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

An Empirical Study on ~~Logging Statement Changes~~ Changes to Logs During Bug Fixes

Suhas Kabinna · Weiyi Shang · Ahmed E. Hassan

Received: date / Accepted: date

Abstract Logs are leveraged by software developers to record and convey important information during the execution of a system. These logs are a valuable source of information for developers to debug large software systems. Prior research has shown that ~~logging statements~~ logs are changed during field debugging. However, little is known about how ~~logging statements~~ logs are changed during bug fixes. In this paper, we perform a case study on three large open source platform software namely *Hadoop, HBase* and *Qpid*. We find that logs are added, deleted and modified statistically significantly more ~~in~~ during bug fixes than other code changes. Furthermore, we find identify four different types of modifications that developers make to ~~logging statements~~ logs during bug fixes, including: (1)~~logging level change~~modification to logging level, (2)~~text modification~~ modification to logging text, (3)~~variable change~~ modification of logged variable and (4)~~logging statement relocation~~ relocation of log. We find that ~~bugs that are fixed with logging statement changes~~ bug fixes that contain changes to logs have larger code churn, but involve fewer developers, require less time and have less discussion during the bug fix. This suggests that given two bugs of similar complexity, the one which leverages logs and has log changes, has likelihood of being resolved faster. We build a regression model to explore the relationship between ~~metrics from logging statement changes~~ log churn metrics and the resolution time of bugs. We find that ~~these metrics from logging statement changes~~ log churn metrics can complement traditional metrics, i.e., # of developers and # of comments, in explaining the bug resolution . In particular, we find a negative relationship between modifying ~~logging statements~~ logs and the resolution time of bugs. ~~Our results suggests that there is a relationship between changing logging statements and~~

Suhas. Kabinna, Weiyi. Shang and Ahmed E.Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, Ontario
E-mail: {kabinna,swy,ahmed}@cs.queensu.ca

~~the resolution time of bugs~~This means that bug fixes with log modifications have higher likelihood of being resolved faster.

1 Introduction

Platform software provides an infrastructure for a variety of applications that run on top of it. Platform software often relies on logs to monitor the applications that run over ~~them~~it. Such logs are generated through simple *printf* statements or through the use of logging libraries such as 'Log4j', 'Slf4j', and 'JCL'. Each logging statement contains a static 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~~verbosity level that indicates when the log should be output. An example of a ~~logging statement~~ log 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);
```

In our paper we use 'log' to refer to the logging statements in the source code, and use 'output log' to refer to the logs generated during system execution. We use the term 'bug fix' to refer to the commits made to fix a bug, and 'non-bug fix' refers to commits made during improvements, tests, new features and other tasks.

Research has shown that logs are used by developers extensively during the development of software systems [?]. Logs are leveraged for anomaly detection [?,?,?], system monitoring [?], capacity planning [?] and large-scale system testing [?]. The valuable information in logs has created a new market for log maintenance platforms ~~like~~ such as Splunk [?], XpoLog [?], and Logstash [?], which assist developers in analyzing logs.

Logs are extensively used to help developers fix bugs in platform software. For example, in the JIRA issue HBASE-3403[1], a bug was reported when a region is orphaned when ~~there is system failure~~ system failure occurs during a split. Developers leveraged logs to identify the point of failure. After fixing the bug, the logs were updated to prevent similar bugs ~~in the system. Prior~~ from re-occurring. A recent study shows that ~~the changes to logging statements~~ changes to logs have a strong relationship with code quality [?]. However, there exists no large scale study that investigate how ~~logging statements~~ logs are changed during bug fixes.

This information is necessary as previous research shows that 16-32% of logs changes are made during field debugging [?], but how these changes are made is not understood.

In this paper, we perform an empirical study on the changes ~~to logging statements~~ that occur to logs during bug fixes in three open source platform

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

software, i.e., *Hadoop*, *HBase* and *Qpid*. In particular, we sought to answer the following research questions.

RQ1: Are ~~logging statements~~ logs changed more often during bug fixes?
We find that logs are changed more ~~in bug fixing commits~~ often during bug fixes than non-bug ~~fixing commits~~fixes. In particular, we find that adding and modifying ~~logging statements appear statistically significantly more in bug fixing commits~~ logs occurs more often in bug fixes than non-bug ~~fixing commits~~ fixes (statistically significant with non-trivial effect size). We identified four types of modifications to ~~logging statements, including~~ logs, including ~~*Logging*~~ *modification to logging level*~~*change*~~, ~~*Text*~~ *modification to logging text*, ~~*Variable change*~~*modification of logged variable* and ~~*Logging statement*~~ *relocation of log*. We find that ~~*Text*~~ *modification to logging text*, ~~*Variable change*~~*modification of logged variable* and ~~*Logging statement*~~ *relocation of log* ~~exist~~ occur more often in bug fixes than non-bug fixes (statistically significantly more with medium to high effect sizes~~in bug fixing commits than non-bug fixing commits.~~
).

RQ2: ~~Are bugs fixed faster with logging statement changes~~Is there relation between log change and resolution time of bug fixes?
We find that bug ~~fixing commits with logging statement changes have higher code churn~~fixes with log churn have higher total code churn than bug fixes with no log churn. After normalizing the code churn, ~~the bugs with logging statement changes~~ bug fixes with log churn take less time to get resolved, involve fewer developers and have less discussions during the bug fixing process. This means that given two metrics of similar complexity the one with log churn is more likely to be resolved faster and involve fewer developers and less discussion.

RQ3: Can log churn metrics ~~from logging statement changes help in explaining~~ explain the resolution time of bugs?
Using ~~metrics from logging statement changes and the~~ log churn metrics (e.g., new logs added, deleted logs) and traditional metrics (i.e., # comments and # developers) we trained regression models for the resolution time of bug fixes. We find that ~~metrics from logging statement changes~~ log churn metrics are statistically significant in the models and have negative impact on resolution time. This suggests ~~that~~ there is a ~~relationship between logging statement changes and the~~ relation between log churn metrics and resolution time of bugs. The relation shows that log churn has impact on resolution time which should be studied further.

The rest of this paper is organized as follows. Section 2 presents ~~the~~ our methodology for extracting data for our study. Section 3 presents ~~the case studies and the results to answer the~~ our case study and the answers to our three research questions. Section ~~?? discuss the change to logs through a manual study. Section~~ 4 describes the prior research that is related to our

Table 1: An overview of the platform softwares

Projects	Hadoop		HBase		Qpid	
	Bug ~~fixing~~ Fixing	Non-Bug Fixing	Bug ~~fixing~~ Fixing	Non-Bug Fixing	Bug ~~fixing~~ Fixing	Non-Bug Fixing
Total # of ~~Revisions~~ commits	~~7,366~~ 1,808 (49.9 %)	~~12,300~~ 1,809 (50.1 %)	~~5,149~~ 1924 (56.8 %)	~~7,784~~ 1463 (43.2 %)	~~1,824~~ 953 (52.1 %)	~~5,684~~875 (47.9 %)
~~Code Churn~~ Code Churn (LOC)	~~4,09K~~ 246 K	~~3.2~~1.8 M	~~1.4M~~ 653 K	~~2.18~~1.5 M	~~175k~~ 106 k	~~2.3M~~597 K
Log Churn (LOC)	~~4,311~~ 3,536	~~23,838~~ 16,980	4,~~566~~ 672	~~12,005~~ 10,335	~~597~~ 972	~~10,238~~4,958
% of Commits with Log churn	24.0 % (433)	46.2 % (656)	36.2 % (648)	42.1 % (616)	22.1 % (211)	32.8 % (287)

work. Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper.

2 Methodology

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

The aim of this paper is to understand ~~logging statement changes~~ how logs are changed during bug fixes and its relation to resolution time. We conduct a case study on three open source platform software, i.e., *Hadoop*, *HBase* and *Qpid*. All three platform softwares have extensive logging in their source code. Table 1 highlights the overview of the three platform softwares.

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 extensively studied in prior research [**?**,**?**,**?**]. We study the changes to ~~logging statements~~ logs from *Hadoop* releases 0.16.0 to 2.0.

HBase[2]: *HBase* is a distributed, scalable, big data software, ~~using~~ which uses *Hadoop* file-systems. We study the changes to ~~logging statements~~ logs in *HBase* from release 0.10 to 0.98.2.*RC*0. This covers more than four years of development in *HBase* from 2010 to 2014.

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

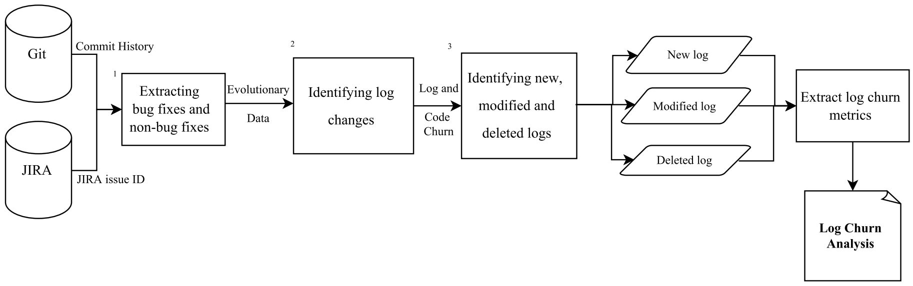


Fig. 1: Overview of our ~~east~~ case study approach

Qpid[3]: *Qpid* is an open source messaging platform that implements an Advanced Message Queuing Protocol (AMQP). We study *Qpid* release 0.10 to release 0.30 that are from 2011 till 2014.

Figure 1 shows a general overview of our approach, which consists of four steps: (1) We mine the Git repository of each ~~subject~~ studied system to extract all commits and identify ~~bug fixing~~ the bug fixes and non-bug ~~fixing commits~~fixes. (2) We identify ~~logging statement~~ log changes in both bug ~~fixing~~ fixes and non-bug ~~fixing commits~~fixes. (3) We categorize the ~~logging statement~~ log changes into *'New logs'*, *'Modified Logs'* and *'Deleted logs'*. (4) We calculate churn metrics for each category and use statistical tool R [**?**], to perform experiments on the data to answer our research questions. In the reminder of this section we describe the first three steps.

2.1 Study Approach

In this section we present the approach of our case study.

2.1.1 Extracting ~~bug-fixing~~ bug fixes and non-bug ~~fixing commits~~ fixes

The first step in our approach is to extract ~~bug fixing~~ commits associated with bug fixes and non-bug ~~fixing commits~~fixes. First, we extract a list of all commits from ~~Git~~the Git repository of each project. To avoid the branching and merging commits, we enable the 'no-merges' option in the Git *log* command~~to exclude all the merging operations in the systems~~. This flattens all the changes made to file in different branches but only exclude the merge between branch and trunk. We also filter the non-Java and 'test' files present in the commits.

Next, we extract a list of all the JIRA issues that have the type 'bug'. Developers often mention the JIRA issue ID's in the commit messages. We

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

search JIRA issue IDs in the commit messages to identify all the bug ~~fixing commits. If a commit message does not contain a JIRA issue ID, 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~~ fixes. We exclude commits which do not have JIRA IDs because we cannot extract developer metrics for those issues.

2.1.2 Identifying ~~logging statement~~ log changes

To identify the ~~logging statement~~ log changes in the datasets, we first manually explore ~~logging statements~~ logs in the source code. Some ~~logging statements~~ logs are specific to a particular project. For example, a ~~logging statement~~ log from Qpid invokes ‘QPID_LOG’ to print logs as follows:

> *QPID_LOG(error, “Rdma: Cannot accept new connection (Rdma exception): ” + e.what());*

Some ~~logging statements~~ logs leverage logging libraries to print logs. For example, *Log4j*[2] is used widely in *Hadoop* and *HBase*. In both projects, ~~logging statements~~ logs have a method invocation ‘LOG’, followed by logging-level. The following ~~logging statement that uses *Log4j*to print logs:~~ log uses *Log4j*:

> *LOG.debug(“public AsymptoticTestCase(String”+ name +“) called”)*

Using regular expressions to match these ~~logging statements~~logs, we automate the process of finding all the ~~logging statements in our datasets.~~logs in the studied projects.

2.1.3 Identifying new, modified and deleted ~~logging statements~~logs

Since, Git *diff* does not provide a feature to track modification to ~~a file~~the code, modifications to ~~logging statements are shown as~~ a log is shown as a deletion followed by an addition. To track these added and deleted ~~logging statements. We~~ logs we used Levenshtein ratio [?]~~to identify modifications to logging statements~~. For every pair of added or deleted ~~logging statement~~ logs in a commit, we compare the text in parenthesis after removing the logging method (e.g, LOG) and the log level (e.g, info). We calculate the Levenshtein ratio between the added and deleted ~~logging statement~~ log similar to prior research [?]. We consider a pair of added and deleted ~~logging statements as log~~ log as a modification if they have a Levenshtein ratio of 0.6 or higher. For example, the ~~logging statements~~ logs shown below have Levenshtein ratio of 0.86. Hence ~~this logging statement change~~ such a log churn is categorized as a log modification.

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

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

If an added or deleted ~~logging statement matches with more than one deleted or added logging statements with over~~ log has a levenshtein ratio higher than 0.6~~Levenshtein ratio~~, we consider the pair of added and deleted ~~logging statements~~ logs with the highest ~~Levenshtein ratio as modifications to logging statements~~levenshtein ratio as a log modification. After identifying ~~log modifications to logging statements, we identify modified, new and deleted logging statements~~all log modifications we identify three categories of logs in a commit namely 'newly added logs', 'deleted logs' and 'modified logs'.

3 Study Results

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

~~RQ1: Are logging statements changed more during bug fixes?~~RQ1: Are logs changed more often during

Motivation

Prior research ~~has shown~~ finds that up to 32% of the ~~logging statements changes are~~ churn to logs, is due to field debugging [?]. During debugging, developers change ~~logging statements~~ logs to gain more run-time information about ~~the systems. Therefore,~~ their system. These log changes may also assist developers in resolving future occurrences of a similar bug~~may be resolved easily with the updated logging statements~~. However, to the best of our knowledge, there exists no large scale empirical study to show whether ~~logging statements~~ logs are changed more often during bug fixes than other ~~activities during development~~ development activities. In addition, we want to investigate how ~~logging statements~~ logs are changed during the bug ~~fixing.~~

fixes. This helps in understanding what developers value as important knowledge during bug fixes.

Approach

We compare the number of changes to ~~logging statements between bug-fixing and non-bug-fixing commits~~logs between bug fixes and non bug fixes. In previous section, we identified three types of changes to ~~logging statements~~logs, i.e., modified, new and deleted ~~logging statements~~logs. Therefore, we compare the number ~~of each type of changes to logging statements between bug-fixing and non-bug-fixing commits~~changes to logs within the three types in bug fixes and non-bug fixes. Since commits with higher total code churn may have a higher number of changes to ~~logging statements. Therefore~~logs, we calculate total code churn for every commit and use it to normalize *# modified, # new* and *# deleted* ~~*logging statements*~~*logs*. The three new metrics are:

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

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

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

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

To future understand how ~~logging statements~~ logs are modified during bug fixes, we perform a manual analysis on the modified ~~logging statements~~ logs to identify the different types of log modifications. We first collect all the commits that modify ~~logging statement~~log. We select a random sample of 357 commits. The size of our random sample achieves a 95% confidence level and 5% confidence interval. We follow an iterative process, similar to prior research [?], to identify the different types of log modifications, until we cannot find any new types of modifications.

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

To ~~determine~~ identify what type of knowledge developers favor during bug fixes, we find whether there is a statistically significant difference ~~of these metrics, in bug-fixing and non-bug-fixing commits, we perform~~ in the log churn metrics, between bug fixes and non-bug fixes. To do this we use the *MannWhitney U test* (Wilcoxon rank-sum test) [?]~~. We choose MannWhitney U testbecause~~ , as our metrics are highly skewed. Since the *MannWhitney U test* is a non-parametric test, it does not have any assumptions about the distribution of the sample population. A p-value of ≤ 0.05 means that the difference between the ~~two data sets~~ bug fixes and non-bug fixes, is statistically significant and we may reject the null hypothesis~~(i.e., there is no statistically significant difference of our metrics between bug-fixing and non-bug-fixing~~

Table 2: Comparing the log density between bug fixes and non-bug fixes (the value of 1.9 for Hadoop means logs are modified 1.9 times more during bug fixes compared to non-bug fixes).

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

~~commits)~~. By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us that there is a statistically ~~significantly difference of~~ significant difference for our metrics between ~~bug-fixing and non-bug-fixing commits~~bug fixes and non-bug fixes.

We also ~~use~~ calculate the *effect sizes* ~~to measure how big is the difference of~~ in order to quantify the differences in our metrics between the ~~bug-fixing and non-bug-fixing commits. Unlike~~ bug fixes and non-bug fixes. Unlike the *MannWhitney U test*, which only tells us whether the difference between the two distributions ~~are~~ is statistically significant, ~~effect sizes quantify~~ the effect size quantifies the difference between the two distributions. Researchers have shown that reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, the p-value ~~can~~ are likely to be small even if the difference is trivial). We use *Cohen's d* to quantify the ~~effects. Cohen's dmeasures the effect sizestatistically, and has been used in prior engineering studies~~effect size [?,?]. *Cohen's d* is defined as:

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

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

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

~~***Results***~~

Results

Developers are more likely to add new ~~logging statements more during bug fixes~~logs when fixing bugs. From Table 2, we find that new log density is 1.2-1.4 times in bug fixes than non-bug fixes. This shows that new logs are 1.2-1.4 times more likely to occur in bug fixes that non-bug fixes. Table 3 shows that ~~*new logging statements ratio* in bug-fixing commits is~~

Table 3: Comparing ~~logging statement changes~~ log churn metrics between the ~~bug-fixing~~ bug fixes and ~~non-bug-fixing commits~~non-bug fixes. The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size ~~means~~ shows that ~~bug-fixing commits have larger metric values and p-values~~ the log changes are ~~bold if they~~ more frequent during bug fixes and P-values are smaller than 0.05.

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

~~statistically significantly higher than non-bug-fixing commits in all subject systems~~ *new logs ratio* in bug fixes is higher than non-bug fixes in all studied systems (statistically significant with non-trivial effect sizes~~. This suggests that developers add new logging statements~~). This suggest that developers may need additional information during bug fixes ~~more than non-bug-fixing commits. We find that effect size of new logging statement is higher in *Qpid* than *Hadoop* and *HBase*. This may be because that *Qpid* is a relatively newer system. Some important source code may not be well logged. Therefore, developers may have to add additional logging statements to assist in bug-fixing.~~

and they add new logs to help fix bugs.

Developers ~~may not delete logging statements~~ are less likely to remove logs during bug fixes. We find that although the difference of *deleted logging statements ratio* between ~~bug-fixing commits and non-bug-fixing commits~~ bug fixes and non-bug fixes is statistically significant in all projects, the effect sizes is trivial for *HBase* and negative for *Qpid* (see Table 3). ~~This result shows that developers of Qpid delete logging statements more during non-bug-fixing commits than bug-fixing commits~~Moreover, we find that the log density varies between 0.5-2.4. This suggests developers are more likely to remove logs during bug fixes in *Qpid*. Such results confirm the findings from prior research that deleted ~~logging statements~~ logs do not have a strong relationship with code quality [**?**].

Logging statements are ~~modified~~ more likely to be modified in ~~bug-fixing commits than non-bug-fixing commits~~bug fixes and non-bug fixes. From Table 2 we find that logs are 1.9-4.1 times more likely to be

Table 4: Distribution of four types of log modifications.

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

modified during bug fixes than non bug fixes. Table 3 shows that ~~*modified logging−statements ratio* is statistically significantly higher n bug-fixing commits than non-bug-fixing commits for all subject systems and the effect sizesare non-trivial in all these systems~~*modified log− ratio* in bug fixes is higher than non-bug fixes in all studied systems (statistically significantly with non-trivial effect sizes). Such results show that developers often change the information provided by ~~logging statements~~ logs to assist in ~~bug-fixing~~bug fixes. Developers may need different information to the information that is provided by the ~~logging statements~~ logs to fix the bugs. Prior research ~~also finds that 33~~find that 66 % of ~~logging statements are modified at-least once as after-thoughts~~the logs are modified when 1) the condition the log code depends on is changed, 2) the logged variable is changed or 3) the function name which is also referred in static text of log is modified [?]. Therefore, we further explore ~~how developers modify logging statements~~the different types of modifications to logs.

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

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

Developers leverage different information from the log output, more in bug fixes. From Table 2 we find that variables in logs are 1.7-4.4 times more likely to be modified during bug fixes. We find that changes to the logged variables is higher in bug fixes and non-bug fixes as seen in Table 6 (statistically significant with medium or small effect sizes). This suggests that developers change the logged variables in order to provide useful information in the log outputs.

Table 5: Comparing ~~logging modification metrics between~~ the ~~bug-fixing and non-bug-fixing commits. The p-value is from MannWhitney U tests~~ log density between bug fixes and ~~the effect sizes are calculated using Cohen's d~~non-bug fixes. ~~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~~

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

Prior research also shows that 16-32% of log changes are made during field debugging [?]. Therefore, to better understand how developers change logged variables during bug fixes, we categorize the changes into three types: a) variable addition, b) variable deletion and c) variable modification.

Table 7 shows that developers modify ~~variables statistically significantly more in *Hadoop* and *Qpid*~~ the logged variables more in bug fixes than non-bug fixes(statistically significant with medium effect sizes~~. This~~). This modification may be because developers need different information in the output logs, than provided by existing ~~logging statements~~logs. For example, when we observe the patch notes for bug QPID-2370[3], we find that developers modify the existing ~~logging statement~~ log to capture the newly defined token. The other reason may be that developers change the name of the existing variable to a more meaningful name. For example, in bug MAPREDUCE-2264[4], we find that developers rename variables and modify the ~~logging statements accordingly.~~logs accordingly.

From Table 7, we observe that the developers add variables ~~into logging statements statistically significantly more in bug-fixing commits than non-bug-fixing commits in *HBase* (~~more in bug fixes than non-bug fixes(statistically significant with medium effect size ~~) and *Qpid* (~~in *HBase* and large effect size ~~). This suggests that the~~ in *Qpid*). We find from Table 2 that developers are 1.3-25.2 times more likely to add new variables during bug fixes. This addition of variables suggests, existing variables may not have all the needed information ~~for fixing bugs. Developers~~ in the log output and developers have to add new variables ~~into logging statements during bug-fixes~~during bug fixes. We also find that developers do not delete variables in ~~logging statements~~logs. Deleting variables in ~~logging statements~~ logs may change the format of the ~~logging statements~~output logs. There may be log processing tools that rely on these ~~logging statements~~output logs. Deleting variables may impact the correctness

[3] https://issues.apache.org/jira/browse/QPID-2370
[4] https://issues.apache.org/jira/browse/MAPREDUCE-2264

Table 6: Comparing logging modification metrics between the bug fixes and non-bug fixes . The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size shows that the log changes are more frequent during bug fixes and p-values are smaller than 0.05

Metrics	Hadoop		HBase		Qpid	
	P-values	Effect Size	P-values	Effect Size	P-values	Effect Size
~~Logging statement relocation~~ Relocation of log	**1.1e-10**	0.330 (small)	**3.0e-11**	0.170 (small)	**1.8e-08**	0.700 (medium)
~~Text modification~~ Modification to logging text	-	-	**0.0075**	0.525 (small)	**4.5e-06**	0.976 (medium)
~~Variable change~~ Modification of logged variable	**1.3e-04**	0.351 (small)	**0.0010**	0.420 (small)	**1.2e-04**	1.17 (medium)
~~Logging level change~~ Modification to logging level	-	-	-	-	-	-

of these log processing applications [?]. Therefore, developer may be award of this and try to avoid deleting variables from ~~logging statements~~logs.

Developers modify logged text more during bug fixes. We find that ~~text modification is statistically significantly more in bug-fixing commits than non-bug-fixing commits~~ modification of logged text is higher in bug fixes and non-bug fixes with non-trivial effect sizes (see Table 6). In some cases, the text description in ~~logging statements~~ logs is not clear and developers need to improve the text to ~~help fix bugs~~provide more clarity. For example, in *HBase* HBASE-6665[5] developers modify the ~~logging statement~~ log to provide more information ~~region splits~~about the regions being split. Prior research shows that ~~there is a challenge to understand logging statements in practice~~ up-to 39% of modifications to logged text is due to inconsistency between the execution event and the message conveyed by log output [?]. Our results ~~show~~ suggests that developers may have faced such challenges ~~and may need to improve~~ during bug fixes and may have modified the text in ~~logging statements for better bug-fixing~~logs for clarification.

5 https://issues.apache.org/jira/browse/HBASE-6665

Table 7: Comparing ~~variable change metrics~~ the different types of changes to variables between the ~~bug-fixing~~ bug fixes and ~~non-bug-fixing commits~~non-bug fixes. The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. A positive effect size ~~means~~ shows that ~~bug-fixing commits have larger metric values~~ the log changes are more frequent during bug fixes 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
Variable addition	-	-	**0.00049**	0.659 (medium)	**0.005**	1.40 (large)
Variable deletion	-	-	-	-	-	-
Variable modification	**4.11e-05**	1.045 (medium)	-	-	**0.0016**	0.949 (medium)

~~**Logging statement relocation:** The logging statement is not changed but moved to a different place in the file. **Text modification:** The text that is printed in the logging statements is modified. **Variable change:** One or more variables in the logging statements are changed (added, deleted or modified). **Logging level change:** The verbosity level of logging statements are changed. This means that in most cases the existing logging messages do not convey all the information necessary for a bug fix. An example of this type of change is shown below:~~
~~+ log.trace("setConnectionURL(" + Util.maskUrlForLog(connectionURL) ")");~~
~~- log.trace("setConnectionURL(" + connectionURL + ")");~~

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

~~**Developers modify variables more in bug-fixing commits.** We find that variable changes are statistically significantly more in bug-fixing commits than non-bug-fixing commits in all the subject systems with small or medium effect sizes (see Table 6). Developers modify the variables that are 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 changes into three types: a) variable addition , b) variable deletion and c) variable modification .~~

Projects	Variable addition density ratio	Variable deletion density ratio	Variable modification density ratio
Hadoop	25.2	4.6	4.7
HBase	1.3	1.6	0.8
Qpid	25.2	4.6	4.7

Logging statement relocation occurs more in bug fixes. Table 4, shows that there are a large number of logging changes that only relocate ~~logging statements~~logs. Table 6 shows that such relocation of ~~logging statements~~ logs is statistically significantly more in ~~bug-fixing commits than non-bug-fixing commits~~bug fixes and non-bug fixes (2.0-4.9 times more likely in bug fixes as shown in Table 2). We manually examine such commits and find that developers often forget to leverage exception handling or using proper condition statements in the code. After fixing the bugs, developers often move existing ~~logging statements~~ logs into the *try/catch* blocks or after condition statements. For example, in the YARN-289 [6] of Hadoop, ~~logging statements~~ logs are placed into the proper *if-else* block.

Logging levels are not modified often during bug fixes. We find that logging level changes are statistically indistinguishable ~~between bug-fixing and non-bug-fixing commits in all subject~~ betweenbug fixes and non-bug fixes in all studied systems. The reason may be that developers are able to enable all the ~~logging statements during bug-fixing~~logs during bug fixes, despite of what level a ~~logging statement~~ log has. In addition, prior research shows that developers do not have a good knowledge about how to choose a correct logging level [**?**].

> *Developers change logs statistically significantly more in bug fixes than non-bug fixes in given file. In particular, developers modify logs to add or change the variables in logs during bug fixes. This suggests that developers often realize the needed information to be logged as after-thoughts and change the variables in log to assist in fixing bugs.*

~~RQ2: What types of modifications to logs are more frequent during bug fix ?~~RQ2: Is there relation betw

Motivation

~~From RQ 1 we found~~ In RQ1, we find that logs are ~~modified more during~~ changed more frequently in bug fixes. ~~In this RQ, we want to know how logs are leveraged~~ However, we do not know if leveraging and changing logs is beneficial during bug fixes~~, in particular the different types of modifications to logs.~~

~~*Approach*~~

~~We performed a manual analysis on the modified logging statements to identify the different types of log modifications. We first collected all the commits which had logging statement changes in our projects. We selected a 5random sample from all the commits with logging statement changes. We followed a iterative process to identify the different types of logging modifications, that developers make in the source code till we cannot find any new types of modifications.~~

~~For every commit, we found calculate~~ . To answer this, we look at the effort spent to fix the bug. We measure effort spent based on the time taken to resolve a bug, the number of developers involved during the resolution of a bug and the number of ~~modifications in each category and used total churn as the controlling measure. The four metrics are- (1) Relocating log Churn, (2)~~

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

~~Text Modification Churn, (3) Variable Modification Churn and (4) Logging Level Churn.~~

~~*Results*~~

~~**RQ2: Are bugs fixed faster with logging statement changes?**~~

~~*Motivation*~~

~~In RQ1, we find that logging statements are changed more frequently in bug fixes. However, there is no study to show if logging statements are useful in debugging process. As a first step of exploring the usefulness of logging statement changes during bug fixes, we try to find out whether bug fixes with logging statement changes are fixed faster than bug fixeswithout logging statement changes.~~

discussions posts on JIRA. We try to find whether there is a correlation between resolution time, developer involvement and log churn during bug fixes.

Approach

To find out whether bugs are fixed faster with ~~logging statement changes~~log churn, we collect all JIRA issues with type 'bug' from the three ~~subject~~ studied systems. We obtained the code commits for each of these JIRA issues by searching for the issue id from the commit messages. We identify the ~~logging statement changes~~log churn, and the code churn for fixing each issue. We then split the JIRA issues into (1) bugs that are fixed with ~~logging statement changes~~ log churn and (2) bugs that are fixed without ~~logging statement changes~~log churn. We use the code churn to measure the complexity of the issue. We then extracted three metrics from JIRA issues to measure the effort of fixing a bug:

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

Table 9: Comparing code and developer metrics between the ~~bug-fixing commits~~ bug fixes with log ~~changes~~ churn and ~~bug-fixing commits~~ bug fixes without log ~~changes~~churn. The p-value is from MannWhitney U tests and the effect sizes are calculated using Cohen's d. ~~A positive~~ P-values are smaller than 0.05 and negative effect size ~~means~~ implies that ~~bug-fixing commits with~~ metrics for bug fixes without log ~~changes~~ churn have ~~larger metric~~ higher values~~and p-values are bold if they are smaller than 0.05~~.

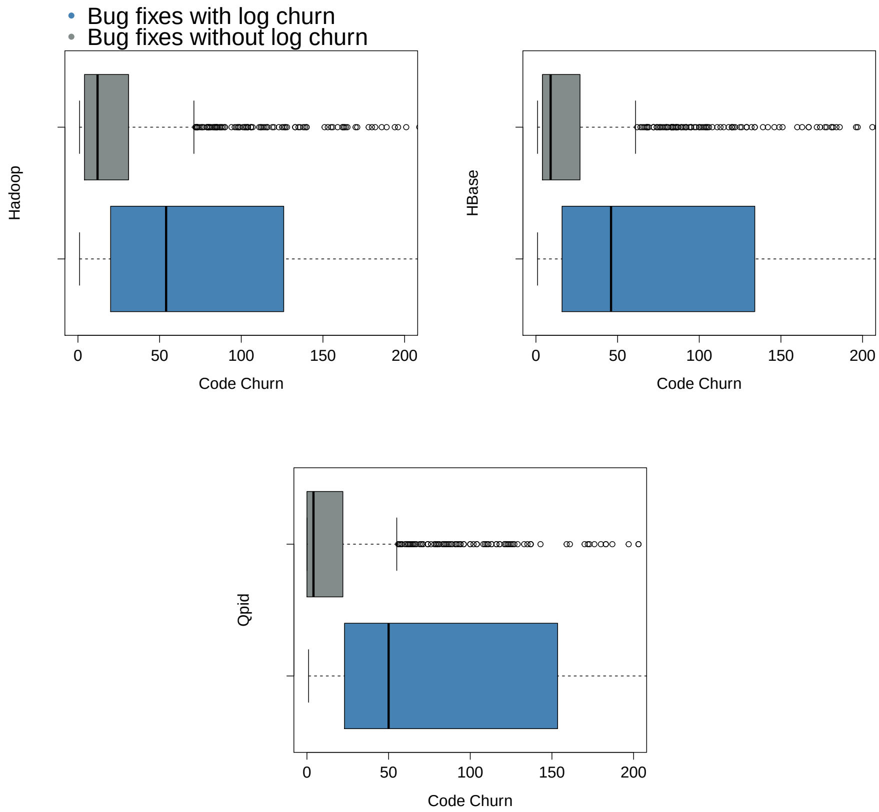
Metrics	Hadoop		HBase		Qpid	
	p-values	Effect size	p-values	Effect Size	p-values	Effect size
Code churn	**<2.2e-16**	0.178 (small)	**<2.2e-16**	0.023	**<2.2e-16**	0.155
Resolution time	**4.7e-14**	-0.095	**<2.2e-16**	-0.188 (small)	**7.7e-08**	-0.276 (small)
# of comments	**2.2e-16**	-0.573 (small)	**<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)

~~We first compare the code churn of bug fixes with and without logging statement changes. Similar to RQ1, we use MannWhitney U testto 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 logging statement changes~~Prior research has shown that complexity of software can be measured using different metrics, including source lines of code change [?]. Intuitively, ~~fixing a more complex bug requires longer time , more people and more discussion~~a complex bug fix might have more code change and in turn takes longer time to be resolved, more developers being involved and more discussions on JIRA. Therefore, we use code churn to normalize the resolution time, the number of comments, and the number of developers ~~when compare them.~~ during bug fixes. We use these normalized effort metrics to find if there is a statistically significant different between bug fixes with log churn and bug fixes without log churn. We use the *MannWhitney U* test to find the $|\rho|$ values and *Cohen's d* test to measure the effect size, similar to RQ1.

Results

We find that the ~~logging statements~~ logs changes are ~~used to fix~~ more likely to occur during complex bugs fixes. We find that the average code churn for fixing bugs is significantly higher with ~~logging statement changes than without logging statement changes~~ log churn than without log churn (see Table 9 and Figure 2). This ~~implies that developers may change logging statements to fix more complex bugs~~suggests that complex bug fixes are more likely to have log churn (statically significant with non-trivial effect size).

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



~~Boxplot of code churn of bug fixing commits with logging statement change (shown in blue) against bug fixing commits without logging statement change (shown in grey).~~

We find that bugs that are fixed with ~~logging statement changes~~log churn, take shorter time with fewer comments and fewer people. After normalizing the code churn, we find that the resolution time, the number of comments and the number of developers are all statistically significantly smaller in the bug fixes with ~~logging statement changes~~ log churn than the ones without ~~logging statement changes~~log churn. This result suggests that given two bugs of the same complexity, the one with ~~logging statement changes~~ log churn usually take less time to get resolved and needs a fewer number of developers involved with fewer discussions. Logging statements may provide useful information to assist developers in discussing, diagnosing and fixing bugs. For example, when fixing bug HBASE-3074[7], developers left the first comment to

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

provide additional details in the ~~logging statement~~ log about where the failure occurs. In the source code, developers add the name of the servers into the the ~~logging statements~~logs. Such additional data helps diagnose the cause of the failure and helps fix the bug.

To further understand how developers change logs in bug fixes, we finally conduct a qualitative analysis. We collected all the bug fixes with log churn for our studied systems. We selected a 5% random sample (266 for *HBase*, 268 for *Hadoop* and 83 for *Qpid*) from all the commits. For the sampled commits, we analyze the code changes made in Git and the corresponding JIRA issue reports to find different patterns of log use in bug fixes. We follow an iterative process, similar to prior research [**?**], until we cannot find any new types of patterns. We find three reasons of changing logs during bug fixes as shown in Table 11. We find that these three reasons can co-occur within a single commit.

Table 10: Log change reasons during bug fix

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

– *Bug diagnosis*

Developers use logs to detect runtime defects. Developers change log to print extra or different information into logs during the execution of the system. We categorize a log change as bug diagnosis only when there is log change has added and deleted code prior to it (code block is changed). For example, to fix HADOOP-2725 [8] we observe that developers notice a discrepancy when a 100TB file is copied across two clusters. To help in debugging we observe that developers modify the log variable which outputs the sizes of the files into human readable format instead of bytes. These log changes are committed along with bug fix, as it clarifies the log and helps in easy understanding.

– *Future bugs prevention*

After fixing a bug, developers may insert log into the code. Such logs monitor the execution of the system to track for similar bugs in the future. We categorize a log change as future bug prevention, when prior to the

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

log change there is addition of new blocks (i.e., if, if-else, try-catch and exception) with code deletion. For example in HADOOP-2890 [9] we see that developers leverage logs to identify the reason behind blocks getting corrupted. In the commit, we observe that the developers fix this bug by adding new checks to verify where the block gets corrupted and they add*try catch* block with new logs to catch these exceptions. The log will notify developers with useful information to diagnose the bug if a similar bug appears.

– *Code quality assurance during bug fixes*

Sometime, developers need to introduce a large amounts of code to fix a bug with no code deletion. The introduction of new code, may introduce new bugs into the system. To ensure the quality of these bug fixes, developers insert logs into the bug-fixing code. For example in HBASE-3787 [10], where developers encounter a non-idempotent increment which causes an error in the application. This fix involves over 13 developers and 112 discussions over the two years. The developers add several new files and functions during the bug fix and new logs to assure the code quality of the fix.

> *Logging statements are changed during fixing more complex bugs. After normalizing the complexity of bugs using code churn, we find that bug fixes with log churn are resolved faster with fewer people and fewer discussions.*

RQ3: Can log churn metrics ~~from logging statement changes~~ help in explaining the resolution time of bugs?

Motivation

~~In RQ2, we find that bugs that are fixed with logging statement changes take shorter time to get resolved than bugs fixed without logging statement changes.~~ Prior research has shown that resolution time is correlated to the number of developers and the number of comments in an issue report [**?**]. ~~We want to see whether the metrics from logging statement changes (as shown in RQ1) can complement the number of developers and the number of comments in modelling the resolution time of bugs~~However, this does not look at the type of changes made by developers.

In RQ2, we find that bug fixes with log churn take shorter time to get resolved than bug fixes without log churn. To explore this correlation we build prediction models using metrics from prior research and log churn metrics. This helps in understanding the effect of log changes during bug fixes and also identify which types of log changes can be beneficial during bug fixes.

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

Approach

To better understand the usefulness of ~~logging statement changes~~ log churn metrics on the resolution time for fixing bugs, we build a non-linear regression model~~for~~. Prior research has shown that linear modelling can help in predicting the resolution time of ~~bugs.~~bug [?]. However, the relation between resolution time of bug fixes and log churn metrics may be non-monotonic. By building a non-linear regression model we can more appropriately approximate the relationship between resolution time and log churn metrics, during bug fixes.

A non-linear regression model fits the curve of the form $y = \alpha + \beta_1 x_1 + \beta_2 x_x + .. + \beta_n x_n$ to the data, where y is the dependent variable (i.e., resolution time of bugs) and every x_i is an explanatory metrics. The explanatory metrics include the ~~metrics from logging statement changes~~ log churn metrics (as shown in RQ1). Since prior research finds that the number of developers and the number of comments in an issue report are correlated to the resolution time of the issue report, we include the number of developers and the number of comments as explanatory metrics. We find that bugs with more complex fixes may take longer time to resolve (see RQ2). Therefore, we also include code churn as explanatory metrics. We use the *rms* package [?] from R, to build the non-linear regression model. The overview of the modeling process is shown in Figure 3 and is explained below.

(C-1) Calculating the degrees of freedom

During predictive modeling, a major concern is over-fitting. An over-fit model is biased towards the dataset from which it is built and will not well fit other datasets. In non-linear regression models, over-fitting may creep in when a explanatory metrics is assigned more degrees of freedom than the data can support. Hence, it is necessary to calculate a budget of degrees of freedom that a dataset can support before fitting a model. We budget $\frac{x}{15}$ degrees of freedom for our model as suggested by prior research [?]. Here x is the number of rows (i.e, # bugs) in each project.

(C-2) Correlation and redundancy analysis

Correlation analysis is necessary to remove the highly correlated metrics from our dataset. We use Spearman rank correlation to assess the correlation between the metrics in our dataset. We use Spearman rank instead of Pearson correlation because Spearman rank correlation is resilient to data that is not normally distributed. We use the function *varclus* in R to perform the correlation analysis. From the hierarchical overview of explanatory metrics constructed by the *varclus* function, we exclude one metric from the sub-hierarchies which have correlation $|\rho| > 0.7$.

Correlation analysis does not indicate redundant metrics, i.e, metrics that can be explained by other explanatory metrics. The redundant metrics can

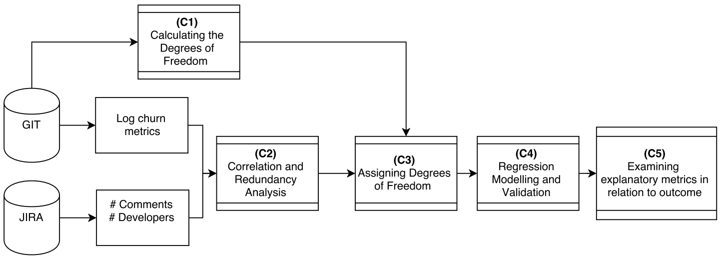


Fig. 3: Overview of our non-linear OLS model construction(C) for the resolution time of bugs

interfere with the one another and the relation between the explanatory and dependent metrics is distorted. We perform redundancy analysis to remove such metrics. We use the ~~function *redun* provided in~~ *redun* function that is provided in the *rms* package to perform the redundancy analysis.

(C-3)Assigning degrees of freedom

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

(C-4)Regression modeling and validation

After budgeting degrees of freedom to our metrics we build a non-linear regression model using the function OLS (Ordinary Least Squares) ~~command~~ that is provided by the *rms* package. We use the *restricted cubic splines* to assign the knots to the explanatory metrics in our model. As we are trying to identify the relationship between ~~metrics from logging statement changes~~ log churn metrics and the resolution time of bug fixes, we are primarily concerned if the ~~metrics from logging statement changes~~ log churn metrics are significant in our models. Therefore, we use ~~the backward (step-down) metric selection method~~ chunk test (a.k.a Wald test) to determine the statistically significant metrics ~~that are~~ to included in our final ~~models~~model. We choose ~~the backward selection method since prior research shows that backward selection method usually~~

~~performs better than the forward selection approach . The backward selection process starts with using all the metrics as predictors in the model~~chunk test as some of our explanatory variables are allocated several degrees of freedom and have to be tested jointly, as done in previous research [?]. At each step ~~, we remove the metrics that is the least significant in the model. This process continues until all the remaining metrics are significant. We use the fastbw (Fast Backward Variable Selection)~~ in the 'wald' test, we measure the significance of each metric according to its p-value. We consider only those metrics which have p-value lower than 0.05 for the final model. We use 'wald.test' function provided by the R package ~~rms~~ *aod* [?] to perform the ~~backward metric selection process.~~

chunk test.

(C-5) Examining explanatory metrics in relation to outcome

After identifying the significant metrics in our datasets we find the relation between each explanatory metric and the resolution time of bugs. In our regression models, each explanatory metric can be explained by several model terms. To account for the impact of all model terms associated with an explanatory metric, we plot the changes to resolution time against each metric, while holding the other metrics at their median value using the *Predict* function in the *rms* package [?]. The plot follows the relationship as it changes directions at ~~knot~~ the spline (knot) locations(C-3).

~~However, to quantify the effects of the significant metrics on resolution time we adopt method suggested in prior research . We first set all the significant metrics to their means and then increase one metric by one standard deviation. We use the *predict* function to calculate the variable Y_2. The difference $\Delta Y = Y_2 - Y$ describes the effect of each metrics from logging statement changes on the resolution time of bug fixing commits with logging statement changes. A positive effect means a higher value of the metrics from logging statement changes increases the resolution time of the bug, whereas a negative effect decreases the resolution time of the bug.~~

We would like to point out that although non-linear regression models can be used to build accurate models for the resolution time of bugs, our purpose of using the non-linear regression models in this paper is not for predicting the resolution time of bugs. Our purpose is to study the explanatory power ~~of metrics from logging statement changes~~ log churn metrics and explore their empirical relationship to the resolution time of bugs.

Results

In this subsection, we describe the outcome of the model construction and analysis outlined in our approach and Figure 3

(C-1) Calculating degrees of freedom. Our data can support between 123 ($\frac{1,925}{15}$ in Hadoop), 63($\frac{953}{15}$ in Qpid) and 183($\frac{2,755}{15}$ in HBase) degrees of

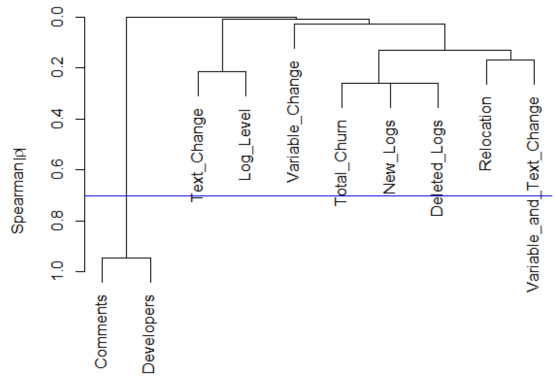


Fig. 4: Correlation between metrics in Qpid. Dashed line indicates cut of set to 0.7

freedom. As we have ~~higher number of knots~~ large degrees of freedom in each project, we can be liberal in ~~their allocation~~ the allocating splines (knots) to the explanatory variables during model construction.

(C-2) Correlation and redundancy analysis Figure 4 shows the hierarchically clustered Spearman ρ values of the three systems. The ~~grey dashed~~ blue line indicates our cut-off value ($|\rho| = 0.7$). Our analysis reveals that # *Comments* and # *developers* are highly correlated in *Qpid* and *HBase*. We chose to remove #*developers* from our model since #*comments* is a simpler metric than #*developers*. We find that there are no redundant metrics in our metrics in all the ~~subject~~ studied systems.

(C-3)Assigning degrees of freedom Figure 5 shows the Spearman multiple ρ^2 of the resolution time against each explanatory metric. Metrics that have higher Spearman multiple ρ^2 have higher chance of benefiting from the additional degrees of freedom to better explain resolution time. Based on Figure 5, we split the explanatory metrics into three groups. The first group consists of #*comments*, the second group consists of #*log level changes*, #*log variable changes*, #~~*logging statement*~~ *kig relocation* and #*new* ~~*logging statements*~~*logs*. The last group consists the remaining metrics. We allocate five degrees of freedom i.e, knots, to the metrics in the first group, three to metrics in second group and no knots to metrics in last group similar to prior research [**?**].

(C-4)Regression modeling and validation. After allocating the knots to the explanatory metrics, we build the non-linear regression model and use the *validate* function in the *rms* package to find the significant metrics in our

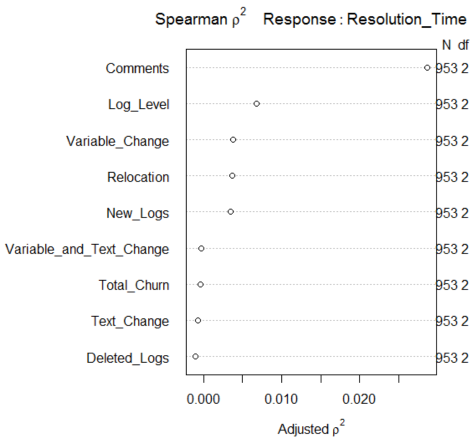


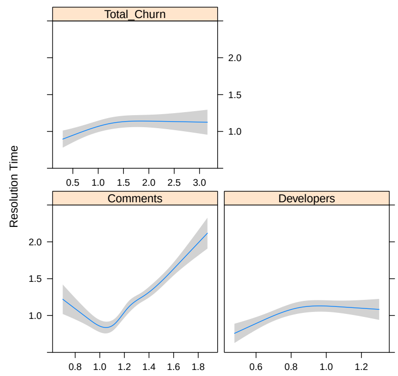
Fig. 5: Spearman multiple ρ^2 of each explanatory metric against Resolution Time of bug ~~fixing commits~~ fixes with ~~logging statement changes~~log churn. Larger values indicate more potential for non-linear relationship

~~subject~~ studied systems. We find that ~~metrics from logging statement changes~~ log churn metrics are significant in Qpid and HBase systems for predicting resolution time of bug fixes.

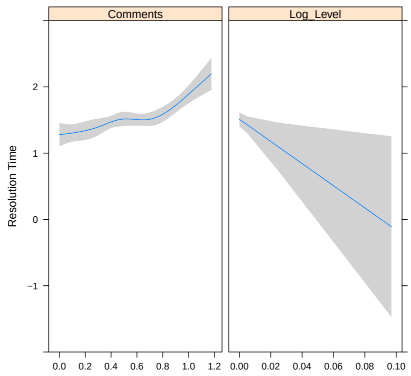
(C-5) Examining explanatory metrics in relation to outcome. Figure 6 shows the direction of impact of ~~metrics from logging statement changes~~ log churn metrics on the resolution of bug ~~fixing commits with logging statement changes~~ fixes with log churn in *HBase*. We find that log modifications have a negative impact on the resolution time of bug fixes. Shown in ~~Table ??~~Figure 6, we find that in *HBase* and *Qpid*, modifications to ~~logging statements~~logs, i.e, Log level changes and variable changes are significant in the models and have negative correlations with the resolution time of bugs.

We find that *log level changes* are statistically significant in *Qpid* and *HBase*. Even though developers often do not change log levels during bug fixes as seen in RQ1, our model shows that changes to log levels can help in faster resolution of bugs. This trend of not changing log levels may be because, ~~prior research has shown that~~ developers are confused when estimating the cost and benefit of each verbosity level in a ~~logging statement~~log as shown by prior research [?]. ~~This suggests that developers should focus more in picking the right verbosity of logging statement. For example, in~~ An example where developers overestimate the log verbosity level is in the issue HADOOP-9593[11] ~~the logging statement~~ , where the log is set to 'error' and causes confusion ~~to~~ among system administrators. On the other hand ~~,~~ in HDFS-1955[12], we see that ~~logging statements are~~ developers underestimate the verbosity level

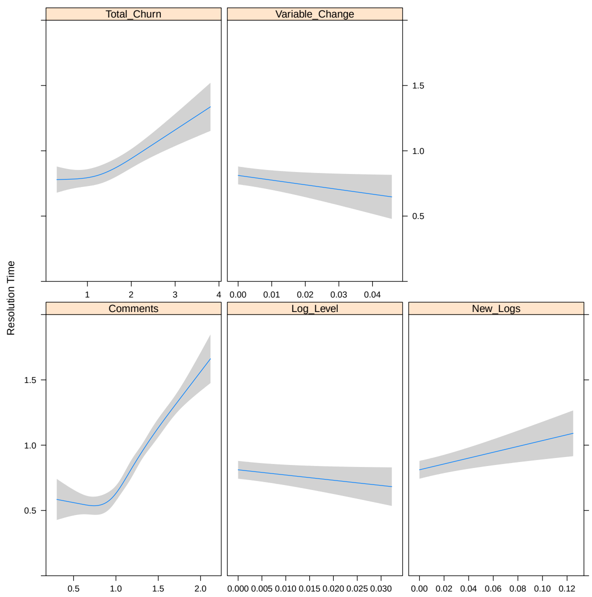
[11] https://issues.apache.org/jira/browse/HADOOP-9593
[12] https://issues.apache.org/jira/browse/HBASE-1955



a: Hadoop



b: Qpid



c: HBase

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

and the log is set to default 'info' and ~~have~~ has to be updated to 'error' and 'warn' to remove unnecessary logs being generated.

We find that *variable changes* is significant in the model for *HBase* with negative effect on the resolution time of bugs. This ~~suggests that changing~~

~~logged variables may help in the faster resolution of bugs. We find in RQ1 that developers change variables in the logging statements more in bug-fixing commits than non-bug-fixing commits. We find that such changesto variables may assist in fixing bugs faster in practice~~may be because having different information in the output logs may benefit developers during bug fixes. The other reason may be that developers add new functions or keywords and leverage the new information in the logs to ensure the functionality of the new code. In RQ1, we find that variable changes; especially addition and modification of logged variables is higher in bug fixes than non-bug fixes. These changes to the logged variables highlight the fact that developers recognize the significance of correct information in logged output.

~~Effect of metrics from logging statement changes 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** 0.0996 Comments 0.2741 Comments 0.3562 Comments-0.0691 Log Level Changes 0.0258 Total Churn 0.1157 Total Churn0.0191 Developers 0.0517 New Logs -0.0281 Log Level Changes-0.0367 Variable Changes~~We find that New ~~logging statements~~ logs have a positive impact on the resolution time of bug fixes in *HBase* project. We find that the average code churn of ~~bug-fixing commits with new logging statements~~ bug fixes with new logs is almost twice that off average code churn of all commits. We also find that the average resolution time is higher for ~~bug-fixing commits with new logging statements~~bug fixes with new logs. These results suggest that during very complex bug fixes, developers might not know the exact cause of the bug and have to add large amount of extra code and new ~~logging statements~~ logs to ensure the functionality of the added code. For example ~~to fix~~ in HBASE-7305[13], ~~which is resolved almost a full year after being open, we observe that developers add over 120 new logging statements and many new feature into~~ developers implement a read write lock for table operations, which previously causes race conditions. We find that this issue has total code churn over 2.5 K and has addition of over 70 new log lines in the code. Because of the addition of new ~~feature into the code, this takes longer time to resolve than bugs with only log modifications.~~

features this issues takes over two months to be resolved with 44 discussion posts on the issue.

> *Log churn metrics can complement the number of comments, the number of developers and total code churn in modelling the resolution time of bugs. We find that logging modifications have a negative effect on resolution time of bug fixes and help in increasing the resolution time of bugs. Such results imply that there is a relationship between log churn and the resolution time of bugs.*

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

4 ~~Discussion of results~~

~~To further understand how developers change logging statements in bug fixes, we finally conduct a qualitative analysis. We collected all the bug fixing commits with logging statement changes for our subject systems. We selected a 5random sample (266 for *HBase*, 268 for *Hadoop* and 83 for *Qpid*) from all the commits. For the sampled commits, we analyze the code changes made in Git and the corresponding JIRA issue reports to find different patterns of log use in bug fixes. We follow an iterative process, similar to prior research , until we cannot find any new types of patterns. We find three reasons of changing logging statements during bug fix as shown in Table 11. Each logging statement changes may have more than one reasons.~~

Table 11: Log change reasons during bug fix

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

– *Bug diagnosis*

Developers use logs to detect runtime defects. Developers change ~~logging statement~~ log to print extra or different information into logs during the execution of the system. We categorize a log change as bug diagnosis only when there is log change has added and deleted code prior to it (code block is changed). For example, to fix HADOOP-2725 [14] 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 of the files into human readable format instead of bytes. These ~~logging statement~~ log changes are committed along with bug fix, as it clarifies the ~~logging statement~~ log and helps in easy understanding.

– *Future bugs prevention*

After fixing a bug, developers may insert ~~logging statement~~ log into the code. Such ~~logging statements~~ logs monitor the execution of the system to

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

track for similar bugs in the future. We categorize a log change as future bug prevention, when prior to the log change there is addition of new blocks (i.e., if, if-else, try-catch and exception) with code deletion. For example in HADOOP-2890 [15] we see that developers leverage logs to identify the reason behind blocks getting corrupted. In the commit, we observe that the developers fix this bug by adding new checks to verify where the block gets corrupted and they add*try catch* block with new ~~logging statements~~ logs to catch these exceptions. The ~~logging statement would~~ log will notify developers with useful information to diagnose the bug if a similar bug appears.

– *~~Bug-fixing code~~ Code quality assurance during bug fixes*

Sometime, developers need to introduce a large amounts of code to fix a bug ~~. However, developers may introduce extra bugs with these bug-fixing code~~with no code deletion. The introduction of new code, may introduce new bugs into the system. To ensure the quality of these ~~bug-fixing code~~bug fixes, developers insert ~~logging statements~~ logs into the bug-fixing code. For example in HBASE-3787 [16], where developers encounter a non-idempotent increment which causes an error in the application. This fix involves over 13 developers and 112 discussions over the two years. The developers add several new files and functions during the bug fix and new ~~logging statements~~ logs to assure the code quality of the fix.

~~Our manual analysis results confirms that developers often change logging statements during bug fixes.~~

4 Related Work

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

4.1 Log Analysis

Prior work leverage ~~logging statements~~ logs for testing and detecting anomalies in large scale systems. Shang *et al.* [**?**] propose an approach to leverage ~~logging statements~~ logs in verifying the deployment of Big Data Analytic applications. Their approach analyzes ~~logging statements~~ logs in order to find differences between running in a small testing environment and a large field environment. Lou *et al.* [**?**] propose an approach to use the variable values printed in ~~logging statements~~ logs to detect anomalies in large systems. Based on the variable values in ~~logging statements~~logs, their approach creates invariants (e.g., equations). Any new ~~logging statements~~ logs that violates the

[15] https://issues.apache.org/jira/browse/HADOOP-2890
[16] https://issues.apache.org/jira/browse/3787

invariant are considered to be a sign of anomalies. Fu *et al.* [?] built a Finite State Automaton (FSA) using unstructured ~~logging statements~~ logs and to detect performance bugs in distributed systems.

Xu *et al.* [?] link logs to ~~logging statements~~ logs in source code to recover the text and and the variable parts of ~~logging statements~~output logs. They applied Principal Component Analysis (PCA) to detect system anomalies. Tan *et al.* [?] 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.* [?] 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.* [?] 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.* [?] propose an approach to automatically infer the failure scenarios when a log is printed during a failed run of a system.

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

The extensive prior research of log analysis motivate our paper to study how ~~logging statements~~ logs are leveraged during bug fixes. As a first step, we study the changes to ~~logging statement~~ log during bug fixes. Our findings show that ~~logging statements~~ logs are change more during bug fixes than other types of code changes. The changes to ~~logging statements~~ logs have a relationship with a faster resolution of bugs with fewer people and less discussion.

4.2 Empirical studies on ~~logging statements~~logs

Prior research performs an empirical study on the characteristics of ~~logging statements~~logs. Yuan *et al.* [?] studies the logging characteristics in four open source systems. They find that over 33% of all ~~logging statement~~ log changes are after thoughts and ~~logging statements~~ logs are changed 1.8 times more than entire code. Fu *et al.* [?] performed an empirical study on where developer put ~~logging statements~~logs. They find that ~~logging statements~~ logs are used for assertion checks, return value checks, exceptions, logic-branching and observing key points. The results of the analysis were evaluated by professionals from the industry and F-score of over 95% was achieved.

Shang *et al.* [?] 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~~ logs and logs outputted during run time [?,?]. They find that ~~logging statements~~ logs are co-evolving with the software systems. However, ~~logging statements~~ logs are often modified by developers without considering the needs of operators. Furthermore, Shang *et al.* [?] find that understanding logs is challenging. They examine user mailing lists from three large open-source projects and find that users of these systems have various issues in understanding logs outputted by the system. Shang *et al.* propose to leverage different types of development knowledge, such as issue reports, to assist in understanding logs.

Prior research by Yuan *et al.* [?] 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 the logging statements~~ in the logs thereby improving the output logs. Log Advisor is another tool by Zhu *et al.* [?] which helps in logging by learning where developers log through existing logging instances.

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

5 Limitations and Threats to Validity

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

External Validity

Our case study is performed *Hadoop, HBase* and *Qpid*. Even though these three ~~subject~~ studied systems have years of history and large user bases, the three ~~subject~~ studied systems are all Java based platform software. Systems in other domain may not rely on ~~logging statements~~ 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~~ studied systems. The quality of the data contained in the repositories can impact the internal validity of our study.

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

Construct Validity

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

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

We use Levenshtein ratio and choose a threshold to identify modifications to ~~logging statements~~logs. However, such threshold may not accurately identify modifications to ~~logging statements~~logs. Further sensitivity analysis on such threshold is needed to better understand the impact of the threshold to our findings.

We build non-liner regression models using ~~metrics from logging statement changes~~log churn metrics, to model the resolution time of bugs. However, the resolution time of bugs can be correlated to many factors other than just logs, such as the complexity of code fixes. To reduce such a possibility, we normalize the ~~metrics from logging statement changes~~ log churn metrics by code churn. However, other factors may also have an impact on the resolution time of bugs. Furthermore, as this is the first exploration (to our best knowledge) in modeling resolution time of bugs using ~~metrics from logging statement changes~~log churn metrics, we are only interested in understanding the correlation between the two. Future studies should build more complex models, that consider other factors to study if there is any causation.

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

6 Conclusion and Future Work

Logs are used by developers for fixing bugs. This paper is a first attempt (to our best knowledge) to understand whether ~~logging statements~~ logs are changed more during bug fixes and how these changes occur. The highlights of our findings are:

- We find that ~~logging statements~~ logs are changed more during bug ~~fixing commits than non-bug-fixing commits~~fixes than non-bug fixes. In particular, we find that ~~logging statements~~ logs are modified more frequently during bug fixes. Variables and textual information in the ~~logging statements~~ logs are more frequently modified during bug fixes.
- We find that ~~logging statements~~ logs are changed more during complex bug fixes. However, bug fixes that change ~~logging statements~~ logs are fixed faster, need fewer developers and have less discussion.
- We find that ~~metrics from logging statement changes~~ log churn metrics can complement the traditional metrics such as the number of comments and the number of developers in modeling the resolution time of bugs.

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