An Empirical Study On Leveraging Logs During Bug Fixes

Suhas Kabinna
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 2N8
Email: kabinna@queensu.ca

Weiyi Shang
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 2N8
Email: swy@queensu.ca

Ahmed E. Hassan
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 2N8
Email: 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 3 open source systems like Hadoop, HDFS and Qpid from the Apace Foundation. We found that logs are statistically significantly more changed in bug fixes than other code changes. We find 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 add more variables to log 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. This demonstrates the importance logs play 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

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

Logs are widely used in practice. Prior research has shown that logs are leveraged for detecting anomalies [1], [2], [3], monitoring performance of systems[4], maintenance of large scale systems [3] and debugging [8]. The valuable information in logs leads to a new market for log maintenance platforms like Splunk[4], XpoLog [5], Logstash [6]which assist developers in analyzing logs.

Logs are used 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.

Research has shown that logs are used by developers extensively during the development of software systems [7]. Prior research performs a manual study on bug fixing changes. The results show that 41% of the log changes are used to field debugging[8]. 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, adding and modifying logs is statistically significant during bug fixing commits than non-bug fixing commits, with non-trivial effect size. This suggests that developers leverage logs during bug fixes.

RQ2: What types of modifications to logs are more frequent during bug fix?

We manually identified four types of log modifications from our case studies. The 4 types of changes are *'Logging level change', 'Text Modification', 'Variable Modification' and* 'Relocating logs'. We found that *'Text Modification', 'Variable Modification' and* 'Relocating' are statistically significant, with medium to high effect sizes in bug fixing commits. This shows that developers believe that adding more information to logs can help fix bugs.

RQ3: 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.

The rest of this paper is organized as follows. Section 2 is the related work in field of logging, Section 3 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 4 contains the details case studies and the results we obtained. Section 5 will contain the limitations of our study and any threats to validity. Section 6 will contain further work we intend to do and will conclude the paper.

II. Related Work

In this section we look at (1) the different usage of logs in improving software quality, (2) the tools which help in logging (3) the manual sampled studies done so far on analysis and characterization of logs, so we can compare the results from our empirical study.

Log analysis:

The purpose of this section is to highlight the research that shows logs are used in debugging process. This is related to our work because (1) we show that logs are used in all the processes of a software life cycle, (2) how logs are useful in the improving the quality of the software.

Most prior work on log analysis focuses on studying the usefulness of logs during testing and detecting anomalies in large scale systems. *Shang et al.*[10] showed mining execution logs in Big Data Analytic (BDA) applications, helps in finding differences in behavior of the system under different tests. Similar work has been done to show how mining execution logs in large systems can help detect anomalies and ensure good load tests *Malik et al*[11]. *Lou et al*[12] discuss how invariants, mined from console logs can help to detect anomalies in large systems. *Fu et al*[13] shows analysis of unstructured logs helps in detecting performance bugs in Distributed systems. Tools have also been developed to diagnose failures by mining execution logs *Tan et al*[14].

Work has been done on mining execution logs in Big Data Analytic (BDA) applications to find difference in behavior of the system, when tests are run on small testing data and actual real world data [10]. Similar work has been done to show how mining execution logs in large systems can help in detecting anomalies and ensure good load tests [11]. In the paper execution logs are mined and event pair are formed. By finding patterns in these event pairs dominant behavior of the system is identified. Any event which does not follow this is tagged as anomaly and can be reported to the developers. These papers show that logs are not used only for the purpose of debugging but can assist a developer in a variety of tasks.

Research has been to detect anomalies by invariant mining[12]. This paper focuses on constructing structured logs from console logs. After construction and grouping of log messages, invariants are mined. Any log which violates a invariant is considered to be an anomaly. Unstructured log analysis has shown to be useful in detecting in even Distributed Systems Several [13]. In the paper a finite state automaton is built and trained to automatically detect performance bugs in the system. These papers consider print messages and other unstructured logs to extract information. Then they cluster or categorize the logging messages into groups and logs which fall into this cluster are treated as anomalies. Tools have also been developed to analyze system logs to obtain state-machine view, control and data flow models and other related statistics. SALSA[14] helps in deriving failure diagnosis techniques and visualization for different workloads in Hadoop project by mining execution logs. In our study, though we use the same system software (Hadoop),we try to understand how logs can be useful in debugging and not on trying to find the bug itself.

Tools for logging:

In this section we try and find out how developers log. Is it done manually or do developers use established logging libraries. From our work on studying logging libraries in the Apache foundation we know that Slf4j, Log4j and Jakarta common logging are the most used libraries for logging. This helps us to find the logging statements in our case studies. Additionally there are many tools which help developers in logging [15]. The authors conduct a manual analysis and find that logs help in understanding the control and data flow in bug fixes but do not contain all the information to conclusively answer why the bug occurs. The paper addresses this issue through a tool called 'Log Enhancer' to enhance existing logs by providing additional details in the logs. This was tested with earlier version of the software and it was observed that 95% of logs added by the tool were added by developer's overtime. This paper highlighted that do not log all the details in the first deployment and modify the logs in later revisions. This conforms to our findings as well, as we found that developers modify logs during debugging than other software process.

Studies on logs:

We finally look at other research work where logging practices have been analyzed to find where logs have been useful. Logs have been characterized in open-source systems and it has been shown that developers take multiple attempts to write correct log messages[7]. In this this paper, authors found that logs are actively maintained, they are changed more often and can help in diagnosing production-run failures. Our paper explores similar aspects, but we try to understand how logs are useful in debugging process in particular.

Similar to the previous paper research has been done to study the logging practices in Industry [16]. This paper identified the most common places developers places logs in Industry. They found 5 places where developers use logs and found the reasoning behind these decisions. The results were evaluated by professionals from the industry and F-score of over 95% was achieved in predicting where logs are written.

There has been substantial work in also trying to bridge the gap between operators of the software system and developers [17]. This is crucial as logs act as mediators in this regard, as they help in providing the operational information for the mediators and for developers it helps them track the system flow. Research has also been done to understand the needs of operators and administrators [18]. In this paper, by manually analyzing email threads in three large open-source projects,

TABLE I
OVERVIEW OF THE DATA

Projects	Hadoop		Hbase		Qpid	
	Buggy	Non-Buggy	Buggy	Non-Buggy	Buggy	Non-Buggy
# of Rev	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

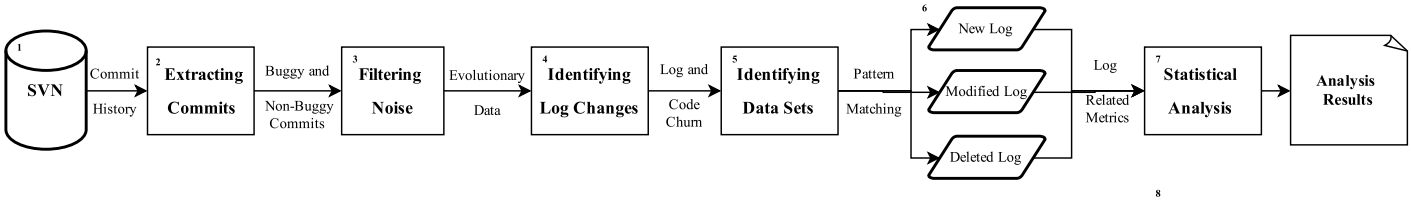

Fig. 1. Overview of our cast study approach

the authors find 5 different types of development knowledge that are sought by practitioners from logging statements.

There has been study also done on Hadoop and Darkstar file systems where console logs are mined for identifying performance and system bugs [2]. This paper looked into two large scale systems and parsed over 24 million lines of logging data. It utilized both source code parsing and console parsing to construct meaningful log messages. Then using Principal Component Analysis (PCA) they detect the anomalies and provide visualizations. This is similar to our work because, we also use source code parsing in finding the logs. But instead of trying to build structured logs we try to classify them using levenshtein distances.

Other studies have looked at the how logs change in large scale systems and how the information conveyed by these logs change over time [19]. In this paper the author explores how communicated information through through logs are changed often and 40%-60% can be avoided. It also shows, that CI with implementation details are changed more often.

We also know logging practices can also assess the code quality of platform software [8]. This paper showed that logging characteristics can help in predicting defect prone modules in software, as developers tend to log more when they feel a module has a higher chance to be prone to bugs. This paper looked at Hadoop and Jboss and found log related metrics complement traditional metrics and increase the explanatory power of defect proneness by 40%. The other imortant aspect of this paper is the manual analysis done to identify the different reasons being logging statements. We did similar manual analysis in our paper to find the types of logging changes that occur (section 4.2) and to understand the different usage of logs in debugging (section 5).

All this prior work tries to look at usage of logs from different perspectives but none of them attempt to measure if logs are actually helpful in the development process, especially in debugging. Some of the work is done through manual analysis and has not been done on different systems to verify the results.

III. METHODOLOGY

In this section, we describe our method for data-gathering to answer our research questions and some of our assumptions when analyzing the data.

A. Subject setup

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 projects in Apache Software Foundation. We found that Hadoop, Hbase and Qpid have extensive logging and best suit our case study.Table 1 highlights some of the statistics of these 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. We used Hadoop releases 0.16.0 to 2.0.

HBase[2]: Apache HBase is a distributed, scalable, big data store using Hadoop database. We used Hbase releases 0.10 till 0.98.2.RC0. This covers more than 4 years of development in Hbase from 2010 till 2014.

Qpid[3]: Qpid is an open source messaging system which implements Advanced Message Queuing Protocol. Qpid we used

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

all data from 0.10 till 0.30 release. This covers development from 2011 till 2014.

We used Apache software versioning system, SVN to collect the commit history for all the projects in our case study. We used JIRA to collect the buggy reports for all the projects in our case study. We extracted the reports in XML format for parsing convenience.

B. Data Collection and filtering

We used SVN to study the evolution of Java source code in the 3 projects, similar to C-REX [20] and J-REX [8]. Through SVN 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 data extraction is to separate the buggy and non-buggy commits. To do this we extracted the list of all commits from SVN and the commit messages from the developers. Next we extracted a list of all JIRA issues related to bug fixes. As developers mention the JIRA issue numbers in the commit messages[1], we matched the commit messages against the JIRA issues to identify all the defect fixing changes. We also included commits which contained words like 'fix' or 'bug' as prior research has shown that these heuristics identify bug-fixing revisions with high accuracy [21].

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 all such commits from both the datasets. To achieve this we wrote scripts which checks if a particular commit file has '.java' files and calcualtes churn for those files only.

After calculating total churn for each revision, we found that some commits have high code churn because of branch and merge operations (commit-952410). To filter such noisy data, we consider only those revisions which have added churn and deleted churn more than 5% of total churn. In the commit above the deleted churn is 0 so we exclude this revision from our study. We also use the change-list file to remove all the branching commits. In a change-list file all commits which are related to branch or merge operations fall under 'BRANCH_SYNC' category. All such commits are excluded from our data-set. Even after this we observed that some commits have churn values over to 30,000. Most of these commits were Improvements to the projects (revision 1,334,037) and cannot be excluded.

3) Identifying Log changes : To identify the logging 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 -

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 -

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

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 these projects logging statements have method invocation like "LOG", followed by log-level and other information. We count every such invocation as a logging statement.

C. Identifying data sets.

After identifying the logging statements in each commit, we found two types of log changes namely

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

Deleted Log: This 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 [2] . A pair of added and deleted logging statement is said to be a modification, if the levenshtein distance of lesser than 5 or ratio of higher than 0.5. 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 this categorization we obtained 3 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) Deleted 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.

Using this new data set we can answer the question whether logs are leveraged during bug fixes more.

IV. 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.

RQ 1: Are logs leveraged more during bug fixes?

Motivation: Prior research has shown that logs are used in the debugging process. When debugging developers update logging statements also, so future occurrences of a similar bug can be resolved easily with the updated information in the log statements. However, there has been no large scale empirical

[1]SVN Commit 1162209 - http://svn.apache.org/viewvc?limit_changes=0&view=revision&revision=1162209

[2]http://xlinux.nist.gov/dads/HTML/Levenshtein.html

Fig. 2. P values and Effect Size of for comparison . A positive effect size means bug is larger

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

study to show how extensively logs are leveraged in debugging or, has there been a comparison against other developmental activities.

Approach: Its intuitive that logs are generally modified when bugs are fixed. This is because prior research has already proved that a module that has been modified continually, has higher chance of having bugs than modules which have not been modified [22]. So, 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 Deleted logs, and we calculated churn for each data set. We used 'Total Churn' of a revision to control the other metrics. The 3 new metrics are:

$Modified\,Churn = Modified\,Log/Total\,Churn$

$New\,Churn = New\,Log/Total\,Churn$

$Deleted\,Churn = Deleted\,Log/Total\,Churn$

To measure the statistical significance of these metrics, in buggy and non-buggy commits we use the *MannWhitney U test* (Wilcoxon rank-sum test)[3]. *Wilcoxon test* gives p-value as the result. A p-value of ≤ 0.05 means that the test was statistically significant and we may reject the null hypothesis (i.e., the two populations are different). By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us if one population is statistically significantly larger than the other. We use *Wilcoxon test* because we our metrics are highly skewed and as it is a non-parametric test, the distribution of population does not factor.

We also use *effect sizes* to measure how big is the difference between the bug and non-bug fixing commits. Unlike *Wilcoxon-test*, which only tells us if the differences of the mean between 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 [23]. *Cohen's d* measures the effect size statistically, and has been used in prior engineering studies. *Cohen's d* is defined as:

$$\textit{Cohen's } d = \frac{\bar{x}_1 - \bar{x}_2}{s}, \tag{1}$$

where $\bar{x}_1$ and $\bar{x}_2$ are the mean of two populations, and s is the pooled standard deviation [24]. As software re-engineering has different thresholds for *Cohen's d* [9], 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: **Logs are modified more in bug fixes than other developmental tasks** (see figure 2). This is because developers often think a lot before adding new logs, and modifying existing log to provide additional information is more convenient. Research has shown that 36% of log messages are modified at-least once as after-thoughts [7]. This means that in most cases the existing logging messages do not convey all the information necessary for a bug fix. An example of this type of change is shown below:

+ log.trace("setConnectionURL(" + Util.maskUrlForLog (connectionURL) ")");

- log.trace("setConnectionURL(" + connectionURL + ")");

Developers add new logs more during bug fixes. We observe that new Logs have effect sizes higher than 0.13 in Hbase and Qpid. This implies in some cases developers completely ignore logging during initial development and have to spend more time starting from scratch. This is an extra effort which can be avoided by good logging practices.

Developers do not delete logs during bug fixes. We observed that in all 3 projects the effect sizes are smaller than 0.13. This implies that in both buggy and non-buggy commits, developers do not remove logging statements. Prior research has shown this is because developers only delete log lines only when confident about their source code [8]. In Hadoop we observe that the effect size is negative which implies logging statements are removed more from non-bug fixing commits than bug fixing commits.

> ***Finding****: Developers log more during bug fixes than other types of changes.*
> ***Implication:*** *Developers modify logs more during the debugging process. This implies that, developers have fair idea which modules can lead to bugs, so they write logging statements in those modules to assist them. But as these logs do not convey all the information necessary so they are modified later by the developers*

[3]http://www.ime.unicamp.br/ dias/Ch10.wilcoxon.pdf

TABLE II
DISTRIBUTION 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

RQ 2: What types of modifications to logs are more frequent during bug fix ?

Motivation: From RQ 1 we found that logs are modified more during bug fixes. However, as this information is not sufficient to understand the usefulness of logs in debugging process; in this RQ we study the different types of modifications done to logs. This will provide more information and insight into how different types logging changes assist in the debugging process.

Approach: We performed a manual analysis on the modified logging statements to identify the different types of changes which occur. We first collected all the commits which had logging statement changes in our projects. We then selected a 5% random sample from all the commits with logging statement changes. We then follow a iterative process [25] to identify the different types of logging changes, that developers make in the source code till we cannot find any new types of changes. We identified 4 different types of log changes and their distribution is shown in table 2. The four types of changes are described below:

Relocating : In this only white spaces are changed or the logging statement is kept intact but moved to a different place in the file.

Text Modification : In this type, message part of log is modified, keeping the variables constant.

```
- Logger.warn( " Sample Text Goes Here " + printAVariable );
+ Logger.warn( " New Sample Text Goes Here " + printAVariable);
```

Variable Modification : In this type, variable part of log is modified , keeping the text constant.

```
- Logger.warn( " Sample Text Goes Here " + printAVariable );
+Logger.warn( " Sample Text Goes Here " + printAVariable + printBVariale);
```

Logging Level Change : In this type only change is the log level change.

```
- Logger.warn( " Sample Text Goes Here " + printAVariable );
+ Logger.debug(" Sample Text Goes Here " + printAVariable );
```

Using *Levenshtein* measures , we automated the process of categorization into the four categories. This is done by parsing all the changed logging statements in each commit and comparing to the 4 categories found. We also obtained the distribution for each type of change as show in table 2.

After categorization, we found the churn for these categories and used *total churn* as the controlling measure. The 4 metrics obtained are-

1) **Relocation Churn:** As explained above we categorized all the logging statements into 4 types. This metric counts the total number of logs of type 'Relocating' were present in each commit.
2) **Text Modification Churn:** This metric is the aggregate of log changes of type 'Type Modification' are present in each commit
3) **Variable Modification Churn:** This metric is the aggregate of log changes of type 'Variable Modification' are present in each commit.
4) **Logging Level Churn:** This metric is the aggregate of log changes of type 'Logging Level Change' are present in each commit.

After finding the churn in each of these categories we tried to find how significant are these compared to the non bug fixing commits. We consider only those commits which have log churn to do this comparisons. The total churn is used as a controlling measure. We used *Wilcoxon* test and *Cohens.d*, to measure the significance of bug fixing commits.

Results: **We found that developers modify variables more in bug fixes.** From table 3, we observe that variable modification has the highest effect size among all the types of log changes and is statistically significant in all 3 projects. This implies that developers modify the parameters in their logs,to provide more information about the system. This is useful because if any other bugs originate in that module, the additional data can help in easy resolution. It also means that developers do not provide all the details the first time logs are written and update them as after-thoughts.

We found that Text Modification is more in bug fixes. From table 3, we see that textual modification is significant in all the projects in our case study. This implies that developers value more in providing contextual data to an existing log rather than writing new logging statements.

We found log relocation is usually done when the same log statements are moved/copied to multiple places as part of code change. We looked at over 100 commits across all three projects to understand this behavior. We observed that in majority of the cases, developers use logging statements before try, catch blocks or do not check all the conditions. When a bug is raised the logs help in fixing the issue, but along with the fix they place the original logging statement in a try catch block and handle further exceptions properly.For example in

TABLE III
P-VALUES AND EFFECT SIZE FOR LOG MODIFICATIONS

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

792,522 revision of Hadoop, we see that the log messages are not placed in the appropriate 'try', 'catch' blocks. We also see that in some parts of the code the exceptions are not even present and later added by developers. In this commit the same logging message is copied in multiple places and hence its falls under relocation category.

From table 2, we see that this form of changes a major bulk of logging changes that occur in both data sets. This shows us that developers when writing code omit safe coding practices and do not use try, catch and finally blocks in their code.

Another example of this 1,042,282 revision in which a log statement is moved from the beginning of code block to the end of block. From our manual study (section 5) we found that this is mostly done to improve the readability or increase the performance. For example in, HBASE-4288 the logs are moved into new blocks, so they are printed only when 'Trace/Debug' option is enabled.

We observe that log level changes has P-values less than 0.05 only for Hadoop project. We also see that in table 2, logging level changes are the least among all the types of logging changes. This implies developers do not leverage log levels during bug fixes. This can be because -
1. Developers do not know what logging level is correct
2. During bug fixes they enable all levels i.e Debug and Trace, so they do not know which levels are incorrect.

> ***Finding****: Variable modification is more significant when compared to other types of log modifications*
> ***Implication****: Developers believe adding more parameters/variables to logging statements, helps in the debugging process. This shows that logging statements evolve continually and can change multiple times across revisions. It also means developers can spend more time or make use of the logging tools present to log more in first commit so subsequent changes are not necessary and debugging is faster.*

RQ 3: 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 from this data. So, in this RQ we try to find correlations between log leverage and bug fixing efficiency (ie. time taken, developer involvement), to understand the usefulness of leveraging logs during bug fixes.

Approach: To find the usefulness of logs in bug fixes we collected a new data-set different from previous research questions. We identified all JIRA issues of type 'bug' from the 3 subject systems. We obtained the code commits for each of these JIRA issues and calculated the code churn for all the commits. Using the log churn metric we split our data into two sets - (1) bug fixes with log change (2) bug fixes without log change.

After obtaining the data sets we parsed the JIRA issue files for each commit and extracted 3 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 01-2-15 and closed in 05-2-15 it means the time taken to fix the bug was 4 days. We used this because this is a very basic measure and tells us 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 as it gives information about how complex issues are. If its a simple issue, it wont be discussed and will be resolved quickly. But, if its a complex bug there will be more discussion posts and takes longer to be resolved.
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.

Finally we matched this JIRA metrics without our two data segments based on JIRA issue numbers. The statistics of the data sets are show in table 5. We used *Wilcoxon* test and *Cohens.d* to measure the difference between the data-sets.

Results: **We found that the logs are used to fix more complex bugs, where code churn is a complexity measure**.

TABLE IV
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**	0.093	**3.15e-08**	0.270(small)
Resolution Time	**4.26e-03**	-0.145(small)	**7.44e-14**	-0.167	0.0865	-0.119
No of Comments	**2.2e-16**	-0.507(small)	**5.16e-11**	-0.289	**2.34e-03**	-0.227(small)
No. of Developers	**2.2e-16**	-0.577(small)	**2.2e-16**	-0.538	**4.73e-02**	-0.375(small)

Higher the churn implies the bug is more complex to fix. From table 4,5 we see that average churn per commit is several magnitudes higher for commits with log changes and is statistically significant with non-trivial effect sizes. This can be because (1) The information provided by existing logs is insufficient and hence they modify the logs to get additional data. (2) Developers are adding new methods or function and hence add logs to track the control and data flow in the new blocks.

Because total churn is correlated to other metrics in our data set, we used it as our controlling measure, similar to the previous research questions.

TABLE V
OVERVIEW OF THE DATA SETS WITH CODE CHURN

Projects	Metrics	With Log Change	Without Log Change
Hadoop	Average Churn	371	41
	Average Time	62 days	48 days
	Average comments	20	16
	Average users	7	6
HBase	Average Churn	692	295
	Average Time	36 days	26 days
	Average comments	27	19
	Average users	6	6
QPid	Average Churn	1,193	145
	Average Time	96 days	64 days
	Average comments	5	4
	Average users	3	2

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 churn takes lesser time to get resolved and needs lesser number of developers involved in the fix. Because there is less developers involved in the discussion, the number of discussions posts/comments on JIRA is also less. This is because when logs are leveraged, its easier to debug the problem so resolution time is less. Developer involvement is also less because, the bug can be traced and fixed easily and long discussions about root-cause analysis is avoided.

An example of this where logs are used is in the JIRA 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.

***Findings**: Logs are leveraged during complex bugs and help in quicker resolution.*
***Implication**: From table 4 and 5 we see that developers change logging statements during complex bug fixes, with more code churn. But when code churn is controlled, bug fixes with log changes are resolved quicker. This implies logs help developers in debugging faster and need less number of developers involved in the fix.*

V. MANUAL STUDY ON BUG FIXES

From our research question we found that logs help in reducing the time and human effort necessary during the debugging process. To validate our case study, we did 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 5% random sample from all the commits(180 samples) . We then followed a iterative process [25] to identify the different scenarios where logs are leveraged during the fix and why they are changed.

A. Analysis Results

From this manual study we tried to answer two important questions:

Q1. What percent of bugs directly leverage logs 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. It should be noted that in some of the JIRA discussions, developers write ' Let me trace/re-create the problem ' and in the next post they provide a list of conditions which causes the bug. In such posts we cannot know if logs are leveraged.

We observed that 34.3% of the samples (35 commits), were cases in which the logs themselves were bugs. These issues were either trivial from mis-spelt words , haphazard logs to complex issues were logs cause performance issues.

Example- In 'HBASE-5081' we see that a log-splitting in function 'deleteNode' forms a race condition and hangs.

Q2. What type of knowledge/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. Example - In HBASE-4797, developers actually provide an extract of logs generated 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.

48 % 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. In such cases, logs have grammatical errors (example - commit 1333321), are not placed within proper blocks (example - commit 1236977) or cause performance bugs (example - commit 999046). In all these cases, the presence of log is sufficient enough information to debug the problems and developers do not need additional information.

> ***Findings****: 61% of buggy commits made use of both textual and variable part in logs and average churn was 244 LOC (lines of code) to 160 of the latter.*
> ***Implication****: When there is more complex bug to be solved, developers use both the textual and variable (parameters) to solve the problems. This tells us that developers try to record more information, to see where fault occurs. This additional information is used in understanding the specific set of conditions which triggers the bug (rational behind the bug).*

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 haven't looked into other programming languages.

The other limitation is projects in which logging data is less. We ran experiments on several other projects like Cassandra, Cayene, Zookeeper and Lucene. In all these projects the total number of logging statements were less to draw any meaningful conclusions. When we manually examined the data for Cassandra and Lucene we observed many custom logging statements which were not caught from pattern matching. As stated above its almost impossible to catch all instances of these custom logs in projects.

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, we do not claim any causal relationship between the two.There are many other contributing factors which need to be considered when trying to predict resolution time for bugs. The purpose of our study was to demonstrate that logs are indeed useful and motivate developers to log more during initial development.

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 JIRA commits which has type Branching in them and removed these branching commits from our data set. But we later found some of the commits are tagged under the category of UNKNOWN. When its under 'Unknown' category it was impossible to know if its branching or not.

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

In this work we looked at Apache Hadoop, Hbase and Qpid to find out if logs are more useful in bug fixes or not.We first try to establish how we are going to measure this by checking for log modifications that are part of a bug fixing commit. In our initial research questions we find that logs are modified and added during bug fixes more than other types of developmental tasks

We identified 4 different types of logging changes namly 'Text Modification', 'Variable Modification', 'Restructuring' and 'Logging Level Changes'. We saw that 'Variable Modification had the highest effect size in all projects, followed by 'Restructuring' and 'Text Modification'. This showed that developers tend to add or delete or change variables more often during bug fixes. We also saw that developers tend to change the position of logging statements or change the text in logs more frequently during bug fixes.

Next, we studied the bug fixes with log changes and found that 1) Bug fixes with log changes take more time to get resolved, 2) Bug fixes with log changes have more comments and users in the discussion. We found that given 2 bugs of same complexity, the one without log changes takes longer time to be fixed, needs more number of developers to be involved in the fix and has long discussions post also.

Finally we did a manual analysis were we found developers use logging statements to understand the rationale behind logs. We found over 70.7 % of the cases directly used logs in the debugging process. We found that over 61 % of the time logs are used to understand the rationale behind bugs and find which specific conditions triggers bugs.

This proves that logs are in-fact helpful in bug fixes and are useful for quick resolution of bugs with less involvement from developers. It shows that if logs added during bug fixes the resolution time and development can reduce during subsequent bug fixes and help other developers.

REFERENCES

[1] 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 IEEE International Conference on Data Mining*, ser. ICDM '09. IEEE Computer Society, 2009, pp. 588–597.

[2] 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*. New York, NY, USA: ACM, 2009, pp. 117–132.

[3] 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*, Sept 2013, pp. 110–119.

[4] 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.

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

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

[7] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software."

[8] W. Shang, M. Nagappan, and A. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, 2015.

[9] 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.

[10] 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.

[11] H. Malik, H. Hemmati, and A. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems."

[12] 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.

[13] 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.

[14] 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.

[15] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI.

[16] Q. Fu, J. Zhu, W. Hu, J.-G. L. R. Ding, Q. Lin, D. Zhang, and U. . I. .-.-.-.-. d. Tao ACM New York, NY, "2014-05-31where 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 ICSE Companion 2014*. 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.

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

[18] 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.

[19] 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.

[20] A. Hassan, "Mining software repositories to assist developers and support managers."

[21] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *Proceedings ofSoftware Maintenance (ICSM), 2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2011, pp. 273–282.

[22] *T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In Proceedings of ISSRE*, 1996, page 364. IEEE CS Press, 1996.

[23] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[24] J. H. G. K. B. K. Sinha, *Statistical Meta-Analysis with Applications*. Wiley, 2011.

[25] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: Making use of a decade of widely varying historical data," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 149–157.