Understanding the Stability of Logs in Software

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

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

***Abstract*—Logs are system generated outputs, created from logging statements in the code. They help in understanding the system behavior, monitoring choke-points and in debugging. Prior research has demonstrated the importance of logs in operating, understanding and improving software systems which has lead to the development of log management applications and tools. However, logs may change over time due to debugging, improvement or addition of new features and these changes have to be communicated to operators and administrators. To understand the different factors which can affect log stability, in this paper we conduct a case study on four large software systems namely: Liferay, ActiveMQ, Camel and CloudStack. We first perform a manual analysis, where we find that 20-80% of the logs are changed in these software systems. We identify the four possible types of log changes namely 1) text modification, 2) variable modification, 3) log level change and 4) log relocation and categorize all the log changes. Next, we build models to predict if a log added to a file will change in the future. We use change and product metrics calculated at the time of introduction of the log, to build a random forest classifier. Our classifier can predict which logs will change in the future with precision of 89%-91% and recall of 71%-91%. We find that file ownership, developer experience, log density and SLOC are strong predictors of log stability in our models and can help identify which logs are more likely to change in the future.**

I. Introduction

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

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

The unified format of logs has lead to the development of many log processing tools such as *Splunk* [7], *Xpolog*, *Logstash* [8] and in-house tools. However, when logs are changed the log processing tools also have to be updated to reflect the changes to the log. For example in Figure 1

```
Log.info("execute Rout 1: Delivered {} 1000 messages in {}ms ",Queue_length, (endTime-startTime) );

INFO  [2015/10/13 20:03]: execute Route 1: Delivered 1000 messages in  0ms          /* ActimveMQ*/

Log.info("execute Rout 1: Delivered {} 1000 messages ",Queue_length );

INFO  [2015/10/13 20:03]: execute Route 1: Delivered 1000 messages                  /* ActimveMQ*/
```

Fig. 1: Modification of a logging statement

we see that developers remove the time take for completing an event. This can affect log processing tools which rely on that information for other activities. Tools like Salsa [9], log-enhancer [5], chukwa [10] are designed to diagnose as well as improve logging in software systems. These tools can be effected by such changes to logging statements and the data lost is not recoverable.

Research shows that only 40% of the logs at execution level stays the same across releases and the impact of 15-80% of the log changes can be minimized through robust analysis of the code changes [6]. These log changes can affect the log processing tools which heavily depend on them and maintenance cost will be high. In this paper, we track the changes made to logs across multiple releases in four studied open source systems. In order to get a better understanding of the log changes we focus our research on the following RQ's.

RQ1: How much do logs change over time and why do the changes occur?

Based on our quantitative analysis of the studied systems we identify three categories of change frequency in logs. If a log is changed more than four times we categorize it as *'Frequently Changed'*, if it has one to three changes, it is categorized as *'Changed'* and if there are no changes made we categorize it as *'Never Changed'*. We find that in our studied systems, 20-80% of the logs are changed at least once throughout their lifespan.

RQ2: Can metrics from code, log and developer dimension help in explaining the stability of logs?

We collect the product and change metrics from three dimensions namely code, log and developer information. Our

random forest achieves an accuracy of 89%-91% and recall of 71%-91%, when predicting which logs have higher likelihood of getting changed. We also identify significant metrics from each of the three dimensions, that affect the stability of logs. We find developer experience, source lines of code, file ownership, log density in the file are strong predictors in classifying if a log will change in the future.

The rest of this paper is organized as follows. Section II presents the methodology for gathering and extracting data for our study. Section III presents the case studies and the results to answer the two research questions. 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. Methodology

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

In this paper, we aim to find the unstable logs in a system for easier management of log processing tools. We cloned four projects from the Apache git repository, which had more than 10,000 commits. We also verify if the projects utilize a bug tracking system like 'JIRA' or 'Bugzilla' because, this helps to tag commits to specific development activities (i.e., bug fix, improvement, new-features). We use the 'grep' command to recursively search for all log lines within '.java' files in the latest release of each project in the cloned repositories. The four open source projects are: Liferay, Camel, ActiveMQ and CloudStack. All studied systems have extensive system logs and Table I presents an overview of the systems.

TABLE I: An overview of all studied systems

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

Liferay[1]: Liferay is a free and open source enterprise project written in Java. It provides platform features which are used in development of websites and portals. It also has an extensive issue tracking system in JIRA which helps in categorizing the commits as bug fixes, improvements, . We study the releases from 6.1.0 to 7.0.0-m3 which cover more than three years of development from 2010 to 2014

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

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

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

Figure 2 shows a general overview of our approach, which consists of four steps: (1) We mine the git repository of each studied system to extract all commits made for each file.(2) We identify the log changes in the extracted files. (3) We track the changes made to each log across the commits. (4) We extract the JIRA issues for the commits and collect the developer metrics when there is a log change in the commit. We use a statistical tool R [11], to perform experiments on the data to answer our research questions.

A. Study Approach

In the reminder of this section we describe the each of these steps.

1) Extracting Code Evolution: In order to find the stability of logs we have to identify all the 'Java' files in our studied systems. To achieve this, we clone the *master* branch of the git repository of each studied system locally. We use the 'find' command to recursively find all the files which end with pattern '*.java'. To remove the *Java Test* files, we use 'grep' command to filter all files which have 'test' or 'Test' in their pathname.

After collecting all the Java files from the four studied systems, we use their respective git repositories to obtain all the changes made to the file within the time-frame discussed above. We use the 'follow' option to track the file even when they are renamed or relocated within our studied systems. We also flatten all the changes made to files to include the changes made in different branches but exclude the final merging commit. Using this approach, we obtain a complete history of each *Java* file.

2) Identifying Log Changes: From the extracted code evolution data for each *Java* file, we identify all the log changes made in the commits. To identify the log statements in the source code, we manually sample some commits from each studied system and identify the logging library used to generate the logs. We find that the studied systems use *Log4j*[1] [12] 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);
```

[1]http://www.liferay.com/
[2]http://camel.apache.org/
[3]http://activemq.apache.org/
[4]https://cloudstack.apache.org/
[1]http://logging.apache.org/log4j/2.x/
[2]http://www.slf4j.org/
[3]http://logback.qos.ch/

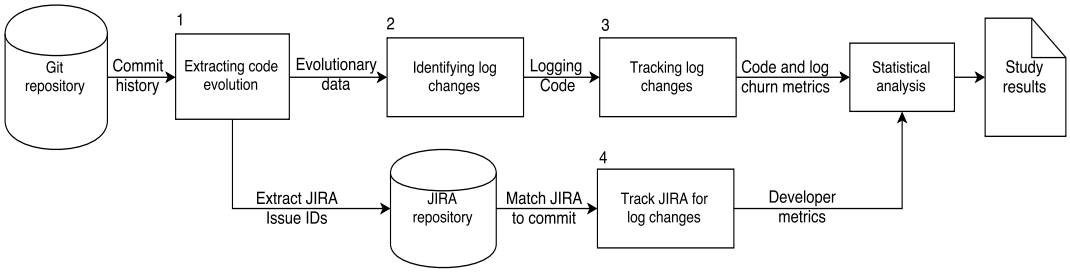

Fig. 2: Overview of the data extraction and case study approach

In CloudStack it is usually done through '_logger' as follows.

```
_logger.warn("Timed out: " + buildCommand-
Line(command));
```

As projects 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 count every such invocation of a logging library followed by a logging level i.e., ('info', 'trace', 'debug', 'error', 'warn') as a log.

3) Tracking Log Changes: After identifying all the log changes made to a file across multiple commits, we track each log individually to find out whether it has changed in subsequent revisions. We first collect all the logs present in a file at the first commit, which form the initial set of logs for the file. Every subsequent commit which has changes to a log appears as an added and deleted log in *git*. To track these log changes made, we used the Levenshtein ratio [13].

To calculate Levenshtein ratio for a pair of added and deleted logs, we first remove the logging method (e.g., LOG) and compare the remaining text. We calculate the levenshtein ratio for each deleted log against all the added logs and pick the pair which has the highest levenshtein ratio as a modification. We then remove this pair from the set and compare the rest of logs recursively to find all the modifications.

For example in the logs shown below, we remove the logging method (i.e., LOG) and 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.

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

If the added log does not match any log in our initial set, it is considered as a new log addition and is added to the initial set for tracking in future commits. From this we can track how many times a log is changed and how many commits are made between the changes.

4) Match JIRA to Log Changes: Using the commit data, we also track the JIRA issues to extract the developer metrics such as developer experience, number of developers involved, issue priority. To achieve this, we extract the JIRA issue IDs from commit messages and use the JIRA repository to extract the JIRA issue. The JIRA issue contains information about the issue such as, type (i.e., bug, improvement, new-feature, task), priority, resolution time and developer information such as number of comments and number of developers involved in the JIRA discussion. We use this information along with code and log churn metrics for answering our research questions.

III. Study Results

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

RQ1: How much do logs change over time and why do the changes occur?

Motivation

Research has shown that logs evolve along with the code [14]. When logs are changed, the log processing tools which are dependent on them also have to get updated. This results in costly maintenance effort. To understand the cost, we have to understand how frequent changes to logs are. Hence, we explore the frequency of changes to logs in our studied systems and why they are changed.

Approach

To find the frequency of log changes, we conduct a quantitative analysis on our studied systems. We use the tracked log data for each studied system as explained in Section II. From each project, we select a random sample with 95% confidence interval. We follow the same iterative process as in prior research [15] to find how frequently logs change in our studied systems.

When logs are changed, they can be changed in five possible ways namely:

1) **Log relocation:** The log is kept intact but moved to a different location in the file because of context changes (code around the log is changed).
2) **Text change:** The text (i.e., static content) of log is changed.
3) **Variable change:** One or more variables in the log are changed (added, deleted or modified).
4) **Change of log level:** The verbosity level of a log is changed.
5) **Text and variable change:** Both text and variables in the logs are changed. This is generally done when developers provide more context information, i.e, text and add/modify the relevant variables in a log.

To automate the process of categorizing log changes into these categories, we first remove the logging method (i.e, LOG) and the log level (i.e, info) from the logs. We then compute the *Levenshtein ratio* between each term within the parentheses. In the example below we find that '+ Integer.toString(listenPort)' has *Levenshtein ratio* of 1, implying they are identical and the *Levenshtein ratio* between 'starting HBase HsHA Thrift server on' and 'starting HBase' is 0.56. This suggests there is some similarity between the two strings and the variable is constant which implies its a text change. (Figure 4 highlights the process of categorizing the log changes.

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

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

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

Results

We find that developers change 20-80% of the logs across our studied systems as shown in Figure 3. Based on frequency of changes, we categorize logs into 3 categories namely: a) Frequently Changed, b) Changed and c) Never Changed as shown in Table II. If a log is changed more than three times it is categorized as 'Frequently Changed'. If it is changed fewer 1 to 3 times it is categorized under 'changed' and if there is no changes made it is categorized under 'Never Changed'. We select three as the threshold as we observe that majority of logs only have 1 to 2 changes as seen in Figure 3. We see that a majority of logs never change in Life Ray and the logs in ActiveMQ and CloudStack are changed at-least once. This may be because Camel and Liferay have fewer logs per source code (i.e., lower log density) when compared to ActiveMQ and CloudStack as seen in Table I.

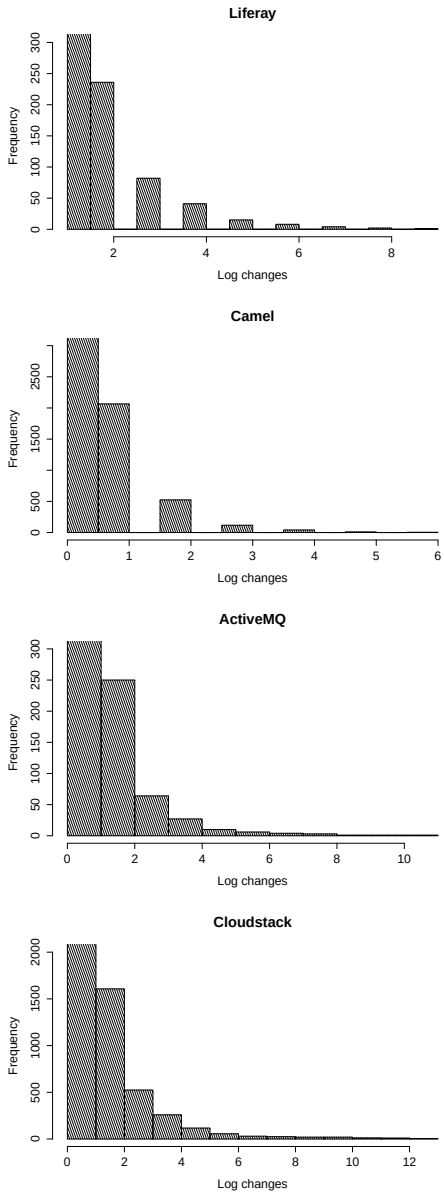

Fig. 3: Frequency of log changes in the studied systems.

We find that log relocation occurs more often than other types of log changes. We find that in all the studied systems log relocation occurs more than 49%. As log relocations have no changes to the text or variables in logs, their impact on log processing tools is limited. Hence we exclude log relocation changes from our datasets and do not consider relocation changes for the classifier.

TABLE II: Distribution of log changes in different projects

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

TABLE III: Distribution of log changes in different projects

Projects	Life Ray	Cloud Stack	Active-MQ	Camel
Log relocation (%)	73.5	78.85	63.27	49.72
Log text change (%)	20.11	6.37	5.97	7.59
Log variable change (%)	3.10	7.21	6.91	8.29
Change of log level (%)	0.87	1.15	1.76	3.79
Text and variable change (%)	2.33	6.39	22.0	30.59

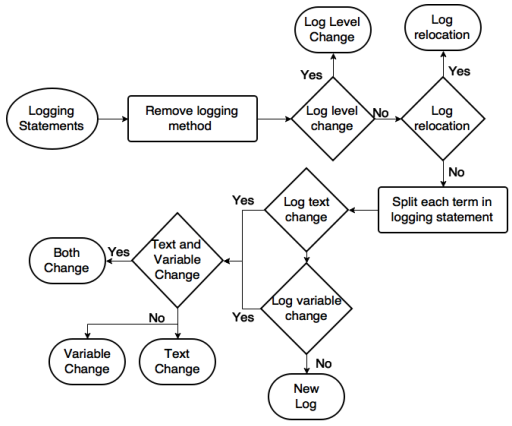

Fig. 4: Flowchart to categorize the different types of log changes that occur

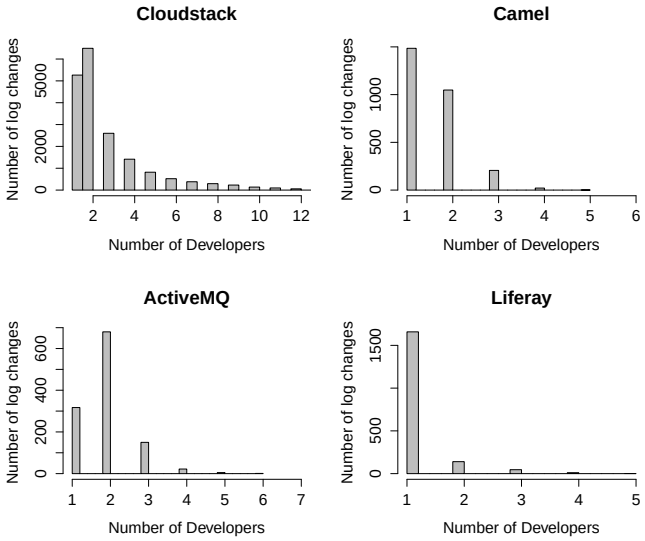

Fig. 6: Distribution of the number of developers responsible for changing a log

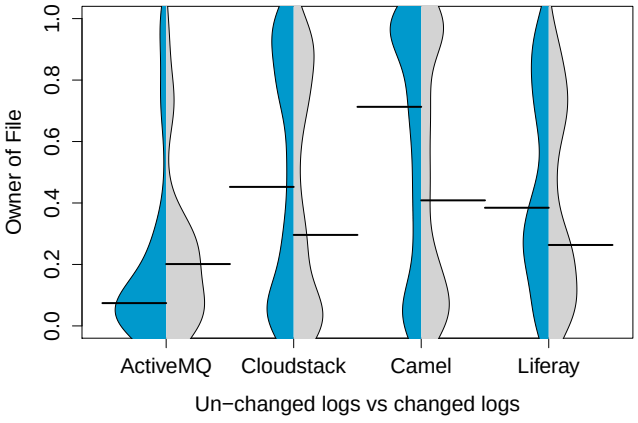

Fig. 5: Distribution of logs which are changed vs un-changed against the ownership of the developer introducing the log

We see that logs are changed by developers who have little ownership over the file. From Figure 5 we see that in three of the studied systems logs are changed by developers who have lesser ownership on files than the developers who introduce the log. We also find that in two of the studied systems majority of logs are changed by two developers as seen in Figure 6. This results suggest that logs are readily changed by developers who access the file and do not have strong ownership characteristics.

45% of the logs are changed atleast once in three of our studied systems. We find that over 49% of these log changes are due to log relocation. The changes are made to the text (i.e., static content), variable and log level. This suggests that developers change logs extensively throughout the lifecyle of a software and developers might dedicate significant amount of time to update and maintain logs

RQ2: Can metrics from code, log and developer dimension help in explaining the stability of logs?

Motivation

In RQ1, we find that 45% of logs are changed at-least once in three of out studied systems. This affects the log processing

tools which run on these studied systems, making developers spend more time on maintenance of those tools. Hence, there is a need to identify these non-stable logs to simplify the job of developers. It can also benefit log processing tool developers to develop more robust applications making them more stable.

Approach

To identify the stability of logs we use product and change metrics collected from code, log and developer dimensions. Change metrics measure the changes to the system at the time of introduction of the log and product metrics measure the source code of the system. We use the git repository to extract the change metrics and product metrics for the studied systems. Table IV lists all the metrics we collect for each dimension. We define each metric and the rationale behind the choice of each metric. We use the product and process metrics because this data can be extracted from control version systems (CVS) easily by developers. It also benefits log processing tool developers as they do not need domain knowledge about the system to understand these metrics.

***Model construction**:* We build random forest models to explain the stability of logs in our studied systems. 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 [23]. In our model the product and change metrics are the explanatory variables and the dependent class variable is if the log is changed or not (i.e., 'False' for not changed and 'True' for changed).

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

(C1 - Correlation analysis)

Correlation analysis is necessary to remove the highly correlated metrics from our dataset. 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 [24] to remove these correlated metrics from 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 [25] because Spearman is resilient to data that is not normally distributed. We use the function 'varclus' in R to perform the correlation analysis.

Figure 8 shows the hierarchically clustered Spearman ρ values in 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 exclude one metric from the sub-hierarchies which have correlation $|\rho| > 0.7$. The gray line indicates our cutoff value ($|\rho|$ = 0.7). We use cutoff value of ($|\rho|$ = 0.7) as used by prior research [26] to remove the correlated metrics before building our model.

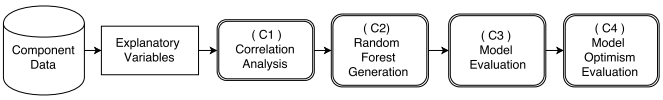

Fig. 7: Overview of Model construction(C) and Evaluation

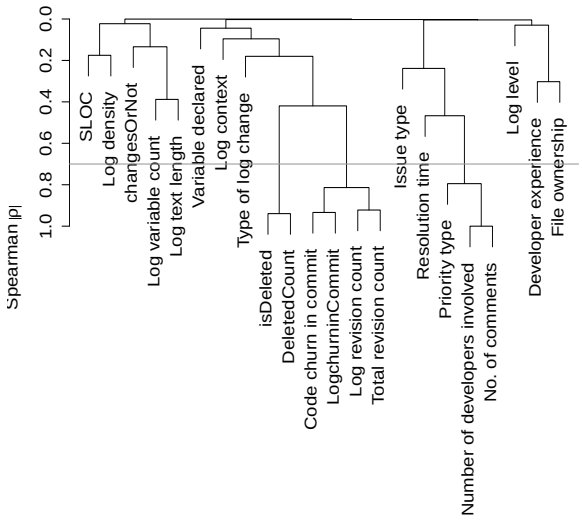

Fig. 8: Hierarchical clustering of variables according to Spearmans ρ in ActiveMQ

(C2 - Random forest generation)

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

Given a dataset of *n* logs for training, $D = \{(X_1Y_1, ..., (X_nY_n)\}$ where $X_i, i = 1...n$, is a vector of descriptors (i.e, explanatory metrics) and Y_i is the flag which indicates whether a log is changed or not, the algorithm follows these steps and is shown in Figure 9.

1) From the training set of *n* logs, a random sample of *n* components is selected with replacement (i.e., bootstrap sample) [26].
   We use the 'boot' function from the 'boot' package in R to generate bootstrap samples. The boot function generates a set of random indices, with replacement from the integers 1:*n*.
2) From the sampled selection, a tree is grown with one difference compared to normal decision trees. The random forest is split at each node using the best among subset of predictors, rather than all. This strategy helps to make

TABLE IV: Taxonomy of metrics considered for model construction

Dimension	Metrics	Values	Definition (d) – Rationale (r)
Change Metrics	Log revision count	Numeric	d: The number of prior commits which had log changes.
			r: This helps to identify if the file is prone to log changes.
	Old Log	Boolean (0 -1)	d: Check if the log is added to the file after creation or it wad added when file was created.
			r: This helps to identify if the logs added into file after creation are changed more than logs added at creation of file.
	Is deleted	Boolean (0 -1)	d: Check if the log is deleted from the file
			r: Identify the logs which get deleted in the file after they are added
	Deleted count	Numerical	d: Number of commits from the time a log is introduced into the file till it is deleted (removed) from the file.
			r: Find out how long it takes before a log is removed from the file.
	New File	Boolean (0 -1)	d: Check if the log is added in a new file (i.e., newly committed)
			r: This helps to identify which logs where added later in subsequent commits.
	Total revision count	Numeric	d: Total number of commits made to the file before the log is added. This value is 0 for logs added in the initial commit but not for logs added overtime.
			r: This helps to find out if the file is changed heavily which can result in log changes [16].
	Code churn in commit	Numeric	d: The code churn of the commit in which a log is added.
			r: Log changes are correlated to code churn in files [12].
	Variables declared	Numeric	d: The number of variables which are declared before the log statement. (we limit to 20 lines before log statement).
			r: When new variables are declared, developers may log the new variables to obtain more information [16].
	SLOC	Numeric	d: The number of lines of code in the file.
			r: Large files have more functionality and are more prone to changes [17] and more log changes [16], [12].
Product Metrics	Log context	Categorical	d: Identify the block in which a log is added. (i.e., ‘if’, ‘if-=else’, ‘try-catch’, ‘exception’, ‘throw’, ‘new function’).
			r: Prior research finds that logs are mostly used in assertion checks, logical branching, return value checking, assertion checking [18].
	Log change type	Boolean (0 -1)	d: Check the type of log change the log has undergone before i.e., relocation, text-variable change, level change.
			r: This helps in removing the relocation changes from the dataset and check if logs that have changed once before undergo similar changes again.
	Log variable count	Numerical	d: Number of variables logged.
			r: Over 62% of logs add new variables [16]. Hence fewer variables in the initial log statement might result in addition later.
	Log density	Numerical	d: Ratio of number of log lines to the source code lines in the file.
			r: Research has found that there is on average one log line per 30 lines of code [16]. If it is less it suggests there may be additions in later commits.
	Log level	Categorical	d: Identify the log level (verbosity) of the added log (i.e., ‘info’, ‘error’, ‘warn’, ‘debug’, ‘trace’ and ‘trace’).
			r: Developers spend significant amount of time in adjusting the verbosity of logs [16].
	Log text length	Numerical	d: Number of text phrases logged (i.e., we count all text present between a pair of quotes as one phrase).
			r: Over 45% of logs have modifications to static context [16]. Logs with fewer phrases might be subject to changes later to provide better explanation.
	Resolution time	Numerical	d: The time it takes for the issue to get fixed. It is defined as the time it takes since an issue is opened until closed.
			r: More resolution time might suggest a more complex fix with more log churn.
	Number of developers involved	Numerical	d: Total number of unique developers who comment on the issue report on JIRA.
			r: Components with many unique authors likely lack strong ownership, which in turn may lead to more defects [19] and change logs [12].
	Number of comments	Numericl	d: Total number of discussion posts on the issue.
			r: Number of comments is correlated to the resolution time of issue reports [20]. More comments may also indicate the issue is more complex resulting in more code churn and log changes.
	Developer experience	Numericl	d: The number of commits the developer has made prior to this commit.
			r: Research has shown that experienced developers might take up more complex issues [21] and therefore may leverage logs more [12].
	Issue type	Categorical	d: Identify the type of issue, i.e., ‘Bug’, ‘Improvement’, ‘Task’, ‘New Feature’, ‘Sub-Task’, ‘Test’.
			r: Some issue types might have higher code churn than others (example: Bug and New features might have more code churn when compared to Sub-Tasks) and are committed faster.
	Priority type	Categorical	d: Identify the priority of the issue i.e., ‘Critical’, ‘Blocker’, ‘Major’, ‘Minor’ and ‘Trivial’
			r: Research has shown that priority of issue affects resolution time of bug fixes [22]. A Higher priority indicates the issue will be fixed faster with log changes.
	File ownership	Numerical (0-1)	d: Identify the percentage of the file written by developer introducing the log
			r: The owner of the file is more likely to introduce stable logs than developers who have not edited the file before.

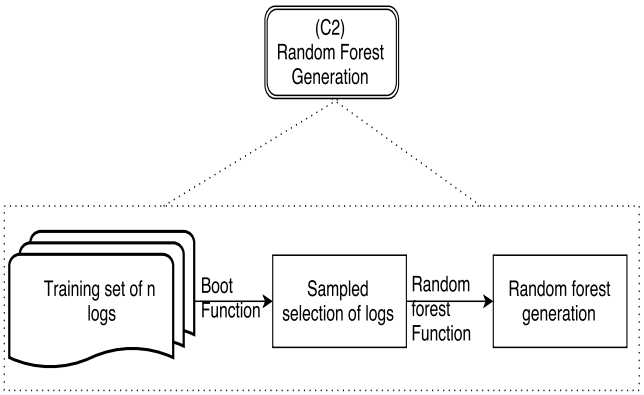

Fig. 9: Overview of random forest generation in C2

		Predicted	
		Log has changed	Log has not-changed
Actual	Log has changed	TP (True Positive)	FN (False Negative)
	Log has not-Changed	FP (False Positive)	TN (True Negative)

TABLE V: Confusion Matrix

the random forests more robust against over-fitting [27]. We use the 'randomForest' function from the 'randomForest' package in R to generate the random forest model.

3) The above steps are repeated until *M* such models are grown.
4) Predict new data by aggregating the prediction of the M models generated.

(C3 - Model evaluation)

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

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

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

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

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

F-Measure is the harmonic mean of precision and recall, combining the inversely related measure into a single descriptive statistic as shown in Equation 3 [28].

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

Brier Score (BS) is a measure of the accuracy of the predictions in our model. It is the mean squared error of the probability forecasts [29]. The most common formulation of Brier Score is shown in Equation 4, where f_t is the probability that was predicted, o_t is the actual outcome of the event at the instance t (0 if log is not changed and 1 if it is changed), and N is the number of forecasting instances.

If the predicted value is 70% and the log is changed, the Brier Score is 0.09 (lower the Brier score, the more likely the event will occur). It reaches 0.25 for random guessing (i.e., predicted value is 50%) and it is 0 when predicted value is 100% and the log is not changed.

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

The model performance measure, described previously only provide insight into how the random forest models fit the observed dataset, but it may overestimate the performance of the model if it is over-fit. We take performance overestimation (i.e., *optimism*) into account by by subtracting the bootstrap averaged optimism [26] from the initial performance measure. The *optimism* of the performance measures are calculated as follows:

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

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

(C4 - Importance of each metric in relation to log stability) 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_ith metric of which we want to find importance for, are randomly permuted in the test dataset and the accuracy of the model is recomputed. The decrease in accuracy as a result of this permutation is averaged over all

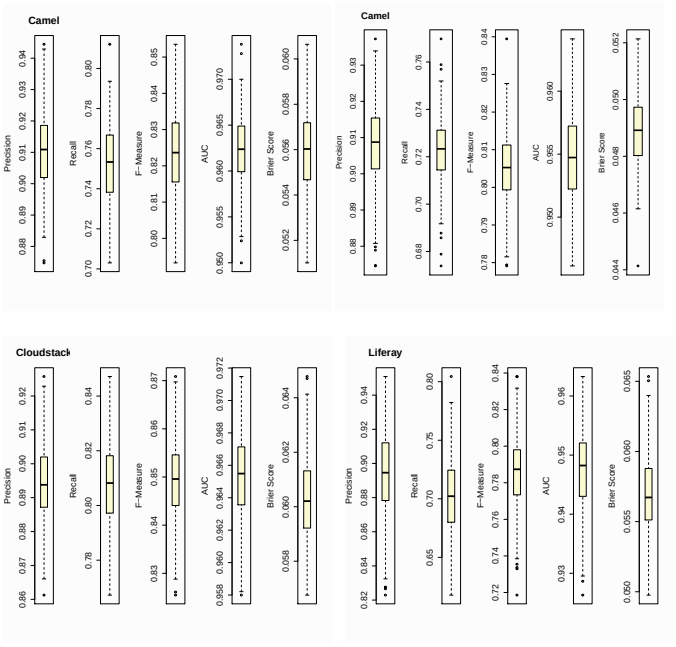

Fig. 10: The optimism reduced performance measures of the four projects

trees, and is used as a measure of the importance of metric X_ith in the random forest.

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

As we obtain 1000 data sets for each metric because of bootstrapping process, we use the **Scott-Knott** clustering to group the metric based on their means [30], [31]. This is done to group metrics which are strong predictors of likelihood of log change. The SK algorithm uses the hierarchical clustering approach to divide the metrics and uses the likelihood ratio test to judge the difference between the groups. This assures the means of metrics within a group not to be statistically significantly different. We use the 'SK' function in the 'ScottKnott' package of R and set significance threshold of 0.05 to cluster the metrics into different groups.

Results

The random forest classifier achieves a precision of 0.89-0.91 and recall of 0.71-0.91 for our studied systems. Figure 10 shows the optimism-reduced values of *precision*, *recall*, *F-measure* and *Brier score* for each project. We find the recall for random forest classier in Liferay is not as high as the other projects. This may be because Liferay has the lowest percentage of logs which are changed at 20%, compared to 45-70% in the other projects. Because of the lower percentage of logs which are changed, the random forest classifier has fewer nodes (trees) and likelihood of false negatives is higher.

The random forest classifier outperforms random guessing. The classifier achieves Brier scores between 0.04 and 0.07 across all projects. As the Brier score measure the total difference between the event and the forecast probability of the event, a perfect classifier will have Brier score of 0 and perfect misfit classifier will have Brier score of 1 (predicts probability of log change when log not changed). This means lower the Brier score value, the better our random forest classifier.

***Important metrics for log stability**:* From Table VI we see that 'file ownership', 'SLOC', 'log density', and 'developer experience' are common across all the projects in our studied systems.

We find that *log density* and *log variable count* are important metrics in our studied systems as shown in Table VI. We find that log density has negative correlation in all the studied systems. This suggests that when source code is well logged i.e., more logs per lines of code, the logs may communicate the necessary information making them more stable. Whereas, in files with fewer logs per lines of code, the logs have to be changed more to convey the necessary information. We also find *log variable count* has a positive correlation, implying that more text in a log results in a higher likelihood that a log will be changed. This may be because there are inconsistencies between logs and the actual needed information intended as shown by prior research [16] and developers have to update logs to resolve the inconsistencies.

We find that *SLOC*(source lines of code), *Variable declared* are strong predictors of log change across all projects. *SLOC* has positive correlation suggesting that logs in larger files have higher likelihood of getting changed. We find that *Variable declared* has positive correlation in three of the studied systems, which suggests that when developers add new variables in the commit there is a higher likelihood of log change as they may add or modify the log to output the new data.

We find that *file ownership* is a strong predict of log change and has positive correlation in three of the studied systems. This suggests that logs introduced by developers who have little ownership are more unstable and have to be changed. This is seen RQ1 Figure 5, where in in three of the studied systems, the logs which change are introduced by developers who have lower ownership of a file.

We find that *developer experience* is also a strong predictor of log change in all our models but the correlation is split within the projects. We find that in Cloudstack and Camel, developer experience has negative correlation on log change, where as in ActiveMQ and Liferay developer experience has positive correlation. The positive correlation may be because of strong code ownership seen ActiveMQ and liferay where two developer are responsible for over 50% of the total commits within the projects.

Our Random Forest classifier achieves a precision of0.89-0.91 and recall of 0.71-0.91 across all studied systems. We find file ownership, SLOC, developer experience and log density to be strong predictors of log change in our studied systems.

TABLE VI: The important values of the metrics (top 8), divided into homogeneous groups by Scott-Knott clustering. The '+' sign signify positive correlation of the metric on log stability i.e., increase in metric increases the likelihood of log change and vice-versa.

	Active MQ			**CloudStack**	
Rank	**Factors**	**Importance**	**Rank**	**Factors**	**Importance**
1	SLOC	0.174 +	1	Code churn in commit	0.138 +
2	File ownership	0.162 +	2	File ownership	0.136 +
	Log variable count	0.161 +		Developer experience	0.135 -
3	Developer experience	0.140 +	3	Log density	0.122 -
4	Log density	0.120 -	4	SLOC	0.122 +
5	Variable Declared	0.090 -	5	Log variable count	0.113 +
6	Deleted count	0.082 -	6	Log text length	0.102 +
7	Log level	0.078 -	7	Variable Declared	0.076 +
8	Log churn in commit	0.060 +	8	Type of log change	0.063 +
	Life Ray			**Camel**	
Rank	**Factors**	**Importance**	**Rank**	**Factors**	**Importance**
1	Log density	0.152 -	1	File ownership	0.207 +
2	File ownership	0.137 -	2	Log density	0.175 -
	Developer experience	0.137 +		Developer experience	0.174 -
3	SLOC	0.127 +	3	SLOC	0.168 +
4	Issue Id	0.096 -	4	Log variable count	0.124 +
5	Variable Declared	0.093 +	5	Log level	0.116 -
	Log variable count	0.091 +	6	Elapsed time	0.109 -
6	Log level	0.081 -		Issue type	0.108 +
7	Elapsed time	0.072 +		Variable Declared	0.108 +

IV. Related Work

In this section, we present prior research in which log behavior in software systems 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 (FSA) 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*. [32] 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.* [33], [34], [35], [36] proposed log analysis approaches to assist in automatically verifying results from load tests. Their log analysis approaches first automatically abstract logs into system events [33]. Based on the such events, they identified both functional anomalies [34] and performance degradations [35] in load test results. In addition, they proposed an approach that leverage logs to reduce the load test that are performed in user environment [36]. Shang *et al.* [14] propose an approach to leverage logs in verifying the deployment of Big Data Analytics applications. Their approach analyzes logs in order to find differences between running in a small testing environment and a large field environment.

Logs are also leveraged in debugging software systems. Jiang *et al*. [37] study the leverage of logs in troubleshooting issues from storage systems. They find that logs assist in a faster resolution of issues in storage systems. Beschastnikh *et al*. [38] designed automated tools that infers execution models from logs. These models can be used by developers to understand the behaviors of concurrent systems. Moreover, the models also assist in verifying the correctness of the system and fixing bugs.

The extensive prior research of log analysis shows that logs are leveraged for different purposes and changes to logs can affect performance of log analysis tools. This motivates our paper to study how much logs are changed by developers. As a first step, we study how many logs are stable and how many undergo changes across commits. Our findings show that more 20-80% of the logs are changed at-least once and 3-11% of the logs are changed frequently.

B. Empirical Studies on Logs

Prior research performs an empirical study on the characteristics of logs. Yuan *et al*. [16] studies the logging characteristics in four open source systems. They find that over 33% of all log changes are after-thoughts and logs are changed 1.8 times more 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-score of over 95% was achieved.

Tan *et al*. [9] 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] shows that logs need to be improved by providing additional . Their tool named *Log Enhancer* can automatically provide additional control and data flow parameters into the logs thereby improving the logs. While these works focus more on enhancing existing logs in the system, our paper focuses more on informing developers which logs are more likely to get changed and identifying which factors explain the change of logs.

Research also shows that logs are 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 [12], [39]. 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 [40]. Furthermore, Shang *et al.* [15] 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 effect both. Our work tries to tackle this problem by identifying logs which are more likely to change in the early stages so developers and system operators so they can be made more stable.

V. Threats to Validity

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

External Validity

Our case study is performed on Liferay, ActiveMQ, Camel and CloudStack. Though these studied systems have years of history and large user bases, these systems are all Java based platform softwares. Software in other domains may not rely on logs and may not use log processing tools. Our projects are all open source and we do not verify the results on any commercial platform systems. More case studies on other domains and commercial platforms, with other programming languages are needed to see whether our findings can be generalized.

Construct Validity

Our heuristics to extract logging source code may not be able to extract every log in the source code. Even though the studied systems-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 keywords to identify bug fixing commits when the JIRA issue ID is not included in the commit messages. Although such keywords are used extensively in prior research [12], we may still fail to identify bug fixing commits or branching and merging commits.

We use Levenshtein ratio and choose a threshold to identify modifications to logs. However, such 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.

Our models are incomplete, i.e., we have not measured all potential dimensions that impact logging stability like file and log ownership metrics, code complexity measures. However, we feel that our metrics are good and additional new dimensions can compliment our dimension to improve the explanation.

Internal Validity

Our study is based on the data obtained from git and JIRA for all the studied systems. The quality of the data contained in the repositories can impact the internal validity of our study.

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

VI. Conclusion

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

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

- We find that 20-80 % of logs are changed at-least once.
- Our random forest classifier for predicting whether a log will change achieves a precision of 89%-91% and recall of 71%-91%.
- We find that logs introduced by developers who have little ownership are more unstable and are more likely to be changed.
- We find that log density has negative correlation in all projects which suggests that when source code is well logged i.e., more logs per lines of code, the logs convey the needed information and are more stable.

Our findings highlight that we can predict which logs have a higher likelihood of getting changed when they are introduced. This information can be used by system administrators and practitioners, to identify logs which have a higher chance of

affecting the log processing tools and prevent failure of these tools.

REFERENCES

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

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

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *ICSE'13: Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 402–411.

[15] 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.

[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] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 192–201.

[20] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 2010, pp. 52–56.

[21] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 491–500.

[22] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. ACM, 2011, p. 11.

[23] 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

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

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

[26] 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.

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

[28] 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.

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

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

[31] 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.

[32] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *ASPLOS '10: Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2010, pp. 143–154.

[33] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *Journal of Software Maintenance Evolution*, vol. 20, no. 4, pp. 249–267, Jul. 2008.

[34] Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *ICSM '08: Proceedings of theIEEE International Conference on Software Maintenance*, Sept 2008, pp. 307–316.

[35] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *ICSM '09: Proceedings of the IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 125–134.

[36] Z. M. Jiang, A. Avritzer, E. Shihab, A. E. Hassan, and P. Flora, "An industrial case study on speeding up user acceptance testing by mining execution logs," in *SSIRI '10: Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement*. Washington, DC, USA: IEEE Computer Society, pp. 131–140, 2010.

[37] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou, "Understanding customer problem troubleshooting from storage system logs," in *FAST '09: Proceedings of the 7th Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2009, pp. 43–56.

[38] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 267–277.

[39] 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.

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