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 created by logging statements and they assist in understanding system behavior, monitoring choke-points and debugging. Prior research has demonstrated the importance of logs in operating, understanding and improving software systems. The importance of logs has lead to a new market of log management applications and tools. However, logs are often unstable i.e., their generating logging statements are often changed without the consideration of other stakeholders, causing misleading results and failure of log analysis tools. In order to proactively mitigate such issues that are caused by unstable logging statements, in this paper we empirically study the stability of logging statements in four large software applications namely: Liferay, ActiveMQ, Camel and CloudStack. We find that although around half of the logs are never changed, some logs are changed up to 10 times during development and more than half of the changed logs are changed within 7 commits after being added into the applications.

We use metrics that are calculated from context and log information, to build a random forest classifier for predicting whether a log added to a file will be changed later. We show that our classifiers achieve 89%-91% precision, 71%-83% recall in the four studied applications. We find that file ownership, developer experience, log density and SLOC are important predictors of log stability in our models. Our findings can help practitioners avoid depending on such unstable logs through critical analysis and develop more robust log processing tools

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 fixing bugs [1, 2, 3], analyzing load tests [4], monitoring performance [5] and transferring knowledge [6]. Logging can be done through the use of log libraries or more archaic methods such as *print* statements. Every log[1] contains a textual part, which provides information about the context, a variable part providing context information about the event and a log level, which shows the verbosity of the logs. An example of a log is in Figure 1.

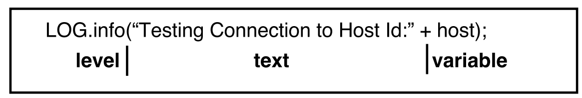

Fig. 1: Example of log statement

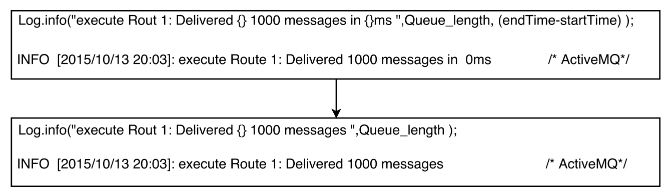

Fig. 2: Modification of a log statement

The rich knowledge in 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 analyze as well as improve logs in software applications. However, when logs are changed, the associated log processing tools may also need to be updated. For example, Figure 2 demonstrates a case in which a developer removes the time taken for completing an event. This can affect log processing tools that rely on the removed information inorder to monitor the health of the application. Prior research shows that 60% of the logs which are output during system execution are changed and affect the log processing tools that heavily depend on such logging statements [6].

As changes to logging statements can affect the log processing tools depending on them, it is necessary to identify the cause for these logging statement changes. We identify three main factors which can cause changes in logging statements namely: 1) who introduces and changes logs, 2) where the logging statement are placed, and 3) what is content of the logged statement.

To identify the influence of these factors, in this paper we first study how much logging statements are changed across multiple releases in the four software applications. We find that 35%-50% of the logging statements are changed at least once during their lifespan in the studied applications. We also find that a single log changes between 0 to 10 times within its lifetime and can be changed by more than one developer. These results suggest that log processing tools have to be constantly monitored at every release to prevent breakage by

[1] In the rest of this paper, we use log to refer to the logging statements in the source code.

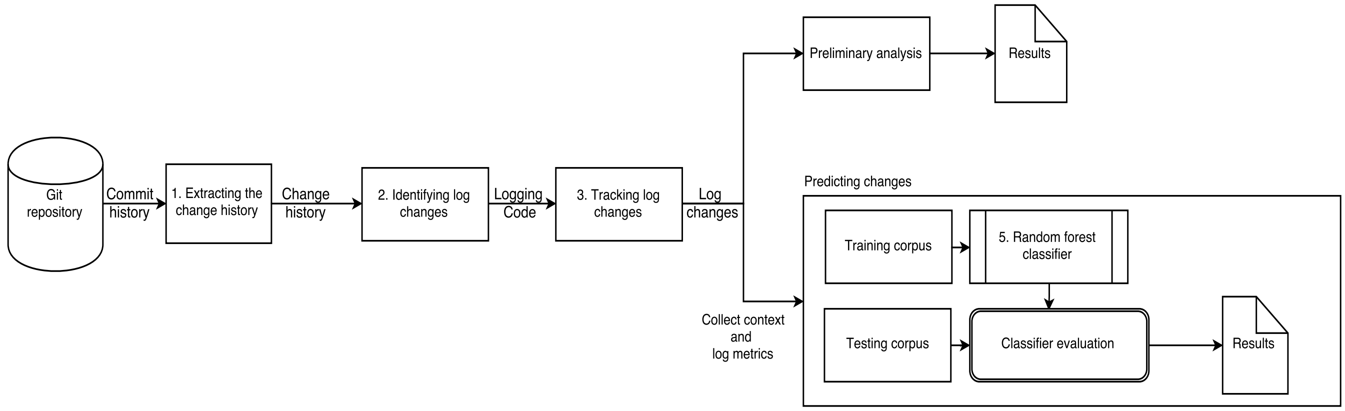

Fig. 3: Overview of the data extraction and empirical study approach

the changes to logging statements.

To identify which factors the stability of logging statements and model which logs will change in the future, we build a random forest classifier using context and log metrics. The most important observations in this paper are:

1) We can predict which logs will be changed using a *random forest* classifier with a precision of 89%-91% and recall of 71%-83%

2 a) Logging statements changed by developers who have less ownership in that file are 0.4 - 1.4 times more likely to be changed again than logs changed by owners of the file in three of the applications.

b) Developer experience is negatively correlated to log stability in the studied applications, suggesting that logs added by more experienced developers are more stable. This findings suggest that knowledge of the code in a file is important when considering the stability of a logging statement. As a result, maintainers of log processing tools can take special care when importing log statements written by developers who have less experience or with no ownership of the file in which the statement was added.

3) Large files (i.e., SLOC is 2x-3x the median) with lower log density are more likely to have changes made to their logs. This suggests that maintainers of log processing tools should be careful when importing log statements written in larger file with less number of logs.

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 study the changes that are made to logs across multiple releases in open source applications. In order to understand and build a random forest classifier to predict log changes, it is necessary to identify the extent logs are changed within these applications.Hence, in this section we present our rationale for selecting the applications that we studied and present the results of our preliminary analysis on the four studied applications.

A. Studied Applications

We evaluate our approach through an empirical study on four open source applications. We selected these projects on the following three criteria:

- **Log usage -** The selected applications must make extensive use of logs in their source code
- **Project activity -** The applications must have a large user base and commit history
- **Programming language -** To simplify the implications of our empirical study, we opted to select only applications written in Java and published on Git repository

To find the log usage 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 code repositories and select projects which have more than 10,000 commits. We find four open source projects from the Apache Git repository which fit these criteria. ActiveMQ[1] is an open source message broker and integration patterns server. Camel[2] is an open source integration platform based on enterprise integration patterns. CloudStack[3] is an open source application designed to deploy and manage large networks of

[1]http://activemq.apache.org/
[2]http://camel.apache.org/
[3]https://cloudstack.apache.org/

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

virtual machines. Liferay[4] is an open source application for websites and portals deployment. Table I presents an overview of the applications. We pick the releases after incubation for each application, as during incubation the applications might not be used by log processing tools.

B. Data Extraction Approach

The data extraction approach from the four studied applications consists of three steps: (1) We clone the Git repository of each studied application to extract the change history made for each file (2) We identify the logs present in the file, and using the change history we identify the log changes made in each file (3) We track the log changes that are made to each log in a file across the commits. We use R [12], to perform experiments and answer our preliminary analysis and empirical study. Figure 3 shows a general overview of our approach and we discuss each step discussed above in further detail.

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 their Git repositories to obtain all the changes that are made to the files within the time-frame shown in Table I. We use the *follow* option to track a file even when it is renamed or relocated. We exclude log changes that are made in non-merged branches as they are unlikely to 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* [13] and *Slf4j*[2] widely and *logback*[3] 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 that are 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 change to a log in the subsequent commits appears as an added and deleted log in Git. To identify added, deleted and modified logs, we leverage the Levenshtein ratio [14]. 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.

To identify log modification we calculate the Levenshtein ratio between all the deleted log and added logs and select the pair which has the highest Levenshtein ratio. 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.

As we cannot identify changes to a log which is added near the end of the time frame, we have to exclude such logs from our analysis. We find that in the studied applications, a new log added changes within 390 commits of being added, as seen from Table II. We exclude all logs added into the application 390 commits before the last commit of our analysis.

C. Results

Developers change 35%-50% of the logs across our studied applications.

[4]http://www.liferay.com/
[2]http://www.slf4j.org/
[3]http://logback.qos.ch/

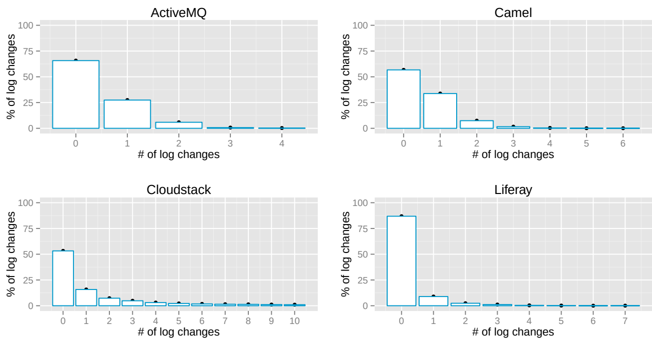

Fig. 4: Distribution of log changes in the studied applications

TABLE II: Summary of the number of commits before a new log added is changed in the studied applications

Project	Min	1st Qu	Median	Mean	3rd Qu	Max
ActiveMQ	1	2	7	9	14	37
Camel	1	1	2	4	5	117
Cloudstack	1	1	3	17	14	390
Liferay	1	1	1	7	1	130

Figure 4 shows the percentage values for the number of times a log is changed in each of the studied applications. This shows that logs change extensively throughout the lifetime of an application which can affect the log processing tools.

75% of the new logs which change, are changed within 15 commits since their addition. From Table II, we find that majority, i.e., 75% are changed within 15 commits since addition. We also find that the median code churn during these log changes is less than 50 lines of code in three of the studied applications as seen in Table III. This suggests that the log changes are more likely to be changed due to rewording changes rather than major changes to the added feature. This means that new logs which are introduced prior to 15 commits in the studied applications, are less likely to break the log processing tools which might depend on them.

III. Building a log change prediction model

In our preliminary analysis, we find that 35%-50% 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, so that these factors can be taken into account by developers of log processing tools.

A. Approach

We use context and log metrics to build the random forest classifier. Context metrics measure the file context at the time of adding the log. Log metrics measure the details about the added log. We use the Git repository to extract the context metrics and log metrics for the studied applications. Table IV lists the metrics we collect. We define each metric and the rationale behind the choice of each metric. We use the context

TABLE III: Summary of total code churn in the commits where a new log is changed

Project	Min	1st Qu	Median	Mean	3rd Qu	Max
ActiveMQ	2	25	47	141	163	493
Camel	2	13	32	98	133	456
Cloudstack	2	66	234	410	574	4121
Liferay	2	6	14	28	27	278

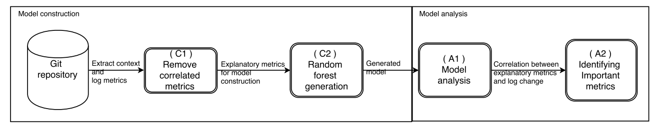

Fig. 5: Overview of model construction (C), analysis (A) and flow of data in random forest generation

and log metrics because this data can be extracted from the source code repository 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 [15] to predict whether a log will change 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 ever changed or not (i.e., '0' for not changed and '1' for changed).

Figure 5 provides an overview of the construction steps (C1 to C3) for building a random forest model and steps (A1 and A2) provides an analysis of the results. 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 [19]. 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 [20] 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 [21] because Spearman is resilient to data that is not normally distributed. We use the function *varclus* in R to perform the correlation analysis.

Figure 6 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

TABLE IV: 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 was added when file was created. r: Logs added into a file after creation might be more likely to be changed than the logs added during file creation.
	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: Logs present in a file which is changed heavily, have higher chance of being changed as prior research shows that average log churn rate is twice that of entire code [16]. Hence, the more commits to a file, the higher the likelihood of logs being changed.
	Code churn in commit	Numerical	d: The code churn of the commit in which a log is added. r: Logs added during large code changes like feature addition might be more stable than logs added during bug fixes which have lesser code changes.
	File ownership	Numerical	d: The percentage of the file written by developer adding the log. r: The owner of the file is more likely to add 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 a large # of variables are declared, there is higher chance that any of the variables can be changed afterwards.
	SLOC	Numerical	d: The number of lines of code in the file. r: Large files have more functionality and are more prone to changes [17] and log changes [13, 16].
	Developer experience	Numerical	d: The number of commits the developer has made prior to this commit. r: More experienced developers are more likely to add more stable logs, since they have a better understanding of the application and may not make mistakes in the logs.
Log Metrics	Log context	Categorical	d: 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 and return value checking [18]. Therefore, logs used in logical branching and assertion checks, i.e., if-else blocks, may not be as stable as logs in exception blocks.
	Is re-added	Boolean	d: Check if the log is re-added into a file. r: Logs which are added, removed and re-added into a file suggest that developers are unsure of the purpose of the log making them very unstable and prone to changes.
	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: Changes to log text, variable and verbosity level can make logs more unstable than relocation changes.
	Log variable count	Numerical	d: Number of variables logged. r: Over 62% of logs add new variables [16]. Hence, fewer variables in the initial log statement might result in addition later.
	Log density	Numerical	d: Ratio of the number of log lines to the source code lines in the file. r: Research has found that files with logs tend to be more defect-prone [13]. Hence, files with higher log density might be more defect-prone, forcing developers to leverage logs to assist in debugging and making them more unstable.
	Log level	Categorical	d: The log level (verbosity) of the added log, i.e., *info,error, warn, debug, trace* and *trace*. r: Research has shown that developers spend significant amount of time in adjusting the verbosity of logs [16]. Hence, higher level logs such as *warn* and *error* might be more carefully placed than default level *info* logs and the higher level info are less likely to be changed than default level logs.
	Log text length	Numerical	d: Number of text phrases logged. We count all text present between a pair of quotes as one phrase. r: Over 45% of logs have modifications to static context [16]. Logs with fewer phrases might be subject to changes later to provide a better explanation.

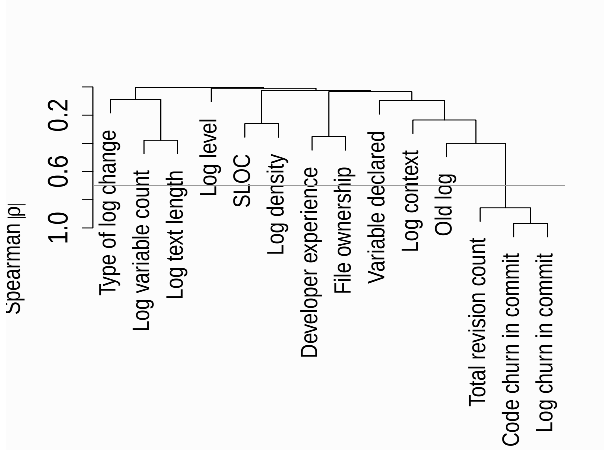

Fig. 6: Hierarchical clustering of variables according to Spearmans ρ in ActiveMQ

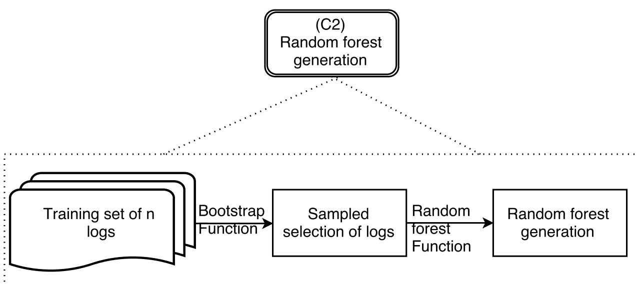

Fig. 7: Overview of random forest generation in C2

prior research [22] to remove the correlated metrics before building our model. We find that *total revision count* is highly correlated with *code and log churn in commit*, because a file with more commits has higher chance of having a large commit with log changes, than a file with fewer commits. We exclude *total revision count* and *log churn in commit* and retain *code churn in commit* as it is the simpler matrix to compute.

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. Figure 7 explains the construction of the random forest classifier, where from a training set of m logs a random sample of *n* components is selected with replacement [22] and using the *randomForest* function from the *randomForest* package in R, a random forest model is generated.

TABLE V: 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, AUC and Brier Score. These measures are functions of the confusion matrix as shown in Table V and are explained below.

Precision (P) measures the correctness of our model in predicting which log will 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.

$$P = \frac{TP}{TP + FP} \quad (1)$$

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.

$$R = \frac{TP}{TP + FN} \quad (2)$$

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].

$$F = \frac{2 \times P \times R}{P + R} \quad (3)$$

Area Under Curve (AUC) is used to measure the overall ability of the model to classify changed and unchanged logs. The value of AUC ranges between 0.5 (worst) for random guessing and 1 (best) where 1 means that our model can correctly classify every log as changed or unchanged.

Brier score (BS) is a measure of the accuracy of the predictions in our model [24]. It explains how well the model performs compared to random guessing, i.e., a perfect classifier will have a Brier score of 0 and a perfect misfit classifier will have a 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 measures, 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 overfit. To account for the overfitting in our models, we use the *optimism* measure, as used by prior research [22]. 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 classifier model built from the bootstrap sample on both the bootstrap and original data sample, calculating precision, recall, F-measure and Brier score for both data samples.
4) Calculate the *optimism* by subtracting the performance measures of the bootstrap sample from 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, AUC 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 precision of the model is recomputed. The decrease in precision 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 every time during the bootstrapping process to obtain 1,000 importance scores for each metric in our dataset.

As we obtain 1,000 data sets for each metric because of bootstrapping process, we use the **Scott-Knott** (SK) clustering to group the metric based on their means [25, 26]. This is done to group metrics which are important predictors for the 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, recall of 0.71-0.83 and outperforms random guessing for our studied applications

Figure 8 shows the optimism-reduced values of *precision*, *recall*, *F-measure* and *Brier score* for each project. The model achieves an AUC of 0.95-0.96 across the studied applications. 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 will change. Brier score reaches 0.25 for random guessing (i.e., predicted value is 50%).

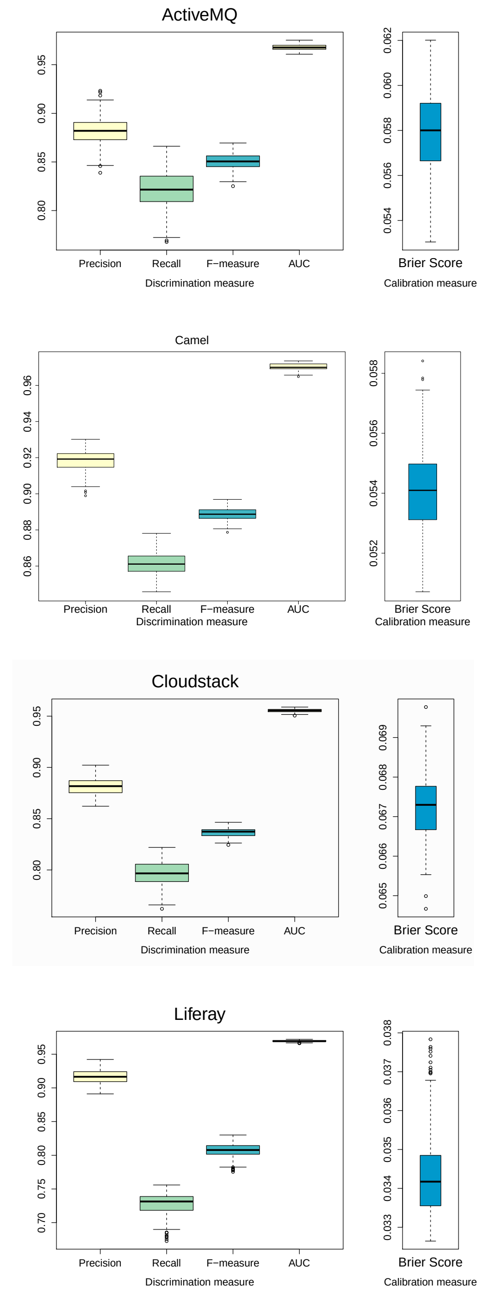

Fig. 8: The optimism reduced performance measures of the four projects

B2. Important metrics for log stability

Developer experience has negative correlation with log stability in the studied applications. From Table VI, we see that developer experience has negative correlation to log stability in three of the studied applications. This suggests that logs which change are added by less experienced developers in ActiveMQ, Camel and Liferay. This can be seen in Figure 9, where logs which change three times or more are changed by less experienced developers, when compared to unchanged logs. For example, when we inspect *git diff* for the file *Ses-*

TABLE VI: 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

ActiveMQ			Camel		
Rank	**Factors**	**Importance**	**Rank**	**Factors**	**Importance**
1	Developer experience	0.258 -	1	Developer experience	0.297 -
2	SLOC	0.188 +	2	Ownership of file	0.161 -
3	Ownership of file	0.170 +	3	Log level	0.128
4	Log density	0.166 +	4	SLOC	0.112 +
5	Log variable count	0.089 +	5	Log density	0.108 -
6	Log level	0.078		Type of log change	0.106
7	Type of log change	0.069	6	Log variable count	0.090 +
8	Variable declared	0.055 -	7	Old log	0.071 -
9	Log context	0.043	9	Log context	0.061
CloudStack			**Liferay**		
Rank	**Factors**	**Importance**	**Rank**	**Factors**	**Importance**
1	Type of log change	0.268	1	SLOC	0.192 +
2	Code churn in commit	0.243 +	2	Developer experience	0.174 -
3	SLOC	0.232 +	3	Ownership of file	0.170 -
4	Log density	0.208 -	4	Log density	0.158 -
5	Ownership of file	0.154 -	5	Log variable count	0.143 +
6	Developer experience	0.119 +	6	Variable declared	0.118 +
7	Log text length	0.095 +	7	Log context	0.106
8	Log variable count	0.091 +	8	Log text length	0.071 +
9	Variable declared	0.097 -	9	Type of log change	0.058

TABLE VII: Contribution of top 3 developers

	Total logs	Changed logs	Total # of contributors
ActiveMQ	956 (50.4%)	301 (31.4%)	41
Camel	3060 (63.1%)	1460 (47.7%)	151
Cloudstack	5982 (35.7%)	2276 (38.0%)	204
Liferay	3382 (86.7%)	609 (18.0%)	351
Average	**3345 (59%)**	**1161 (33.75%)**	747

sionBatchTransactionSynchronization.java in Camel, we find that all the logs introduced by a new developer are fixed by a more experienced developer. In this case, the experienced developer changes the log contexts, i.e., relocates the logs and adds additional context information to provide more meaning to the logs.

We also find that in ActiveMQ, Camel and Liferay, the top three developers are responsible for more than 50% of the logs. Only 30% of the logs added by these developers are changed as seen in Table VII. This result recommend that new developers should get more experience about the application by actively making more commits to write more stable logs.

File ownership is an important predictor of log change and has negative correlation in three of the studied applications. This suggests that logs added by developers who have little ownership are more unstable and more likely to be changed. This is show by Figure 10, where in Camel and Liferay, the unchanged logs are introduced by developers who have more ownership of the file. We also find that the most unstable logs i.e., (3 + changes) in Camel, Cloudstack and Liferay are done by developers who have lesser ownership of the file, accounting for the negative correlation in these three applications. These results suggest that developers who are not owners of a file, should be more cautious when adding or changing logs in the file. For example when we inspect the *git diff* for the file *SSLContextServerParameters.java* in Camel, we find that the owner of the file fixes the changes made by a non-owner by increasing the logging level to reduce a flood of logs being generated.

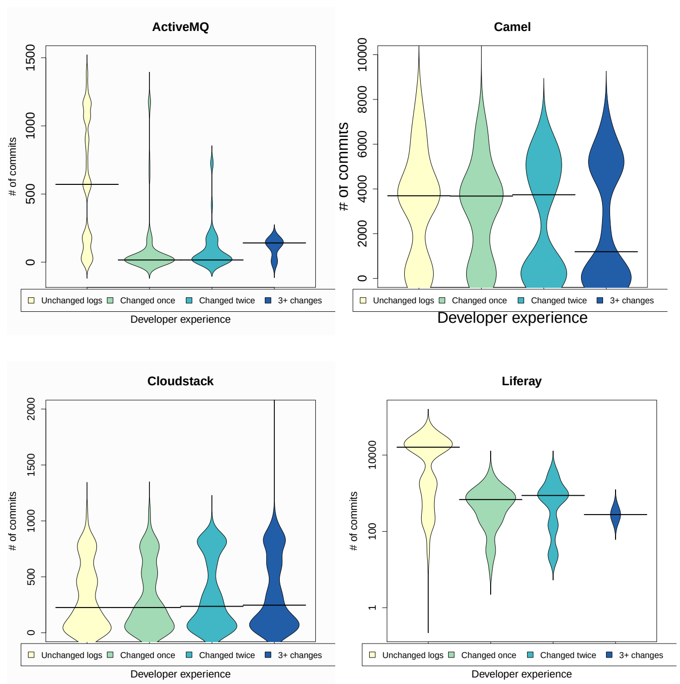

Fig. 9: Comparing experience of developers who add unchanged logs against logs changing more than once

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 Camel, Cloudstack and Liferay as seen in Table VI. We find that in these applications, the logs which change are present in files with lower log

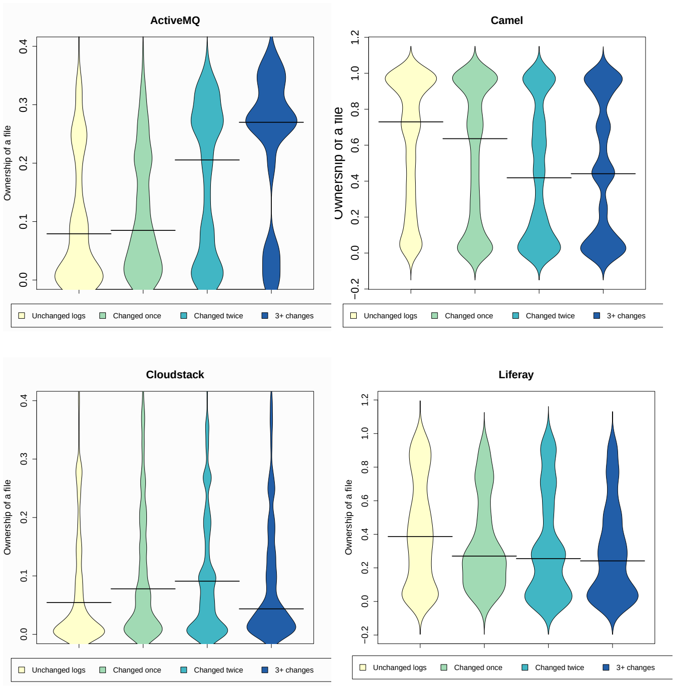

Fig. 10: Comparing file ownership of developers who add unchanged logs against logs changing more than once

density than unchanged logs. When we measure the median file sizes we find that, logs which change more are present in files with significantly higher SLOC (2x-3x higher). This suggests that large files which are not well logged are more likely to have unstable logs, than well logged files.

Our Random Forest classifier achieves a precision of 89%-91% and recall of 71%-83% across all studied applications. We find file ownership, SLOC, developer experience and log density to be important predictors of log change in our 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. [27] 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. [28, 29, 30, 31] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [28]. Based on the such events, they identified both functional anomalies [29] and performance degradations [30] 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.

C. Empirical Studies on Logs

Prior research performs an empirical study on the characteristics of logs. Yuan et al. [16] 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. [18] 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 [13, 32]. 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 [33]. Furthermore, Shang et al. [34] 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 empirical 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 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, added 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 35%-50% 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 are important 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] 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.

[13] 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.

[14] 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.

[15] 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

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.