', 'Variable Changes', 'Log Relocating' and 'Logging Level Changes'. Of these four types, we find developers modify variables more. This implies that developers require more specific data to fix bugs and leverage logs to collect it. We also find that developers leverage logs during complex bug fixes, where complexity is measured using total churn of a commit. We find that bug fixes which leverage logs are resolved quicker, need fewer developers and less discussions. From the logistic regression model built we find that adding new logs reduces the resolution time of bug fixes.

Finally we did a manual analysis were we find developers use logging statements to understand the rationale behind logs. We find in over 70 % of the bug fixing issues, developers used logs in the debugging process. We also find that in over 61 % of the bug fixing issues, developers leverage both the contextual and variable information in logging statements to fix the bug.

Though its know logs are useful, our study presents the first empirical study on log leverage during bug fixes. Our study demonstrates the importance of logging during bug fixes and shows the need for more specific information from logging statements.

References

[1] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 102–112. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337236

[2] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 588–597. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2009.19

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 117–132. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629587

[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 *Proceedings of ICSM 2013: the 29th IEEE International Conference on Software Maintenance , Year = 2013, Month = Sept, Pages = 110-119, Doi = 10.1109/ICSM.2013.22, ISSN = 1063-6773, Keywords = .*

[5] L. Bitincka, A. Ganapathi, S. Sorkin, and S. Zhang, "Optimizing data analysis with a semi-structured time series database," in *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, ser. SLAML'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1928991.1929002

[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. [Online]. Available: http://dx.doi.org/10.1002/smr.1579

[7] W. Shang, M. Nagappan, and A. 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. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9274-8

[8] Xpolog, "http://www.xpolog.com/."

[9] logstash, "http://logstash.net."

[10] J.-G.-L. Q.-F.-S. Y. Y-Xu and J-Li., "Mining invariants from console logs for system problem detection." in *In Proc. of 2010 USENIX Annual Technical Conference(ATC 10).*, 2010.

[11] 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.

[12] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, 1966.

[13] E. A. Gehan, "A generalized wilcoxon test for comparing arbitrarily singly-censored samples," *Biometrika*, vol. 52, no. 1-2, pp. 203–223, 1965.

[14] 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.

[15] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 249–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685068

[16] U. L. L. U. D. Knowledge, "Weiyi shang, meiyappan nagappan, ahmed e. hassan, zhen ming jiang," in *The 30th International Conference on Software Maintenance and Evolution (ICSME 2014)*, 2014.

[17] P. Anbalagan and M. Vouk, "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.

[18] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 300–310.

[19] A. Mockus, "Organizational volatility and its effects on software de-

fects," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 117–126.

[20] 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 *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 402–411. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486842

[21] Q. F. J. L. Y. Wang and J. Li., "Execution anomaly detection in distributed systems through unstructured log analysis." in *In Proc. of 9th IEEE International Conference on Data Mining (ICDM 09)*, 2009.

[22] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Salsa: Analyzing logs as state machines," in *Proceedings of the First USENIX Conference on Analysis of System Logs*, ser. WASL'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 6–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855886.1855892

[23] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou, "Understanding customer problem troubleshooting from storage system logs," in *Proccedings of the 7th Conference on File and Storage Technologies*, ser. FAST '09. Berkeley, CA, USA: USENIX Association, 2009, pp. 43–56. [Online]. Available: http://dl.acm.org/citation.cfm?id=1525908.1525912

[24] 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. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025151

[25] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568246

[26] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736038

[27] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *J. Softw. Maint. Evol.*, vol. 20, no. 4, pp. 249–267, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1002/smr.v20:4

[28] Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, Sept 2008, pp. 307–316.

[29] ——, "Automated performance analysis of load tests," in *In Proceedings of ICSM 2009: the 2009 IEEE International Conference on Software Maintenance*, Sept 2009, pp. 125–134.

[30] 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 *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, ser. SSIRI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 131–140. [Online]. Available: http://dx.doi.org/10.1109/SSIRI.2010.15

[31] 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 2014 Companion Proceedings of the 36th International Conference on Software Engineering*. ACM New York, NY, USA 2014 ISBN: 978-1-4503-2768-8 doi¿10.1145/2591062.2591175, 2014-05-31 2014, pp. Pages 24–33.

[32] W. Shang, "Bridging the divide between software developers and operators using logs," in *Software Engineering (ICSE), 2012 34th International Conference on*.

[33] 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. [Online]. Available: http://dx.doi.org/10.1002/smr.1579

[34] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 4:1–4:28, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2110356.2110360