Examining the Stability of Logging Statements

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 logging statements 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 logging statements are never changed, some logging statements are changed up to 10 times during development and more than half of the changed logging statements are changed within 17 days after being added into the applications.

We use metrics that are calculated from context and log information, to build a random forest classifier to understand which factors can increases the likelihood of a log change. We show that our classifiers achieve 83%-91% precision, 65%-85% recall in the four studied applications. We find that file ownership, developer experience, log density and SLOC are important metrics which affect the likelihood of a log change. By understanding the important metrics, practitioners can avoid the logs which have a higher likelihood of being changed and develop robust log processing tools.

I. Introduction

Developers use logging statements to yield useful information about the state of an application during its execution. This information is collected in files (logs) and contains details which would otherwise be difficult to collect, such as the value of variables. Logs are used during various development activities such as fixing bugs [1, 2, 3], analyzing load tests [4], monitoring performance [5] and transferring knowledge [6]. Logging statements make use of logging libraries or more archaic methods such as *print* statements. Every logging statement 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 logging statement. An example of a logging statement is in Figure 1.

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 logs as well as improve logging statements. However, when logging statements 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 elapsed time for an event. Removing information from a logging statement can affect log processing tools that rely on the removed information in order to monitor the health of the application. Prior research shows that 60% of the logging statements that generate output during system execution are changed and affect the log processing tools that heavily depend on the logs that are generated by these logging statements [6].

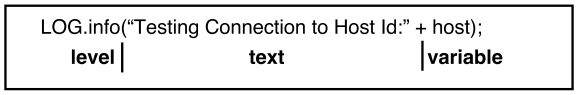

Fig. 1: Example of a logging statement

Knowing whether a logging statement is likely to change in the future is helpful to reduce the effort required to maintain log processing tools. If a developer of a log processing tool knows that a logging statement is likely to change, the developer can opt not to depend on the parts of the log that are generated by this logging statement. Instead, the developer can let the log processing tool depend on output generated by logging statements that are likely to remain unchanged. Depending on logging statements that remain unchanged will reduce the maintenance effort required for keeping the log processing tool consistent with ever-changing logs.

To decide whether a logging statement will change in the future, we must understand which factors play an important role during such a change. The following factors can influence whether a logging statement will change:

1) The contents of the logging statement
2) The location of the logging statement
3) The developer who added the logging statement to the source code

In this paper, we examine which of these factors can help to decide whether a logging statement will change. First, we present a preliminary study which was done to get a better understanding of the stability of logging statements in four

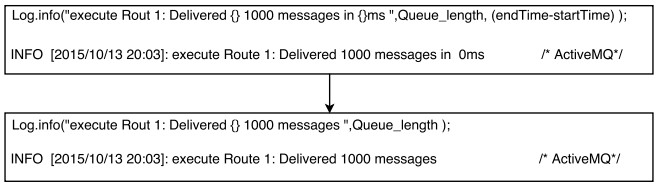

Fig. 2: Modification of a logging statement

open source projects. In this preliminary study, we find that 25%-45% of the logging statements are changed at least once during their lifespan in the studied applications, which shows that developers of log processing tools have to carefully select the logging statements they will depend on.

Second, we extract the factors that are important for explaining the stability of a logging statement using a random forest classifier. This classifier uses metrics that describe the three factors mentioned above to decide the likelihood of a log change. The most important observations in this paper are:

1) We can decide which logging statements will be changed in the future using a *random forest* classifier with a precision of 83%-91% and recall of 65%-85%.
2) Logging statements added by highly experienced developers and very new developers are less likely to be changed. We find that in three of the studied applications the top three developers add more than 60% of the logging statements and 70% of their logging statement remain untouched.
3) Logging statements added by developers who have less ownership on the file have higher likelihood of being changed. We find that 27%-67% of all log changes, are done on logging statement added by developers who own less than 20% of the file.
4) Large files (i.e., SLOC is 2x-3x the median) with lower log density, are more likely to have logging statement changes than well logged files.

From the above findings, maintainers of log processing tools can be more selective when importing logging statements to be used by their tools and decrease the maintenance effort.

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 projects. The goal of our study is to present a classifier for deciding whether a logging statement is likely to change in the future. This classifier can assist developers of log processing tools to decide on which logging statements they want their tool to depend. First, we perform a preliminary analysis, in which we examine how often logging statements change, to motivate our work. In this section, we present our rationale for selecting the projects that we studied and present the results of our preliminary analysis of the four studied projects.

A. Studied Projects

In this paper, we examine the logging statements in open source projects. We selected these projects based on the following three criteria:

- **Log usage** - The selected projects must make extensive use of logs in their source code
- **Project activity** - The projects must have a mature development history (i.e., more than 10,000 commits)
- **Technology used** - To simplify the implementation of our study, we opted to select only projects that are written in Java and are available on a Git repository

To select projects matching these criteria, we first selected all Java projects from the list of Apache Foundation Git repositories[1] that have more than 10,000 commits. Next, we counted the number of logging statements in all `*.java` files in a repository using the `grep` command in Listing 1.

```
grep -icR
"\(log\.*\)\.\(|info\|trace\|debug\|error\|warn\)(" .
| grep "\.java"
```

Listing 1: Counting logging statements

This command counts the occurrences in a file of an invocation of a logging library (e.g., `log` or `_logger`) followed by the specification of a log level. We sum the occurrences in all files of a project to get the total number of logging statements shown in Table I.

We select the four projects (ActiveMQ, Camel, Cloudstack and Liferay) with the highest number of logging statements for further analysis. ActiveMQ[2] is an open source message broker and integration patterns server. Camel[3] is an open source integration platform based on enterprise integration patterns. CloudStack[4] is an open source project designed to deploy and manage large networks of virtual machines. Liferay[5] is an open source platform for building websites and web portals. Table I presents an overview of the studied projects.

B. Data Extraction Approach

The data extraction approach from the four studied projects consists of three steps, which will be explained further in this section:

1) We clone the Git repository of each studied project in order to extract the change history of each file
2) We identify the logging statements in the repository
3) We track the changes that are made to each logging statement across commits

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

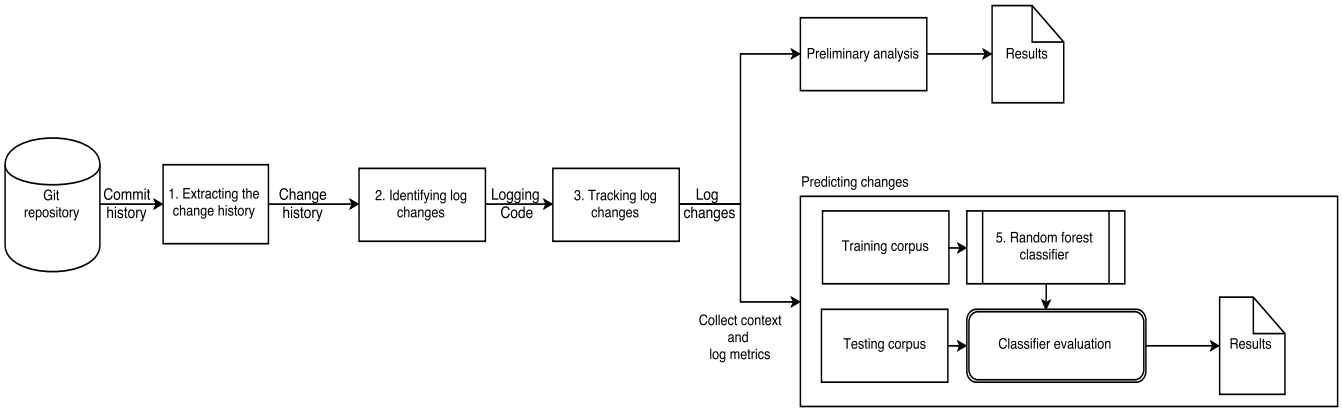

Fig. 3: Overview of the data extraction and empirical study approach

TABLE I: An overview of the studied projects (all metrics calculated using the latest HEAD of the repository)

	ActiveMQ	**Camel**	**CloudStack**	**Liferay**
Logging statements	5.1k	6.1k	9.6k	1.8k
Commits	11K	21K	29K	143K
Years in repository	8	8	4	4
Added lines of code	261k	505k	1.09M	3.9M
Deleted lines of code	114k	174k	750k	2.8M
Added logging statements	4.5k	5.1k	24k	10.4k
Deleted logging statements	2.3k	2.4k	17k	8.1k
Logging-related changes	1.8%	1.1%	2.3%	0.3%

We use R [12], to perform experiments and answer our preliminary analysis. Figure 3 shows a general overview of our approach and we detail below each of the aforementioned steps.

B.1. Extracting the change history of Java files: To examine the changes made to logging statements, we must first obtain a complete history of each Java file in the latest version of the main branch. We collect all the Java files in the four studied projects and we use their Git repositories to obtain all the changes that are made to the files. We use Git's *follow* option to track a file even when it is renamed or relocated. We include only the changes to logging statements that are made in the main branch as other logging statements are unlikely to affect log processing tools.

B.2. Identifying logging statements: From the extracted change history of each Java file, we identify all the logging statements. First, we manually examine the documentation of each studied project to identify the logging library used to generate the logs. We find that the studied projects use *Log4j* [13], *Slf4j*[6] and *logback*[7]. Using this information, we manually identify the common method invocations that invoke the logging library. For example, in ActiveMQ and Camel, a logging library is invoked by a method named *LOG* as shown below.

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

As a project can use multiple logging libraries throughout its lifetime, we use regular expressions to search for all the common log invocation patterns (i.e., *LOG, log, _logger, LOGGER, Log*). We identify every successful match of this regular expression that is followed by a log level (*info, trace, debug, error, warn*) as a logging statement.

B.3. Tracking changes to logging statements: After identifying all the logging statements, we track the changes made to these statements after their introduction. We extract the change information from the Git commits, which show a *diff* of added and removed code. To distinguish between a change in which a new logging statement is added and a change to an existing logging statement, we must track the changes made to a logging statement starting from the first project commit. Because there may be multiple changes to logging statements in a commit, we must decide to which existing logging statement a change maps.

We first collect all the logging statements in the initial project commit as the initial set of logging statements. Then, we analyze the next commit to find changes to logging statements until we reach the latest commit in the repository. To distinguish between added, deleted and changed logging statements and to map the change to an existing logging statement, we use the Levenshtein ratio [14].

[6]http://www.slf4j.org/
[7]http://logback.qos.ch/

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

Listing 2: Selecting the best matching logging statement

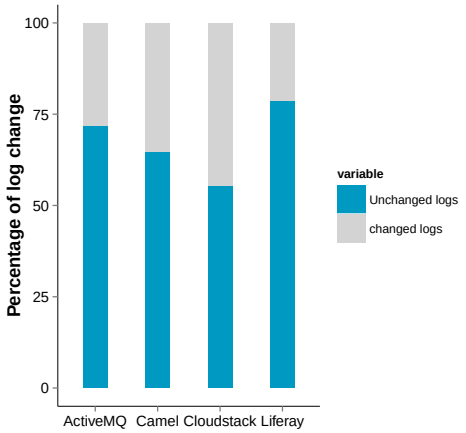

Fig. 4: Distribution of log changes in the studied projects

We use the Levenshtein ratio instead of string comparison, because Levenshtein ratio quantifies the difference between the strings on a continuous scale between 0 and 1 (the more similar the strings are, the closer the ratio approaches 1). This continuous scale is necessary to decide between multiple logging statements which can have a similar match to a change. Selecting the best matching logging statement is demonstrated by the example in Listing 2. In this example, there are two changes made to logging statements: one change and one addition.

To identify the change we calculate the Levenshtein ratio between each deleted and all the added logging statements and select the pair which has the highest Levenshtein ratio. This calculation is done iteratively to find all the changes within a commit. In our example, we find that the Levenshtein ratio between the deleted statement and statement *a1* is 0.86 and between the deleted statement and statement *a2* 0.76. Hence, we consider *a1* as a change. If there are no more deleted logging statements, *a2* is considered a newly added instead of a changed logging statement.

We extend the initial set of logging statements with every newly added logging statement. As we do not have change information for logging statements which are added near the end of the lifetime of the repository, we exclude these logging statements from our analysis. We find that in the studied projects, the maximum number of commits between the addition of a logging statement and its first change is 390, as shown in Table II. We exclude all logs added to the project 390 commits before the last commit of our analysis.

TABLE II: The number of commits before a newly added log is changed in the studied projects *[CP: BOXPLOTS]*

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

TABLE III: Time taken in days before for a new log into the application is changed

Project	Min	1st Qu	Median	Mean	3rd Qu	Max
ActiveMQ	1	1	7	86	99	680
Camel	1	1	17	127	145	1139
Cloudstack	1	1	1	46	1	1113
Liferay	1	1	2	34	2	945

C. Results

Developers change 25%-45% of the logs across our studied applications.

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

We find that 12%-40% of the logging statement changes, are due to the added logging statements which change. From table III, we observe that upto 75% of the added logging statement changes, are done within four months after the log is added. This means that any log processing tool, which utilizes a logging statement introduced in the previous releases is more susceptible to breakage, by changes to the added logging statement.

III. Building a Log Change Classifier

In our preliminary analysis, we find that 35%-50% of logs are changed in our subject applications. These log changes affect the log processing tools which run on the studied applications, forcing developers spend more time on maintenance of the log processing tools. By analyzing the factors which can increase the likelihood of a log change, developers of log processing tools can decrease the effort spent on maintenance.

A. Approach

In this section we construct a random forest classifier for explaining the likelihood of log changes in the studied applications. We then evaluate the performance of our random forest classifier and use it to understand which factors can increase the likelihood of a log change in the studied applications.

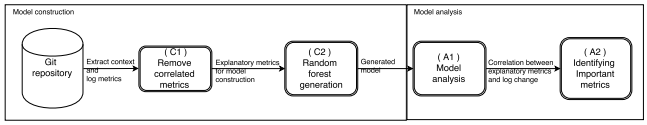

Fig. 5: Overview of random forest classifier construction (C), analysis (A) and flow of data in random forest generation

We use context and log metrics to train 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 defines each metric collected and the rationale behind our choice of each metric. We use the context metrics as it describes the conditions in which the was added into the application and log metrics provides information about the log added. These metrics also benefit log processing tool developers as they do not need domain knowledge about the application to understand these metrics.

We build random forest classifier [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 classifier, 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 classifier and steps (A1 and A2) provides an analysis of the results. We use the statistical tool R to model our data using the *RandomForest* package.

Step C1 - Removing Correlated Metrics

Correlation analysis is necessary to remove the highly correlated metrics from our dataset [18]. Correlated metrics can severely impact the calculation of importance in the random forest classifier, as small changes to one correlated metric can affect the values of the other correlated metrics, causing large changes on dependent class variable.

We use Spearman rank correlation [19] 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 [20] 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

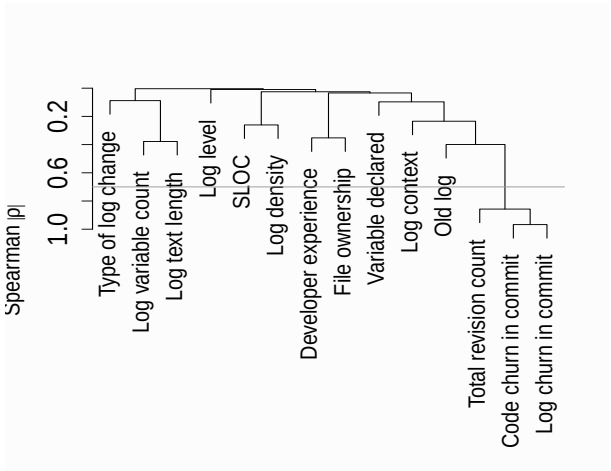

Fig. 6: Hierarchical clustering of variables according to Spearmans ρ in ActiveMQ

TABLE VI: 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)

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 [21] 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 a simpler metric 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. From a training set of m logs a random sample of *n* components is selected with replacement [21] and using the *randomForest* function from the *randomForest* package in R, a random forest model is generated.

Step A1 - Model validation

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 VI and are explained below.

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

TABLE IV: The investigated metrics in our clasifier

Dimension	Metrics	Values	Definition (d) – Rationale (r)
Context Metrics	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 often changed, have higher a likelihood of being changed [16]. Hence, the more commits to a file, the higher the likelihood of in that file 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 can be very different from logs added during bug fixes which have lesser code changes.
	File ownership	Numerical	d: The percentage of the file written by the developer who added 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 in that function. 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 may add more stable logs than a new developer who has little understanding of the application.
	Log context	Categorical	d: The block in which a log is added i.e., *if, if-else, try-catch, exception, throw, new function*. r: Logs used in logical branching and assertion checks, i.e., if-else blocks, may be very different from the logs in try-catch, exception blocks.
Log Metrics	Log addition	Boolean	d: Check if the log is added to the file after creation or it was added when file was created. r: New logs added to a file are more likely to be changed than log previously left untouched in previous commits.
	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 variable count	Numerical	d: Number of logged variables. r: Over 62% of log changes add new variables [16]. Hence, fewer variables in the initial log statement might result in addition of new variables later.

TABLE V: The investigated metrics in our clasifier

Dimension	Metrics	Values	Definition (d) – Rationale (r)
Log Metrics	Log density	Numerical	d: Ratio of the number of log lines to the source code lines in the file. r: Files which are well logged (i.e., higher log density) may not need additional logs and the logs are less likely to be changed.
	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 thought out than default level *info* logs.
	Log text count	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.
	Log churn in commit	Numerical	d: The number of logging statements changed in the commit. r: Logging statements can be added as part of specific change or part of bigger change.

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 correctly classify all the logs which will get changed in our dataset. Recall 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 [22].

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

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 [23]. Brier score explains how well the model performs compared to random guessing as explained in Equation 4, where P_t is the probability of log change, O_t is the actual log being changed or not. A perfect classifier will have a Brier score of 0, a perfect misfit classifier will have a Brier score of 1 (predicts probability of log change when log is not changed) and for random guessing Brier score reaches the value of 0.25. This means lower the Brier score value, the better our random forest classifier.

$$BS = (P_t - O_t)^2 \tag{4}$$

The performance measures described previously, may overestimate the performance of the classifier due to over fitting. To account for the overfitting in our models, we use the *optimism* measure, as used by prior research [21]. The *optimism* of the performance measures are calculated as follows:

1) From the original dataset with *m* records, we select a bootstrap sample with *n* records 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, less of an overfit the original classifier is.

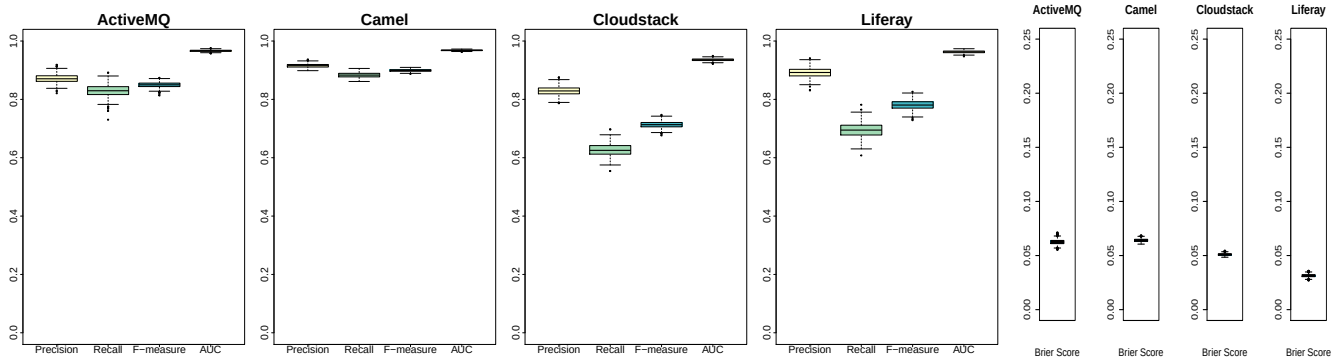

Fig. 7: The optimism reduced performance measures of the four projects

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 [24, 25]. 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.

Step A3 - Plotting the Important Metrics

To understand the effect each metric has in our random forest classifier, it is necessary to plot the predicted probabilities of log change against the metrics. By plotting the predicted probabilities of log change, we obtain a clearer picture of how the random forest classifier uses the important metrics to classify the data.

Using the *randomForest* package in R, we build a classifier as explained in C2, and we use the *predict* function in *R*, to calculate the probabilities of log change for the classifier. We plot each predicted probability against the value of the metric, to understand how changes in the metric values affect the probability of log change.

TABLE VIII: 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

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 7 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%).

B2. Important metrics for log stability

Developer experience is an important metric to explain the likelihood of log change. From Table VII, we see that developer experience is one of top four metrics, to help explain the likelihood of a log being changed in all the studied applications. Figure 8 shows the probabilities of a log being changed as developer experience increases, and it is interesting to note that in all the projects logs introduced by new developers have lower probability of being changed, when compared to more experienced developers. We also observe that as developers get more experience the probability of log change decreases in ActiveMQ, Camel and Liferay. This downward trend maybe be because in ActiveMQ, Camel and Liferay, the top three

TABLE VII: 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	0.246	1	Developer experience	0.272
2	Ownership of file	0.175	2	Ownership of file	0.151
3	Log density	0.163	3	Log level	0.138
4	Log variable count	0.101	4	SLOC	0.112
5	Log level	0.063	5	Log addition	0.090
6	Variable declared	0.048		Log density	0.088
7	Log context	0.069	6	Log variable count	0.063
8	Log text length	0.022	7	Log context	0.052
			9	Variable declared	0.051
	CloudStack			**Liferay**	
Rank	**Factors**	**Importance**	**Rank**	**Factors**	**Importance**
1	Log density	0.224	1	Log density	0.192
2	Ownership of file	0.215		Developer experience	0.195
3	SLOC	0.192	2	Ownership of file	0.190
4	Developer experience	0.182		SLOC	0.188
5	Log text length	0.120	3	Log variable count	0.162
6	Log variable count	0.115	4	Log level	0.148
7	Log level	0.102	5	Log context	0.091
8	Variable declared	0.092	6	Variable declared	0.080
9	Log context	0.061	9	Log text length	0.071

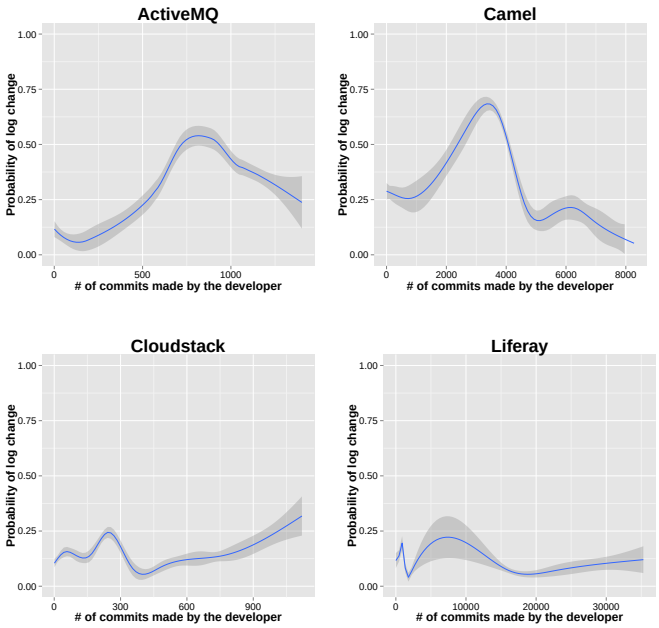

Fig. 8: Comparing experience of developers who add unchanged logs against logs changing more than once

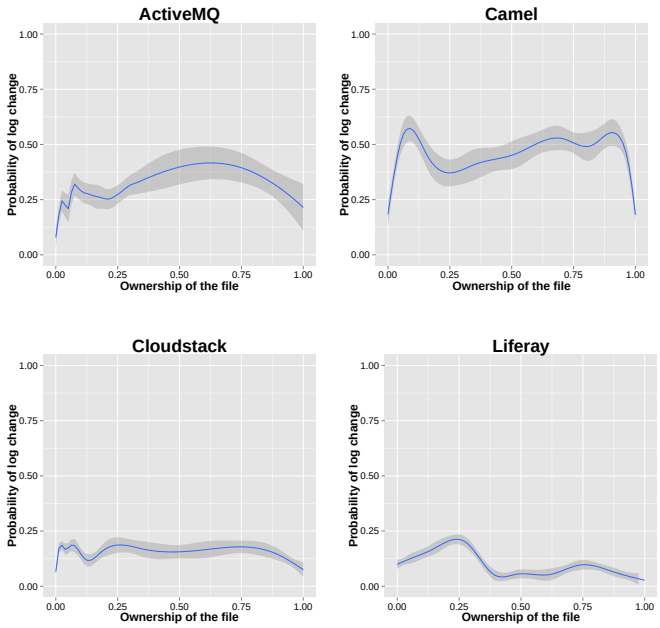

Fig. 9: Comparing file ownership of developers who add unchanged logs against logs changing more than once

developers are responsible for adding more than 50% of the logs as seen in Table VIII. We also find that only 30% of the logs added by these top developers ever change. These findings, suggest that developers of log processing tools should be more cautious when using a log written by developers who have some experience in the application rather than new developers.

Ownership of the file is an important metric to explain

the likelihood of log change. From Table VII, we see that ownership of the file is one of top four metrics after developer experience to help explain the likelihood of a log being changed in all the studied applications. From Figure 9, we observe that in all the applications that logs introduced by developers who own more than 75% of the file are less likely to be changed. We also observe that developers who own less than 20% of the file are responsible for 27%-67% of the log changes in the studied applications, which is seen as upward trend from 0 to 0.20 in Figure 9. These results suggest that developers of log processing tools should be more cautious when using a log written by developers who have contributed to less than 20% of the file in the studied applications.

Log density is an important metric in our studied applications to explain the likelihood of log change. From Table VII, we see observe that log density has the highest importance in Liferay and Cloudstack. We find that in these two applications, the logs which change are present in files with lower log 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 *[CP: FIXME (discuss relation)]*

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. [26] 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. [27, 28, 29, 30] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [27]. Based on the such events, they identified both functional anomalies [28] and performance degradations [29] 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 Logging statements

Prior research performs an empirical study on the characteristics of logging statements. Yuan et al. [16] study the logging characteristics in four open source systems. They find that over 33% of all changes to logging statements are after-thoughts and that logging statements are changed 1.8 times more often than regular code. Fu et al. [31] performed an empirical study on where developers put logging statements. They find that logging statements are used for assertion checks, return value checks, exceptions, logic-branching and observing key points. The results of the analysis were evaluated by professionals from the industry and an 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 logging statements are co-evolving with software systems. However, logging statements are often modified by developers without considering the needs of operators which even affects the log processing tools which run on top of the logs produced by these statements. 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 logging statements 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 logging statements 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 logging statement in the source code. Even though the studied applications leverage logging libraries to generate logs at run-time, there may still exist user-defined logs. By manually examining the source code, we believe that we extract most of the logging statements. Evaluation on the coverage of our extracted logging statements can address this threat.

In our model, we use the first change after the introduction of a logging statement only. While the first change is sufficient for deciding whether a logging statement will change, we need more information to decide how likely it is that a logging statement will change. In future work, we will extend our model to give more specific details about a future change.

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. For example, merging commits or rewriting the history of the repository (i.e., by *rebasing* the history) may affect our results.

Our analysis of the relationship between metrics that are important factors in predicting the stability of logging statements cannot claim causal effects, as we are investigating 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

Logging statements are snippets of code, added by developers to yield valuable information about the execution of an application. Logging statements generate their output in logs, which are used by a plethora of log processing tools to assist in software testing, performance monitoring and system state comprehension. These log processing tools are completely dependent on the logs and hence are affected when logging statements are changed.

In order to reduce the effort required for the maintenance of log processing tools, we examine changes made to logging statements in four open source projects. The goal of our work is to help developers of log processing tools decide whether a logging statement is likely to change in the future. We consider our work an important first step towards helping developers to build more robust log processing tools, as knowing whether a log will change in the future allows developers to let their log processing tools rely on logs generated by logging statements that are likely to remain unchanged. The highlights of our work are:

- We find that 12%-40% of logs are changed at-least once.
- Our random forest classifier for predicting whether a log will change achieves a precision of 83%-91% and recall of 65%-85%.
- We find that log density, SLOC, developer experience, file ownership are important predictors of log stability in the studied applications.

Our findings highlight that we can correctly classify the likelihood of a log change, when log is added into the application. The important metrics from the classifier help in determining the likelihood of a log change, and developers can use this knowledge to be more selective when importing logging statements into their processing tools.

References

[1] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *ACM SIGOPS '09: Proceedings of the 22nd Symposium on Operating Systems Principles*, New York, NY, USA, pp. 117–132.

[2] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection." in *USENIX Annual Technical Conference*, 2010.

[3] Q. F. J. L. Y. Wang and J. Li., "Execution anomaly detection in distributed systems through unstructured log analysis." in *ICDM 09: In Proceedings of 9th IEEE International Conference on Data Mining*.

[4] H. Malik, H. Hemmati, and A. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in *Software Engineering (ICSE), 2013 35th International Conference on*, May 2013, pp. 1012–1021.

[5] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems*, vol. 30, pp. 4:1–4:28, Feb. 2012.

[6] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 3–26, 2014.

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

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

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

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[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.