Understanding the Stability of Logs in Software

Suhas Kabinna, Cor-Paul Bezemer and Ahmed E.Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, Ontario
Email:{kabinna,bezemer,ahmed}@cs.queensu.ca

Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec
Email: shang@encs.concordia.ca

***Abstract*—Logs are system generated outputs, created by logging statements in the code. They help in understanding the system behavior, monitoring choke-points and in debugging. Prior research has demonstrated the importance of logs in operating, understanding and improving software systems which has lead to the development of log management applications and tools. However, logs may change over time due to debugging, improvement or addition of new features and these changes have to be communicated to operators and administrators. In this paper, we study the different factors which can affect log stability. We conduct a case study on four large software applications namely: Liferay, ActiveMQ, Camel and CloudStack. We find that 45%-55% of the logs are changed in these software applications. We identify the log changes which effect the static text, variables or verbosity level of log and exclude other changes to logs. Next, we build models to predict if a log added to a file will change in the future. We use context and log metrics calculated at the time of introduction of the log, to build a random forest classifier. Our classifier can predict which logs will change in the future with 89%-91% precision and 71%-83% recall. We find that file ownership, developer experience, log density and SLOC are strong predictors of log stability in our models and can help identify which logs are more likely to change in the future.On the one hand, 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, which can reduce their maintenance costs.**

I. Introduction

Logs are leveraged by developers to record useful information during the execution of an application. Logs are recorded during various development activities such as bug fixing [1], [2], [3], load test analysis [4], monitoring performance [5] and for knowledge transfer [6]. Logging can be done through the use of log libraries or more archaic methods such as *print* statements. Every log contains a textual part, which provides information about the context, a variable part providing information about the event and a 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* is the context information and *host*, which is the variable part, provides information about the logging context.

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

The unified format of logs has lead to the development of many enterprise log processing tools such as *Splunk* [7], *Xpolog* [8], *Logstash* [9] and research tools such as Salsa [10], log-enhancer [5] and chukwa [11] which are designed to diagnose as well as improve logging in software applications. However, when logs are changed the log processing tools also have to be updated to reflect the changes to the log. For example, Figure 1 demonstrates a case in which a developer removes the time taken for completing an event. This can affect log processing tools that rely on that information for other activities and the data lost is not recoverable. Prior research also shows that 60% of the logs which are ouputted at system execution are changed across releases [6]. These log changes can affect the log processing tools which heavily depend on them and maintenance cost will be high [6].

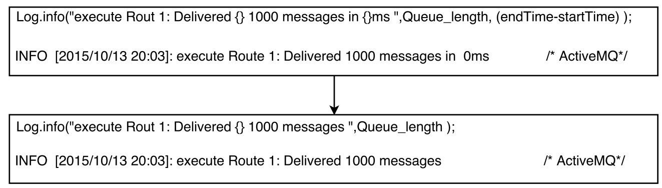


Fig. 1: Modification of a logging statement

In this paper, we study the changes made to logs across multiple releases in four studied open source applications. We find that 45%-55% of the logs are changed at least once during their lifespan in the studied applications. We find that a single log changes between 1 to 12 times within its lifetime and can be changed by more than one developer. To identify which factors play a vital role in the stability of logging statements and predict which logs will change in future, we build a random forest classifier using context and log metrics. The most important observations in this paper are:

1) Our *random forest* achieves an precision of 89%-91% and recall of 71%-83%, when predicting which logs will be changed.
2) Logs introduced in a file by developers who have less ownership of that file are more likely to be changed later than logs written by owners of the file.
3) Files with a higher log density are less likely to have changes made to their logs than files with a lower log

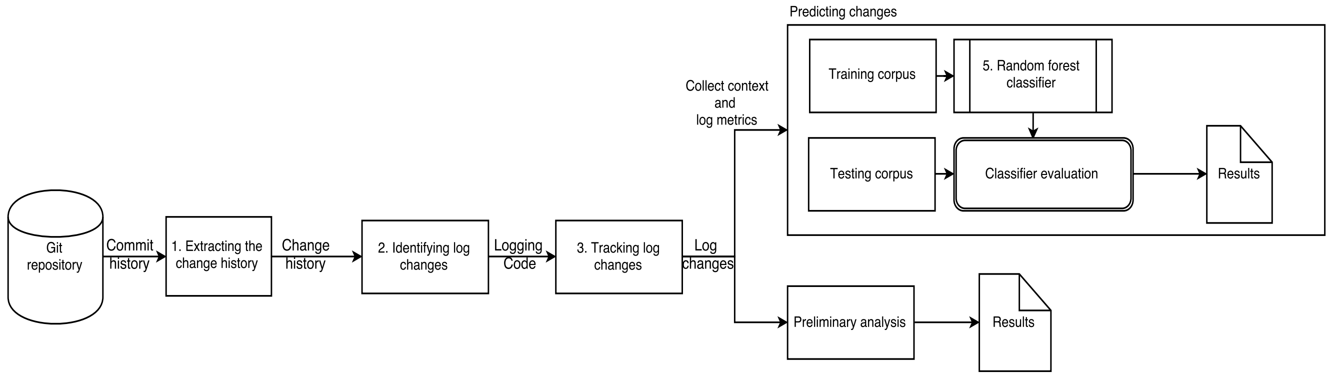


Fig. 2: Overview of the data extraction and case study approach

density.

4) Developer experience is negatively correlated in the studied applications, suggesting that logs introduced by more experienced developers are more stable.
5) Change metrics such as SLOC, number of variables logged and number of variables declared are strong predictors of log stability within the studied applications.

The remainder of this paper is organized as follows. Section II presents the preliminary analysis to motivate our study. Section III describes the random forest classifier and the analysis results. Section IV describes the prior research that is related to our work. Section V discusses the threats to validity. Finally, Section VI concludes the paper.

II. Preliminary analysis

In this paper we aim to get a better understanding of the unstable logs in an application so that we can improve the maintenance of log processing tools. In this section we present our rationale for selecting the applications we studied and present the results of our preliminary analysis in the four studied applications.

A. Subject systems

We evaluate our approach through a case study on four open source applications. We select these projects on the following two criteria:

- **Log usage -** We select applications that make use of extensive logging in their source code.
- **Project activity -** We pick applications which have a large user base and commit history.
- **Programming language -** We pick applications written in Java as it is one of the most popular languages today [12].

To find the number of logs present in an application we use the 'grep' command to search all lines of code within the '.java' files. Next, using git log we find the total number of commits in the open source projects and pick ones which have more than 10,000 commits. We find four open source projects from the Apache git repository which fit these criteria: 1) ActiveMQ[1] is an open source message broker and integration patterns server, 2) Camel[2] is an open source integration platform based on enterprise integration patterns, 3) CloudStack[3] is open source software designed to deploy and manage large networks of virtual machines and 4) Liferay[4] is an open source software to deploy websites and portals. Table I presents an overview of the applications.

TABLE I: An overview of all studied applications

Projects	ActiveMQ	Camel	CloudStack	Liferay
Starting release	4.1.1	1.6.0	2.1.3	6.1.0-b3
End release	5.9.0	2.11.3	4.2.0	7.0.0-m3
Total # log lines	5.1k	6.1k	9.6k	1.8k
Total # of releases	19	43	111	24
Total added code	261k	505k	1.09M	3.9M
Total deleted code	114k	174k	750K	2.8M
Total # added logs	4.5k	5.1k	24k	10.4k
Total # deleted logs	2.3k	2.4k	17k	8.1k

B. Study Approach

The data extraction approach from the four studied applications consists of four steps: (1) We clone the git repository of each studied application to extract all commits made for each file. (2) We identify the log changes in the extracted files. (3) We track the changes made to each log across the commits. (4) We categorize the log changes in the commit and collect the context and log metrics for each log change in the commit. Figure 2 shows a general overview of our approach. We use

[1]http://activemq.apache.org/
[2]http://camel.apache.org/
[3]https://cloudstack.apache.org/
[4]http://www.liferay.com/

R [13], to perform experiments and answer our preliminary analysis and case study

B.1. Extracting the change history: In order to find the stability of logs we have to identify all the 'java' files in our studied applications. To achieve this, we use the 'grep' command to search for all the '*.java' files in the cloned repositories and we exclude the 'test' files.

After collecting all the Java files from the four studied applications, we use the respective git repositories to obtain all the changes made to the files within the time-frame shown in Table I. We use the 'follow' option to track the file even when they are renamed or relocated. We exclude the log changes made in non-merged branches as they might not affect log processing tools. We use the '–no-merges' option to flatten the changes to a file and exclude the final merging commit. Using this approach, we obtain a complete history of each Java file in the latest version of the master branch.

B.2. Identifying log changes: From the extracted change history of 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 studied application and identify the logging library used to generate the logs. We find that the studied applications use *Log4j* [14] and *Slf4j*[1] widely and *logback*[2] sparingly. Using this information, 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);
```

As a project can have multiple logging libraries throughout its life-cycle, we use regular expressions to match all the common log invocation patterns (i.e., 'LOG','log','_logger','LOGGER','Log'). We consider every invocation of a logging library followed by a logging level ('info', 'trace', 'debug', 'error', 'warn') a log.

B.3. Tracking log changes: After identifying all the log changes made to a file across multiple commits, we track each log individually to find out whether it has changed in subsequent revisions. We first collect all the logs present in a file at the first commit, which form the initial set of logs for the file. Every subsequent commit which has changes to a log appears as an added and deleted log in *git*. To track these log changes made, we used the Levenshtein ratio [15]. We use Levenshtein ratio instead of string comparison, because Levenshtein ratio quantifies the difference between the strings compared within the range 0 to 1 (more similar the strings the ratio approaches 1). This is necessary to compare multiple logs which can be similar, which is not possible using string comparison.

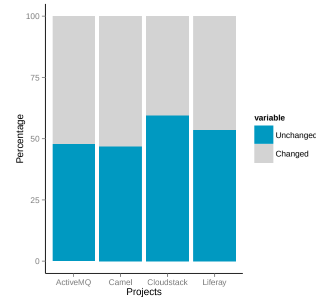
We calculate the Levenshtein ratio for each deleted log against all the added logs and pick the pair which has the highest Levenshtein ratio as a modification. This is done recursively to find all the modifications within a commit. For example in the logs shown below, we find that the Levenshtein ratio between the added and deleted pair (a1) is 0.86 and (a2) 0.76. Hence, we consider (a1) as a log modification and compare (a2) with next deleted log. If there are no more deleted log pairs, (a2) is considered as addition of new log into the file.

```
- LOG.debug("Call: " +method.getName()+ " " + callTime);
+ LOG.debug("Call: " +method.getName()+" took "+ callTime + "ms");        – (a1)
+ LOG.debug("Call: " +method.setName()+" took "+ callTime + "ms");        – (a2)
```

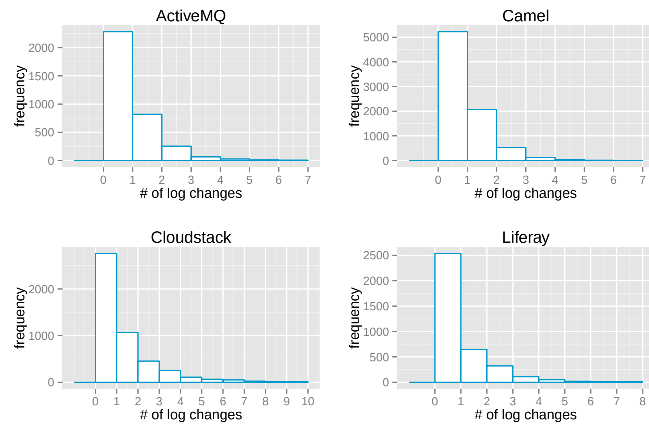
This way we track when a log is added into a file and the log is added to the initial set for tracking in future commits. From this, we track how many times a log is changed and how many commits are made between the changes.

[1]http://www.slf4j.org/

[2]http://logback.qos.ch/



(a) Percentage of log changes



(b) Frequency of log changes

Fig. 3: (a) shows the percentage of changed vs unchanged logs and (b) shows the number of times logs are changed within the studied applications.

C. Results

C.1. Change frequency:

> *Developers change 45%-55% of the logs across our studied applications.*

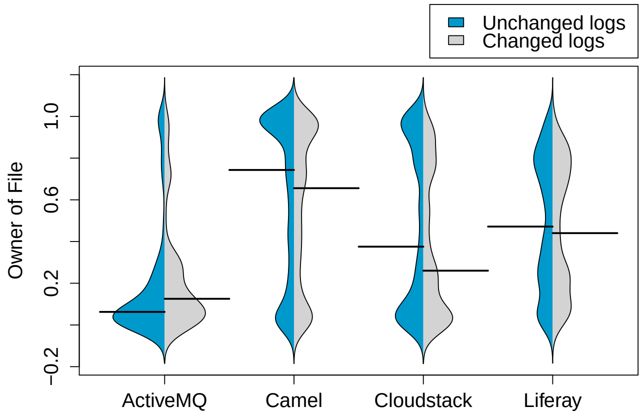


Fig. 4: Distribution of logs which are changed vs un-changed against the ownership of the developer introducing the log

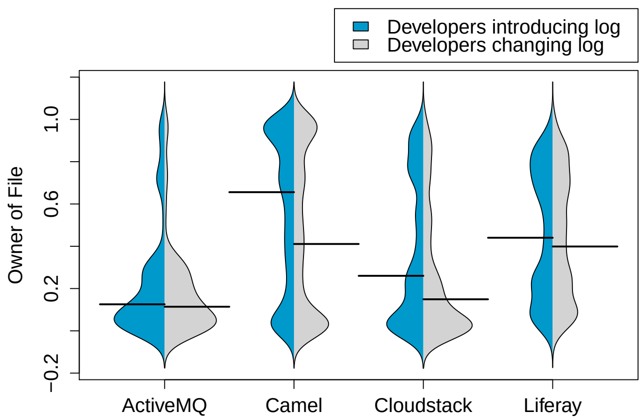


Fig. 5: Distribution of file ownership against developers introducing the log vs developers changing the log.

Figure 3 shows the percentage of changed logs in each of the studied applications. This shows that log change extensively throughout the lifetime of an application which can affect the log processing tools.

C.2. Developer impact: After identifying the frequency of changes within the studied applications, we find the number of the developers responsible for the log changes and also if they own the file which contains the log. We use the developer name available from the 'git log' to count the number of developers who change a log. To decide whether a developer owns a file we calculate the ratio of number of lines written by him to the total lines of code using the 'blame' command available in git. This is recursively done till the first commit of the file and the contribution of the developer at each commit is recorded. We take the mean average of his contribution across all commits to obtain his ownership metric for the file.

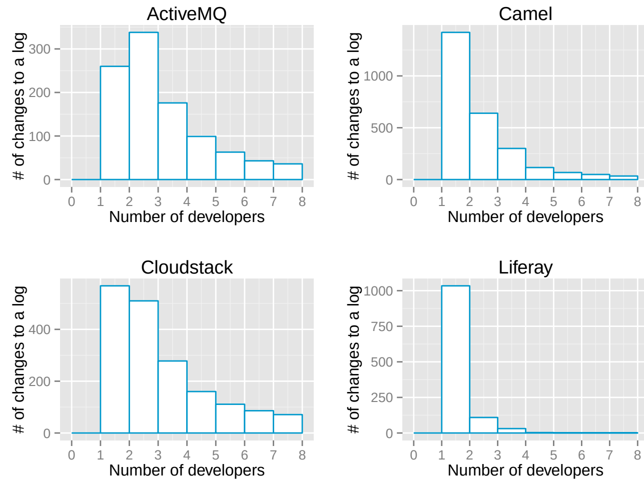


Fig. 6: Distribution of the number of developers responsible for changing a log.

We see that logs which change are introduced by developers who have little ownership over the file.

Figure 4 shows that in Camel, Cloudstack and ActiveMQ, the logs which change are more likely to be introduced by developers who have less ownership on the files, than logs which are never changed. This suggests that logs can be introduced by non-owners of a file, which leads to logs being changed later.

We see thats logs are also changed by developers who have lesser ownership than the ones introducing them. Figure 5 shows that in all the studied applications the logs are more likely to be changed by developers who have lesser ownership on the file than the developers who introduce the log. We find that in one of the studied applications the majority of logs are changed by two or more developers as seen in Figure 6. These results suggest that logs are readily changed by developers who access the file but do not have strong ownership characteristics.

III. Building a Log Change Prediction Model

From our preliminary analysis, we find that 45%-55% of logs are changed in our subject applications. This affects the log processing tools which run on these studied applications, making developers spend more time on maintenance of those tools. In this section we construct a random forest model for predicting log changes. We use this model to identify the most important factors which describe whether a log will change in the future.

A. Approach

We use context and log metrics to build the random forest classifier. Context metrics measure the changes to the applications at the time of introduction of the log and log metrics, collect the information about the introduced log. We use the git repository to extract the context metrics and log metrics for the studied applications. Table II lists all the

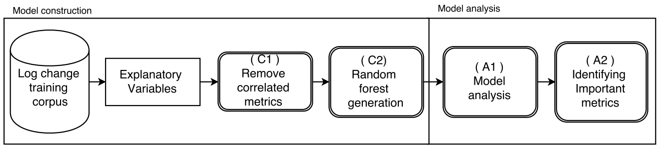


Fig. 7: Overview of model construction(C), analysis(A) and flow of data in random forest generation

metrics we collect. We define each metric and the rationale behind the choice of each metric. We use the context and log metrics because this data can be extracted from control versioning systems easily by developers. It also benefits log processing tool developers as they do not need domain knowledge about the application to understand these metrics.

We build random forest models [16] to explain the stability of logs in our studied applications. A random forest is a collection of largely uncorrelated decision trees in which the results of all trees are combined to form a generalized predictor. In our model the context and log metrics are the explanatory variables and the dependent class variable is a boolean variable that represents whether the log is changed or not (i.e., '0' for not changed and '1' for changed).

Figure 7 provides an overview of the four construction steps (C1 to C4) for building a random forest model and evaluating the model. We adopt the statistical tool R to model our data and use the 'RandomForest' package to generate the random forests.

Step C1 - Removing correlated metrics

Correlation analysis is necessary to remove the highly correlated metrics from our dataset [21]. Collinearity between metrics can affect the performance of a model because small changes in one metric can affect the values of other metrics causing large changes on the dependent class variable.

We use Spearman rank correlation [22] to find correlated metrics in our data. Spearman rank correlation assesses how well two metrics can be described by a monotonic function. We use Spearman rank correlation instead of Pearson [23] because Spearman is resilient to data that is not normally distributed. We use the function 'varclus' in R to perform the correlation analysis.

Figure 8 shows the hierarchically clustered Spearman ρ values in the 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. We include one metric from the sub-hierarchies which have correlation $|\rho| > 0.7$. The gray line indicates our cutoff value ($|\rho|$ = 0.7). We use cutoff value of ($|\rho|$ = 0.7) as used by prior research [24] to remove the correlated metrics before building our model.

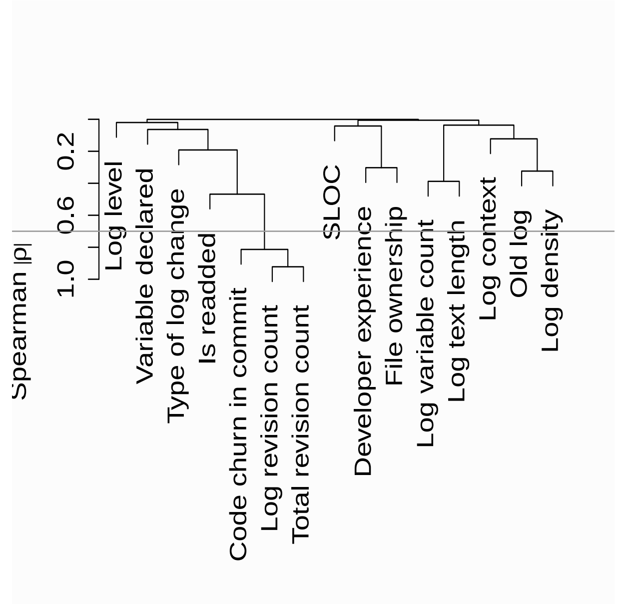


Fig. 8: Hierarchical clustering of variables according to Spearmans ρ in ActiveMQ

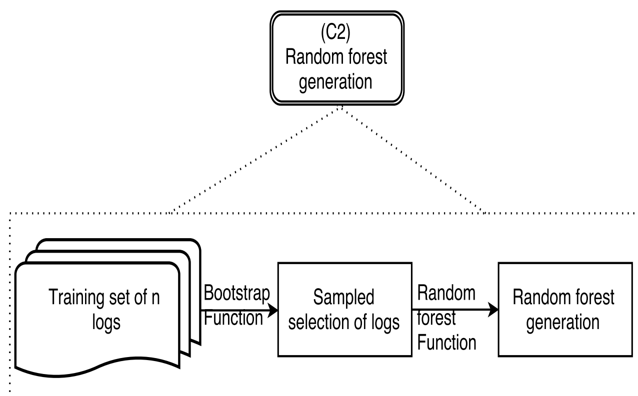


Fig. 9: Overview of random forest generation in C2

Step C2 - Random forest generation

After we eliminate the correlated metrics from our datasets, we construct the random forest model. Random forest is a black-box ensemble classifier, which operates by constructing a multitude of decision trees on the training set and uses this to classify the testing set.

Given a dataset of *m* logs for training, $D = \{(X_1Y_1), ..., (X_mY_m)\}$ where $X_{i,i=1...m}$, is a vector of descriptors (i.e., X are the metrics which are left after correlation analysis) and Y_i is the flag which indicates whether a log is changed or not. Figure 9 explains the construction of the random forest classifier and the steps are explained below.

1) From the training set of *m* logs, a random sample of *n* components is selected with replacement (i.e., bootstrap sample) [24].

TABLE II: Taxonomy of metrics considered for model construction

Dimension	Metrics	Values	Definition (d) – Rationale (r)
Context Metrics	Old log	Boolean	d: Check if the log is added to the file after creation or it wad added when file was created.
			r: This helps to identify if the logs added into file after creation are changed more than logs added at creation of file.
	Total revision count	Numerical	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 [17].
	Code churn in commit	Numerical	d: The code churn of the commit in which a log is added.
			r: Log changes are correlated to code churn in files [14].
	File ownership	Numerical	d: Identify the percentage of the file written by developer introducing the log
			r: The owner of the file is more likely to introduce stable logs than developers who have not edited the file before.
	Variables declared	Numerical	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 [17].
	SLOC	Numerical	d: The number of lines of code in the file.
			r: Large files have more functionality and are more prone to changes [18] and more log changes [17], [14].
	Developer experience	Numerical	d: The number of commits the developer has made prior to this commit.
			r: Research has shown that experienced developers might take up more complex issues [19] and therefore may leverage logs more [14].
Log Metrics	Log context	Categorical	d: Identify the block in which a log is added. (i.e., 'if', 'if-=else', 'try-catch', 'exception', 'throw', 'new function').
			r: Prior research finds that logs are mostly used in assertion checks, logical branching, return value checking, assertion checking [20].
	Is readded	Boolean	d: Check if the log is readded into a file
			r: Identify the logs which get readded into a file after they are removed.
	Log change type	Categorical	d: Check the type of log change the log has undergone before i.e., relocation, text-variable change, level change.
			r: This helps in removing the relocation changes from the dataset and check if logs that have changed once before undergo similar changes again.
	Log variable count	Numerical	d: Number of variables logged.
			r: Over 62% of logs add new variables [17]. Hence fewer variables in the initial log statement might result in addition later.
	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 on average one log line per 30 lines of code [17]. If it is less it suggests there may be additions in later commits.
	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 logs [17].
	Log text length	Numerical	d: Number of text phrases logged (i.e., we count all text present between a pair of quotes as one phrase).
			r: Over 45% of logs have modifications to static context [17]. Logs with fewer phrases might be subject to changes later to provide better explanation.
	Log revision count	Numerical	d: The number of prior commits which had log changes.
			r: This helps to identify if the file is prone to log changes.

We use the 'boot' function from the 'boot' package in R to generate bootstrap samples. The boot function generates a set of random indices, with replacement from the integers 1:n.

2) From the sampled selection, many classification trees are grown without pruning. Each classification tree gives a vote, i.e., 'true' if log is changed and 'false' if not. The random forest chooses the class with most number of votes over all the trees in the forest. This strategy helps to make the random forests more robust against over-fitting [25].
We use the 'randomForest' function from the 'randomForest' package in R to generate the random forest model.
3) The above steps are repeated until M such models are grown.
4) Predict new data by aggregating the prediction of the M models generated.

TABLE III: Confusion Matrix

		Predicted	
		Log changed	Log not changed
Actual	Log change	True positive (TP)	False negative (FN)
	Log not changed	False positive (FP)	True negative (TN)

Step A1 - Model analysis

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

Precision (P) measures the correctness of our model in predicting which log will undergo a 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) measures 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 [26].

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

Brier Score (BS) is a measure of the accuracy of the predictions in our model [27]. It explains how well the model performs compared to random guessing i.e., a perfect classifier will have Brier score of 0 and perfect misfit classifier will have Brier score of 1 (predicts probability of log change when log not changed). This means the lower the Brier score value, the better our random forest classifier.

These 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 the model is over-fit.

To account for the overfitting in our models, we use *optimism* measure, as used by prior research [24]. The *optimism* of the performance measures are calculated as follows.

1) From the original dataset with *m* records, select a bootstrap sample with *m* components with replacement.
2) Build random forest as described in (C2) using the bootstrap sample.
3) Apply the classier model built from bootstrap sample 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.

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 the original model fit is.

Step A2 - Identifying important metrics

To find the importance of each metric in a random forest model, we use a permutation test. In this test, the model built using the bootstrap data (i.e., two thirds of the original data) is applied to the test data (i.e., remaining one third of the original data).

Then, the values of the $X_i\ ^{th}$ metric of which we want to find importance for, are randomly permuted in the test dataset and the accuracy of the model is recomputed. The decrease in accuracy as a result of this permutation 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.

As we obtain 1000 data sets for each metric because of bootstrapping process, we use the **Scott-Knott** (SK) clustering to group the metric based on their means [28], [29]. This is done to group metrics which are strong predictors of likelihood of log change. 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 the significance threshold parameter to 0.05 to cluster the metrics into different groups.

B. Results

> *The random forest classifier achieves a precision of 0.89-0.91 and recall of 0.71-0.83 for our studied applications.*

Figure 10 shows the optimism-reduced values of *precision*, *recall*, *F-measure* and *Brier score* for each project. The model achieves AUC of 0.95-0.96 across the studied applications. We find the recall of the random forest classier in Liferay is not as high as the other projects. This may be because Liferay has the lowest total number of log lines and close to 50% of the log changes are log relocations as seen in Table **??**. Because of the lower percentage of logs which are changed, the random forest classifier has fewer nodes (trees) and likelihood of false negatives is higher.

> *The random forest classifier outperforms random guessing.*

The classifier achieves Brier scores between 0.04 and 0.07 across all projects. If the model achieves a Brier score of 0.07, it means our model can forecast with 73% probability a log

TABLE IV: The importance values of the metrics (top 10), divided into homogeneous groups by Scott-Knott clustering. The '+' and '-' signs signifing positive and negative correlation of the metric on log stability.

Active MQ			Camel		
Rank	**Factors**	**Importance**	**Rank**	**Factors**	**Importance**
1	SLOC	0.186 +	1	Ownership of file	0.218 +
2	Ownership of file	0.180 +	2	Developer experience	0.200 -
	Developer experience	0.179 +	3	Log density	0.188 -
3	Log density	0.132 -	4	SLOC	0.187 +
4	Code churn in commit	0.093 +	5	Log level	0.135 -
5	Log variable count	0.072 +	6	Log variable count	0.133 +
6	Variable declared	0.067 -	7	Variable Declared	0.122 +
7	Log level	0.061 +	8	Code churn in commit	0.103 +
8	Log text length	0.055 +	9	Log context	0.060 -
CloudStack			**Life Ray**		
Rank	**Factors**	**Importance**	**Rank**	**Factors**	**Importance**
1	Developer experience	0.152 -	1	Log density	0.199 -
2	Ownership of file	0.149 +	2	Developer experience	0.180 +
	Code churn in commit	0.148 +	3	SLOC	0.165 +
3	Log density	0.139 -		Log level	0.165 +
4	SLOC	0.135 +	4	Ownership of file	0.151 -
5	Log variable count	0.122 +		Variable Declared	0.150 +
6	Log text length	0.107 +	5	Log variable count	0.074 +
7	Variable Declared	0.084 +	6	Log context	0.059 -
8	Type of log change	0.066 +		Log text length	0.059 +

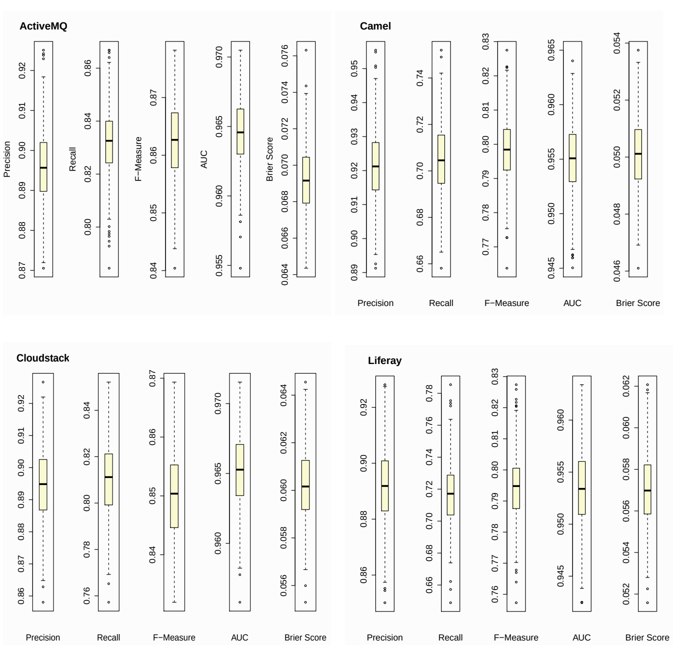


Fig. 10: The optimism reduced performance measures of the four projects

will change. Brier score reaches 0.25 for random guessing (i.e., predicted value is 50%).

Important metrics for log stability

> *We find that* log density *is an important metric in our studied applications.*

We find that log density has negative correlation with log stability (i.e., increase in log density decreases probability of log change), in all the studied applications as seen in Table IV. This suggests that when source code is well logged i.e., more logs per lines of code, the logs may communicate the necessary information making them more stable.

> *We find* log variable count *has a positive correlation with log stability as shown in Table IV.*

This implies that more variables in a log results in a higher likelihood that a log will be changed. This may be because there are inconsistencies between logs and the actual needed information intended as shown by prior research [17] and developers have to update logs to resolve the inconsistencies.

> *We find that* file ownership *is a strong predictor of log change and has positive correlation in three of the studied applications.*

This suggests that logs introduced by developers who have little ownership are more unstable and have to be changed. This is seen Figure 4, where in three of the studied applications, the logs which change are introduced by developers who have lower ownership of a file.

> *We find that* developer experience *has negative correlation in the studied applications.*

Even though ActiveMQ and Liferay has positive correlation in Table IV, we find that these projects have strong code ownership and two developer are responsible for over 50% of the total commits within the projects. To remove this strong ownership, we exclude the log changes made by these top developers in ActiveMQ and Liferay and we find that developer experience has negative correlation in both ActiveMQ and Liferay. This suggests that logs which are introduced by more experienced developers are less likely to change in all of the studied applications.

IV. Related Work

In this section, we present prior research in which log behavior in software applications is analyzed. In addition, we discuss tools developed to assist in logging.

A. Log analysis

Prior work leverages logs for detecting anomalies in large scale systems. Lou et al. [2] propose an approach to use the variable values printed in logs to detect anomalies in large systems. Based on the variable values in logs, their approach creates invariants (e.g., equations). Any new logs that violates the invariant are considered to be a sign of anomalies. Fu et al. [3] built a finite state automaton using unstructured logs to detect performance bugs in distributed systems. Xu et al. [1] link output logs to logs in source code to recover the text and and the variable parts of logs in source code. They applied Principal Component Analysis (PCA) to detect system anomalies. To assist in fixing bugs using logs, Yuan et al. [30] propose an approach to automatically infer the failure scenarios when a log is printed during a failed run of a system.

Logs are leveraged during load testing of large scale systems. The data collected from logs during load tests helps developers diagnose faults in the system. Jiang et al. [31], [32], [33], [34] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [31]. Based on the such events, they identified both functional anomalies [32] and performance degradations [33] in load test results. The extensive prior research on log analysis shows that logs are leveraged for different purposes and changing logs can affect the performance of log analysis tools.

B. Log tools

Tan et al. [10] propose a tool named SALSA, which constructs state-machines from logs. The state-machines are further used to detect anomalies in distributed computing platforms. Yuan et al. [5] show that logs need to be improved by providing additional . Their tool named *Log Enhancer* can automatically provide additional control and data flow parameters into the logs thereby improving the logs. While these works focus more on enhancing existing logs in the system, our paper focuses more on informing developers which logs are more likely to get changed and identifying which factors explain the change of logs.

C. Empirical Studies on Logs

Prior research performs an empirical study on the characteristics of logs. Yuan et al. [17] study the logging characteristics in four open source systems. They find that over 33% of all log changes are after-thoughts and that logs are changed 1.8 times more often than regular code. Fu et al. [20] performed an empirical study on where developers put logs. They find that 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 a F-measure of over 95% was achieved.

Research also shows that logs are a source of information about the execution of large software systems for developers and end users. Shang et al. performed an empirical study on the evolution of both static logs and logs outputted during run time [14], [35]. They find that logs are co-evolving with software systems. However, logs are often modified by developers without considering the needs of operators which even affects the log processing tools which run on top of them. They highlight the fact that there is a gap between operators and developers of software systems, especially in the leverage of logs [36]. Furthermore, Shang et al. [37] 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. These research works highlight that developers and system operators leverage logs and changing logs can affect both.

V. 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 studied applications have years of history and large user bases, these applications are all Java based. Other languages may not use logs as extensively. Our projects are all open source and we do not verify the results on any commercial platform applications. More case studies on other domains and commercial platforms, with other programming languages are needed to see whether our findings can be generalized.

Construct Validity

Our heuristics to extract logging source code may not be able to extract every log in the source code. Even though the studied applications-leverage logging libraries to generate logs at runtime, there may 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 Levenshtein ratio and choose a threshold to identify modifications to logs. However, this threshold may not accurately identify modifications to logs. Further sensitivity analysis on this threshold is needed to better understand the impact of the threshold to our findings.

Internal Validity

Our study is based on the data obtained from git for all the studied applications. 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 logs cannot claim causal effects, as we are investigating 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 in future studies.

VI. Conclusion

Logs are snippets of code, introduced by developers to record valuable information. The recorded 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 affected when logs are changed.

In this paper we study the stability of logs using a random forest classifier. The classifier is used to predict which logs are more likely to change in the future using context and log data. The highlights of our study are:

- We find that 45%-55% of logs are changed at-least once.
- Our random forest classifier for predicting whether a log will change achieves a precision of 89%-91% and recall of 71%-83%.
- We find that log density, SLOC, developer experience, file ownership and log variable count are strong predictors of log stability in the studied applications.

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] D. Carasso, "Exploring splunk," *published by CITO Research, New York, USA, ISBN*, pp. 978–0, 2012.

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

[9] X. Xu, I. Weber, L. Bass, L. Zhu, H. Wada, and F. Teng, "Detecting cloud provisioning errors using an annotated process model," in *Proceedings of the 8th Workshop on Middleware for Next Generation Internet Computing*. ACM, 2013, p. 5.

[10] 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.

[11] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang, "Chukwa, a large-scale monitoring system," in *Proceedings of CCA*, vol. 8, 2008, pp. 1–5.

[12] "Java language ranking -http://www.tiobe.com/paperinfo/tpci/index.html."

[13] R. Ihaka and R. Gentleman, "R: a language for data analysis and graphics," *Journal of computational and graphical statistics*, vol. 5, no. 3, pp. 299–314, 1996.

[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] 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.

[16] J. Albert and E. Aliu, "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

[17] 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.

[18] 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.

[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] 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.

[21] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013.

[22] J. H. Zar, *Spearman Rank Correlation*. John Wiley & Sons, Ltd, 2005. [Online]. Available: http://dx.doi.org/10.1002/0470011815.b2a15150

[23] R. J. Serfling, *Approximation theorems of mathematical statistics*. John Wiley & Sons, 2009, vol. 162.

[24] 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.

[25] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[26] 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.

[27] D. S. Wilks, *Statistical methods in the atmospheric sciences*. Academic press, 2011, vol. 100.

[28] A. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, pp. 507–512, 1974.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.