# 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*—Logging statements produce logs that assist in understanding system behavior, monitoring choke-points and debugging. Prior research 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 and processing tools. However, logs are often unstable, i.e., the logging statements that generate logs are often changed without the consideration of other stakeholders, causing misleading results and failures of log processing 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 open source applications namely: Liferay, ActiveMQ, Camel and CloudStack. We find that 20-45% of the logging statements in our studied applications change throughout their lifetime. The median number of days between the introduction of a logging statement and the first change to that statement is between 1 and 17 in our studied applications. These numbers show that in order to reduce maintenance effort, developers of log processing tools must be careful when selecting the logging statements on which they will let their tools depend.**

**In this paper, we make an important first step towards assisting developers of log processing tools in determining whether a logging statement is likely to remain unchanged in the future. Using random forest classifiers, we examine which metrics are important for understanding whether a logging statement will change. We show that our classifiers achieve 83%-91% precision and 65%-85% recall in the four studied applications. We find that file ownership, developer experience, log density and file-size are important metrics for determining whether a logging statement will change in the future. Developers can use this knowledge to build more robust log processing tools, by making those tools depend on logs that are generated by logging sCtatements that are likely to remain unchanged.**

## I. Introduction

Developers use logging statements to yield useful information about the state of an application during its execution. Such information is collected into files (logs) and contains details which would otherwise be difficult to collect, such as the values 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 (e.g., Log4j [7]) 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 shown in Figure 1.

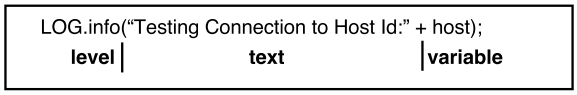


Fig. 1: An example of a logging statement

The rich knowledge in logs has lead to the development of many log processing tools such as *Splunk* [8], *Xpolog* [9], *Logstash* [10] and research tools, such as Salsa [11], Log Enhancer [5] and Chukwa [12], that are designed to analyze logs as well as to 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 [6]. Such changes may affect the log processing tools that heavily depend on the logs that are generated by these logging statements.

Knowing whether a logging statement is likely to change in the future helps to reduce the effort that is 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 logs 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 that is required for keeping the log processing tool consistent with the ever-changing logs [6, 13].

To determine whether a logging statement will change in the future, we must understand which factors influence such a change. The following factors can influence whether a logging statement will change:

1) the content of the logging statement,
2) the context of the logging statement (i.e., where the statement resides in the source code),

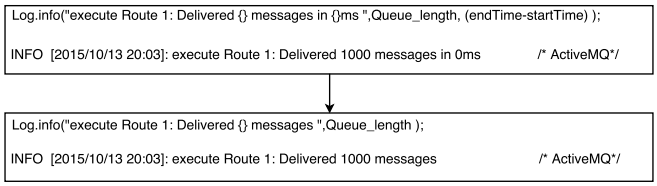


Fig. 2: Modification of a logging statement

3) and the developer who added the logging statement into the source code

In this paper, we examine which of these factors influence the likelihood of a logging statement changing. First, we present a preliminary study which was done to get a better understanding of the changes made to logging statements in the four studied open source applications namely ActiveMQ, Camel, Cloudstack and Liferay. Our preliminary study finds that 20%-45% of the logging statements are changed at least once during their lifetime in the studied applications. Therefore, developers of log processing tools have to carefully select the logging statements on which to depend.

Second, we examine the factors that influence the likelihood of a logging statement changing using a random forest classifier. This classifier uses measures that quantify the three above-mentioned factors to determine the likelihood of a logging statement changing. The most important observations in this paper are:

1) We model whether a logging statement will change in the future using a *random forest* classifier with 83%-91% precision and 65%-85% recall.
2) Logging statements that are 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 the logging statements that are added by the top three developers remain unchanged.
3) Logging statements added by developers who have little ownership on the file that contains the logging statements have a higher likelihood of being changed. We find that 27%-67% of all log changes, are done on logging statements added by developers who own less than 20% of the file.
4) Large files (i.e., ones with SLOC that is 2×-3× the median) with a low log density are more likely to have changes to their logging statements than well logged files.

The above findings can assist the maintainers of log processing tools with selecting the logging statements on which their tool will depend. The remainder of this paper is organized as follows. Section II presents the preliminary analysis that motivates our study. Section III describes the random forest classifier and the analysis results. Section IV describes 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 logging statements in open source applications. The goal of our study is to present a classifier for determining whether a logging statement is likely to change in the future. This classifier can assist developers of log processing tools in determining 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 process for selecting the applications that we studied and present the results of our preliminary analysis of the four studied applications.

### A. Studied Applications

We selected our studied applications based on the following three criteria:

- **Usage of logging statements.** The applications must make extensive use of logging statements in their source code.
- **Application activity.** The applications 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 only select applications that are written in Java and are available through a Git repository.

To select applications that match these criteria, we first selected all Java applications 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

Listing 1 counts the occurrences in Java files of invocations 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 an application to get the total number of logging statements shown in Table I.

We select the four applications (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 application that is designed to deploy and manage large networks of virtual

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

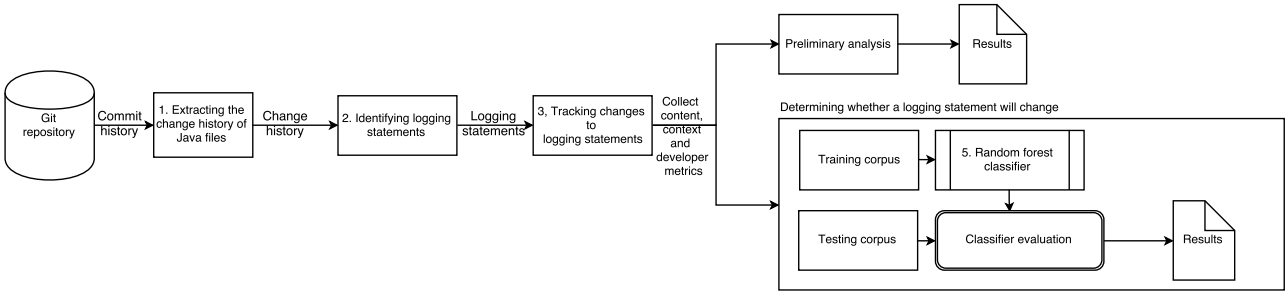


Fig. 3: Overview of the data extraction and empirical study approach

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

| | **ActiveMQ** | **Camel** | **CloudStack** | **Liferay** |
|---|---|---|---|---|
| # of logging statements | 5.1K | 6.1K | 9.6K | 1.8K |
| # of commits | 11K | 21K | 29K | 143K |
| # of years in repository | 8 | 8 | 4 | 4 |
| # of contributors | 41 | 151 | 204 | 351 |
| # of added lines of code | 261K | 505K | 1.09M | 3.9M |
| # of deleted lines of code | 114K | 174K | 750K | 2.8M |
| # of added logging statements | 4.5K | 5.1K | 24K | 10.4K |
| # of deleted logging statements | 2.3K | 2.4K | 17K | 8.1K |
| % of logging-related changes | 1.8% | 1.1% | 2.3% | 0.3% |

machines. Liferay[5] is an open source platform for building websites and web portals. Table I presents an overview of the studied applications.

### B. Data Extraction Approach

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

1) We clone the Git repository of each studied application 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.

We use R [14] to perform 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 that are 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 application 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 application to identify the logging library that is used to generate the logs. We find that the studied applications use *Log4j* [15], *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 an application 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 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 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 [16].

We use the Levenshtein ratio instead of string comparison, because the 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

[5]http://www.liferay.com/

[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

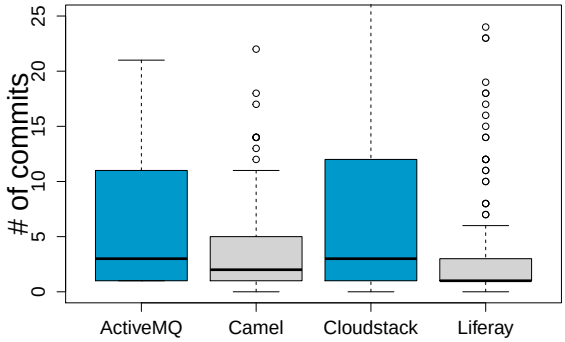


Fig. 4: Number of commits before an added logging statement is changed in the studied applications (Cloudstack has outliers which are not shown due to their large numbers)

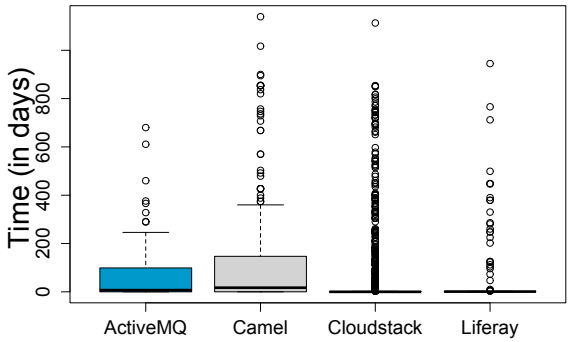


Fig. 5: Number of days before an added logging statement is changed in the studied applications

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 to logging statements, 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 applications, the maximum number of commits between the addition of a logging statement and its first change is 390, as shown in Figure 4. We exclude all logs added to the application 390 commits before the last commit of our analysis.

### C. Results

> *20%-45% of the logging statements in the studied applications are changed. The median number of days between the addition of a logging statement and its first change is between 1 and 17.*

We observe that 28%, 35.2%, 44,6% and 21.2% of the logging statements are changed in ActiveMQ, Camel, Cloudstack and Liferay, respectively, during their lifetime. The observed values show that logging statements change extensively throughout the lifetime of an application, which can affect log processing tools.

From Figure 5, we observe that 75% of the changes to logging statements are done within 145 days after the log is added. In fact, the largest median number of days between the addition of a logging statement and its first change is 17 in our studied applications. This number shows that, all too often, the changes to logging statements happen in a short time after the logging statement being added. Hence, it is important for developers of log processing tools to not depend on logging statements that are likely to change, as this dependency will require additional maintenance within a short time.

## III. Determining the likelihood that a Logging Statement Will Change in the Future

In our preliminary analysis, we find that 25-45% of the logging statements are changed in our studied applications. These logging statement changes affect the log processing tools that depend on the logs that are generated by these statements, forcing developers to spend more time on maintenance of their tools. By analyzing the metrics which can influence the likelihood that a logging statement will change, developers of log processing tools can reduce the effort spent on maintaining their tools, by letting their tool depend on logging statements that are likely to remain unchanged. In this section, we train a random forest classifier for determining the likelihood that a logging statement will change in the future. We then evaluate the performance of our random forest classifier and use the classifier to understand which metrics increase the likelihood of a change to a logging statement.

### A. Approach

We use metrics that measure the context, the content and the developers of the logging statements to train the random forest classifier. Context metrics measure the file context at

the time of adding the logging statement. Content metrics collect information about the logging statement. Developer metrics collect information about the developer who added the logging statement. Table II defines each metric collected and the rationale behind our choice of each metric. We use the Git repository to extract the context, content and developer metrics for the studied applications.

We build random forest classifier [17] to determine the likelihood whether a logging statement will change in our studied applications. A random forest is a collection of decision trees in which the produced classification of all trees are combined to form a global classification. In our classifier, the context, content and developer metrics are the explanatory variables and the dependent class variable is a boolean variable that represents whether the logging statement ever changed or not (i.e., '0' for not changed and '1' for changed).

Figure 6 provides an overview of the construction steps (C1 to C2) for building a random forest classifier and steps (A1 and A3) for analyzing the results. We use the statistical tool R to model and to analyze our data using the *RandomForest* package.

*Step C1 - Removing Correlated and Redundant Metrics*

Correlation analysis is necessary to remove the highly correlated metrics from our dataset [20]. Correlated metrics can lead to incorrect determination 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 the Spearman rank correlation importances [21] 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 [22] because Spearman is resilient to data that is not normally distributed. We use the function *varclus* in R to perform the correlation analysis.

Figure 7 shows the hierarchically clustered Spearman $\rho$ values in the ActiveMQ. The solid horizontal lines indicate the correlation value of the two metrics that are connected by the vertical branches that descend from it. We include one metric from the sub-hierarchies which have correlation $|\rho| > 0.7$. The dotted blue line indicates our cutoff value ($|\rho|$ = 0.7). We use cutoff value of ($|\rho|$ = 0.7) as used by prior research [23] to remove the correlated metrics before building our classifier.

We find that *total revision count* is highly correlated with *code churn in commit*, *log churn in commit* and *log addition*, because a file with more commits has higher chance of having a large commit with log changes, than a file with less commits. We exclude *total revision count*, *log churn in commit* and *log addition* and retain *code churn in commit* as it is a simpler metric to compute.

Correlation analysis does not indicate redundant metrics, i.e, metrics that can be explained by a combination of other explanatory metrics. The redundant metrics can interfere with the one another and the relation between the explanatory and dependent metrics is distorted. We perform redundancy

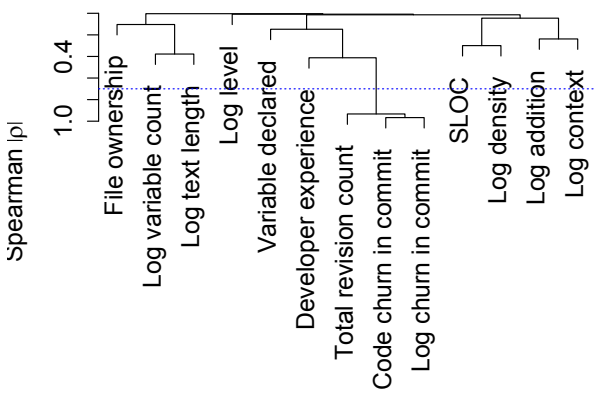


Fig. 7: Hierarchical clustering of variables according to Spearmans $\rho$ in ActiveMQ

analysis to remove such metrics. We use the *redun* function that is provided in the *rms* package to perform the redundancy analysis. We find after removing the correlated metrics, that there are no redundant metrics.

*Step C2 - Random Forest Generation*

After we eliminate the correlated metrics from our datasets, we construct the random forest classifier. 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 $m$ logging statements a random sample of *n* components is selected with replacement [23] and using the *randomForest* function from the *randomForest* package in R, a random forest classifier is generated.

*Step A1 - Model Validation*

After we build the random forest classifier, we evaluate the performance of our classifier using precision, recall, F-measure, AUC and Brier Score. These measures are functions of the confusion matrix and are explained below.

*Precision (P)* measures the correctness of our classifier in predicting which logging statement will change in the future. Precision is defined as the number of logging statements which were correctly classified as changed over all logging statements classified to have changed as explained in Equation 1.

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

*Recall (R)* measures the completeness of our classifier. A classifier is said to be complete if the classifier can correctly determine all the logging statements which will get changed in our dataset. Recall is defined as the number of logging statements which were correctly classified as changed over number of logging statements which actually change as explained in Equation 2.

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

TABLE II: The investigated metrics in our classifier

| Dimension | Metrics | Values | Definition (d) – Rationale (r) |
|---|---|---|---|
| Context Metrics | Total revision count | Numerical | d: Total number of commits made to the file before the logging statement is added. This value is 0 for logging statements added in the initial commit of the project but not for logging statements added over time.<br>r: Logging statements present in a file which is often changed, have a higher likelihood of being changed [18]. Hence, the more prior commits to a file, the higher the likelihood of a change to a logging statement. |
| | Code churn in commit | Numerical | d: The code churn of the commit in which a logging statement is added.<br>r: The likelihood of change of logging statements that are added during large code changes, such as feature addition, can be different from that of logging statements added during bug fixes which have less code changes. |
| | Variables declared | Numerical | d: The number of variables which are declared before the logging statement in that function.<br>r: When a large number of variables are declared, there is a higher chance that any of the variables will be added to or removed from a logging statement afterwards. |
| | SLOC | Numerical | d: The source lines of code in the file.<br>r: Large files have more functionality and are more prone to changes [19] and changes to logging statements [15, 18]. |
| | Log context | Categorical | d: The block in which a logging statement is added i.e., *if, if-else, try-catch, exception, throw, new function*.<br>r: The stability of logging statements used in logical branching and assertion checks, i.e., if-else blocks, may be different from the logging statements in try-catch, exception blocks. |
| Developer Metrics | File ownership | Numerical | d: Percentage of the file written by the developer who added the logging statement.<br>r: The owner of the file is more likely to add stable logging statements than developers who have not edited the file before. |
| | Developer experience | Numerical | d: The number of commits the developer has made prior to this commit.<br>r: More experienced developers may add more stable logging statements than a new developer who has less knowledge of the code. |
| Content Metrics | Log addition | Boolean | d: Check if the logging statement is added to the file after creation or it was added when file was created.<br>r: Newly added logging statements may be more likely to be changed than logging statements that exist since the creation of the file. |
| | Log variable count | Numerical | d: Number of logged variables.<br>r: Over 62% of logging statement changes add new variables [18]. Hence, fewer variables in the initial logging statement might result in addition of new variables later. |
| | Log density | Numerical | d: Ratio of the number of logging statements to the source code lines in the file.<br>r: Files that are well logged (i.e., with higher log density) may not need additional logging statements and are less likely to be changed. |
| | Log level | Categorical | d: The level (verbosity) of the added logging statement, i.e., *info,error, warn, debug, trace* and *trace*.<br>r: Research has shown that developers spend significant amount of time in adjusting the verbosity of logging statements [18]. Hence, the verbosity level of a logging statement may affect its likelihood of change. |
| | Log text count | Numerical | d: Number of text phrases logged. We count all text present between a pair of quotes as one phrase.<br>r: Over 45% of logging statements have modifications to static context [18]. Logging statements 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.<br>r: Logging statements can be added as part of a specific change or part of a larger change. |

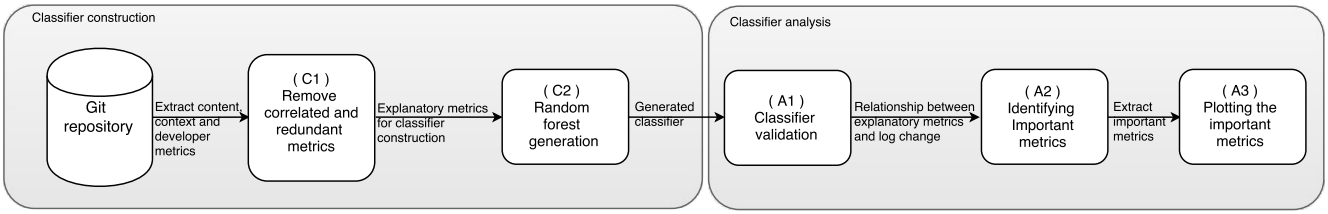


Fig. 6: Overview of random forest classifier construction (C), analysis (A) and flow of data in random forest generation

*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 [24].

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

*Area Under Curve (AUC)* is used to measure the overall ability of the classifier to determine changed and unchanged logging statements. AUC is the area below the curve plotting the true positive rate against false positive rate. The value of AUC ranges between 0.5 (worst) for random guessing and 1 (best) where 1 means that our classifier can correctly determine every logging statement as changed or unchanged. We calculate AUC using the *roc.curve* function from the *pROC* package in R.

*Brier score (BS)* is a measure of the accuracy of the classifications of our classifier [25]. The Brier score explains how well the classifier performs compared to random guessing as explained in Equation 4, where $P_n$ is the probability of whether a logging statement will change and $O_n$ is the boolean that shows whether the statement actually changed. 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). When there is an equal distribution of logging statements being changed and un-changed, the Brier score reaches the value of 0.25 for random guessing (i.e., 50% chances that log is un-changed).

$$BS = \sum_{n=1}^{M} (P_n - O_n)^2 \tag{4}$$

*Optimism:* The previously described performance measures described previously may overestimate the performance of the classifier due to overfitting. To account for the overfitting in our classifier, we use the *optimism* measure, as used by prior research [23]. 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 built from the bootstrap sample on both the bootstrap and original data sample, calculating precision, recall, F-measure and Brier score for both data samples.
4) Calculate the *optimism* by subtracting the performance measures of the bootstrap sample from the original sample.

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

*Step A2 - Identifying Important Metrics*

To find the importance of each metric in a random forest classifier, we use a permutation test [26]. In this test, the classifier 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 classifier 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 the $X_i$ $^{th}$ metric 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 from the bootstrapping process, we use the **Scott-Knott Effect size clustering** (SK-ESD) to group the metric based on their effect size [27]. Such an approach groups metrics based on their importance in predicting the likelihood of logging statement changes. The SK-ESD algorithm uses effect sizes that are calculated using *cohen's* delta [28], to merge any two statistically indistinguishable groups. We use the *SK.ESD* function in the *ScottKnottESD* package of R and set the effect size threshold parameter to negligible, (i.e., $< 0.2$) to cluster the two metrics into the same groups.

*Step A3 - Plotting the Important Metrics*

To understand the effect of each metric in our random forest classifier, it is necessary to plot the predicted probabilities of a change to a logging statement against the metrics. By plotting the predicted probabilities of a change to a logging statement,

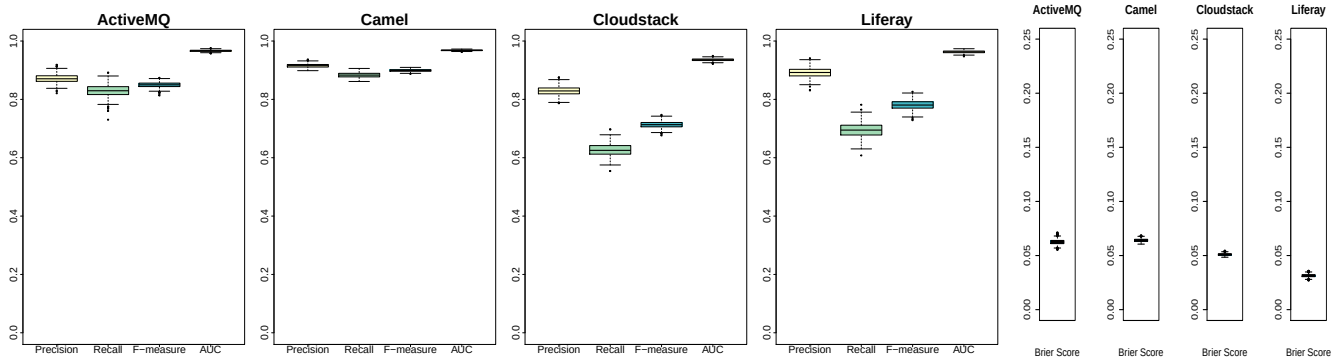


Fig. 8: The optimism reduced performance measures of the four applications

TABLE III: The most important metrics, divided into homogeneous groups by Scott-Knott Effect Size clustering

| | 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 |
| | | | 8 | 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 | 7 | Log text length | 0.071 |

we obtain a clearer picture of how the random forest classifier uses the important metrics to determine the likelihood that a logging statement changes.

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 a change to a logging statement. We plot each predicted probability against the value of the metric, to understand how changes in the metric values affect the probability of a change to a logging statement.

### B. Results

*The random forest classifier achieves 0.89-0.91 precision, 0.71-0.83 recall and outperforms random guessing for our studied applications.*

Figure 8 shows the optimism-reduced values of *precision*, *recall*, *F-measure AUC* and *Brier score* for each studied application. The classifier achieves an AUC of 0.94-0.95.

The Brier scores for ActiveMQ, Camel, Cloudstack and Liferay are 0.061, 0.060, 0.051 and 0.042, respectively. Using Equation 4, we find that for intuitive prediction based on the portion of changed and un-changed logging statements, the Brier score values are 0.20, 0.12, 0.24, 0.17 in ActiveMQ, Camel, Cloudstack and Liferay respectively. Our approach outperforms such an intuitive prediction with a much lower Brier score.

TABLE IV: Contribution of top 3 developers

| | Total logs | Changed logging statements | Total # of contributors |
|---|---|---|---|
| ActiveMQ | 956 (50.4%) | 301 (31.4%) | 41 |
| Camel | 3,060 (63.1%) | 1,460 (47.7%) | 151 |
| Cloudstack | 5,982 (35.7%) | 2,276 (38.0%) | 204 |
| Liferay | 3,382 (86.7%) | 609 (18.0%) | 351 |
| **Average** | **3,345 (59%)** | **1,161 (33.75%)** | 747 |

### *B2. Important Metrics Determining the Likelihood of a for Logging Statement Changing*

*In three out of four studied applications, the top three developers were responsible for adding over 50% of the logging statements. Up to 70% of these logging statements never change.*

Table III shows the important metrics for determining whether a logging statement will change in the future. From Table III, we see that developer experience is in the top four metrics for all studied applications to help explain the likelihood of a logging statement being changed. Figure 9 shows the probabilities of a logging statement being changed as developer experience increases. The grey region around the line indicates the confidence interval of 0.95. In all the studied applications, logging statements that are added by new developers have a lower probability of being changed, when compared to those added by more experienced developers. We account this phenomenon to the fact that inexperienced developers add a very small fraction of the logs (12% in Cloudstack and 1-3% in the other applications).

We also observe that as developers become more experienced the probability of a change to a logging statement

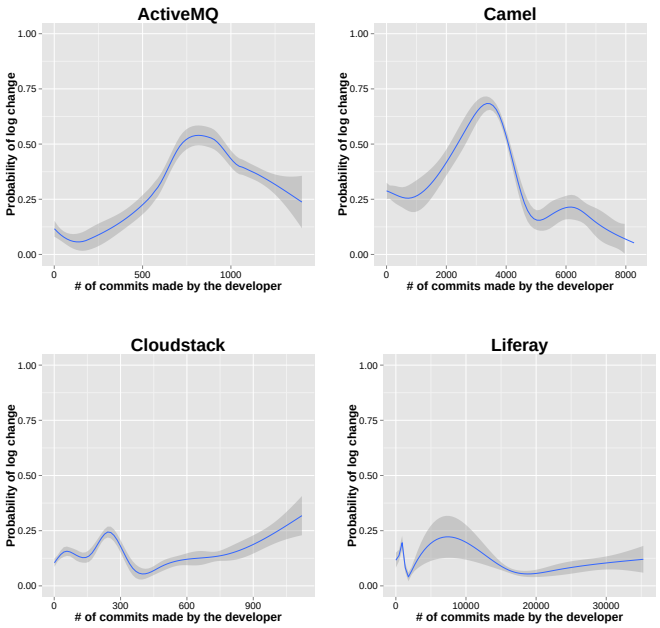


Fig. 9: Comparing the probability of a change to a logging statement against the experience of the developer who adds that logging statement

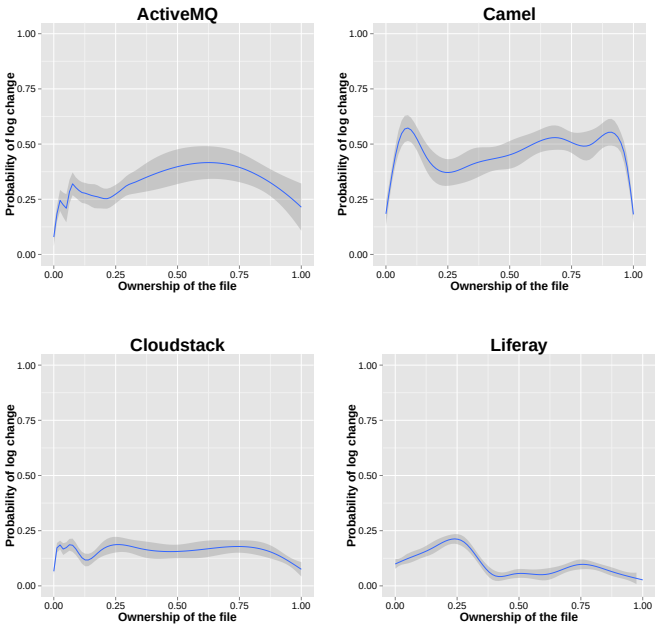


Fig. 10: Comparing the probability of a change to a logging statement against ownership of the file in which the logging statement is added

decreases in ActiveMQ, Camel and Liferay. This downward trend may be explained by the fact that in ActiveMQ, Camel and Liferay, the top three developers are responsible for adding more than 50% of the logging statements as seen in Table IV. In addition, we find that up to 70% of the logging statements added by these top developers never change.

> *Logging statements that are added into a file by developers who own more than 75% of that file are unlikely to be changed in the future.*

From Table III, we see that ownership of the file is in the top two metrics to help explain the likelihood of a change to a logging statement in all the studied applications. From Figure 10, we observe in all the applications that logging statements 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 changes to logging statements in the studied applications, which is seen as upward trend from 0 to 0.20 in Figure 10. These results suggest that developers of log processing tools should be more cautious when using a logging statement written by a developer who has contributed less than 20% of the file.

> *Logging statements in files with a low log density are more likely to change than logging statements in files with a high log density.*

From Table III, we observe that log density has the highest importance in Liferay and Cloudstack. We find that in these two applications, changed logging statements are in files that have a lower log density than the files containing unchanged logging statements. When we measure the median file sizes, we find that logging statements which change more are present in files with significantly higher SLOC (2×-3× higher). This difference in SLOC, suggests that large files that are not well logged are more likely to have unstable logging statements, than well logged files.

> *Developer experience, file ownership, file size, and log density are important metrics for determining whether a logging statement will change in the future.*

## IV. Related Work

We present prior research that is related to this paper.

### A. Log Maintenance Tools

Prior research has explored various approaches in order to assist developers in maintaining logs. Research by Fu et al. [29] explores where developers put logging statements in their code and provides guidelines for more effective logging. A recent work by Zhu et al. [30] helps developer log effectively during development and provides a suggestive tool named *Log Advisor*, to assist in logging. Yuan et al . [31] show that logs can be effectively used to diagnose system failures and provide a tool named *Errlog*, to proactively add logging statements. A follow-up work done by them Yuan et al. [5] show that logs need to be improved by providing additional information. Their tool named *Log Enhancer* can automatically provide additional control and data flow parameters into the logs thereby improving the logs. *Log Enhancer* can improve the quality of log added and mitigate the need for

changes later. Though prior research tries to place the most stable logging statements in software, they do not provide any insight into why some logging statements are more likely to be changed. Our paper tries to determine which logging statements have higher likelihood of being changed and avoid using such logging statements in the log processing tools.

### B. Empirical Studies on Logging Statements

Prior research performs empirical studies to understand the characteristics of logging statements. Yuan et al. [18] study the logging characteristics in four open source systems and finds that logging statements are changed 1.8 times more than regular code. Shang et al. performed an empirical study on the evolution of both static logs and logs outputted during run time [15, 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. Shang 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.

The existing empirical studies on logging statements show that 1) logs are leveraged by developers for different purposes and 2) logging statements are changed extensively by developers without consideration of other stakeholders, which affect practitioners and end users. These findings highlight the need for better understanding of the factors determining the likelihood of a logging statement changing.

## 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 applications 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 use 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 determining whether a logging statement will change, we need more information to determine how likely it is going to be changed again. In future work, we will extend our model to give more specific details about stability of logs (i.e., how likely will a changed log be changed again).

**Internal Validity.** Our study is based on the data from Git repositories of 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 likelihood of change 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 that is required for the maintenance of such log processing tools, we examine changes to logging statements in four open source applications. The goal of our work is to help developers of log processing tools determine 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 20%-45% 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%.
- Logging statements added by very experienced developer are less likely to be changed.
- Logging statements added by a developer who owns more than 75% of a file, is less likely to be changed.
- We find that developer experience, file ownership, log density and file-size play an important role in determining the likelihood of log statement changing.

Our findings highlight that we can correctly determine the likelihood of a log changing. Developers can use this knowledge to be more selective when designing log processing tools around logging statements.

## REFERENCES


[1] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SOPS 2009, 22nd symposium on Operating systems principle*, 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 *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 24–24.

[3] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proceedings of the ICDM 2009, Ninth IEEE International Conference on Data Mining*. IEEE, 2009, pp. 149–158.

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

[5] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *Proceedings of ASPLOS 2011, The 16th Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–14, 2011.

[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] Log4j. [Online]. Available: http://logging.apache.org/log4j/2.x/

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

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

[10] 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 MW4NG 2013, The 8th Workshop on Middleware for Next Generation Internet Computing*. ACM, 2013, p. 5.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of ICSE 2012, The 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 102–112.

[19] D. Zhang, K. El Emam, H. Liu *et al.*, "An investigation into the functional form of the size-defect relationship for software modules," *IEEE Transactions on Software Engineering,*, vol. 35, no. 2, pp. 293–304, 2009.

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

[21] J. H. Zar, *Spearman Rank Correlation*. John Wiley & Sons, Ltd, 2005.

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

[23] 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.

[24] 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.

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

[26] C. Strobl, A.-L. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis, "Conditional variable importance for random forests," *BMC bioinformatics*, vol. 9, no. 1, p. 307, 2008.

[27] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An Empirical Comparison of Model Validation Techniques for Defect Prediction Model," http://sailhome.cs.queensu.ca/replication/kla/model-validation.pdf, 2015, under Review at Transactions on Software Engineering (TSE).

[28] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Soft-*

*ware Technology*, vol. 49, no. 11, pp. 1073–1086, 2007.

[29] 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 *Proceedings of ICSE Companion 2014: The 36th International Conference on Software Engineering,*, pp. Pages 24–33.

[30] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proceedings of ICSE 2015, The 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 415–425.

[31] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: enhancing failure diagnosis with proactive logging," in *in OSDI, 2012*, pp. 293–306.

[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 *Proceedings of ICSE 2012, 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 *Proceedings of ICSME 2014: The International Conference on Software Maintenance and Evolution,*. IEEE, 2014, pp. 21–30.