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 leveraged 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 software systems. Prior research has shown that over 41% of log changes are made during bug fixes. However, little is known about how logs are 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 find that logs are changed ~~a~~ statistically significantly more in bug fixes than other code changes. Furthermore, we find four different types of log modifications that developers make to logging statements during bug fixes: (1)changes to logging level, (2)changes to the textual content of a log, (3)changes to logging parameters and (4)relocation of logs. ~~We~~ In addition we find that developers modify variables more during bug fixes, than adding or deleting logs. Finally, we find that ~~issue reports~~ bug fixing commits with log changes have larger code churn, but involve fewer developers, require less time and have less discussion during the fix process. We train a linear model and find that adding and modifying logging statements is ~~statistically significant and reduce~~ negatively co-related to the resolution time of bug fixes. ~~These findings show that developer should leverage logs more in practice to fix~~ This suggests that leveraging logs helps in the quicker resolution of bugs.**

I. Introduction

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

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

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

Logs are also used extensively to help developers to fix ~~bus~~ bugs in large software systems. For example, in the JIRA issue HBASE-3403 (commit ~~1056484~~1,056,484), a bug is reported when a module does not exit upon system failure.~~In order to fix this bug, developers needed to record more information in the existing logs~~To find the point of failure, developers leverage log data. After the ~~bug was fixed, the added information in the logs has been used by developers~~ fix, the logs or updated to prevent similar bugs in the system.

Prior research performs a manual study on bug fixes. The results show that 41% of log changes are used to assist ~~in~~ debugging [6]. However, there exists no large scale empirical study of how logs are leveraged during bug fixes.

In this paper, we perform an empirical study on how logs are leveraged during the bug fixes of ~~the~~ Hadoop, HBase and Qpid projects. In particular, we sought to answer following research questions.

RQ1: Are logs leveraged more during bug fixes
We find that logs are leveraged more frequently during bug fixing ~~changes~~ commits than non-bug fixing ~~changes~~commits. In particular, we find that log addition and log modification appear more in bug fixing commits than non-bug fixing commits, to a statistically significant degree, with non-trivial effect size. We identified 4 types of log modifications namely *'Logging level change', 'Text Modification', 'Variable Change' and* Log Relocation'. We find that *'Text Modification', 'Variable Change' and* 'Relocating' are statistically 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 find that logs help in a faster resolution of bugs with less developer involvement. We find that bug fixing commits with log changes have higher code churn. This implies that logs are leveraged to fix more complex bugs. After controlling for code churn, we find that the issues with log changes take

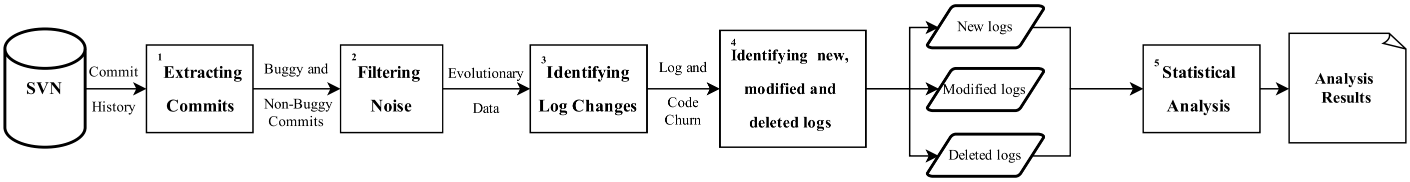

Fig. 1. Overview of our cast study approach

less time to get resolved, involve fewer ~~developer~~ developers and have less discussions during the bug fixing process. Using these metrics we trained a linear model to study the effect that log churn related metrics have on resolution time of bug fixes. We find that log churn related metrics are statistically significant in the model and have a negative effect on resolution time of bug fixes. This shows that developers should leverage logs more in practice to assist in the faster resolution of bugs.

The rest of this paper is organized as follows. Section II presents a qualitative study to motivate the paper. Section III presents the methodology for gathering and extracting data for our study. Section IV presents the case studies and the results to answer the two research questions. Section ~~II presents a manual analysis on bug fixing commits with log changes. Section~~ V describes the prior research that is related to our work. Section VI discusses the threats to validity. Finally section VII concludes the paper.

II. A Manual Study on Leveraging Logs During Bug Fixes

To understand how developers use logs in bug fixes, we first do a qualitative analysis. We collected all the bug fixing commits with log changes for our subject systems. We selected a 5% random sample (266 for *HBase*, 268 for *Hadoop* and 83 for *Qpid*) from all the commits. From the sampled commits, we collect the JIRA issue ID's and analyze the discussion posts on JIRA. We study the posts to find how many commits utilize the information of logs during the bug fixing process. The findings are reported in table II

TABLE II
Distribution of Log Usage in Bug Fixes

Projects	Hadoop (%)	HBase (%)	Qpid (%)
Log Relocation	74	89	63

From table II we find that in all projects developers use logs in the discussion posts before fixing the bugs. We find that in 40% of *Hadoop* discussion posts developers even provide log dumps to help in debugging the bug. We find *HBase* has the highest percentage of log dumps at 50% and it is least in *Qpid* at 22%. This may be because older mature projects like Hadoop, Hbase are already well logged and developers can use the existing logs. In newer projects like *Qpid*, the code may not be logged and developers may have to add new logs to the code during the bug fix, as seen in section IV.

This manual analysis shows that developers use the information provided by logs during bug fixes and update the logs for future use. This motivates us to perform an empirical study to find how logs are leveraged during bug fixes and understand the usefulness of log.

III. Methodology

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

The aim of this paper is to understand how logs are leveraged during bug fixes. We conduct a case study on three open source projects i.e. Hadoop, HBase and Qpid. All three subject systems have extensive logging in their source code. 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. Hadoop uses the MapReduce data-processing paradigm. The logging characteristics of Hadoop have been studied in prior research [6], [9], [3]. We study 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~~ release 0.10 till 0.98.2.RC0. ~~Those release have~~ This covers more than 4 years of development in HBase from 2010 till 2014.

Qpid[3]: Qpid is an open source messaging system that implements an Advanced Message Queuing Protocol (AMQP). We study release 0.10 to release 0.30 of Qpid. ~~These releases are~~ This covers development from 2011 ~~to~~ till 2014.

Figure 1 shows a general overview of our approach. (1) We mine the SVN repository of each subject system to extract all commits and identify each commit as bug fixing or non-bug fixing commit. (2) We remove the noise from our extracted data sets. (3) We identify logging statement changes in both bug fixing and non-bug fixing commits. (4) We categorize the log changes into *'New logs'*, *'Modified Logs'* and *'Deleted logs'*. (5) We calculate churn metrics using these categories and use statistical tools, such as R [10], to perform experiments

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

TABLE I
AN OVERVIEW OF THE SUBJECT SYSTEMS

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

on the data to answer our research questions. In the rest of this section we describe the first four steps in more detail.

A. Study Approach

We used SVN to study the evolution of Java source code in the three ~~projects~~subject systems. We extract the changes made in each commit and using this data calculate the churn metrics to answer our research questions.

1) Extracting Commits : The first step in our approach is to extract buggy and non-buggy commits. ~~We~~ To achieve this we extract a list of all commits ~~from SVN and the commit message from each commit~~with commit messages from SVN. We extract a list of all JIRA issues related to bug fixes. As developers mention the JIRA issue ID's in the commit messages, we matched the commit messages against the JIRA issues to identify all the bug fixing changes. If a commit message does not contain a JIRA issue we search for bug fixing keywords like 'fix' or 'bug'. Prior research has shown that such heuristics can identify bug fixing commits with a high accuracy [6].

2) Filtering Noise: After separating our data into bug fixing and non-bug fixing commits, we calculated the churn of each commit. ~~As commits~~ In our study, churn is code addition and deletion that takes place in a commit and a churn metric is collection of code churns for all commits in the dataset. As a commit may contain changes to non-Java files, we filtered ~~out the changes to~~ the non-Java files from ~~both~~ our 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~~ such commits we search for keywords like 'branch' or 'merge' in the commit messages. If there are no such heuristics found and the total code churn of a commit is larger than 50,000 ~~), we consider only those commits that have both code addition and deletion~~lines of code, we find the total added and deleted lines of code for that commit. If the total code churn of a commit is entirely due to either added or deleted code, we exclude those commits. For example, ~~in the~~ we exclude commit 952,410 ~~the code churn is~~ of HBase, which has code churn of over 100,000 ~~and it has no deletion of code and this commit is filtered out since its a branching commit. To filter branching commits with churn less than 50,000, we search for keywords like 'branch' or 'merge' in the commit messages~~lines because it is entirely due do code addition as it is a branching operation.

3) Identifying new, modified and deleted logs : To identify the log changes in the datasets, we manually sample some commits to find common patterns in the logging statements. Some of the patterns ~~were~~ are specific to a particular project. For example a logging statement from Qpid invokes 'QPID_LOG', as follows:

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 the following sentence uses *Log4j*:

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

Using regular expressions to match these patterns, we automate 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~~ a method invocation "LOG", followed by logging-level. We count the change to every such invocation as a log change. Some logging statements may be split into multiple lines. We consider one log change for each 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 logging statements added in a commit.

Deleted Log: This type includes all logging statements 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~~measure [11]. 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~~ shown below have levenshtein distance of 16 and ~~ration~~ ratio of 0.86 when we compare both the logging statements entirely. Hence this log change 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 modified logs from the added logs.
3) Removed Logs: This includes all those logs which were deleted in a commit. Similar to new logs, we removed all the modified logs from the deleted logs to obtain this.

We use this data to answer the two research questions in the next section.

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.

RQ1: Are logs leveraged more during bug fixes?

***Motivation**:* Prior research has shown that logs are used during bug fixing [6]. During bug fixing, developers update logging 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 whether 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 use the data sets obtained in previous section i.e, modified, new and removed logs, and we calculate code churn for each commit. We use the total code churn of a commit to control *# modified, # new* and *# removed logs*. The three new metrics are:

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

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

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

To understand the different types of log modifications during bug fixing commits, we perform a manual analysis on the modified logging statements to identify the different types of log modifications. We first collect all the commits that have logging statement changes. We select 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 follow an iterative process, as prior research [12], to identify the different types of log modifications, until we cannot find any new types of modifications.

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

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

$$Cohen's\ d = \frac{\bar{x}_1 - \bar{x}_2}{s}, \quad (4)$$

where $\bar{x}_1$ and $\bar{x}_2$ are the mean of two populations, and s is the pooled standard deviation [13]. As software 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.** Table III, shows that *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 that 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, resulting in non-trivial effect size. For mature projects like Hadoop and HBase, source code is well logged so. In these projects 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

TABLE III
P VALUES AND EFFECT SIZE OF FOR COMPARISON . A POSITIVE EFFECT SIZE MEANS BUG FIXING COMMITS ARE LARGER. P-VALUES ARE BOLD IF < 0.05.

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

in HBase and Qpid, the effect sizes are trivial (see table III). ~~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 [5].

TABLE IV
DISTRIBUTION OF FOUR TYPES OF LOG MODIFICATIONS.

Projects	Hadoop (%)	HBase (%)	Qpid (%)
Log Relocation	~~82.6~~ 73.1	~~61.4~~ 70.7	~~55.8~~ 47.4
Text Modification	~~7.85~~ 10.5	~~12.1~~ 13.4	~~18~~ 16.8
Variable Modification	~~7.9~~ 9.9	~~8.4~~ 10.1	~~12.5~~ 18.9
Logging Level Change	~~3.85~~ 6.5	~~5.4~~ 5.8	~~13.6~~ 16.8

Logs are modified more in bug fixing commits than non-bug fixing commits. Table III 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 finds that 36% of log messages are modified at-least once as after-thoughts [1]. Developers may find that they need different information from logs to 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 add new logs but rather improve the existing logs. Prior research shows that too much information provided by logs may have become a burden for developers [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 IV. 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 significantly more in all the subject systems and has small or medium effect sizes (see Table V). 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 categorize the variable change into three types: variable addition, variable deletion and variable modification. Table VI, shows that developers modify variables statistically significantly more frequent 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 variables in logging statements to print the needed values. 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, because the massive amounts of logs can be a burden for developers.

Developers modify log text ~~in logs~~ more during bug fixes. We find that text modification is statistically significantly more in bug fixing commits than non-bug fixing commits with non-trivial effect sizes (see Table V). 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 Qpid 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~~ shows that there is a challenge to understand logs in practice [16]. Our results show that developers may have ~~encountered such challenge and try to improve~~ faced such challenges and improved the text in logs for better bug fixing.

Log relocation occurs more in bug fixes. Table IV, shows that there are a large number of logging changes that only relocate logging statements. Table V shows that such relocation ~~happens statistically significantly more~~ of logs is statistically significant in bug fixing commits. 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 bugs, developers often move existing logging statements into the *try/catch* blocks or after condition

TABLE V
P-VALUES AND EFFECT SIZE FOR LOG MODIFICATIONS. P-VALUES ARE BOLD IF < 0.05

Metrics	Hadoop		HBase		Qpid	
	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 VI
P-VALUES AND EFFECT SIZE FOR THE DIFFERENT TYPES OF VARIABLE CHANGE. P-VALUES ARE BOLD IF < 0.05.

Metrics	Hadoop		HBase		Qpid	
	P-values	Effect Size	P-values	Effect Size	P-values	Effect Size
Variable addition	0.22	-0.069	0.129	0.222(small)	0.486	-0.152
Variable deletion	0.25	-0.114	0.585	0.165(small)	0.22	-0.195(small)
Variable modification	**0.047**	0.221(small)	**0.0032**	0.268(small)	**1.61-05**	0.550(small)

statements. For example, in the revision 792,522 of Hadoop, 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 happen statistically significantly more in Hadoop project. This implies that developers typically do not change log levels during bug fixes. The reason may be that developers are able to enable all the logging statements during bug fixing, despite of what level a logging statement has. In addition, prior research shows that developers do not have a good knowledge about how to choose a correct logging level [1].

Developers change 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 as after-thoughts and change the variables in logging statement to assist in fixing bugs.

RQ2: Are logs useful in bug fixes?

***Motivation**:* In RQ1, we find that logs are leveraged more frequently in bug fixes. However, little is known about the usefulness of logs in bug fixes. In this research question, we try to understand the usefulness of leveraging logs during bug fixes.

***Approach**:* To find the usefulness of logs in bug fixes, we collect all JIRA issues with type 'bug' from the three subject systems. We obtained the code commits for each of these JIRA issues by searching for the issue id from the commit messages. We measure the log churn and the code churn for each issue. We then split the JIRA issues into (1) bugs fixed with log churn (2) bugs fixed without log churn. We use the code churn to measure the complexity of the issue. We then extracted three metrics from JIRA issues to measure the effort of fixing a bug:

1) **Resolution Time:** This metric measures how fast ~~is the bug fixed. The faster~~ the bug is fixed~~, the less effort is spent on the bug. We measure~~. This is defined as the time taken from when the bug is opened till its resolved. For example, if a bug was ~~reported~~ opened on 1^{st} February 2015 and closed on 5^{th} February 2015, the time taken to fix the bug is four days.
2) **# Comments:** This metric measures how ~~much discuss is~~ many discussions are needed to fix a bug. Intuitively, the more discussion in the issue report, the more effort is spent on fixing the bug. We count the total number of comments in the discussion of each issue report.
3) **# Developers:** This metric measures how many developers participated in the discussion of fixing the bug. Intuitively, more people discuss the bug, more effort is spent on fixing the bug. We count the number of unique developers who commented on the issue report. We use the user names to identify developers.

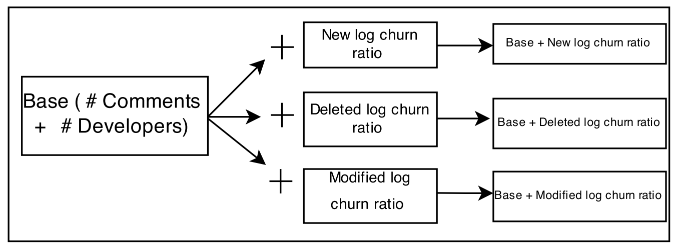

Fig. 2. Overview of our linear models for the resolution time of bugs

We first compare the code churn of bug fixes with and without log churn. Similar to RQ1, we use *MannWhitney U test* to study whether the difference is statistical ~~significance~~ significant and we use *Cohen's D* to measure the size of the difference between code churn of bug fixes with and without log churn. Then, we control the code churn for resolution time, the number of comments, and the number of developers. We compare these metrics in the bug fixes with and without log churn to study with the same complexity of bugs, whether they are fixed faster, with fewer comments and fewer people when logs are leveraged.

To better understand the usefulness of log churn on the time taken for fixing bugs, we build a linear regression model. Prior research has shown that resolution time is correlated to the

TABLE VII
P-VALUES AND EFFECT SIZE FOR COMPARING CODE CHURN, RESOLUTION TIME, # COMMENTS AND # DEVELOPERS IN THE BUG FIXES WITH AND WITHOUT LOG CHURN. THE RESOLUTION TIME, # COMMENTS AND # DEVELOPERS ARE CONTROLLED BY THE CODE CHURN.

Metrics	Hadoop		Hbase		Qpid	
	P -values	Effect Size	P -values	Effect Size	P -values	Effect Size
Code churn	**2.2e-16**	0.563(small)	**2.2e-16**	0.163(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
# comments	**2.2e-16**	-0.507(small)	**5.16e-11**	-0.289(small)	**2.34e-03**	-0.227(small)
# developers	**2.2e-16**	-0.577(small)	**2.2e-16**	-0.538(small)	**4.73e-02**	-0.375(small)

number of developers and the number of comments in a issue report [17]. We want to see whether the metrics from log churn (as shown in RQ1) can complement the number of developers and the number of comments in modelling the resolution time of bugs. The overview of the models is shown in Figure 2. We start with baseline model **BASE(#comments+#developers)** that uses the number of developers and the number of comments as independent variables [17]. For the base model for each project, if the number of developers or the number of comments is not significant in the model, we remove the insignificant metric from the base model. We then build subsequent models in which we add our metrics that are measured in RQ1 as independent variables. The added metrics are $modified\ log\ churn\ ratio$, $new\ log\ churn\ ratio$, and $removed\ log\ churn\ ratio$. We add these metrics separately into the base model to examine whether each log churn related metric can complement the number of developers and the number of comments. We examine whether ~~the~~ each log churn related metric is significant in each model.

To measure the effect of each log churn related 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 *'Resolution Time'*. 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 log churn related metrics increases the *'Resolution Time'*, whereas a negative effect means that a higher value of the log churn related metrics decreases the *'Resolution time'*.

We would like to point out that although linear regression has been used to build accurate models for the resolution time of bugs [17], our purpose of using the linear model in this paper is not for predicting the resolution time of bugs. Our purpose is to study the explanatory power of log churn related metrics and explore its empirical relation to the resolution time of bugs.

Results: **We found that the logs are used to fix more complex bugs**. We find that the average code churn for fixing bugs is significantly higher with log churn than without log churn (see Table VII and Figure 3). Such results imply that developers may leverage logs to fix more complex bugs.

We found that bugs that are fixed with log churn take a shorter time with fewer comments and fewer people. After controlling for code churn, we find that the resolution time, the number of comments and the number of developers are all statistically significantly smaller in the bug fixes with log churn than the ones without log churn. This result means that given two bugs of same complexity, the one with log churns takes less time to get resolved and needs fewer number of developers involved with fewer discussions. This implies that logs provide useful information to assist developers in discussing, diagnosing and fixing bugs. For example, when fixing issue HBASE-3074 (commit 1,005,714), developers left the first comment to provide additional details in the logging message about where the failure occurs. In the source code, developers add the name of the servers into the the logging statements. This additional data helps trace the cause of the failure and helps in fixing the bug.

Log churn related metrics are significant in modelling the resolution time of bugs with negative effect. From table IX shows the effect of log churn related metrics when modeling resolution time. We find that the $new\ log\ churn\ ratio$ is significant in Hadoop and HBase when modelling resolution time of bug fixes and the $modified\ log\ churn\ ratio$ is significant in modelling the resolution time of bugs for Hadoop. Such results shows that there exists a relationship between log churns and the resolution time of bug fixes. We find that all log churn related metrics have negative effects on the resolution time of bugs. The negative effect implies that more the developers leverage logs, the lesser time it takes for them to fix the bug. This is a important correlation and should not be confused with causation.

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

Projects	Hadoop	HBase	Qpid
new log churn ratio	**-4.55** **	**-5.06** **	**-6.00** ◇
removed log churn ratio	-3.24	-4.16	-2.37
modified log churn ratio	**-5.31** *	-3.7	-6.31

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

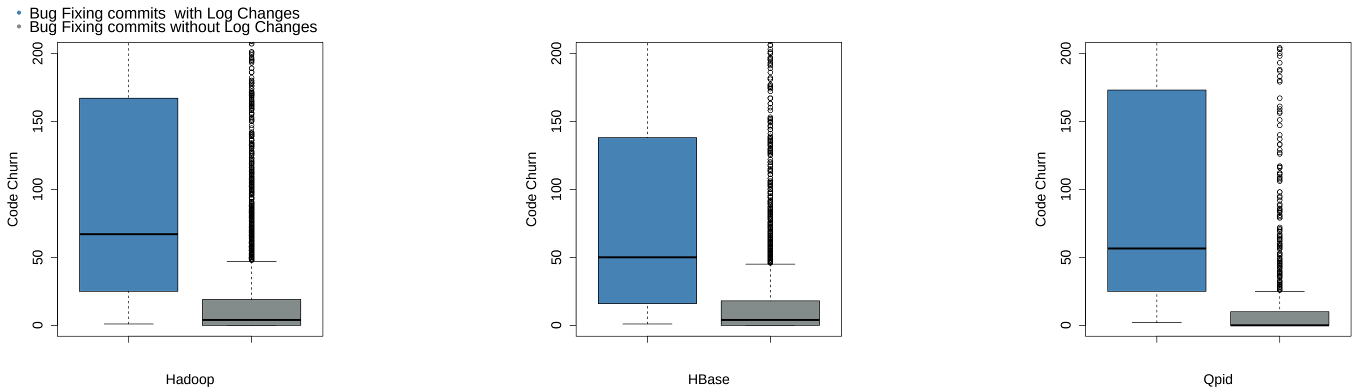

Fig. 3. Boxplot of code churn of bug fixing commits with log churn (shown in blue) against bug fixing commits without log churn (shown in grey).

Logs are leveraged during fixing more complex bugs. Bug fixes with log changes are resolved faster with fewer people and fewer discussions. Log churn related metrics can complement number of comments and the number of developers in modelling the resolution time of bugs with a negative effect. Such results imply that there is a relationship between leveraging logs and a faster resolution of bugs.

***Motivation**:* In the previous research question we look at how logs are leveraged during bug fixes and if logs are useful during a bug fix. However, prior research has shown that resolution time is correlated to the number of developers and the number of comments in a issue report [17]. In this RQ we explore if the log churn can increase the explanatory of the previous model and we study how each metric impacts the resolution time of bug fixes.

***Approach**:* We use logistic regression model, to study the explanatory power of our log churn metrics on resolution time. From previous studies we know that the number of developers and the number of comments are effective in explaining resolution time [17]. Therefore, we include these metrics along with our log churn metrics to increase the explanatory power of our model below,

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

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

To understand the effect of each metric on the model we follow a similar approach used in prior research [18], [19]. To measure the effect 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 the impact of log churn metrics on resolution time of bug fixes.

***Results**:* **We found that New Logs help in decreasing the resolution time of bug fixes**. From table IX 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 IX, 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 modified log churn is not significant in Hbase. This maybe because developers do not modify logs in Hbase as seen from table III, where effect sizes are trivial for HBase.

We see that for new projects like Qpid, the source code is not well logged. So developers may add extra logs during complex bug fixes, increasing the resolution time bug fixes. This is seen in table III where Qpid has the highest effect sizes among all projects for *'New Log churn'* and *'Modified Log churn'*, implying developers log extensively during bug fixes.

V. ~~A MANUAL STUDY ON LEVERAGING LOGS DURING BUG FIXES~~

~~From the results of our research questions, we find that logs are often changed during bug fixes and the bugs with log changes are fixed faster. However, from our data analysis, we do not know whether the changed logs are actually leveraged for bug fixes.~~

~~To further study whether logs are leveraged during bug fixes, we perform a manual analysis to find out whether logs are leveraged during bug fixes. Where and what information provided by logs are used in bug fixes. We first collect all the issue reports with log churn. We then select a random sample from all the issue reports (180 samples) with 95confidence~~

TABLE IX

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	**-1.05** **	**-1.26** **	1.46
Deleted Log Churn	-1	-1.11	0.84
Modified Log Churn	**-0.6** *	-0.6	1.25

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

~~level and 5confidence interval. We read the discussion in each bug report to examine whether and how are log used in fixing the bug.~~

~~**In 70.7% of the manually examined issue reports, logs are leveraged for bug fixes.**~~ ~~We find that developers leverage logs in 128 of examined issue reports (e.g. HBASE-3741). After identifying a bug, developers often leverage logging statements to trace the cause of bugs. We want to note that, for the bug reports that do not leverage logs in the issue report, logs may be used but not discussed in the issue report.~~

~~**In 61% of the bug reports made use of both variable and textual part in the logging statements to resolve bugs.**~~ ~~We observe that developers output the system state, the server/connection name, time, and even object names in the logs. We find that such information are often added by after-thought of developers during bug fixes. For example, to fix HBASE-3832, developers identify the cause of the problem using the logs. However, they fail to identify the exact conditions that cause the bug. Therefore, developers add more information into logs in order to fix the bug. Our results confirm that the text and the variable in the logs do help developers fix bugs.~~

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~~ on large software systems and empirical studies done 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.* [9] 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~~ also show that logs are ~~widely leveraged~~ leveraged more 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~~ performs an empirical study 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~~ were 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 [6], [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~~ provide additional control and data flow parameters into logs.

The most related prior research by Shang *et al.* [6] empirically study the relationship of logging practice and code quality. Their manual analysis sheds ~~some lights~~ light on the fact that some logs are changed due to field debugging. They also show that there is a strong relationship between logging practice and code quality. Our paper focused on understanding how logs are 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

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

External Validity

Our study is performed ~~and~~ Hadoop, HBase and Qpid. Even though these three subject systems have years of history and large user bases, the three subject systems are all ~~based on Java~~ Java based platform systems. More case studies on other software in other domains with other programming languages are needed to see whether our findings can be generalized.

Internal Validity

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

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

Construct Validity

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

We use keywords to identify bug fixing commits when the JIRA issue id is not included in the commit messages. We also use ~~keyword~~ keywords to identify branching and merging commits. Although such ~~keyword~~ keywords are used extensively in prior research [6], we may still miss identify bug fixing commits or branching and merging commits.

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

We build a linear regression to model the resolution time of bugs. However, the relationship between log churn and the resolution time of bugs may not be linear. In addition, there may exist interactions between metrics. For example, the logs in debug logging level may have a higher relationship with the resolution time of bugs. However, as the first exploration of the leverage of logs in bug fixes, we only use linear model to find out whether the leverage of logs have a relationship with the resolution time of bugs. The resolution time of bugs can be correlated to many factors other than just logs, such as the complexity of code fixes. To reduce such a possibility, we control the log churn related metrics by code churn. However, other factors may also have an impact on the resolution time of bugs. Future studies should build more complex models that consider these other factors.

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

VII. Conclusion and Future Work

Logs are used by developers for finding anomalies, monitor performance, software maintenance, capacity planning and also in fixing bugs. The leverage of logs and their usefulness during bug fixes has never been empirically studied before. This paper is a first attempt (to our best knowledge) to understand whether and how are logs leveraged during bug fixes. The highlights of our findings are:

- We find that logs are used more during bug fixing commits. In particular, we find logs are modified more frequently during bug fixes. More specifically we find that variables and textual information in the logs are more frequently modified during bug fixes.
- We find that logs are leveraged in fixing more complex bugs. However, bug fixes that leverage logs are faster, need fewer developers and have less discussion.
- We find that log modification and new logs are significant in modeling the resolution time of bugs. More the log modification or addition of new logs, the shorter the resolution time of bugs.

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

REFERENCES

[1] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 102–112.

[2] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *ICDM '09: Proceedings of the 9th IEEE International Conference on Data Mining*, pp. 588–597.

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *ACM SIGOPS '09: Proceedings of the 22nd Symposium on Operating Systems Principles*, New York, NY, USA, pp. 117–132.

[4] L. Bitincka, A. Ganapathi, S. Sorkin, and S. Zhang, "Optimizing data analysis with a semi-structured time series database," in *SLAML'10: Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*. USENIX Association, pp. 7–7.

[5] 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.

[6] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, vol. 20, no. 1, pp. 1–27, 2015.

[7] Xpolog. [Online]. Available: http://www.xpolog.com/.

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

[9] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection." in *USENIX Annual Technical Conference*, 2010.

[10] 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.

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

[12] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *Software Engineering, IEEE Transactions on*, vol. 25, no. 4, pp. 557–572, 1999.

[13] W. R. Shadish and C. K. Haddock, "Combining estimates of effect size," *The handbook of research synthesis and meta-analysis*, vol. 2, pp. 257–277, 2009.

[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 *OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2014, pp. 249–265.

[16] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in *ICSME '14: Proceedings of the International Conference on Software Maintenance and Evolution,*. IEEE, 2014, pp. 21–30.

[17] P. Anbalagan and M. Vouk, "On predicting the time taken to correct bug reports in open source projects," in *ICSM 2009: Proceedings of IEEE International Conference on the Software Maintenance,*. IEEE, 2009, pp. 523–526.

[18] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *ACM SIGSOFT '11 : Proceedings of the 19th symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 300–310.

[19] A. Mockus, "Organizational volatility and its effects on software defects," in *ACM SIGSOFT '10 : Proceedings of the 18th international symposium on Foundations of software engineering*. ACM, 2010, pp. 117–126.

[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 *ICSE'13: Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 402–411.

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

[22] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Salsa: Analyzing logs as state machines," in *WASL'08: Proceedings of the 1st USENIX Conference on Analysis of System Logs*. USENIX Association, 2008, pp. 6–6.

[23] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou, "Understanding customer problem troubleshooting from storage system logs," in *FAST '09: Proceedings of the 7th Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2009, pp. 43–56.

[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.

[25] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *ICSE '14: Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 468–479.

[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 *ASPLOS '10: Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2010, pp. 143–154.

[27] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *Journal of Software Maintenance Evolution*, vol. 20, no. 4, pp. 249–267, Jul. 2008.

[28] Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *ICSM '08: Proceedings of theIEEE International Conference on Software Maintenance*, Sept 2008, pp. 307–316.

[29] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *ICSM '09: Proceedings of the IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 125–134.

[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 *SSIRI '10: Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement*. Washington, DC, USA: IEEE Computer Society, pp. 131–140, 2010.

[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 '14: Proceedings of the 36th International Conference on Software Engineering,*, pp. Pages 24–33.

[32] W. Shang, "Bridging the divide between software developers and operators using logs," in *ICSE '12 :Proceedings of the 34th International Conference on Software Engineering*.

[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.

[34] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems*, vol. 30, pp. 4:1–4:28, Feb. 2012.