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

Which logs are stable?

First Author · Second Author

Received: date / Accepted: date

Abstract Logs are system generated outputs, inserted by developers into the code. They help in understanding the system behavior, monitor the choke-points and also in debugging. Moreover, recent research has demonstrated the importance of logs in operating, understanding and improving software systems. This has lead to the development of log management applications and log processing tools. However, when logs are changed the changes have to be communicated to operators and administrators. The processing tools and log management applications also suffer due to these changes. To avoid this, in this paper we study the stability of logs on four large software systems namely: Liferay, ActiveMQ, Camel and CloudStack. We conduct a manual analysis and find that 20-80% of the logs are changed in these software systems. We find four different types of log changes namely 1) text modification, 2) variable modification, 3) log level change and 4) log relocation. After characterizing the different types of log changes we build models to predict if a log added to a file will change in the future. We use code churn, log churn and developer metrics to build a random forest classifier. Our classifier can predict which logs will change in future with precision of 89-93 %, and recall of 76-92%. We find that code, log and developer metrics are statistically significant in our models and can help identify which logs are more likely to change in the future. This can help developers of log processing tools to develop more robust applications. On the other hand, system administrators can know before hand which logs are more likely to cause issues in the log processing tools and this can reduce their maintenance costs.

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

S. Author
second address

1 Introduction

Logs are leveraged by developers to record useful information during the execution of a system. Logs are recorded during various developmental activities from bug-fixes [1–3], improvement tasks [4], monitoring performance [5] and also for knowledge transfer [6]. Logging can be done through use of log libraries or archaic methods like *print* statements. Every log statement contains a textual part, which provides information about context, a variable part providing information about the event and log level, which shows the verbosity of the logs. An example of a log is shown below where info is the logging level, *Testing Connection to Host Id* to is the event and the value of variable *host* is the variable. 1

```
LOG.info( “Testing Connection to Host Id:” + host);
```

The rich and unified format of logs has lead to the development of many log processing tools. These tools are used for variety of purposes and not only by developers for developmental activities. They are used to monitor system health, to detect anomalies, to find performance issues and also capacity planning. Because of this new area of log utilization there have been many commercial log processing tools like *Splunk*, *Xpolog*, *Logstash* and in-house tools. These applications reply heavily on the log messages themselves and require continuous maintenance when the format or content of logs are changed.

Research shows that only 40% of the logs stay same across releases and up-to 21.5% of the log statements are modified in *Hadoop* and *PostgreSQL* [6]. These changes can effect the log processing tools, which heavily depend on them and maintenance cost will be high. In this paper, we track the changes made to logs across multiple releases in the four subject systems and use the data to answer the following research questions.

RQ1:How much do logs change overtime?

Based on our quantitative analysis of the subject systems we identify three categories of changes in log statements. If a log statement is changed more than four times we categorize it under *‘Frequently Changed’*, if it has only three changes or lesser its categorized under *‘changed’* and if there is no changes made we categorize under *‘Never Changed’*. We find that 20-80% of logs are changed at-least once across our subject systems. We find four different types of log changes which occur namely ‘log level change’, ‘text modification’, ‘variable modification’ and ‘log relocation’.

RQ2: Can code and developer related metrics help in explaining the stability of logs?

We find that code, log and developer related metrics help in building models which can predict which logs are more likely to change in the future. We use the data from three dimensions namely, code, log and developers. Our *random forest* achieved accuracy of between 89-93% in all subject systems with recall ranging between 76%-92%, when predicting which logs have higher likelihood of getting changed. We also identify significant metrics from each dimension, that affect the stability of logs. We find several metrics(e.g., developer experience, source lines of code, # of comments, log text length) are statistically significant in predicting if the log will be changed in future.

These results show that code, log and developer related metrics can help in identifying unstable log statements in our subject systems. This can help in reducing the effort needed in the maintenance of log processing applications, as system maintainers can flag the log statements with potential of being changed in subsequent releases and track them.

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 describes the prior research that is related to our work. Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper.

2 Methodology

In this section we present our rationale for selecting our subject systems and present our data extraction and analysis approach.

In this paper, we aim to find the unstable logs in a system for easier management of log processing tools. We conduct a cast study on four open source projects i.e Liferay, Camel, ActiveMQ and CloudStack. All subject systems have extensive system logs and Table 1 highlights the overview of the systems.

Table 1: An Overview of all Subject Systems

Projects	Liferay	Camel	ActiveMQ	CloudStack
Starting Release	6.1.0-b3	1.6.0	4.1.1	2.1.3
End Release	7.0.0-m3	2.11.3	5.9.0	4.2.0
Total Added Code	3.9M	505k	261k	1.09M
Total Deleted Code	2.8M	174k	114k	750K
Total Added Log	10.4k	5.1k	4.5k	24k
Total Deleted Log	8.1k	2.4k	2.3k	17k

Liferay[1]: Liferay is a free and open source enterprise project written in JAVA. It provides platform features which are used in development of websites and portals. It also has extensive issue tracking system in JIRA. We study the releases from 6.1.0 till 7.0.0-m3 which covers more than three years of development from 2010 till 2014

Camel[2]: Camel is an open source integration framework based on enterprise integration patterns. We used Camel release 1.6 to 2.11.3 which covers more than five years of development from 2009 till 2013.

[1] http://www.liferay.com/
[2] http://camel.apache.org/

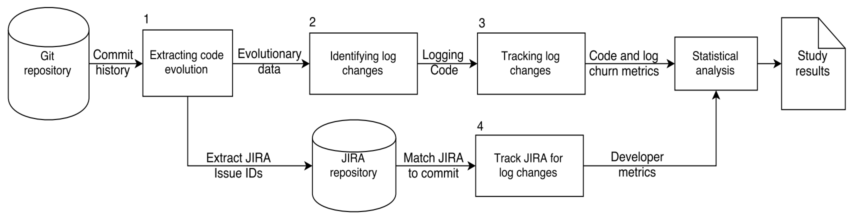

Fig. 1: Flowchart to categorize the different types of log changes that occur

ActiveMQ[3]: ActiveMQ is an open source message broker and integration patterns server. We covered ActiveMQ release 4.1.1 to 5.9.0 which covers more than 6 years of development from 2007 till 2013.

CloudStack[4]: Apache CloudStack is open source software designed to deploy and manage large networks of virtual machines, as a highly available, highly scalable Infrastructure as a Service (IaaS) cloud computing platform. We covered CloudStack release 2.1.3 to 4.20 which covers more than 3 year of development from 2010 till 2014.

In the remainder of this section we present our approach for preparing the data to answer our research questions.

2.1 Extracting code evolution

In order to find the stability of logs we have to identify all the 'java' files in our subject systems. To achieve this, we obtain the *end release* version of each subject system locally. We used keyword based searches to find all the Java files in the version and we filter out the *Java Test* files, through keyword based heuristics. After collecting all the Java files in the four subject systems, we use git repository to obtain all the changes made to the file within the time-frame discussed above. We use the 'follow' option to track the file even when they are renamed or relocated within our subject systems. We flatten all the changes made to files to include the changes made in different branches but exclude the final merging commit. Using this approach, we obtain complete history of each *java* file.

2.2 Identifying log changes

From the extracted code evolution data for each *java* file, we identify all the log changes made in the commits. To identify the log statements in the source code, we manually sample some commits from each subject system and identify the

[3] http://activemq.apache.org/
[4] https://cloudstack.apache.org/

logging library used to generate the logs. We found that all the subject systems use *Log4j* and *Slf4j* widely and *logback* sparingly. Using this we identify the common method invocations that invoke the logging library. For example in ActiveMQ and Camel a logging library is invoked by method named 'LOG' as shown below.

```
LOG.debug("Exception detail", exception);
```

In CloudStack its usually done through '_logger' as follows.

```
_logger.warn("Timed out: " + buildCommandLine(command));
```

Using regular expressions to match these patterns, we automate the process of finding all the logging statements in each commit to a file. We count every such invocation of logging library as a log statement.

2.3 Tracking log changes

After identifying all the log changes made to a file across multiple commits, we track each log individually to find if its changed in subsequent revisions. To track the log changes made, we used levenshtein measure [10]. We first collect all the logs present in a file at the first commit, which form the initial set of log statements for the file. Every subsequent commit which has changes to a log appears as a added and deleted log statement in *git*. If the levenshtein ratio between the deleted log and the added log is higher than 0.6, and the deleted log matches the initial set, its considered as log change. If the added log does not match any log in our initial set, its considered as a new log addition and is added to the initial set for tracking in future commits. From this we can track how many times a log is changed and how many commits are made between the changes.

2.4 Match JIRA to log changes

Using the commit data, we also track the JIRA issues. To achieve this, we extract the JIRA issue ID's from commit messages and use the JIRA repository to extract the JIRA issue. The JIRA issue contains the information about the issue like type (i.e., bug, improvement, new-feature, task), priority, resolution time, # of comments and # of developers involved in the commit. We use this information along with code and log churn metrics for answering our research questions.

3 Study Results

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

RQ1: How much do logs change overtime?

Motivation

Research has shown that logs evolve along with the code [11]. When logs are changed the log processing tools which are dependent on the them also have to get updated. This results in costly maintenance efforts. To understand the costs, we have to understand how frequent are the changes to logs. Hence, we explore the frequency of changes to logs in all our subject systems.

Approach

To find the frequency of log changes, we conduct an quantitative analysis on our subject systems. We use the tracked log data for each subject system as explained in section 2. From each project, we select a random sample which achieves 95% confidence level and 5% confidence interval. We follow an iterative process as prior research [12] to find how frequently logs change in our subject systems.

When logs are changed, they can be changed in several ways. To understand the different types of log changes we perform a manual analysis on the changed logging statements. We select a random sample from each project such that the sample achieves 95% confidence level and 5% confidence interval. After identifying all the types of log changes we automate the process using *Levenshtein ratio* [10]. Figure 2 highlights the process of categorizing the log changes. For example, consider the logging statements shown below.

+ LOG.info("starting HBase HsHA Thrift server on " + Integer.toString(listenPort));

- LOG.info("starting HBase " + implType.simpleClassName() +" server on " + Integer.toString(listenPort));

To identify the type of log change we first remove the logging method (i.e, LOG) and the log level (i.e, info) from the logging statements. We then compute the *levenshtein ratio* between each term within the parenthesis. In the above case we find that '+ Integer.toString(listenPort)' has *levenshtein ratio* of 1, implying there are identical and the *levenshtein ratio* between 'starting HBase HsHA Thrift server on' and 'starting HBase' is 0.56. This suggests there is some similarity between the two strings. Using this information we categorize logs into different types.

Results

We find that developers change 20-80% of the logs across our subject systems as shown in Table 2. Based on frequency of changes, we categorize logs into 3 categories namely: a) Frequently Changed, b) Changed and c) Never Changed. If a logging statement is changed more than three

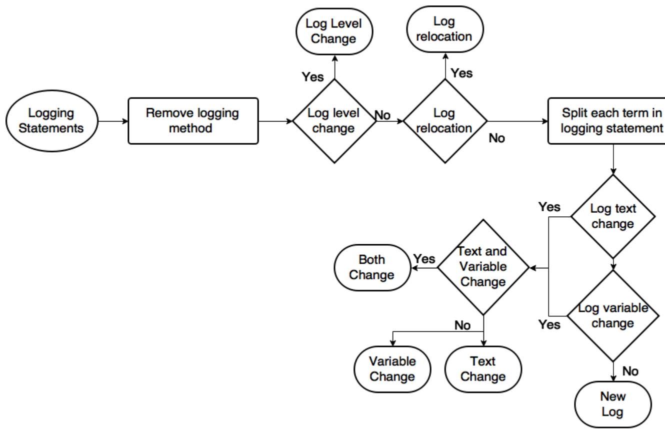

Fig. 2: Flowchart to categorize the different types of log changes that occur

Table 2: Distribution of log changes in different projects

Projects	Never Changed (%)	Changed (%)	Frequently Changed (%)
Life Ray	78.67	19.66	1.66
Camel	55.43	37.32	7.25
ActiveMq	34.78	62.02	3.20
CloudStack	19.68	68.61	11.71

times it is categorized as ‘Frequently Changed’. If it is changed fewer than three times it categorized under ‘changed’ and if there is no changes made it is categorized under ‘Never Changed’. This suggests that logs are not stable and developers have to change logs as afterthoughts in later commits [13].

When developers change logs there can be many types of changes i.e., changes to log context, verbosity levels or logged variables. From our manual analysis, we identify five types of log changes. Table 3 shows their distributions. When there is overlapping of the different types of log changes, we categorize them as newly added log and track changes made to it.

1. **Log relocation:** The log has only white spaces changed with no changes to context, variables or the level.
2. **Text modification:** The text (i.e., context) of 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.
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.

Log relocation is more than other types of log changes. We find that in all the subject systems log relocation occurs more than 40%. As log relocations have no changes to the text or variables in logging statements, their impact on

Table 3: Distribution of log changes in different projects

Projects	Life Ray	Cloud Stack	Active-MQ	Camel
Log Relocation (%)	73.5	78.85	63.27	49.72
Log Text Change (%)	20.11	6.37	5.97	7.59
Log Variable Change (%)	3.10	7.21	6.91	8.29
Log Level Change (%)	0.87	1.15	1.76	3.79
Text and Variable Change (%)	2.33	6.39	22.0	30.59

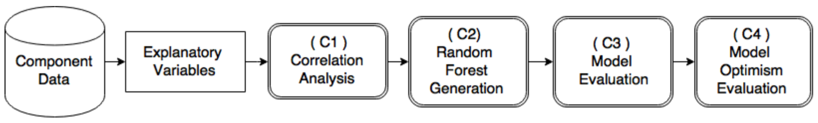
Component Data
Explanatory Variables
(C1) Correlation Analysis
(C2) Random Forest Generation
(C3) Model Evaluation
(C4) Model Optimism Evaluation

Fig. 3: Overview of Model construction(C) and Evaluation

log processing tools is limited. Hence we exclude log relocation changes from our datasets.

45% of the logging statemetns are changed atleast once in three of our subject systems. We find that over 50% of these changes are changes to log context and remaining is spread over text, variable and log level changes. This shows that developers chnage logs extensively and this suggests developers might dedicate significant amount of time to update and maintain these logs

RQ2: Can code and developer related metrics help in explaining the stability of logs?

In RQ1, we find that 20-80% of logs are changed in our subject systems. This effects the log processing tools which run on these subject systems, making developers spend more time on maintenance of logs. Hence, there is a need to identify these non-stable logs and simplify job of developers. It can also benefit log processing tool developers to develop more robust applications making them more stable.

Approach

To find the stability of logs in our subject systems, we extract code and log churn metrics from Git repository and developer metrics from JIRA. We use code churn metrics because prior work has linked logs to development knowledge and issue reports [12].

Table 4 lists all the metrics we collect for each dimension. We define each metric and the rationale behind the choice of each metric. We collect the code and log churn metrics when a log is added into the subject system and track the log statement. If it is changed in subsequent revisions we flag the log as changed and record it.

We adopt the statistical tool R to model our data and use the 'randomForest' package to generate the random forests. The overview of the process in shown in Figure 3 and explained below.

Table 4: Taxonomy of considered metrics for model construction

Dimension	Metrics	Values	Definition (d) – Rationale (r)
Code Metrics	Log revision count	Numeric	d: The number of commits prior which had log statement changes.
			r: This helps to identify if the file is prune to log statement changes.
	New File	Boolean (0 -1)	d: Check if the log is added in a new file (i.e., newly comitted)
			r: This helps to identify which logs where added later in subsequent commits.
	Total Revision Count	Numeric	d: Total number of commits made to the file before the log is added. This value is 0 for logs added in the initial commit but not for logs added overtime.
			r: This helps to find out if the file is changed heavily which can result in log changes [13]
	Code churn in commit	Numeric	d: The code churn of the commit in which log is added.
			r: Log changes are correlated to code churn in files [14]
	Variables declared	Numeric	d: The number of variables which are declared before the log statement. (we limit to 20 lines before log statement)
			r: When new variables are declared, developers may log the new variables to obtain more information [13]
	SLOC	Numeric	d: The number of lines of code in the file.
			r: Large files have more functionality and are more prone to changes [16] and more log changes [13,14]
Log Metrics	Log Context	Categorical	d: Identify the block in which log statement is added. (i.e., ‘if’, ‘if-else’, ‘try-catch’, ‘exception’, ‘throw’, ‘new function’)
			r: Prior research find that logs are used in assertion checks, logical branching, return value checking, assertion checking [17]
	Log Level	Categorical	d: Identify the log level (verbosity) of the added log. (i.e., ‘info’, ‘error’, ‘warn’, ‘debug’, ‘trace’ and ‘trace’)
			r: Developers spend significant amount of time in adjusting the verbosity of logging statements [13]
	Log variable count	Numerical	d: Number of variables logged.
			r: Over 62% of logging statements end adding new variables [13]. Hence fewer variables in inital log statement might result in addition later.
	Log text length	Numerical	d: Number of text phrases logged (i.e., we count all text present between a pair of colons as one phrase)
			r: Over 45% of logs have modifications to static context [13]. Logs with fewer phrases might be subject to changes later to provide better explanation
	Log density	Numerical	d: Ratio of number of log lines to the source code lines in the file.
			r: Research has found that there is one log line per 30 lines of code [13]. If its less it suggests there may be additions in later commits.

Dimension	Metrics	Values	Definition (d) – Rationale (r)
Developer Metrics	Resolution time	Numerical	d: The time it takes for the issue to get fixed. It is defined as the time it takes since an issue is opened till its closed.
			r: More resolution time might suggest more complex fix with more code churn resulting in more log churn.
	Number of developers involved	Numeric	d: Total number of unique developers who comment on the issue report on JIRA
			r: Components with many unique authors likely lack strong ownership, which in turn may lead to more defects [18] and change logging statements [14].
	Number of Comments	Numeric	d: Total number of discussion posts on the issue.
			r: Number of comments is correlated to the resolution time of issue reports [15]. More comments may also indicate the issue is more complex requiring more code churn and logging statement changes.
	Developer experience	Numeric	d: The number of commits the developer has made prior this commit.
			r: Research has shown that experienced developers might take up more complex issues [19] and therefore may leverage logging statemetns more [14].
	Issue type	Categorical	d: Identify the type of issue i.e., 'Bug', 'Improvement', 'Task', 'New Feature', 'Sub-Task', 'Test'
			r: Some issue types might have higher code churn than others (example: Bug and New features might have more code churn when compared to Sub-Tasks) and are committed faster.
	Priority type	Categorical	d: Identify the priority of the issue i.e., 'Critical', 'Blocker', 'Major', 'Minor' and 'Trivial'
			r: Research has shown that priority of issue effects resolution time of bug fixes [20]. Higher the priority indicates the issue will be fixed faster with logging statement changes.

(C1 - Correlation 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.

Figure 4 shows the hierarchically clustered Spearman ρ values in ActiveMQ project. The solid horizontal lines indicate the correlation value of the two metrics that are connected by the vertical branches that descend from it. The gray line indicates our cutoff value ($|\rho| = 0.7$).

(C2 - Random forest generation)

After we eliminate the correlated metrics from our datasets, we construct the random forest model. A random forest is collection of largely uncorrelated decision trees where they trees combine their results to form a generalized predictor [21].

Given a dataset of n logging statements for training, $D = \{(X_1Y_1, ..., (X_nY_n)\}$ where $X_i, i = 1...n$, is a vector of descriptors (i.e, explanatory metrics) and Y_i

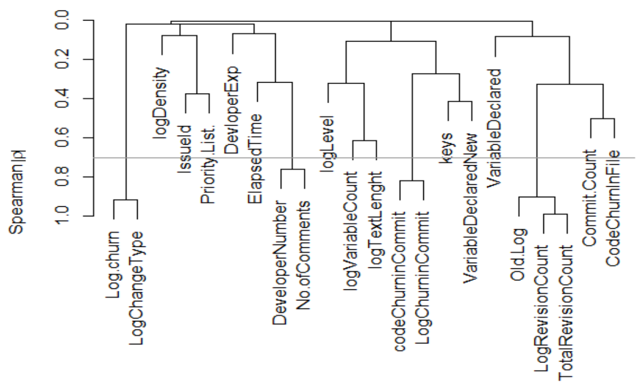

Fig. 4: Hierarchical clustering of variables according to Spearmans ρ in ActiveMQ

	Predicted logging statement changed	Predicted Logging statement not-changed
Actual logging statement changed	TP (True Positive)	FN (False Negative)
Actual logging statement not-Changed	FP (False Positive)	TN (True Negative)

Table 5: Confusion Matrix

is the flag which indicates of logging statement is changed or not, the algorithm follows these steps.

– From the training set of n logs, a random sample is selected with replacement (i.e., bootstrap sample).
– From the sampled selection, tree is grown with one difference compared to normal decision trees. The random forest is split at each node using the best among subset of predictors, rather than all. This counterintuitive strategy helps to make the random forests more robust against over-fitting [22]
– The above steps are repeated until M such trees are grown.
– Predict new data by aggregating the prediction of the M trees generated.

(C3 - Model evaluation)

After we build the random forest model, we evaluate the performance of our model using precision, recall, F-measure and Brier Score. All these measures are functions of the confusion matrix as shown in Table 5 and are explained below.

Precision (P) measure the correctness of our model in predicting which log will undergo change in the future. It is defined as the number of logs which were accurately predicted as changed over all logs predicted to have changed as explained in Equation 1.

$$P = \frac{TP}{TP + FP} \tag{1}$$

Recall (R) measure the completeness of our model. A model is said to be complete if the model can predict all the logs which will get changed in our dataset. It is defined as the number of logs which were accurately predicted as changed over all logs which actually change as explained in Equation 2.

$$R = \frac{TP}{TP + FN} \tag{2}$$

F-Measure is the harmonic mean of precision and recall, combining the inversely related measure into a single descriptive statistic as shown in Equation 3 [23].

$$F = \frac{2 \times P \times R}{P + R} \tag{3}$$

Brier Score (BS) is a measure of the accuracy of the predictions in our model. It is the mean squared error of the probability forecasts [24]. The most common formulation of Brier Score is shown in Equation 4, where f_t is the probability that was predicted, o_t is the actual outcome of the event at the instance t (0 if logging statement is not changed and 1 if it is changed), and N is the number of forecasting instances.

If the predicted value is 70% and the logging statement is changed, the Brier Score is 0.09. It reaches 0.25 for random guessing (i.e., predicted value is 50%) and it is 0 when predicted value is 100% and the log is not changed.

$$BS = \frac{1}{N} \Sigma_{t=1}^{N} (f_t - o_t)^2 \tag{4}$$

The model performance measure, described previously only provide insight into how the random forest models fit the observed dataset, but it may overestimate the performance of the model if it is over-fit. We take performance overestimation (i.e., *optimism*) into account by by subtracting the bootstrap averaged optimism [25] from the initial performance measure. The *optimism* of the performance measures are calculated as follows:

1. From the original dataset with n records, select a bootstrap sample with n components with replacement.
2. Build random forest using the bootstrap sample.
3. Apply the model built from bootstrap data on both the bootstrap and original data sample, calculating precision, recall, F-measure and Brier score for both data samples.
4. Calculate *Optimism* by subtracting the performance measures of the bootstrap sample against the original sample.

Table 6: The optimism reduced performance measures of the four projects

Projects	Life Ray	ActiveMQ	Camel	CloudStack
Precision	0.91	0.90	0.93	0.89
Recall	0.76	0.67	0.89	0.92
F-Measure	0.83	0.77	0.91	0.90
Breir Score	0.06	0.04	0.06	0.07

The above process is repeated 1,000 times and the average (mean) *optimism* is calculated. Finally, we calculate *optimism-reduced* performance measures for precision, recall, F-measure and Brier score by subtracting the averaged optimism of each measure, from their corresponding original measure. The smaller the optimism values, the more stable that the original model fit is.

(C4 - Importance of each metric in relation to logging statement stability) To find the importance of each metric in a random forest model, we use permutation test. In this, for every tree grown in the forest, we use the test dataset (out-of-bag) to predict the accuracy of the model. Then, the values of the X_ith metric of which we want to find importance for, is randomly permuted in the test dataset and the accuracy of the model is recomputed. The decrease in accuracy as a result of this permuting is averaged over all trees, and is used as a measure of the importance of metric X_ith in the random forest.

We use the 'importance' function defined in 'randomForest' package of R, to calculate the importance of each metric. We call the 'importance' function each time during the bootstrapping process to obtain 1000 importance scores for each metric in our dataset.

We use the **Scott-Knott** clustering to group the metric based on their means [26,27]. The SK algorithm uses the hierarchical clustering approach to divide the metrics and uses the likelihood ratio test to judge the difference between the groups. This assures the means of metrics within a group not to be statistically significantly different. We use the 'SK' function in the 'ScottKnott' package of R and set significance threshold of 0.05 to cluster the metrics into different groups.

Results

The random forest classifier achieves precision of 0.89-0.93 and recall of 0.76-0.92 for our subject systems. Table 6 shows the optimism-reduced values of *precision*, *recall*, *F-measure* and *Brier score* for each project. We find the performance of the classier for Liferay is not as good as the other projects. This is because Liferay has the lowest percentage of logs which are changed at 20%, compared to 45-70% in the other projects.

The random forest classifier outperforms random guessing. The classifier achieves Brier scores between 0.06 and 0.09 across all projects, which is much lower than 0.25 (Brier score for random guessing). This indicates out model classier achieves better performance than random guessing.

Table 7: The importance values of the metrics (top 10), divided into homogeneous groups by Scott-Knott clustering.

Active MQ			CloudStack			Life Ray			Camel		
Groups	**Factors**	**Importance**	**Groups**	**Factors**	**Importance**	**Groups**	**Factors**	**Importance**	**Groups**	**Factors**	**Importance**
1	Developer experience	0.144 -	1	Log revision count	0.116 +	1	Log context	0.276 -	1	Log text length	0.157 +
2	Log text length	0.103 +	2	SLOC	0.078 +	2	Log density	0.270 +	2	Developer experience	0.147 -
3	Developer number	0.098 +	3	Developer Experience	0.073 -	3	Resolution time	0.162 -	3	Log variable count	0.086 +
4	SLOC	0.097 +	4	Priority list	0.067 +	4	Developer experience	0.156 -	4	Log density	0.082 +
5	Log density	0.082 +	5	# of comments	0.064 -	5	SLOC	0.143 +	5	Log revision count	0.081 +
6	Priority list	0.076 -	6	Log density	0.063 -	6	Log level	0.097 -		# of comments	0.081 -
7	Resolution time	0.070 +	7	Code churn in commit	0.056 +	7	Log variable count	0.090 +	6	Code churn in commit	0.079 -
	# of comments	0.070 +	8	Variable declared	0.042 +		# of comments	0.089 -	7	Resolution time	0.078 -
8	Code churn in commit	0.067 +	9	Log text length	0.039 +	8	Priority list	0.079 +	8	SLOC	0.077 +
	Log revision count	0.067 +	10	Log context	0.037 -	9	Log text length	0.0 +	9	Log level	0.065 -

We find that 'Developer Experience' to be the most important factor for developer metrics. From Code Metrics we find that 'SLOC', 'Code Churn in Commit' and '# of comments' to be the important factors and from log metrics we find that 'log density' to be important factors. Table 7 shows the important factors of the predictor metrics, divided into statistically distinct groups. A '+' or '-' sign following the predictor metric indicates if the metric has positive or negative effect on the stability of logs. This is obtained by calculating the correlation between the predictor metrics and the response metrics.

We find *log density*, *log text length*, *log revision count* are significant in our subject systems as shown in Table 7. We find *log text length* has a positive correlation, implying more text in a log, the more likelihood it can be changed. This may be because there is inconsistencies between logs and the actual needed information intended as shown by prior research [13]. *Log revision count* also has positive correlation across all subject systems. This is because, if a logs in file are changed often, there is higher chance of them getting changed in future, than logs in file with no prior changes [28].

We find that *SLOC* and *# of comments*is significant from *Code Churn* dimension across all projects. *SLOC* has positive correlation suggesting that logs in larger files have higher likelihood of getting changed. We find that *# of comments* has negative correlation which suggests that when developers provide information through comments, logs around those comments are less likely to get changed.

We find that *developer experience* is significant in all our models and it has negative correlation across all projects. This implies that logs are more likely to be changed by less experienced developers. We find that other metrics from Developer dimension are not consistent among the subject systems. This may be because each project might have different philosophy of development, for example we find that *resolution time* has negative effect in Liferay and Camel but has positive effect in Active MQ. This suggests that logs are more likely to change in Active MQ when the resolution time of issue increases, but less likely in Liferay and Camel.

Random Forest classifier achieves precision of 0.89-0.93 and recall of 0.76-0.92 across all subject systems. We find deeveloper experience, SLOC, # of comments and log density are significant metrics for predicting the stability of logs.

4 Related Work

In this section, we present the prior research that performs log analysis on large software systems and tools developed to assist in logging.

4.1 Log Analysis

Prior work leverage logging statements for testing and detecting anomalies in large scale systems. Shang *et al.* [11] propose an approach to leverage logging statements in verifying the deployment of Big Data Analytic applications. Their approach analyzes logging statements in order to find differences between running in a small testing environment and a large field environment. Lou *et al.* [2] propose an approach to use the variable values printed in logging statements to detect anomalies in large systems. Based on the variable values in logging statements, their approach creates invariants (e.g., equations). Any new logging statements that violates the invariant are considered to be a sign of anomalies. Fu *et al.* [3] built a Finite State Automaton (FSA) using unstructured logging statements and to detect performance bugs in distributed systems.

Xu *et al.* [1] link logs to logging statements in source code to recover the text and and the variable parts of logging statements. They applied Principal Component Analysis (PCA) to detect system anomalies. Tan *et al.* [29] 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.* [30] 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.* [31] 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.* [32] propose an approach to automatically infer the failure scenarios when a log is printed during a failed run of a system.

Jiang *et al.* [33–36] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [33]. Based on the such events, they identified both functional anomalies [34] and performance degradations [35] in load test results. In addition, they proposed an approach that leverage logs to reduce the load test that are performed in user environment [36].

The extensive prior research of log analysis shows that logs are leveraged for different purposes and changes to logging statements can effect performance of log analysis tools. This motivate our paper to study understand how much logs are changed by developers. As a first step, we study how many logging statements are stable and how many undergo changes across commits. Our findings show that more than 40% of the logging statements are changed at-least once and 3-11% of the logging statements are changed frequently.

4.2 Empirical studies and tools developed on logging statements

Prior research performs an empirical study on the characteristics of logging statements. Yuan *et al.* [13] studies the logging characteristics in four open source systems. They find that over 33% of all logging statement changes are after thoughts and logging statements are changed 1.8 times more than entire code. Fu *et al.* [17] 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.* [37] 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 logs outputted during run time [14, 38]. They find that logging statements are co-evolving with the software systems. However, logging statements are often modified by developers without considering the needs of operators. Furthermore, Shang *et al.* [12] 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.

The most related research by Yuan *et al.* [5] 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 thereby improving the logs. Our paper focuses more on informing developers which logging statements are more prone to get changed and identifying which factors explain the change of logging statements.

5 Threats to Validity

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

External Validity

Our case study is performed on Liferay, ActiveMQ, Camel and CloudStack. Though these subject systems have years of history and large user bases, these systems are all Java based platform softwares. Softwares in other domains may not rely on logs and may not use log processing tools. More case studies on other domains with other programming languages are needed to see whether our findings can be generalized.

Construct Validity

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

We use keywords to identify bug fixing commits when the JIRA issue ID is not included in the commit messages. Although such keywords are used extensively in prior research [14], we may still miss identify bug fixing commits or branching and merging commits.

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

Our models are incomplete, i.e., we have not measured all potential dimensions that impact logging stability. Other metrics that we may have overlooked may better explain the stability of logs.

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 metrics that are important factors in predicting the stability of logging statements cannot claim causal effects, as we are investigating which metrics correlation and not causation. The important factors from our random forest models only indicate that there exists a relationship which should be studied in depth through empirical studies.

6 Conclusion

Logs are snippets of code, introduced by experts to record valuable information. This information is used by a plethora of log processing tools to assist in software testing, monitoring performance and system state comprehension. These log processing tools are completely dependent on the logs and hence are impacted when logs are changed.

In this paper we study the stability of logs, which effects the performance of the processing tools which utilize them. We build a Random Forest classifier to predict which logs are more likely to change in future using data from three dimensions namely: *code, log* and *developers*. We collect this data when logs are added in the source code and we predict which log will change in the future using this data. The highlights of our study are:

– We find that 20-80 % of logs are changed at-least once.
– Our RF classifier achieves prediction of 89-93% and recall of 76-92%.
– We find a negative correlation between developer experience and log changes from the developer dimension. We find that No. of comments in source code, also has negative correlation towards log changes.

Our findings highlight that we can predict which logs have higher likelihood of getting changed when they are introduced. This information can be used by system administrators and practitioners, to identify logs which have higher chance of affecting the log processing tools and prevent failure of these tools.

References

1. 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.
2. 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.
3. 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.*
4. H. Malik, H. Hemmati, and A. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in *Software Engineering (ICSE), 2013 35th International Conference on*, May 2013, pp. 1012–1021.
5. 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.
6. W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 3–26, 2014.
7. Xpolog. [Online]. Available: http://www.xpolog.com/.
8. S. Alspaugh, B. Chen, J. Lin, A. Ganapathi, M. Hearst, and R. Katz, "Analyzing log analysis: An empirical study of user log mining," in *28th Large Installation System Administration Conference (LISA14).* Seattle, WA: USENIX Association, Nov. 2014, pp. 62–77. [Online]. Available: https://www.usenix.org/conference/lisa14/conference-program/presentation/alspaugh
9. logstash, "http://logstash.net."
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.

15. E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering.* ACM, 2010, pp. 52–56.
16. D. Zhang, K. El Emam, H. Liu *et al.*, "An investigation into the functional form of the size-defect relationship for software modules," *Software Engineering, IEEE Transactions on*, vol. 35, no. 2, pp. 293–304, 2009.
17. 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.
18. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 2014, pp. 192–201.
19. F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011, pp. 491–500.
20. L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering.* ACM, 2011, p. 11.
21. J. Albert, E. Aliu, H. Anderhub, P. Antoranz, A. Armada, M. Asensio, C. Baixeras, J. Barrio, H. Bartko, D. Bastieri, J. Becker, W. Bednarek, K. Berger, C. Bigongiari, A. Biland, R. Bock, P. Bordas, V. Bosch-Ramon, T. Bretz, I. Britvitch, M. Camara, E. Carmona, A. Chilingarian, S. Ciprini, J. Coarasa, S. Commichau, J. Contreras, J. Cortina, M. Costado, V. Curtef, V. Danielyan, F. Dazzi, A. D. Angelis, C. Delgado, R. de los Reyes, B. D. Lotto, E. Domingo-Santamara, D. Dorner, M. Doro, M. Errando, M. Fagiolini, D. Ferenc, E. Fernndez, R. Firpo, J. Flix, M. Fonseca, L. Font, M. Fuchs, N. Galante, R. Garca-Lpez, M. Garczarczyk, M. Gaug, M. Giller, F. Goebel, D. Hakobyan, M. Hayashida, T. Hengstebeck, A. Herrero, D. Hhne, J. Hose, S. Huber, C. Hsu, P. Jacon, T. Jogler, R. Kosyra, D. Kranich, R. Kritzer, A. Laille, E. Lindfors, S. Lombardi, F. Longo, J. Lpez, M. Lpez, E. Lorenz, P. Majumdar, G. Maneva, K. Mannheim, M. Mariotti, M. Martnez, D. Mazin, C. Merck, M. Meucci, M. Meyer, J. Miranda, R. Mirzoyan, S. Mizobuchi, A. Moralejo, D. Nieto, K. Nilsson, J. Ninkovic, E. Oa-Wilhelmi, N. Otte, I. Oya, M. Panniello, R. Paoletti, J. Paredes, M. Pasanen, D. Pascoli, F. Pauss, R. Pegna, M. Persic, L. Peruzzo, A. Piccioli, N. Puchades, E. Prandini, A. Raymers, W. Rhode, M. Rib, J. Rico, M. Rissi, A. Robert, S. Rgamer, A. Saggion, T. Saito, A. Snchez, P. Sartori, V. Scalzotto, V. Scapin, R. Schmitt, T. Schweizer, M. Shayduk, K. Shinozaki, S. Shore, N. Sidro, A. Sillanp, D. Sobczynska, F. Spanier, A. Stamerra, L. Stark, L. Takalo, P. Temnikov, D. Tescaro, M. Teshima, D. Torres, N. Turini, H. Vankov, A. Venturini, V. Vitale, R. Wagner, T. Wibig, W. Wittek, F. Zandanel, R. Zanin, and J. Zapatero, "Implementation of the random forest method for the imaging atmospheric cherenkov telescope {MAGIC}," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 588, no. 3, pp. 424 – 432, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900207024059
22. L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.
23. G. Hripcsak and A. S. Rothschild, "Agreement, the f-measure, and reliability in information retrieval," *Journal of the American Medical Informatics Association*, vol. 12, no. 3, pp. 296–298, 2005.
24. D. S. Wilks, *Statistical methods in the atmospheric sciences.* Academic press, 2011, vol. 100.
25. 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.
26. A. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, pp. 507–512, 1974.
27. E. G. Jelihovschi, J. C. Faria, and I. B. Allaman, "Scottknott: a package for performing the scott-knott clustering algorithm in r," *TEMA (São Carlos)*, vol. 15, no. 1, pp. 3–17,

2014.
28. A. E. Hassan, “The road ahead for mining software repositories,” in *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE, 2008, pp. 48–57.
29. 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.
30. 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.
31. 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.
32. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. W. Shang, “Bridging the divide between software developers and operators using logs,” in *ICSE '12 :Proceedings of the 34th International Conference on Software Engineering.*
38. 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.