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

An Empirical Study On Log Changes During Bug Fixes in Platform Systems

First Author · Second Author

Received: date / Accepted: date

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 a valuable source of information for developers in debugging large software systems. Prior research has shown that over 32% of log changes are made during bug fixes. However, little is known about how much more logs are leveraged during bug fixes. In this paper, we sought to study the changing of logs during bug fixes through case studies on three large open source platform software namely Hadoop, HBase and Qpid. We find that logs are changed 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 namely: (1)changes to the logging level, (2)changes to the textual content of a log, (3)changes to the logging parameters and (4)relocation of logs. In addition we find that developers modify and add variables more during bug fixes, than deleting logs. Finally, we find that bugs that are fixed with log changes have larger code churn, but involve fewer developers, require less time and have less discussion during the fix process. We build a non-linear regression model to explore the explanatory power of log related metrics on bug resolution time. We find that log related metrics can complement traditional metrics i.e, # of developers and # of comments, in explaining the bug resolution time. In particular, we find that modifying logging statements is negatively co-related to the resolution time of bug fixes. Our results

Suhas. Kabinna
first address
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: fauthor@example.com

S. Author
second address

show that there is a relationship between changing logs and the resolution speed of bugs.

1 Introduction

Platform softwares provide an infrastructure for a variety of applications that run on top of it. Hadoop, HBase are two popular platform systems and Qpid is a messaging system providing multi-platform support. These software systems rely on logs to monitor the applications which run over them. These logs are also leveraged during development for field debugging [1] and performance monitoring [2,3].

Logs can be generated through simple *printf* statements or through the use of 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 knowledge about the events, and 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 variable *host* contains the information about the logged event.

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

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

Logs are used extensively to help developers fix 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.To identify the point of failure, developers leverage logs. After fixing the bug, logs are updated to prevent similar bugs in the system.

Prior manual study on bug fixes shows that over 55% of logs changes occur during feature changes and 32% during field debugging [1]. However, there exists no large scale study of how logs are changed during bug fixes.

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

RQ1: Are logs changed more during bug fixes?
We find that logs are changed more frequently during bug fixing commits than non-bug fixing 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 four types of log modifications namely *'Logging level change',*

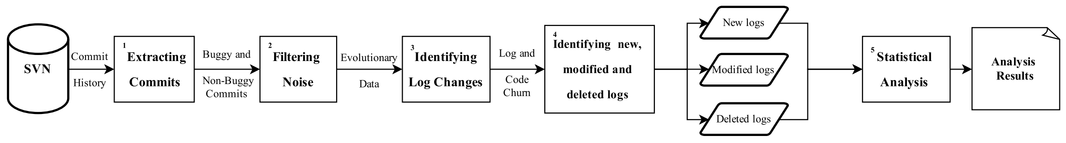


Fig. 1: Overview of our cast study approach

'Text modification', 'Variable change' and Log relocation'. We find that in log modification, *'Text modification', 'Variable change' and* 'Log relocation' are statistically significant, with medium to high effect sizes in bug fixing commits.

RQ2: Are bugs fixed faster with log changes ?
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 less time to get resolved, involve fewer developers and have less discussions during the bug fixing process. Using log churn related metrics we trained a non-linear regression model for the resolution time of bug fixes. We find that log modification related metrics are statistically significant in the model and have a negative effect on resolution time of bug fixes. This suggests that there is a relationship between log changes and faster resolution of bugs.

The rest of this paper is organized as follows. Section 2 presents the methodology for gathering and extracting data for our study. Section 3 presents the case studies and the results to answer the two research questions. Section 4 presents a qualitative study to strengthen the findings in our paper. Section 5 describes the prior research that is related to our work. Section 6 discusses the threats to validity. Finally, Section 7 concludes the paper.

2 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 logs changes during bug fixes. We conduct a case study on three open source platform systems i.e. Hadoop, HBase and Qpid. All three subject systems have extensive logging in their source code. Table 1 highlights the overview of the three subject systems.

Hadoop[1]: Hadoop is an open source software framework for distributed storage and processing of big data on clusters. Hadoop uses the MapReduce data-processing paradigm. The logging characteristics of Hadoop have been studied in prior research [1,10,6]. We study Hadoop releases 0.16.0 to 2.0.

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

Table 1: 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

HBase[2]: Apache HBase is a distributed, scalable, big data store using Hadoop file-systems. We used HBase release 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 that implements an Advanced Message Queuing Protocol (AMQP). We study release 0.10 to release 0.30 of Qpid. This covers development from 2011 till 2014.

Figure 1 shows a general overview of our approach, which consists of five steps: (1) We mine the Git repository of each subject system to extract all commits and identify as commit bug fixing or non-bug fixing. (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 tool R [11], to perform experiments on the data to answer our research questions. In the reminder of this section we describe the first four steps.

2.1 Study Approach

We use Git to study the evolution of Java source code in the three subject systems. We extract the changes in each commit and use this data to calculate the metrics to answer our research questions.

2.1.1 Extracting Commits

The first step in our approach is to extract buggy and non-buggy commits. To achieve this we extract a list of all commits with their commit messages from Git. Next, we extract a list of all JIRA issues that have the type "bug". As developers mention the JIRA issue ID's in the commit messages, we match 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 [1].

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

2.1.2 Filtering Noise

After segregating our data into bug fixing and non bug fixing commits, we filter the non-Java and 'Test' files present in the commits. We also find that some commits correspond to branching and merging operations. To filter the this we enable the 'no-merges' option in Git *log* to exclude all the merging operations in the systems.

2.1.3 Identifying new, modified and deleted logs

To identify the log changes in the datasets, we manually explore logging statements in the source code. Some logging statements 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 logging statements leverage the same logging libraries. For example the following logging statement uses *Log4j* library:

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

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

2.1.4 Identifying types of log changes.

After identifying the logging statements in every commit, we found two types of log changes.

Added Log: This type includes all the logging statements that are added in a commit.

Deleted Log: This type includes all the logging statements that are deleted in a commit.

Since Git *diff* does not provide a built in feature to track modification to a file , modifications to logging statements are shown as added and deleted code. To track these modifications, we used levenshtein ratio [12]. For every pair of added or deleted logging statement in a commit, we remove the logging method (e.g, LOG) and the log level (e.g, info) and compare the text in the parenthesis using levenshtein ratio as done in prior research [13]. For example, the logging statements shown below have levenshtein ratio of 0.86. 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 all identifying log modifications we obtain 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.

3 Study Results

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

RQ1: Are logs changed more during bug fixes?

Motivation

Prior research has shown that close to 32% of logs are changed during field debugging [1]. During debugging developers update logging statements in order to gain more run-time information about the systems. Therefore, the future occurrences of a similar bug may be resolved easily with the updated logs. However, to the best of our knowledge, there exists no large scale empirical study to show whether logs are changed more during bug fixes than other activities during development. In addition we also examine how logs are changed during the bug fixing process.

Approach

We compare the changes to logging statements between bug fixing and non-bug fixing commits with respect to log churn. We use the data sets obtained in previous Section i.e, modified, new and removed logs. Commits with higher total code churn may also have higher log churn. Therefore, we calculate total code churn for every commit and use it to control *# modified, # new* and *# removed logs*. The three new metrics are:

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

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

$$\textit{Removed log churn ratio} = \frac{\#\ \textit{removed log churn}}{\textit{code churn}} \tag{3}$$

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 this set. The size of our random sample achieves 95% confidence level and 5% confidence interval. We follow an iterative process, similar to prior research [14], to identify the different types of log modifications, until we cannot find any new types of modifications.

After we identify the types of log modifications, we create an automated tool to label log modifications into the identified types. We calculate the number of log modifications of every type in each commit and 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) [15]. We choose *MannWhitney U test* because our metrics are highly skewed. Since *MannWhitney U test* is a non-parametric test, it does not have any assumptions about the distribution of the sample population. A p-value of ≤ 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 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 [16,17]. *Cohen's d* is defined as:

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

where $\bar{x}_1$ and $\bar{x}_2$ are the mean of two populations, and s is the pooled standard deviation [18]. As software engineering has different thresholds for *Cohen's d* [19], 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

Table 2: P values and effect size of the metrics between the bug fixing and non-bug fixing commits. A positive effect size means that bug fixing commits have larger metric values and P-values are bold if they are smaller than 0.05.

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

Developers add new logs more during bug fixes. Table 2, shows that *new log churn ratio* in bug fixing commits is statistically significantly larger than non-bug fixing commits in all subject systems with non-trivial effect size. This suggests that developers add new logging statements during bug fixes more than non-bug fixing commits. We find that effect size of new logs is higher in Qpid when compared to Hadoop and HBase. This suggests that Qpid being a relatively newer system, some important source code may not be well logged. Therefore, forcing developers to add additional logging statements to assist in bug fixing.

Developers do not delete logging statements during bug fixes. We find that although *removed log churn ratio* in bug fixing commits is statistically significantly larger than non-bug fixing commits in all projects the effect sizes are non-trivial only in Hadoop (see table 2). We find that in newer projects like Qpid, the effect size is negative implying developers delete logging statements more during non-bug fixing commits than bug fixing commits. This suggests that developers may not have clear understanding of the bug and require more information in newer projects. Such results confirm the findings from prior research that deleted logs do not have a strong relationship with code quality [7].

Table 3: Distribution of four types of log modifications.

Projects	Hadoop (%)	HBase (%)	Qpid (%)
Log Relocation	73.1	70.7	47.4
Text Modification	10.5	13.4	16.8
Variable Modification	9.9	10.1	18.9
Logging Level Change	6.5	5.8	16.8

Table 4: 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	**1.1e-10**	0.330(small)	**3.0e-11**	0.170(small)	**1.8e-08**	0.700(medium)
Text modification	0.38	-0.231(small)	**0.0075**	0.525(small)	**4.5e-06**	0.976(medium)
Variable change	**1.3e-04**	0.351(small)	**0.0010**	0.420(small)	**1.2e-04**	1.17(medium)
Logging level change	0.097	0.389(small)	0.51	-0.05	0.398	-0.02
Text and Variable Change	**9.7e-07**	0.253(small)	**1.2e-07**	0.446(small)	**0.004**	0.393(small)

Logs are modified more in bug fixing commits than non-bug fixing commits. Table 2 shows that *modified log churn ratio* is statistically significantly higher for all subject systems and the effect sizes are non-trivial in all projects. Such results show that developers often change the information provided by logging statements to assist in bug fixing. This is because developers may need different information than provided by logs to fix the bugs. We also find that the 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. This is substantiated by prior research which finds that 33% of log messages are modified at-least once as after-thoughts [4]. Prior research shows that too much information provided by logs may have become a burden for developers [20]. Such finding may explain the reason why developer choose modifying logs over adding new logs.

As log modification is more prominent than addition and deletion of logs, we find the different types of changes within log modifications. To achieve this we collect all commits with log modifications and select a random sample which achieves 95% confidence level and 5% confidence interval. We then follow an iterative process [14] to identify the different types of log modifications, until we cannot find any new type of modification.

From our manual analysis, we identify five types of log modifications. Table 3 shows their distributions.

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 that is 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.

Table 5: 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	0.09	0.107	**0.00049**	0.659(medium)	**0.005**	1.4(large)
Variable deletion	0.098	-0.075	0.347	0.364(small)	0.19	1.54 (small)
Variable modification	**4.11e-05**	1.045(medium)	0.58	0.0016	**0.0016**	0.949(medium)

5. **Text and Variable change.** The text and variables in the logging statements are changed. This is generally done when developers provide more context information i.e, text and add/modify the relevant variables in a logging statement.

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 4). This suggests that developers 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: a) variable addition, b) variable deletion, c) variable modification.

Table 5 shows that developers modify variables statistically significantly more frequent in Hadoop and Qpid with medium effect sizes. This implies that developers may not know what exact information is needed when they add a logging statement into the source code. The developers realize the need of more specific information and modify the 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 [4]. From Table 5, we also observe that the variable addition is statistically significant in Hbase and Qpid, with large effect size in Qpid. This suggests the developers add new variables more in newer projects like Qpid as the logging statements might not contain the necessary information. In older projects like HBase its lesser and we find its trivial in case of Hadoop, where developer prioritize modifying more. We also find that developers do not delete variables in logging statements. The reason maybe that deleting variables may impact the performance of the log analysis tools which rely on the log variables.

Developers modify log text 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 4). This suggests that in some cases, the text description in logs is not clear and developers improve the text to understand the logs better to fix the bugs. For example, in Qpid commit 1405354, developers modify the logging statement to provide more information about the cause of an exception being raised. Prior research shows that there is a challenge to understand logs in practice [21]. Our results show that developers may have faced such challenges and improved the text in logs for better bug fixing.

Log relocation occurs more in bug fixes. Table 3, shows that there are a large number of logging changes that only relocate logging statements. Table 4 shows that such relocation of logs is statistically significantly more in bug fixing commits than non-bug fixing commits. We manually examine such commits and find that developers often forget to leverage exception handling or using proper condition statements in the code. After fixing the bugs, developers often move existing logging statements into the *try/catch* blocks or after condition 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 are not statistically significant in any project. This suggests that developers 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 [4].

Developers change logs more in bug fixing commits than non-bug fixing commits. In particular, developers modified logs to add or 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 bugs fixed faster with log changes ?

Motivation

In RQ1, we find that logs are changed more frequently in bug fixes. However, there is no study to show if logs are useful in debugging process. To answer this, we explore to find if bugs fixes with log changes are fixed faster than bug fixes without log changes.

Approach

To find if bugs are fixed faster with log changes, we collect all JIRA issues with type ‘bug’ from the three platform 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 that are fixed with log change and (2) bugs that are fixed without log change. 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:

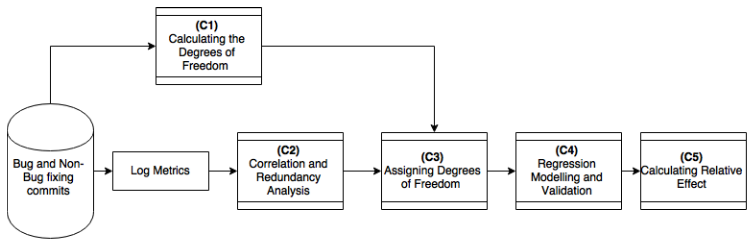

(C1) Calculating the Degrees of Freedom
Bug and Non-Bug fixing commits
Log Metrics
(C2) Correlation and Redundancy Analysis
(C3) Assigning Degrees of Freedom
(C4) Regression Modelling and Validation
(C5) Calculating Relative Effect

Fig. 2: Overview of our non-linear OLS model for the resolution time of bugs

1. **Resolution Time:** This metric measures how fast the bug is fixed. This metric is defined as the time taken from when the bug is opened till it is resolved. For example, if a bug was opened on 1^{st} February 2015 and closed on 5^{th} February 2015, the resolution time of the bug is four days.
2. **# of Comments:** This metric measures how 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. **# of Developers:** This metric measures how many developers participated in the discussion of fixing the bug. Intuitively, more people who discuss the bug, more effort is spent on fixing the bug. We count the number of unique developers who comment on the issue report. We use the user names in the JIRA discussion posts to identify the developers.

We first compare the code churn of bug fixes with and without log change. Similar to RQ1, we use *MannWhitney U test* to study whether the difference is statistically significant and we use *Cohen's D* to measure the size of the difference between code churn of bug fixes with and without log change. Intuitively fixing a more complex bug requires longer time, more people and more discussions. Therefore, we divide resolution time, the number of comments, and the number of developers by code churn. We study these bug fixes which are controlled for complexity to find whether bugs are fixed faster, with fewer comments and fewer people when logs are changed.

To better understand the usefulness of log change on the resolution time for fixing bugs, we build a non-linear regression model. Prior research has shown that resolution time is correlated to the number of developers and the number of comments in a issue report [22]. We want to see whether the metrics from log change (as shown in RQ1) can complement the number of developers and the number of comments in modelling the resolution time of bugs.

A non-linear regression model fits the curve of the form $y = \alpha + \beta_1 x_1 + \beta_2 x_x + .. + \beta_n x_n$ to the data, where y is the dependent variable and every x_i is an explanatory variable. In our model, *resolution time* is the dependent variable y and the log churn metrics are the explanatory variables.

We adopt the statistical tool R to model our data and use the *rms* package provided [23] to generate the regression model. The overview of the model generation process is shown in Figure 2 and is explained below.

(C-1) Calculating the Degrees of Freedom

During predictive modelling, a main concern is over-fitting. An over-fit model is biased towards the dataset it is built and wont be scalable to other datasets. In non-linear regression models over-fitting may creep in when a explanatory variable is assigned more degrees of freedom than the data can support. Hence, it is necessary to calculate a budget of degrees of freedom that a dataset can support before fitting a model. We budget $\frac{x}{15}$ degrees of freedom for our model as suggested by [24]. Here x is the number of rows (i.e, bug fixing commits) in each project.

(C-2) Correlation and Redundancy Analysis

Correlation analysis is necessary to remove the highly correlated metrics from our dataset. We use Spearman rank correlation instead of Pearson because Spearman is resilient to data that is not normally distributed. We use the function *varclus* in R to perform the correlation analysis.

Correlation analysis does not indicate redundant variables i.e, variables which can be explained by other explanatory variables, we perform redundancy analysis. The redundant variables interfere with the one another and the relation between the explanatory and dependent variables is distorted. We use the function *redun* provided in *rms* package to perform the redundancy analysis.

(C-3)Assigning Degrees of Freedom

After removing the correlated and redundant metrics from our datasets, we spend the budgeted degrees of freedom efficiently. We have to identify the metrics which can use the benefit from the additional degrees of freedom (knots) in our models. To identify these metrics we use the Spearman multiple ρ^2 between the explanatory and dependent variable. A strong relation between explanatory variable x_i and the dependent variable y indicates that, x_i will benefit from the additional knots and improve the model. We use *spearman2* function in the *rms* package to calculate the Spearman multiple ρ^2 values for our metrics.

(C-4)Regression Modeling and Validation

After budgeting degrees of freedom to our metrics we build OLS (Ordinary Least Squares) regression model. We use the *restricted cubic splines* to assign the knots to the explanatory variables in our model. As we are trying to identify the relation log related metrics have on the resolution time of bug fixes, we are primarily concerned if log related metrics are significant in our models. To do this, we use the validate function in the *rms* package. We boot strap the data and perform 1,000 iterations with backward elimination enabled to retain the

Table 6: P-Values and Effect Size for comparing code churn, resolution time, # comments and # developers in the bug fixes with and without log change. 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**	0.178(small)	$<$ **2.2e-16**	0.023	$<$ **2.2e-16**	0.155
Resolution time	**4.7e-14**	-0.095	$<$ **2.2e-16**	-0.188(small)	**7.7e-08**	-0.276(small)
# of comments	**2.2e-16**	-0.573(small)	$<$ **2.2e-16**	-0.436(small)	$<$ **2.2e-16**	-0.304(small)
# of developers	$<$ **2.2e-16**	-0.539(small)	$<$ **2.2e-16**	-0.617(medium)	$<$ **2.2e-16**	-0.440(small)

significant metrics in our models. If the metric has ρ value > 0.05 its removed from the model.

(C-5) Calculating Relative Effect

After identifying the significant metrics in our datasets, we find the effect of each explanatory variable on the dependent variable. To do this, we set all the significant metrics to their means and increase one metric by one standard deviation [25,26]. We use the *predict* function to calculate the variable Y_2. The difference $\Delta Y = Y_2 - Y$ describes the effect of each log related metric on the resolution time of bug fixing commits with log changes. A positive effect means a higher value of the log change related metrics increases the resolution time of the bug, whereas a negative effect means that a higher value of the log change related metrics decreases the resolution time of the bug.

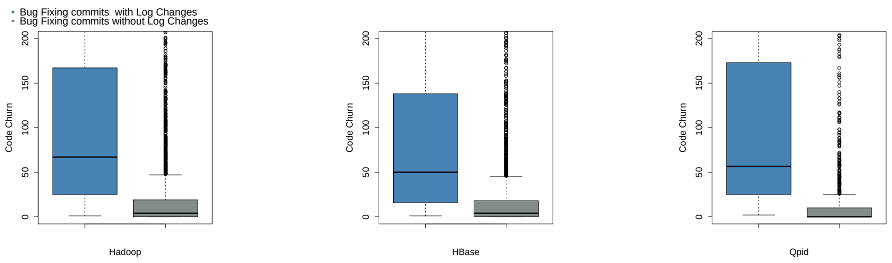
We would like to point out that although non-linear regression has been used to build accurate models for the resolution time of bugs [22], our purpose of using the linear model in this paper is not for predicting the resolution time of bugs. Our purpose is to study the explanatory power of log change 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 change than without log change (see Table 6 and Figure 3). This implies that developers may change logs to fix more complex bugs.

We found that bugs that are fixed with log change 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 change than the ones without log change. This result suggests that given two bugs of same complexity, the one with log changes usually take

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



less time to get resolved and needs fewer number of developers involved with fewer discussions. This suggests that logs provide useful information to assist developers in discussing, diagnosing and fixing bugs. For example, when fixing an issue HBASE-3074 (commit 1005714), 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. Such additional data helps trace the cause of the failure and helps in fixing the bug.

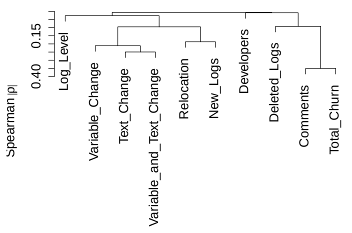
Regression Model Results

In this section, we describe the outcome of the model construction and analysis outlined in our approach and Figure 2

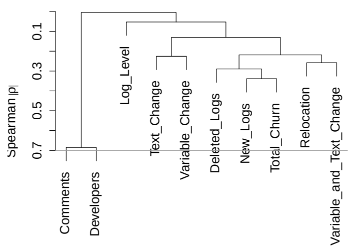
(C-1) Calculating Degrees of Freedom. Out data can support between 123 ($\frac{1,925}{15}$ in Hadoop) and 63($\frac{953}{15}$ in Qpid) degrees of freedom. As we have higher number of knots we can be liberal in their allocation during model construction.

(C-2) Correlation and Redundancy Analysis Figure 4 shows the hierarchically clustered Spearman ρ values of the three systems. The gray dashed line indicates our cutoff value ($|\rho| = 0.7$). Our analysis reveals that # *Comments* and # *developers* are highly correlated in Qpid and HBase. As calculating #*comments* is easier than # *developers*, we exclude #*developers* from our model. We find that there are no redundant variables in our metrics in all the subject systems. Hence, no metrics are eliminated in this step.

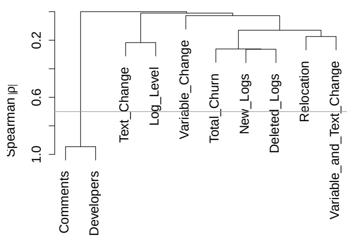
(C-3)Assigning Degrees of Freedom. Figure 5 shows the Spearman multiple ρ^2 of the resolution time against each explanatory variable. Variables which have higher Spearman multiple ρ^2 have higher chance of benefiting from the additional degrees of freedom and can explain resolution time better. From figure 5c we split the explanatory variables into three groups. The first consists of #*comments*, the second consists of #*log level changes*, #*log variable changes*, #*log relocationss* and #*new logging statemetns*. The last group consists remaining metrics as shown in Figure 5c . We allocate five degrees of



a: Correlation between metrics in Hadoop



b: Correlation between metrics in HBase



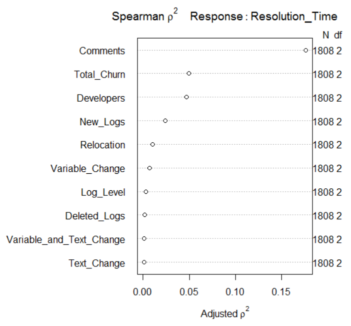
c: Correlation between metrics in Qpid

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

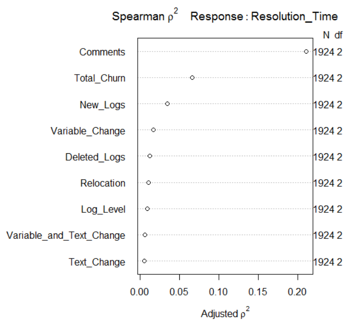
freedom i.e, knots, to the metrics in the first group, three to metrics in second group and no knots to variables in last group as done in prior research [27]. Similar approach is used to assign knots to all subject systems.

(C-4)Regression Modeling and Validation. After allocating the knots to the explanatory variables, we build the non-linear regression model and use the *validate* function in the *rms* package to find the significant metrics in our subject systems. We find that log related metrics are significant in Qpid and HBase systems for predicting resolution time of bug fixes. Figure 6 shows the direction of impact of log related metrics on the resolution of bug fixing commits with log changes in HBase. To conserve space we only present the plot for HBase.

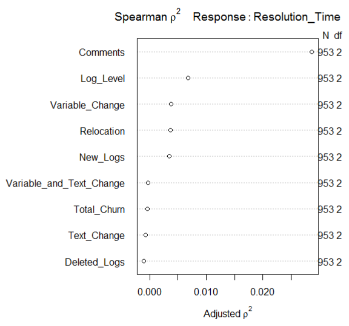
(C-5) Calculating Relative Effect. We find that log modifications have negative impact on the resolution time of bug fixes. From Table 7 we find that in HBase and Qpid, log modifications i.e, Log level changes, Variable Changes are significant and have negative correlation towards resolution time. This may be because when developers that file or module may be well logged and



a: Spearman multiple ρ^2 in Hadoop



b: Spearman multiple ρ^2 in HBase



c: Spearman multiple ρ^2 in Qpid

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

Table 7: Effect of log change 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.

Delta Y	Variable	Delta Y	Variable	Delta Y	Variable
Qpid		**Hadoop**		**HBase**	
0.099604	Comments	0.274173	Comments	0.356256	Comments
-0.069142	Log Level Chanes	0.025857	Total Churn	0.115751	Total Churn
		0.019192	Developers	0.051755	New Logs
				-0.028189	Log Level Chanes
				-0.036716	Variable Changes

developers can identify the problem faster from the existing logs and modify the logs for debugging purposes.

We find that New logging statements have a positive impact on the resolution time of bug fixes in HBase project. This suggests that during some bug fixes, developers might not know the exact cause of the bug and have to add new logs to find where the problem lies. For example in (HBASE-7305) which

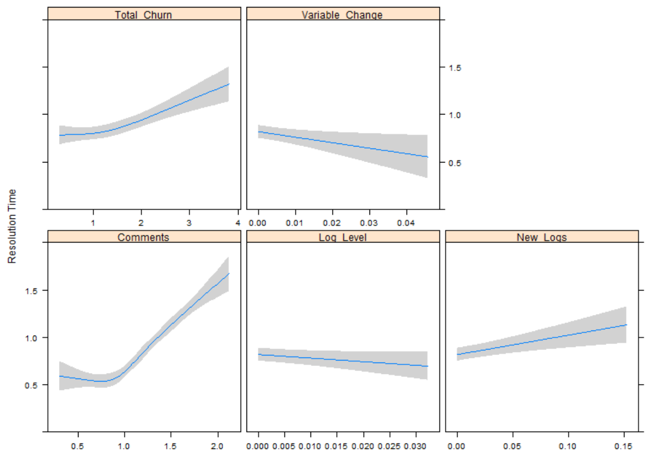


Fig. 6: Relation between the explanatory variables and resolution time of bug fixing commits with log changes in Hbase. Increasing graph shows increase in explanatory variable increases the resolution time and decrease reduces the resolution time

is resolved almost a full year after being open, we observe that developers add over 120 new log lines and many new functionalities in the code. Because of the addition of new functionalities in the code, this takes longer time to resolve than bugs with only log modifications.

Logs are changed during fixing more complex bugs. After controlling the complexity of bugs using code churn, we find that bug fixes with log changes are resolved faster with fewer people and fewer discussions. Log churn related metrics can complement the 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.

4 Quantitative analysis of log change during bug fixes

To further understand how developers change logs in bug fixes, we finally conduct 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. For the sampled commits, we analyze the code changes made in Git and the corresponding JIRA issue reports to find different patterns of log use in bug fixes until we cannot find any new patterns.

4.1 Analysis of Git commits

We analyze the code changes in Git for the same JIRA issue reports from the previous section. We find three reasons of log change during bug fix as shown in Table 8.

Table 8: Log change reasons during bug fix

Projects	Hadoop	HBase	Qpid
Field debugging	157	175	49
Feature changes	156	170	42
Code Refactoring	93	78	18

– *Field Debugging*

  Developers use logs to detect runtime defects. For example in commit 620071 (HADOOP-2725) we observe that developers notice a discrepancy when a 100TB file is copied across two clusters. To help in debugging we observe that developers modify the log variable which outputs the sizes into human readable format instead of bytes. These log changes are committed along with bug fix, as it clarifies the log and helps in easy understanding. These findings are consistent with prior findings where majority of log changes are made during debugging [1].

– *Feature changes*

  Developers use log dumps to find the bug and provide additional logs to make sure the fix does not cause future bugs. For example in commit 637724 (HADOOP-2890) we see that developers leverage logs to identify the reason behind a read file exception. In the commit, we observe that the developers fix this bug by adding new *try catch* block and add new log statements to verify blocks. As these log changes are integral part of the new *try catch* blocks they are committed along with the bug fix.

– *Code Refactoring*

  When developers find critical bugs in the code, they might re-factor the complex functions. For example in commit 663886 (HADOOP-2393), where developers encounter bad jobs, resulting in system timeouts. The developers leverage trace logs to identify that timeouts occur due to expensive synchronization methods in the file. To reduce the costs the developers move the methods into separate functions and add logs in both parent and re-factored function to track the changes made in both functions.

From Table 8 we find that logs are changed primarily for 'Field debugging' and 'Preventative Measures'. We observe that most commits have an overlap of several categories, where developers change logs as preventative measure and also re-factor the code adding new logs in the process. We observe the biggest overlap between field debugging and bug prevention. This is intuitive where developers commit the log changes in debugging to help in future commits.

Prior research finds that 9% of log changes are deleted as they are not necessary [1]. From our quantitative analysis we do not find these cases during bug fixes. In our RQ, we also find that developers do not delete logs during bug fixes.

From this analysis we find that developers leverage logs for several reasons during bug fixes. From analyzing the code changes we find that during bug fixes, logs are primarily used for debugging and also when a bug change involved feature changes. This suggests that logs are an asset to developers during bug fixes and may help in faster resolution of bugs.

5 Related Work

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

5.1 Log Analysis

Prior work leverage log analysis for testing and detecting anomalies in large scale systems. *Shang et al.* [28] propose an approach to leverage logs in verifying the deployment of Big Data Analytic applications. Their approach analyzes logs in order to find differences between running in a small testing environment and a large field environment. *Lou et al.* [10] propose an approach to use the variable values printed in logs to detect anomalies in large systems. Based on the variable values in logs, their approach creates invariants (e.g., equations). Any new logs that violates the invariant are considered to be a sign of anomalies. *Fu et al* [29] built a Finite State Automaton (FSA) using unstructured logs and to detect performance bugs in distributed systems.

Xu et al [6] 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. [30] 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. [31] 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. [32] 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.* [33] propose an approach to automatically infer the failure scenarios when a log is printed during a failed run of a system.

Jiang et al. [34–37] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [34]. Based on the such events, they identified both functional anomalies [35] and performance degradations [36] in load test results. In addition, they proposed an approach that leverage logs to reduce the load test that are performed in user environment [37].

The extensive prior research of log analysis motivate our paper to study how logs are leveraged during bug fixes. Our findings also show that logs are 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.

5.2 Empirical studies on logs

Prior research performs an empirical study on logs and logging characteristics. Yuan et al. [4] 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.* [38] 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 were evaluated by professionals from the industry and F-score of over 95% was achieved.

Shang et al. [39] 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 [1,40]. 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* [21] 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.* [41] shows that logs need to be improved by providing additional information. Their tool named *Log Enhancer* can automatically provide additional control and data flow parameters into logs.

The most related prior research by Shang *et al.* [1] empirically study the relationship of logging practice and code quality. Their manual analysis sheds light on the fact that some logs are changed due to field debugging. They also show that there is a strong relationship between logging practice and code quality. Our paper focused on understanding how logs are leveraged during bug fixes. Our results show that logs are leveraged extensively during bug fixes and assist in a quick resolution of bugs.

6 Limitations and Threats to Validity

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

External Validity

Our study is performed Hadoop, HBase and Qpid. Even though these three subject systems have years of history and large user bases, the three subject systems are all Java based platform systems. Systems in other platforms may not rely on logs in bug fixes. More case studies on other software in other domains with other programming languages are needed to see whether our findings can be generalized.

Internal Validity

Our study is based on the data obtained from Git and JIRA for all the 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 keywords to identify branching and merging commits. Although such keywords are used extensively in prior research [1], 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 in changing of logs during bug fixes, we only use linear model to find out whether the changing logs has 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.

7 Conclusion and Future Work

Logs are used by developers for detecting system anomalies, monitoring performance, software maintenance, capacity planning and also fixing bugs. This paper is a first attempt (to our best knowledge) to understand whether logs are changed more during bug fixes and how these changes occur. The highlights of our findings are:

– We find that logs are changed more during bug 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 changed more during complex bug fixes. However, bug fixes that leverage logs are faster, need fewer developers and have less discussion.
– We find that log modification is significant in modeling the resolution time of bugs. More the log modification 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. 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.

2. 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.
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 *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 110–119.
4. 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.
5. 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.
6. 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.
7. 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.
8. Xpolog. [Online]. Available: http://www.xpolog.com/.
9. logstash, “http://logstash.net.”
10. 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.
11. R. Ihaka and R. Gentleman, “R: a language for data analysis and graphics,” *Journal of computational and graphical statistics*, vol. 5, no. 3, pp. 299–314, 1996.
12. V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, 1966.
13. M. Mednis and M. K. Aurich, “Application of string similarity ratio and edit distance in automatic metabolite reconciliation comparing reconstructions and models,” *Biosystems and Information technology*, vol. 1, no. 1, pp. 14–18, 2012.
14. C. B. Seaman, “Qualitative methods in empirical studies of software engineering,” *Software Engineering, IEEE Transactions on*, vol. 25, no. 4, pp. 557–572, 1999.
15. E. A. Gehan, “A generalized wilcoxon test for comparing arbitrarily singly-censored samples,” *Biometrika*, vol. 52, no. 1-2, pp. 203–223, 1965.
16. V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, “A systematic review of effect size in software engineering experiments,” *Information and Software Technology*, vol. 49, no. 11, pp. 1073–1086, 2007.
17. B. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, J. Rosenberg *et al.*, “Preliminary guidelines for empirical research in software engineering,” *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 721–734, 2002.
18. 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.
19. 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.
20. 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.
21. 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.
22. 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.

23. F. E. Harrell, *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. Springer Science & Business Media, 2013.
24. F. E. Harrell, K. L. Lee, R. M. Califf, D. B. Pryor, and R. A. Rosati, "Regression modelling strategies for improved prognostic prediction," *Statistics in medicine*, vol. 3, no. 2, pp. 143–152, 1984.
25. 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.
26. 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.
27. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, p. To appear, 2015.
28. 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.
29. 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*.
30. 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.
31. 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.
32. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.
38. 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.
39. W. Shang, "Bridging the divide between software developers and operators using logs," in *ICSE '12 :Proceedings of the 34th International Conference on Software Engineering*.

40. 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.
41. 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.