Noname manuscript No.
(will be inserted by the editor)

Examining the Stability of Logging Statements

Suhas Kabinna[1] · **Cor-Paul Bezemer**[1] · **Weiyi Shang**[2] · **Mark D. Syer**[1] · **Ahmed E. Hassan**[1]

Received: date / Accepted: date

Abstract Logging statements (embedded in the source code) produce logs that assist in understanding system behavior, monitoring choke-points and debugging. Prior work showcases the importance of logging statements in operating, understanding and improving software systems. The wide dependence on logs has lead to a new market of log processing and management tools. However, logs are often unstable, i.e., the logging statements that generate logs are often changed without the consideration of other stakeholders, causing sudden failures of log processing tools and increasing the maintenance costs of such tools. We examine the stability of logging statements in four open source applications namely: Liferay, ActiveMQ, Camel and CloudStack. We find that 20-45% of their logging statements 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 their tools depend.

In order to effectively mitigate the issues that are caused by unstable logging statements, we make an important first step towards determining whether a logging statement is likely to remain unchanged in the future. First, we use a random forest classifier to determine whether a just-introduced logging statement will change in the future, based solely on metrics that are calculated when it is introduced. Second, we examine whether a long-lived logging statement is likely to change based on its change history. We leverage Cox proportionality hazard models (Cox models) to determine the change likelihood

Software Analysis and Intelligence Lab (SAIL) [1]
Queen's University, Kingston, Ontario
E-mail: {kabinna, bezemer, mdsyer, ahmed}@cs.queensu.ca

Department of Computer Science and Software Engineering Concordia [2]
University, Montreal, QC, Canada
E-mail: shang@encs.concordia.ca

of long-lived logging statements in the source code. Through our case study on four open source applications, we show that our random forest classifier achieves a 83%-91% precision, a 65%-85% recall and a 0.95-0.96 AUC. We find that file ownership, developer experience, log density and SLOC are important metrics in our studied projects for determining the stability of logging statements in both our random forest classifiers and Cox models. Developers can use our approach to determine the likelihood of a logging statement changing in their own projects, to construct more robust log processing tools, by ensuring that these tools depend on logs that are generated by more stable logging statements.

1 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 support various development activities such as fixing bugs [49, 27, 11], analyzing load tests [28], monitoring performance [52] and transferring knowledge [37]. Logging statements make use of logging libraries (e.g., Log4j [47]) or more archaic methods such as *print* statements. Every logging statement contains a textual part indicating the event, a variable part providing contextual information about the event and a log level indicating the verbosity of the logging statement. An example of a logging statement is shown in Figure 1.

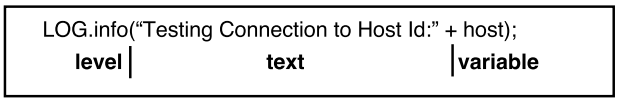


Fig. 1: An example of a logging statement

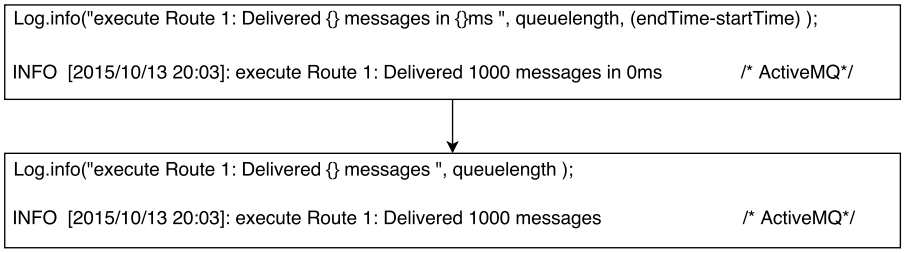


Fig. 2: Modification of a logging statement

The rich knowledge and the wide dependence on logs has lead to the development of many log processing tools such as *Splunk* [4], *Xpolog* [48],

Logstash [50] and research tools, such as Salsa [44] and Chukwa [3], that are designed to analyze logs. 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.

Knowing whether a logging statement is likely to change in the future helps 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 [39, 37]. Even if the tool must depend on a particular log line, having a good idea about the stability of that log line would ensure that developers factor in the realistic maintenance cost associated with such a tool. For example, a developer who must rely on an unstable logging statement could:

- Make the rest of the development team aware that there may be issues with the part of the log processing tool that processes the unstable logging statement in the future,
- Implement additional error handling functionality for the code that processes the unstable logging statement,
- Interact with the development team that is responsible for the logging statement, in order to try and make that important logging statement more stable, or introduce a new logging statement that provides the required information in a more stable way.

We study the likelihood of a logging statement changing in the future by studying the following set of metrics:

M1 The content of the logging statement (i.e., number of variables, log level, length of log text),
M2 The context of the logging statement (i.e., where the statement resides in the source code and the characteristics of the code changes at the time of the introduction of the logging statement),
M3 The characteristics of the developer who introduced the logging statement into the source code.

In this paper, we present an approach that uses this set of metrics to determine the likelihood of a logging statement changing in a project. 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 (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 (or at minimum, factor in the unstable nature of logs in the maintenance efforts of their log processing tools).

Second, we present our approach for determining the likelihood of a logging statement changing. By leveraging a random forest classifier, we can provide early advice to log processing tool developers about the stability of logging statements as soon as they are introduced into the source code. For long-lived logging statements which have already been in the application for several releases, we use the same metrics (M1 to M3). However, the metrics are collected at every release of the application to construct Cox proportionality hazard (Cox) models. These Cox models help developers of log processing tools identify the stable long-lived logging statements for future releases of the application. Our most important results are:

1. **We model the likelihood of a just-introduced logging statement changing in the future using a *random forest* classifier with a precision of 83%-91%, a recall of 65%-85% and an AUC of 0.95-0.96.**
2. **Developer experience is an important metric for determining the change likelihood of a logging statement for both just-introduced and long-lived logging statements in our studied projects.**
3. **Logging statements that are introduced by developers who have little ownership of the file that contains the logging statement, have a higher likelihood of being changed in our studied projects.** We find that 27%-67% of all changes are done on logging statements that are introduced by developers who own (i.e., contributed) less than 20% of the file.
4. **Logging statements that are recently introduced into large files (i.e., files with SLOC that is twice to three times the median SLOC of a project) with a low log density are more likely to be changed than logging statements in well-logged files in our studied projects.**

Our previous work [20] focused on examining whether a just-introduced logging statement will change in the future. In this paper, we extend our prior work to cover long-lived logging statements that have already been in the application for several releases. The presented work in this paper can be used 1) *preventively*, as log processing tool developers can be more selective in picking logging statements on which their log processing tools depend and 2) *proactively*, if log processing tool developers have to depend on a particular logging statement, they are aware of the associated risks of doing so for future releases (i.e., the expected maintenance costs of such tools).

In order to reap the benefits of the approach described in this paper, developers of open source software can apply the approach as described. Developers of tools that process logs of proprietary software often do not have access to the source code history of that software. Hence, they cannot directly apply our

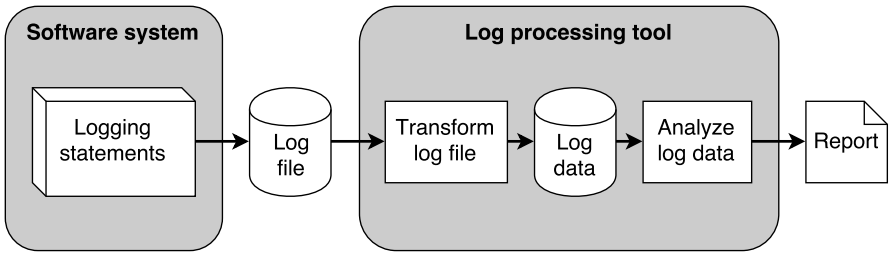


Fig. 3: Overview of a log processing tool

```
INFO [2015/10/13 20:03]: execute Service 1: Delivered 1000 messages in 10ms
INFO [2015/10/13 20:04]: execute Service 1: Delivered 1020 messages in 10ms
INFO [2015/10/13 20:05]: execute Service 1: Delivered 970 messages in 10ms
INFO [2015/10/13 20:06]: execute Service 1: Delivered 960 messages in 10ms
INFO [2015/10/13 20:07]: execute Service 1: Delivered 200 messages in 10ms
```

Listing 1: Example log file (the red line indicates a problem with the service)

approach as described in this paper. A possible solution is that proprietary software vendors use our approach as described and publish the risk factors, i.e., the change likelihood, of each log line with every release of their software. These risk factors can then be used to build robust log processing tools.

The remainder of this paper is organized as follows. Section 2 gives background information on log processing tools and discusses related work. Section 3 discusses several real-world examples of changes to logging statements that break log processing tools to motivate our work. Section 4 presents our experimental setup. Section 5 presents the preliminary analysis that motivates our study. Section 6 describes the random forest classifier and the analysis results. Section 7 describes the construction of the Cox models and the obtained results. Section 8 describes the important metrics from both models. Section 9 discusses the threats to validity. Section 10 concludes the paper and finally, Appendix A presents background information about survival analysis and Cox models.

2 Background

In this section, we give a brief overview of how a log processing tool works, and how it is affected by changes to a logging statement. In addition, we discuss related work.

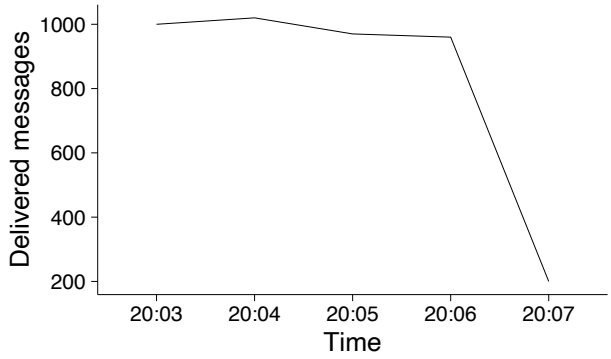


Fig. 4: The graph that can be generated by a log processing tool from the log file in Listing 1

2.1 Log Processing Tools

Figure 3 gives an overview of how a log processing tool works. Log files are generated by logging statements during the execution of a software system. Often, these log files become large and tedious to interpret manually. Therefore, developers employ log processing tools that transform the log into more readable output, such as an aggregated report or graph.

For example, assume that a software system generates the log file that is shown in Listing 1. Lines 1–5 contain information about the number of messages that are delivered by a service that is provided by the software system. Sudden changes in the number of delivered messages (such as in the fifth line) indicate that there is a problem with the service. Hence, the developer of the software system uses the information in the log file to monitor the service. The developer builds a log processing tool that searches for the log lines that contain the words 'execute', and extracts the name of the service and the number of delivered messages from these lines. Then, the extracted information is used to generate the graph in Figure 4 that shows the number of delivered messages of the service over time.

2.1.1 How Different Types of Changes Affect Log Processing Tools

Even though a log processing tool may be designed carefully, changes to a logging statement may break the tool. Li et al. [26] identified three groups of reasons for changing a logging statement. For each group, we discuss whether such a change can affect a log processing tool, if that log processing tool is not updated accordingly.

– **Log improvement.** These changes are made to improve the output that is generated by the logging statement. For example, the debugging capability of the logging statement can be improved by adding variables to

the generated output. Log improvement changes are very likely to break log processing tools, as these changes involve changes to the format of the generated log file.
- **Dependency-driven change.** These changes are made to reflect changes to the code in the log file. For example, a variable or method name is updated. Dependency-driven changes are likely to break a log processing tool when the return type of a method or the unit of a variable changes. An example of such a breaking change was made in the Apache Hadoop project[1], where the unit of a variable was changed from kilobytes to bytes.
- **Logging issues.** These changes are made to fix an issue with a logging statement. For example, the logging level needs to be downgraded from `info` to `debug`. Such changes are very likely to break a log processing tool, as the changes may restrict the amount of information that is generated in the log file.

To summarize, there are many changes to logging statements possible that can break a log processing tool. The symptoms, frequency and severity of the failures of a log processing tool depend on the purpose of the log processing tool. Log processing tools can be used in all stages of software development. For example, a log processing tool can be used to collect information about program exceptions during development, or a log processing tool can be used to collect information about the execution of a program during production. In both cases, the log processing tool may break by throwing an exception when parsing the log file, or by not generating a correct report. The severity of the failure of the log processing tool is difficult to define without domain knowledge. For example, a failure of the log processing tool in the development phase can be catastrophic, if the broken log processing tool causes a developer to overlook a program exception which later ends up in the production code.

In this paper, we present an approach that developers can use to decide whether a logging statement is likely to change in the future. In the remainder of this section, we first discuss related work.

2.2 Related Work

2.2.1 Log Maintenance Tools

Prior research has explored various approaches in order to assist developers in maintaining logs. Research by Fu et al. [10] explores where developers put logging statements in their code and provides guidelines for more effective logging. A recent work by Zhu et al. [55] helps developers log effectively by informing developers where to log and presents a tool named *Log Advisor*, to assist in logging. Yuan et al. [52] show that logs need to be improved by providing additional information and present a tool named *Log Enhancer* can

[1] `http://svn.apache.org/viewvc/hadoop/core/trunk/src/test/org/apache/hadoop/util/TestProcfsBasedProcessTree.java?r1=722760&r2=722759&pathrev=722760`

automatically provide additional control and data flow parameters into the logs thereby improving the logs and avoiding the need for later changes. Follow-up work done by Yuan et al. [51] shows that logs can be effectively used to diagnose system failures and provides a tool named *Errlog*, to pro-actively add logging statements to diagnose failures before occurring. A recent work by Ding et al. [7] presents a tool named *Log*^{2} that condenses the number of irrelevant logs that are generated while preserving the useful logs. Though prior work focuses on introducing effective logging statements in software, prior work does not provide any insight into the stability of logging statements. Our paper presents an approach for determining which logging statements have a higher likelihood of being changed such that developers of log processing tools can avoid depending on such logging statements in their tools.

2.2.2 Empirical Studies on Logging Statements and Logs

Prior work performs empirical studies to understand the characteristics of logging statements. Li et al. [26] study the types of changes that are made to logging statements, and they propose an approach that suggests changes to logging statements when changes to the code are committed. In addition, Li et al. [25] propose an approach for suggesting which log level should be used for a logging statement. Yuan et al. [53] study the logging characteristics of four open source systems and find that logging statements are changed 1.8 times more than regular code. Pecchia et al. [32] study the importance of event logging in industry and find that event logging statements change due the specific needs of the code and that such changes are rarely communicated to the different teams within the company. Shang et al. [41, 38] performed an empirical study on the evolution of both logging statements and the logs that are outputted at run-time. They find that logging statements are changed as software systems mature. However, these changes are done by developers without considering the needs of operators which negatively impacts the log processing tools. Shang et al. highlight the fact that there is a gap between operators and developers of software systems when leveraging of logging statements [36]. Furthermore, Shang et al. [40] find that understanding logs is challenging. They examine user mailing lists from three large open-source projects and find that users of the systems that are produced by these projects have various issues in understanding logs outputted by the system.

The work described above shows that logs continuously evolve and that changes to logging statements are made by developers without consideration for other stakeholders, which affects practitioners and end users. These findings highlight the need for a better understanding of the metrics to determine the likelihood of a logging statement changing. The approach that is presented in this paper facilitates this understanding, as it allows developers to understand which metrics can be used to determine the likelihood of a logging statement changing in their own projects.

3 Two Real-World Examples

In this section, we discuss two real-world examples in which a change to a logging statement breaks a log processing tool.

The first example is issue report `HADOOP-4190`[2] for the Apache Hadoop[3] project. Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware. Hadoop allows the user to store and process very large amounts of data through the execution of tasks (or *jobs*). The JobHistory log file can be processed using a log processing tool (i.e., a parser) that is included in the Hadoop project.

Issue report `HADOOP-4190` explains that the format of the log file was updated, and that the log processing tool was updated accordingly to process only lines that end with a dot ('.'). However, lines in older versions of the log file do not necessarily end with a dot. Therefore, the log processing tool will not function as expected when processing older versions of the log file. The `HADOOP-4190` issue is an example of how a seemingly simple-looking change to a logging statement, i.e., adding a dot to the end of the line, can break a log processing tool. To resolve the reported issue, a 65 Kb patch[4] had to be made to check whether changes made to the log file format affect the Hadoop log processing tool.

The second example is issue report `WICKET-3919`[5] for the Apache Wicket[6] project. Apache Wicket is a Java framework for building web applications. The Wicket project has a request logger which logs requests that are made to the application. However, issue report `WICKET-3919` describes that the log files that are generated by this logger cannot be automatically processed, because of inconsistencies in the format. For example, the `[ResourceStreamRequestTarget[[]]` log line contains an unbalanced number of brackets. Issue report `WICKET-3919` shows that (1) developers use automated tools for processing logs and (2) developers care about the format of logging statements, which implies that they care about changes to this format.

In this paper, we present an approach that can help to prevent issues in log processing tools, by helping developers understand which logging statements are likely to change in the future.

4 Experimental Setup

In this paper we study the changes that are made to logging statements in open source applications. The goal of our study is to determine the likelihood of a logging statement changing in the future. Hence, in this section, we present

[2] `https://issues.apache.org/jira/browse/HADOOP-4190`
[3] `http://hadoop.apache.org/`
[4] `https://issues.apache.org/jira/browse/HADOOP-4191`
[5] `https://issues.apache.org/jira/browse/WICKET-3919`
[6] `https://wicket.apache.org/`

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

Listing 2: Counting logging statements

our process for selecting the studied applications and the modeling techniques used for the analysis of the four studied applications.

4.1 Studied Applications

We selected four applications (ActiveMQ, Camel, Cloudstack and Liferay). ActiveMQ[7] is an open source messaging protocol for delivering messages between two nodes in a network. Camel[8] is an open source source routing engine for constructing route between two nodes in a network. CloudStack[9] is an open source application for deploying and managing large networks of virtual machines. Liferay[10] is an open source platform for building websites and web portals. Table 1 presents an overview of the studied applications. These applications share the following three characteristics:

- **Usage of logging statements.** The applications extensively use logging statements in their source code (i.e., the source code contains more than 1,000 logging statements).
- **Application activity.** The applications have a mature development history (i.e., more than 10,000 commits).
- **Used technology.** 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.

We counted the number of logging statements in all the `*.java` files of the application using the `grep` command in Listing 2. Listing 2 counts the invocations of a logging library (e.g., `log` or `_logger`) followed by the specification of a log level in Java files. We sum the number of invocations in all files of an application to get the total number of logging statements shown in Table 1.

4.2 Modeling Techniques

We use two methods for studying the likelihood of a logging statement changing in the future namely 1) Random forest classifiers and 2) Cox proportionality hazard (Cox) models.

[7] `http://activemq.apache.org/`(last checked April 2016)
[8] `http://camel.apache.org/` (last checked April 2016)
[9] `https://cloudstack.apache.org/` (last checked April 2016)
[10] `http://www.liferay.com/` (last checked April 2016)

Table 1: An overview of the studied applications (all metrics are calculated based on the main branch of the repository on September 2015)

	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 releases	15	25	31	21
# 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%

Random Forest Classifier

A random forest classifier is a collection of decision trees in which the produced classifications of all trees are combined to form a global classification. We use a random forest classifier due to its strong performance in comparison to other classifiers such as SVM, boosted trees, bayes or logistic regressions [12]. Random forest classifiers take a single snapshot of the data as input. Hence, they are ideal for providing early advice on determining the likelihood of a logging statement changing based solely on the metrics that are available when the logging statement is just-introduced.

Cox Proportionality Hazard Model

As an application evolves, the code around the logging statements also evolves which can destabilize logging statements overtime. As random forest classifiers are not capable of using data across different time snapshots for such long-lived logging statements, we leverage survival analysis techniques. We construct Cox models to better understand the change likelihood of long-lived logging statement that has been in the code for some time.

The Cox model is a popular survival analysis model (see Appendix A) that captures the likelihood of an event, e.g., the changing of a logging statement, in relation to the elapsed time. As Cox models are built from snapshots of the data that are collected throughout the lifetime of an application, we can use Cox models to examine the stability of logging statements that have been in the code for some time. As a result, Cox models allow us to calculate the risks of depending on a certain logging statement.

A limitation of Cox models is that when a logging statement is introduced into the application, (i.e., time '0') it will not have any prior observation periods and the calculated risk at time '0' can be erroneous. In such cases it is necessary to use a modeling technique which does not require prior knowledge to determine if a just-introduced logging statement will change in the future.

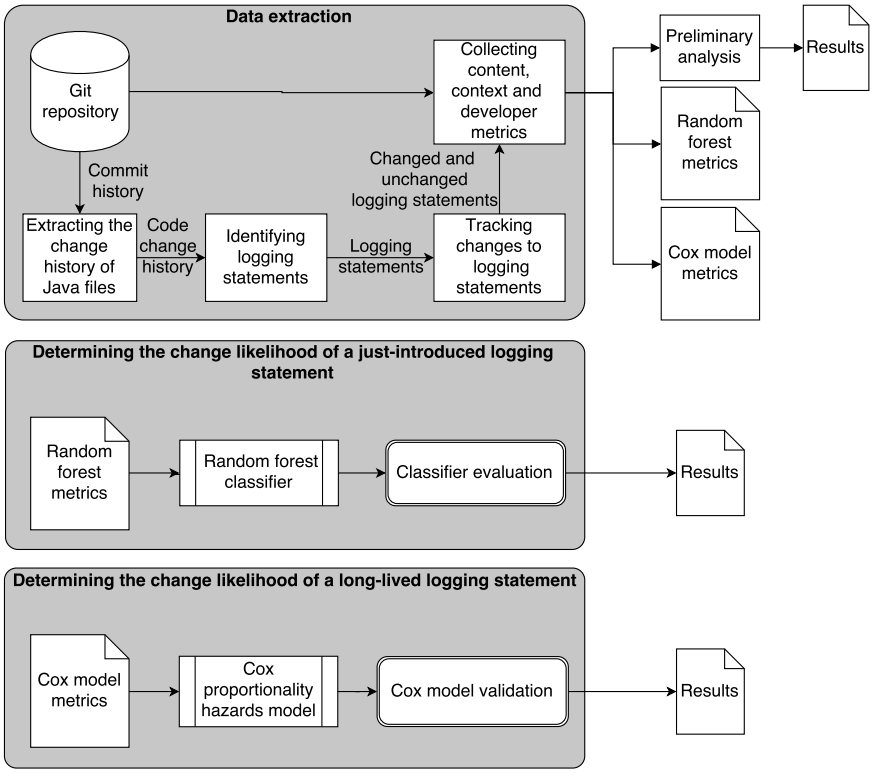


Fig. 5: Overview of the data extraction and empirical study approach

Hence, in the first part of our work we first explore random forest classifiers to determine the change likelihood of just-introduced logging statements and in the second part, we use Cox models to determine the change likelihood of long-lived logging statements.

4.3 Data Extraction Approach

Our data extraction approach for the random forest classifiers and Cox models consists of four steps, which are explained further in this section:

1. We extract the change history of each source code file by cloning the Git repository of each studied application. We then identify logging statements in the repository.
2. We track the changes that are made to each logging statement across commits.

3. We collect the metrics for each logging statement when it is introduced in order to build random forest classifiers.
4. We collect the metrics for each logging statement at each official release of the application in order to build Cox models.

We use the `R` [19] statistical analysis tool to perform our analysis. Figure 5 shows a general overview of our approach and we detail below each of the aforementioned steps.

4.3.1 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 applications and we use the Git repositories of the applications 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 of the applications as changes made to logging statements in other branches are unlikely to affect log processing tools.

4.3.2 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 libraries that are used to generate logs. We find that the studied applications use *Log4j* [41], *Slf4j*[11] and *logback*[12]. 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 [21], 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.

We find that applications can migrate from one logging library to another during development [21]. However, such changes do not affect the log processing tools as only the log invocation patterns are changed. Hence, we exclude the logging statement changes that only have the invoker (e.g., the variable LOG) changed.

[11] `http://www.slf4j.org/` (last checked April 2016)
[12] `http://logback.qos.ch/` (last checked April 2016)

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

Listing 3: Selecting the best matching logging statement

4.3.3 Tracking Changes to Logging Statements

After identifying all the logging statements, we track the changes that are 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 changes in which a new logging statement is introduced 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 map changes to existing logging statements.

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 just-introduced, deleted and changed logging statements and to map the change to an existing logging statement, we use the Levenshtein ratio [30].

We leverage the Levenshtein ratio for both random forest classifiers and Cox models because in both random forest classifiers and Cox models we track a logging statement to observe if the logging statement changes or not. 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 to a change.

Selecting the best matching logging statement is demonstrated by the example in Listing 3. 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 introduced logging statements and select the pair that 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* is 0.76. Hence, we consider *a1* as a change. If there are no more deleted logging statements, *a2* is considered a just-introduced instead of a changed logging statement. We extend the initial set of logging statements with every just-introduced logging statement.

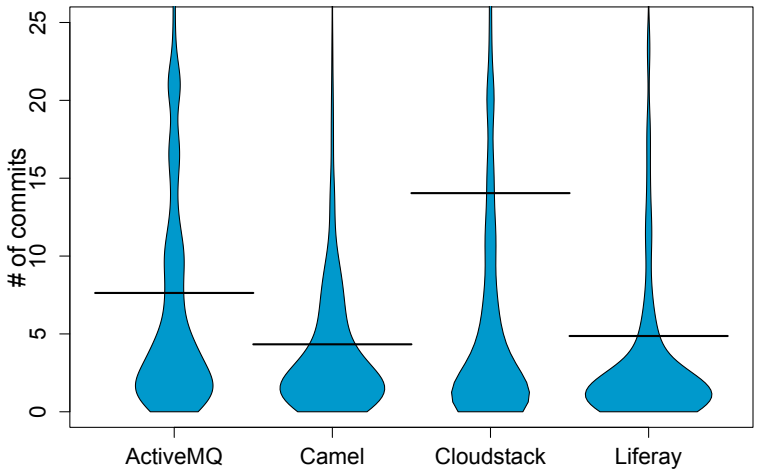


Fig. 6: Number of commits before a just-introduced logging statement is changed in the studied applications (Cloudstack has outliers that are not shown due to their large numbers)

For the random forest classifiers, we do not have change information for logging statements that are introduced at the end of the lifetime of a repository. Hence, 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 6 (we exclude 110 outliers from Cloudstack to make the graph more readable). We exclude all logs that are introduced into the application 390 commits before the last commit when building a random forest classifier. We do not exclude any logging statements for our Cox models.

4.3.4 Collecting Content, Context and Developer Metrics

We collect metrics that measure the context, the content and the developers of the logging statements to train the random forest classifier and Cox models. For the random forest classifier we collect these metrics at the time of the introduction of each logging statement, whereas for the Cox models we collect these metrics at every release.

Context metrics measure the file context and the code changes at the time of adding a logging statement. Content metrics collect information about the logging statement. Developer metrics collect information about the developer who introduced the logging statement. Table 4-6 define each collected metric 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.

Table 2: Example data for survival analysis

Log - ID	Start release	Stop release	Log change	Number of logged variables
$L1$	1	2	0	2
$L1$	2	3	1	4
$L2$	2	3	0	1
$L2$	3	4	0	1
$L2$	4	5	0	1

Table 3: Recurrent changes to logging statements

	Changed once	Changed twice	Changed three times or more
ActiveMQ	25%	4%	0.7%
Camel	31%	2%	1%
Cloudstack	45%	12%	3%
Liferay	20%	4%	2%

4.3.5 Collecting Survival Analysis Data

To construct survival models we have to collect data for all observation periods, i.e., all application releases as shown in Figure 5. We collect the metrics at the start of the observation period (i.e., introduction of logging statement) and every subsequent release of the application, till the end of the study period. Our study period covered four years of development from January 2010 till September 2015 and the statistics of the studied releases are shown in Table 1.

As an example of survival analysis data, Table 2 shows the metrics that are collected for logging statement *L1* for release 1 and the logging statement is changed in release 2. However, *L2* which is introduced into the application at release 2 is never changed. Hence, this logging statement is tracked till the end of the study period.

Similarly, we extract the metrics that are described in Table 4-6 at every official release of an application for all logging statements. We exclude minor releases and release candidates and other 'hotfixes'. However, we find that all the applications have *nightly* and *beta* releases within 5-10 days of one-another which cannot be counted as actual releases of the applications. Hence, we ensure a time difference of 30 days between any two consecutive releases to avoid the *nightly* and *beta* releases in between.

If there is a change made to a logging statement, we tag it as a *log change event* and stop collecting data for that particular logging statement. We do not consider recurrent changes to logging statements in our analysis because we observe from Table 3 that recurrent changes are infrequent to logging statements.

Table 4: The investigated context metrics in our random forest classifier and Cox model

Metric	**Values**	**Definition (d) – Rationale (r)**
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 introduced in the initial commit of the application but not for logging statements introduced over time. r: Logging statements present in a file which is often changed, have a higher likelihood of being changed [53]. Hence, the more prior commits to a file, the higher the change likelihood of a logging statement.
Code churn in a commit	Numerical	d: The amount of code churn in the commit in which a logging statement is added. r: The change likelihood of a logging statement that is introduced during large code changes, such as feature addition, can be different from that of a logging statement that is introduced during bug fixes, which have less code changes.
Declared variables	Numerical	d: The number of variables which are declared before the logging statement in that function. 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 the logging statement throughout its lifetime.
SLOC	Numerical	d: The source lines of code in the file. r: Large files have more functionality and are more prone to changes [54] and to changes to logging statements [53, 41].
Log context	Categorical	d: The block (*i.e., if, if-else, try-catch, exception, throw, new function*) in which a logging statement is introduced. r: The stability of logging statements used in logical branching and assertion checks, i.e., if-else blocks, may be different from the stability of logging statements in try-catch, exception blocks.

Table 5: The investigated developer metrics in our random forest classifier and Cox model

Metric	**Values**	**Definition (d) – Rationale (r)**
File ownership	Numerical	d: Percentage of the file that is written by the developer who introduced the logging statement. In random forest classifiers File ownership is calculated when the log is introduced and in Cox models it is calculated at every release of the application. r: The owner of the file is more likely to add stable logging statements than a developer who has not previously edited the file.
Developer experience	Numerical	d: The number of commits that the developer has made prior to this commit. In both random forest classifiers and Cox models, developer experience is calculated when the log is introduced and the value is not re-calculated. r: More experienced developers may introduce more stable logging statements than a less experienced developer.

Log modification experience	Numerical	d: The number of logging statements modified by the developer who introduces the new logging statement, prior to the addition of this logging statement. r: Developers who frequently changed logging statements in the past are more likely to change logging statements in future changes.

Table 6: The investigated content metrics in our random forest classifier and Cox model

Metric	**Values**	**Definition (d) – Rationale (r)**
Log addition	Boolean	d: Check if the logging statement was introduced to the file after creation or if it was introduced when the file was created. r: Logging statements that are introduced at file creation might be essential statements that are less likely to change.
Log variable count	Numerical	d: Number of logged variables in a logging statement. r: Over 62% of logging statement changes add new variables [53]. Hence, fewer variables in the initial logging statement might result in the addition of new variables later.
Log density	Numerical	d: Ratio of the number of logging statements to the source code lines in the file. r: Files that are well logged (i.e., with a 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 introduced logging statement, i.e., *info, error, warn, debug* and *trace*. r: Research has shown that developers spend a significant amount of time in adjusting the verbosity of logging statements [53]. Hence, the verbosity level of a logging statement may affect its stability.
Log text count	Numerical	d: The number of text phrases that are logged. We count all text present between a pair of quotes as one phrase. r: Over 45% of logging statements have modifications to their textual content [53]. 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. r: Logging statements can be introduced as part of a specific change or part of a larger change.
Log churn ratio	Numerical	d: The ratio of total number of logging statement changes to total code changes in a commit. r: Logging statements which were introduced as a part of large code changes might be different from logging statements that are introduced in a strategical fashion (i.e., changes which introduce many logging statements relative to the introduced code).

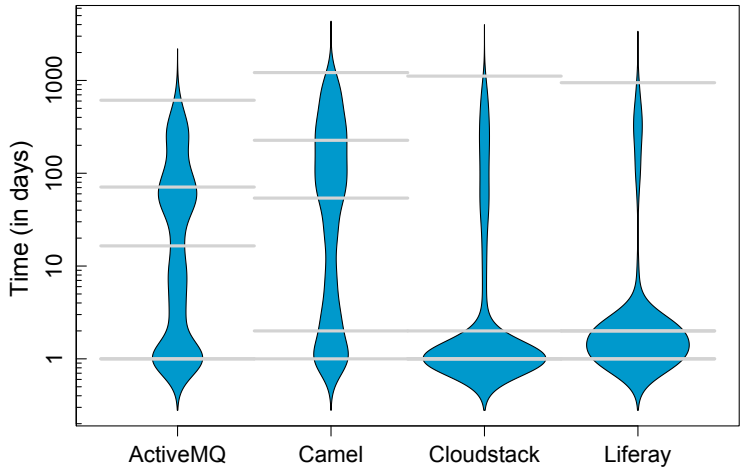


Fig. 7: Number of days before a just-introduced logging statement is changed in the studied applications. The gray lines represent the first, second (median), third and fourth quantiles within each application

5 Preliminary Analysis

In our experimental setup we explain the process for collecting and tracking logging statement changes in the studied applications. In this section, we perform a preliminary analysis, in which we examine how often logging statements change, to motivate our work. Figure 5 provides an overview of our approach for collecting and tracking logging statement changes for the preliminary analysis.

> *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 20%-45% of the logging statements are changed in the studied applications, during their lifetime. The observed values show that logging statements change extensively throughout the lifetime of an application, which can affect the maintenance efforts and costs of log processing tools that depend on such logging statements.

From Figure 7, we observe that 75% of the changes to logging statements are done within 223 days after the log is introduced. In fact, the largest median number of days between the addition of a logging statement and its first change is 17 days in our studied applications. This number shows that, all

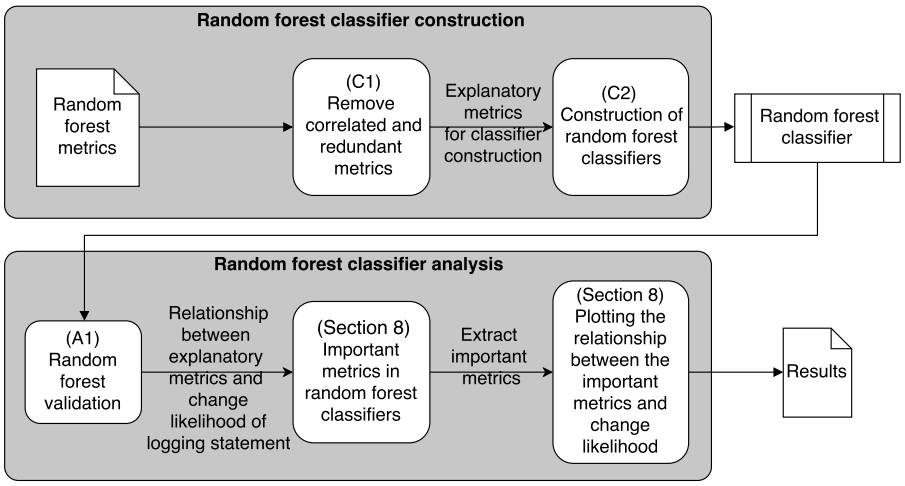


Fig. 8: Overview of random forest classifier construction (C), analysis (A) and the flow of data in the random forest construction

too often, the changes to logging statements happen in a short time after the logging statement being introduced. This result suggests that developers of log processing tools should be cautious when selecting just-introduced logging statements in their log processing tools.

6 Determining Whether a Just-Introduced Logging Statement will Change

In our preliminary analysis, we find that 20-45% 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 maintaining their tools. By analyzing the metrics which can have a strong relationship with the likelihood of a logging statement changing, log processing tools can be more cautious and select more stable logging statements that are likely to remain unchanged.

6.1 Approach

We train a random forest classifier for determining whether a just-introduced logging statement will change in the future, then we evaluate the performance of our random forest classifier. In our classifier, the context, content and developer metrics (see Table 4-6) 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., False for not changed and True for changed).

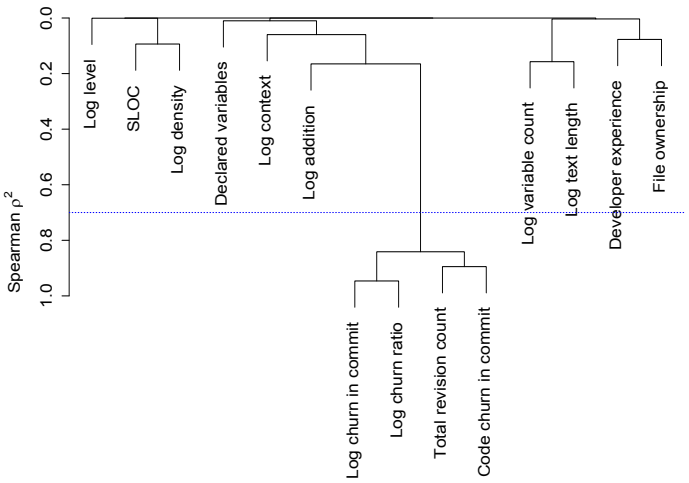


Fig. 9: Hierarchical clustering of variables according to Spearmans ρ^2 in ActiveMQ (the blue dotted line indicates our cutoff value ($|\rho^2| = 0.7$))

Figure 8 provides an overview of the construction steps (C1 and C2) for constructing a random forest classifier and the analysis step (A1) for analyzing the results. We use the statistical tool R to model and analyze our data using the *RandomForest* package.

Step C1 - Removing Correlated and Redundant Metrics

Correlation analysis is necessary to remove highly correlated metrics from our dataset [5]. Correlated metrics can lead to incorrect determination of the importance of a particular metric in the random forest classifier, as small changes to one correlated metric can affect the values of the other correlated metrics.

We use the Spearman square rank correlation [23] to find correlated metrics in our data. Spearman rank correlation assesses how well two metrics can be described by a monotonic function. We use the Spearman square rank correlation instead of the Pearson correlation [35] because the Spearman square correlation is resilient to data that is not normally distributed. We use the function *varclus* in R to perform the correlation analysis.

Figure 9 shows the hierarchically clustered Spearman ρ^2 values for the ActiveMQ application. A solid horizontal line indicates the correlation value of the two metrics that are connected by the vertical branches that descend from it. We keep only one metric from the sub-hierarchies which have correlation $|\rho^2| > 0.7$. The blue dotted line indicates our cutoff value ($|\rho^2| = 0.7$). We

use a cutoff value of ($|\rho^2| = 0.7$) as it represents highly correlated metrics as shown by prior research [1].

We find that *total revision count* is highly correlated with *code churn in commit*, *log churn ratio* and *log churn in commit*. We exclude *total revision count*, *log churn ratio* and *log churn in commit* and retain *code churn in commit* as it is a simpler metric to compute. Similarly, we also find that *developer experience* is highly correlated with *log modification experience*. We retain *developer experience* 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 one another and the relation between the explanatory and dependent metrics would become distorted. We perform redundancy 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 exist no redundant metrics.

Step C2 - Construction of the Random Forest Classifier

After removing the correlated metrics, we construct the random forest classifier. Random forest is an ensemble classifier, which operates by constructing several decision trees using the training set and uses these trees to classify the testing set.

Step A1 - Random Forest Validation

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

Precision (P) measures the correctness of our classifier in determining whether a just-introduced logging statement will change in the future. Precision is defined as the number of just-introduced logging statements which were correctly classified as changed (CC) over all just-introduced logging statements that have changed (TC) as detailed in Equation 1.

$$P = \frac{CC}{TC} \tag{1}$$

Recall (R) measures the ability of our classifier to successfully classify the changed logging statements. A classifier is said to have perfect recall if the classifier can correctly classify all the just-introduced logging statements which change. Recall is defined as the number of just-introduced logging statements which were classified as changed (CC), over the number of all logging statements which are changed (TL) as explained in Equation 2.

$$R = \frac{CC}{TL} \tag{2}$$

F_1-Score also known as F-measure (F) [34], is the harmonic mean of precision and recall, combining the inversely related measure into a single descriptive statistic as shown in Equation 3 [18].

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

Area Under Curve (AUC) is used to measure how well our classifier can discriminate between changed logging statements and unchanged logging statements. AUC is the area below the curve plotting the true positive rate against the 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 classify every logging statement. We calculate AUC using the *roc.curve* function from the *pROC* package in `R`.

Removing Optimism in Performance Measures using Bootstrapping

The previously-described performance measures 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 [29, 14, 20]. The *optimism* of the performance measures is calculated as follows:

1. From the original dataset with m records, we select a bootstrap sample with m records with replacement.
2. Construct a random forest classifier as described in (C2) using the bootstrap sample.
3. Apply the classifier built from the bootstrap sample on both the bootstrap sample and the original data sample, calculating the precision, recall, F-measure and AUC for both data samples.
4. Calculate *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 and AUC by subtracting the averaged optimism of each measure, from their corresponding measure for the original data sample. The smaller the optimism values, the less the chances that the original classifier overfits the data and the more likely that the observed performance on our testing data would generalize to unseen testing data (e.g., data from other applications).

6.2 Results

The random forest classifier achieves a precision of 0.83-0.91, a recall of 0.65-0.85 and outperforms random guessing for our studied applications with an AUC of 0.95-0.96.

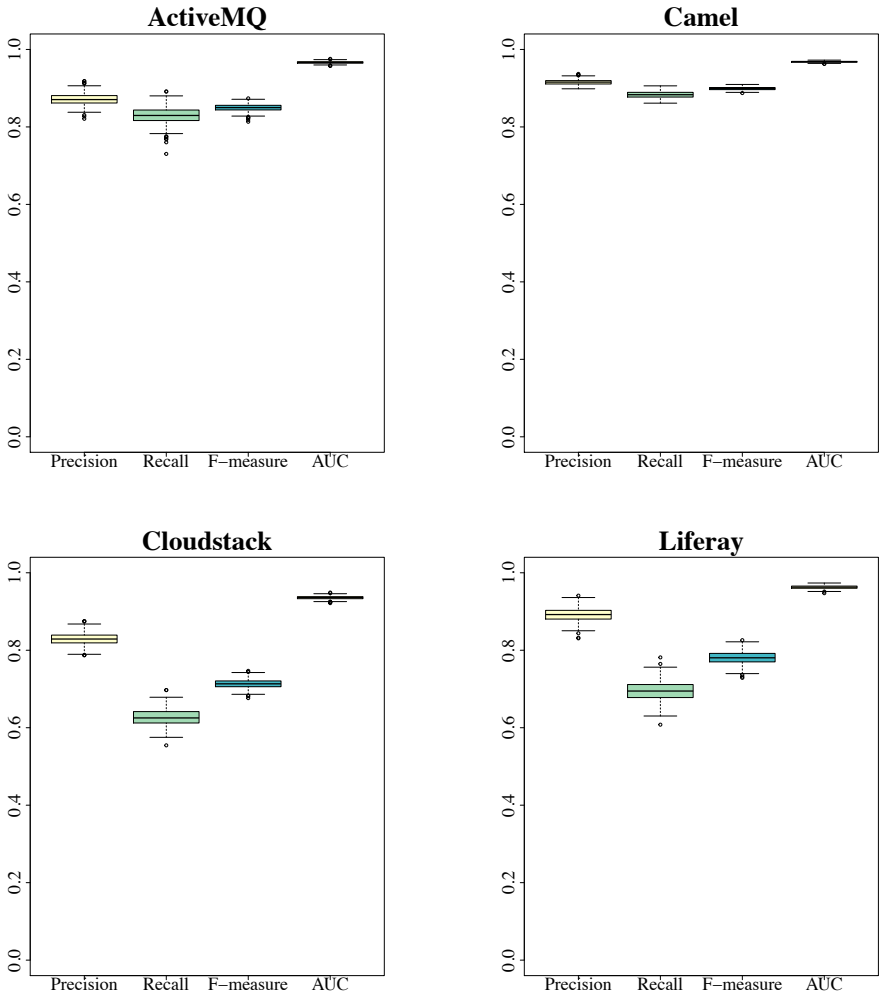


Fig. 10: The optimism-reduced performance measures of the studied applications

Figure 10 shows the optimism-reduced values of precision, recall, F-measure and AUC for each studied application. The classifier achieves an AUC of 0.95-0.96. A random classifier, which randomly ($p = 0.5$) determines whether a logging statement will change in the future, has an AUC of 0.5. Our results show that random forest classifiers can accurately determine whether a just-introduced logging statement will change in the future, with high precision and recall.

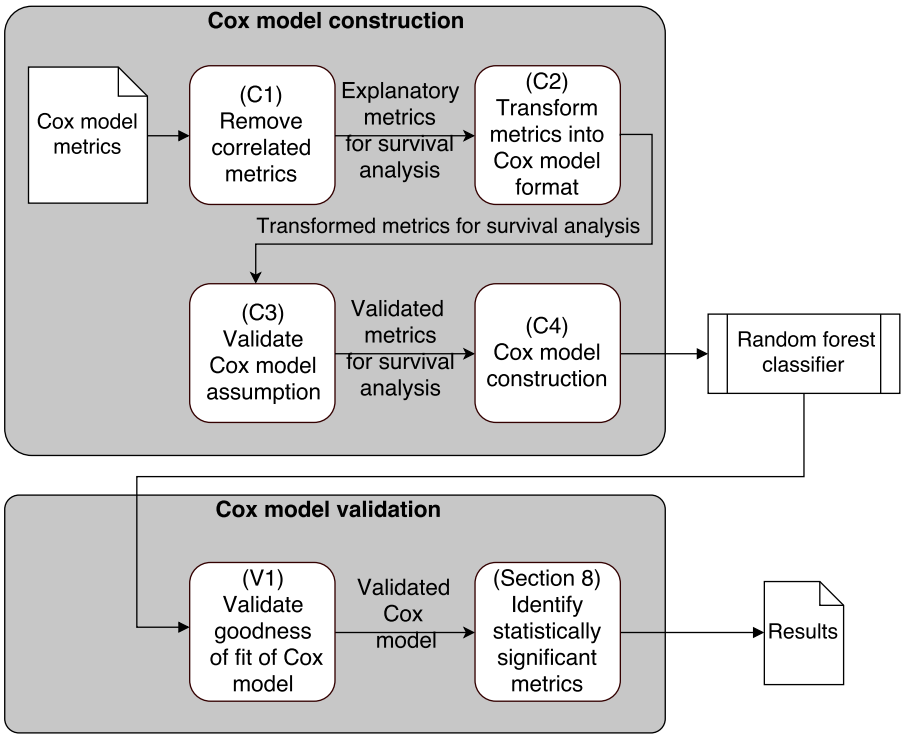


Fig. 11: Overview of Cox model construction (C), validation (V) and flow of data in the Cox model

7 Determining Whether a Long-Lived Logging Statement will Change

Using a random forest classifier we could determine whether a just-introduced logging statement will change in the future. However, the random forest classifier was constructed solely using data that is collected when the logging statement was introduced. Subsequent changes to the file in which the logging statement resides are not considered. Hence, for logging statements that have a change history we leverage survival analysis techniques.

Using survival analysis we aim to determine the most stable logging statements at any give time and explore the relationship of each metric on the change likelihood of long-lived logging statements. This knowledge can help developers of log processing tools avoid using unstable long-lived logging statements, thereby reducing the effort spent of maintaining their log processing tools. It also helps in *proactive analysis*, where developers can use survival analysis to be aware of the associated risks of using a particular logging statement if an alternative, more stable logging statement cannot be found.

In this section, we explain our approach for constructing our survival analysis model and the additional metrics that are collected for this model. Next, we

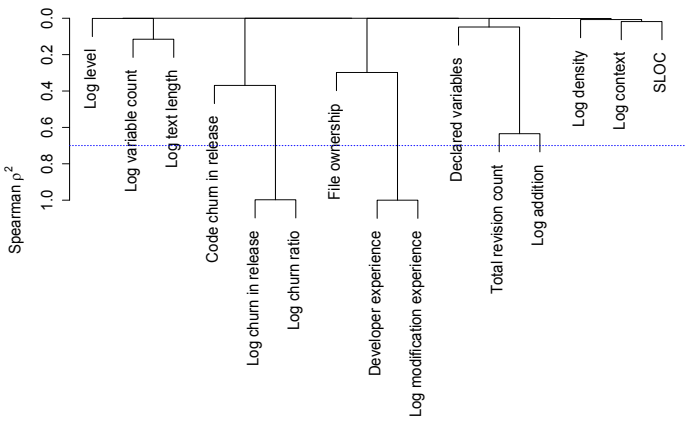


Fig. 12: Hierarchical clustering of variables according to Spearmans ρ^2 in ActiveMQ for Cox models (the blue dotted line indicates our cutoff value ($|\rho^2| = 0.7$))

evaluate the performance of our survival analysis model and use the model to explore the relationship of each metric with the change likelihood of long-lived logging statements.

7.1 Approach

We construct a Cox proportionality hazards (Cox) model [8] to determine the change likelihood of long-lived logging statements. Figure 11 provides an overview of construction steps (C1 to C4) for constructing a Cox model, the step (V1) for the validation of our model and identifying the statistically significant metrics. We use the statistical tool `R` to model our data using the *survival* package.

Step C1: Remove Correlated Metrics

Similar to our random forest classifier, we use the Spearman square rank correlation to find correlated metrics in our data. As Cox models use data that is collected from different releases of the applications, we merge all the releases together and find the correlation between metrics across all releases. Figure 12 shows the hierarchically clustered Spearman ρ^2 values for the ActiveMQ application in Cox models. Similar to random forests we find that the hierarchically clustered Spearman ρ^2 values are similar between *developer experience and log*

modification experience and *log churn in commit and log churn ratio* and we retain *developer experience* and *log churn in commit* respectively from our Cox models. However, we find that *total revision count* and *log churn in release* are not highly correlated as observed in random forest classifiers and hence we retain both these metrics.

Step C2: Transform the Collected Metrics into the Cox Model Format

Before constructing a Cox model, we collect the metrics at every release for every existing logging statement. When a logging statement is changed we identify the release at which the logging statement is changed (i.e., the event) and stop collecting metrics for that changed logging statements. Each observation for a logging statement consists of the following fields:

1. *UID*: We give a unique identifier for each logging statement that is introduced into the application.
2. *Start time*: The end time of the previous release or '0' for the first release of the application.
3. *End time*: The start time of the release plus one, i.e., if the start time is the 11^{th} release of an application, the end time is 12.
4. *Event*: If the logging statement is changed we set event as true, ie., 1 or 0 if the logging statement did not change.
5. *Metrics*: Metrics that are calculated at every release of the application.

Step C3: Validation of the Cox Model Assumptions

Before constructing a Cox model it is essential to validate if the Cox models can be applied to our dataset. To check if Cox models are applicable to our dataset, we verify if the dataset satisfies the Cox model assumptions. The first assumption of a Cox model is the existence of a linear relation between each collected metric in Table 4-6 and the change likelihood of the logging statements in the dataset, i.e., the change likelihood of a logging statement is linearly dependent on the difference between the values of the metrics and this linear relation holds for all time. The second validation is to check for overly influential observations i.e., outliers within the data which can influence the model. We validate these two assumptions prior to the construction of our Cox model.

To assess the existence of a linear relation between each collected metric and the change likelihood of the logging statements, we employ graphical techniques as they help in visually analyzing the linear relationship as numerical methods are widely considered to be insufficient [43, 24]. We leverage Schoenfeld residuals [43, 24, 17], which plot only the changed logging statements to assess the existence of a linear relation between each collected metric and the change likelihood of the logging statement. Schoenfeld residuals can be thought of as the observed values minus predicted values for each collected metric of a

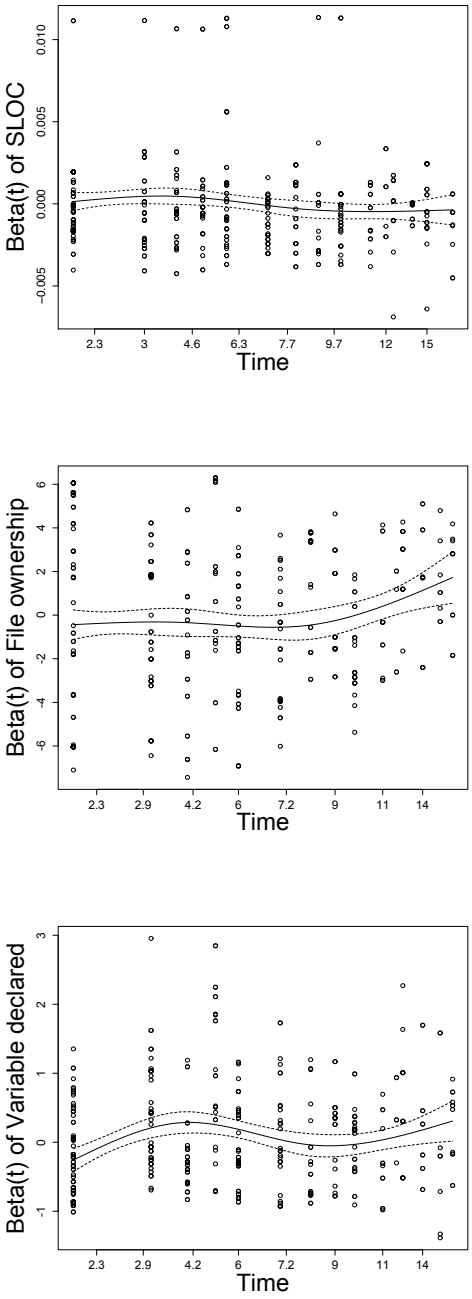


Fig. 13: Three of the scaled Schoenfeld residuals for the Camel application. We did not include all residuals to avoid cluttering the paper.

changed logging statement. If the Schoenfeld residuals have a random pattern against time, i.e., the slope of the Schoenfeld residuals is '0', the linear relation is not violated. However, if a metric is dependent on time, the observed values minus the predicted values for each collected metric can increase or decrease with time. In such a case, the slope of the Schoenfeld residual is non-zero, which implies that the Cox model assumption is violated. For example, in Figure 13 we see that for metric SLOC, the Schoenfeld residuals is nearly constant throughout time (the dotted lines indicate a conference interval of 95%), implying that SLOC is independent of time. We plot the Schoenfeld residuals using the *cox.zph* function from the `R` *rms* package.

Overly influential observations can skew the coefficients of our final model and can affect the validity of Cox models. Hence, to identify the existence of overly influential observations we calculate *dfbeta* residuals as proposed by prior research [43, 24]. *Dfbeta* residuals calculate the influence of each observation by fitting a Cox model with and without the observation. The difference between the coefficients for the metrics with and without the observation shows the influence of that observation in the Cox model. Overly influential observations will have *dfbeta* residuals greater than twice the inverse of the square root of the number of logging statements in the application. We plot *dfbeta* residuals for each metric using the *residuals* function within the `R` *stats* packages and we use the *abline* function to demarcate the overly influential observations. For example, we observe that in Figure 14 many metrics have overly influential observations (i.e., points outside the area marked by the red lines) which suggests that these observations are overly influential (absence of red lines indicates that the metric has no overly influential observations). However, upon further analysis we find that these outliers are valid observations and we do not exclude these observations in our Cox models.

Step C4: Cox Model Construction

After transforming the collected metrics into the Cox model format and ensuring the Cox model assumptions are met, Cox models are constructed using the *cph()* function from the `R` *rms* package.

Step V1: Validation of the Goodness of Fit of the Cox Model

In the previous steps we validated the metrics that are used for building the Cox models. However, to validate the goodness of fit of the Cox models, we leverage the Martingale residuals. Martingale residuals are widely used in the field of biometrics and cancer research to measure the goodness of fit of the Cox models [46, 33, 6]. A Martingale residual considers both unchanged and changed logging statements and the plot ranges between $(-\infty, 1)$ on the Y-axis and the logging statements on the X-axis. If the Martingale residuals plot has more negative values below -1 or positive values close to 1 it indicates that

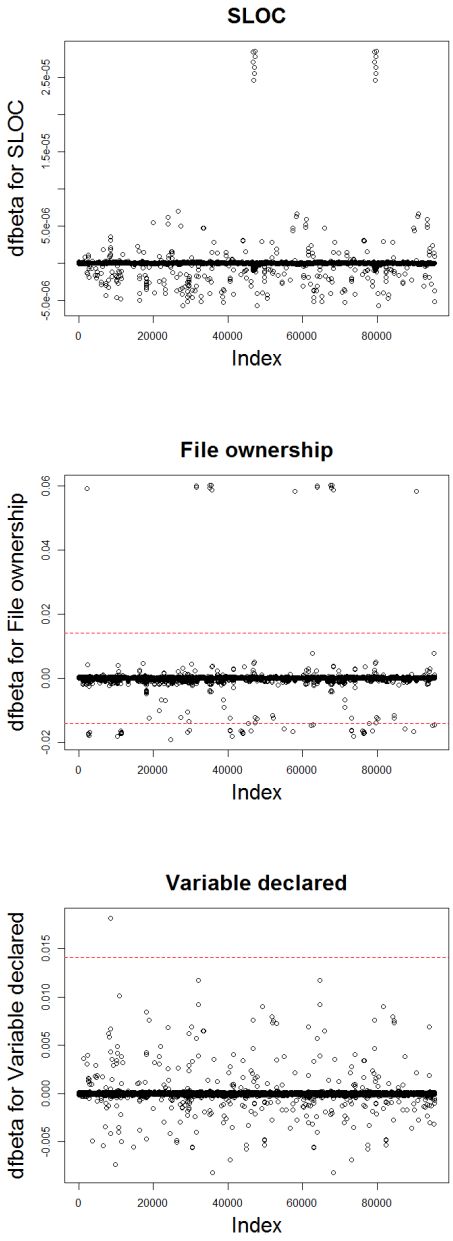


Fig. 14: Three of the *dfbeta* residuals for the Camel application. We did not include all residuals to avoid cluttering the paper.

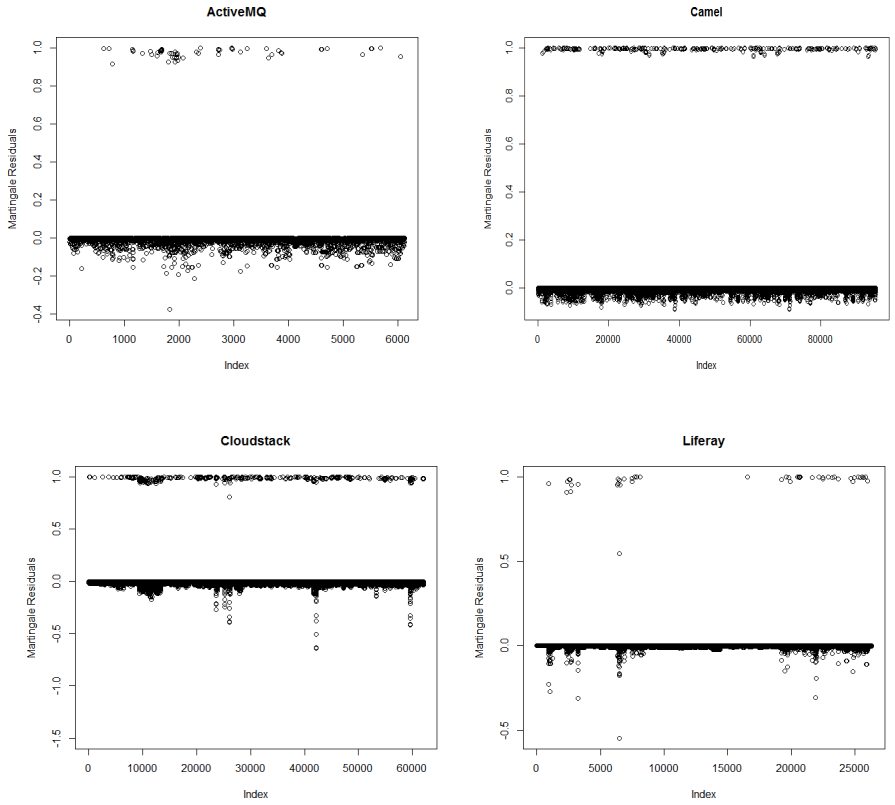


Fig. 15: Martingale residuals to validate the fit of the Cox models for the studied applications

our Cox model is not a good fit. We use the *cph()* function from the R *rms* package to plot the Martingale residuals for the studied applications.

The residuals can be interpreted as the difference between the observed number of logging statement changes in the data and the number of predicted logging statement changes by our Cox model. If a logging statement is changed and our Cox model accurately classifies the change, the Martingale residual has a value of 1. If a logging is not changed and our Cox models accurately classifies the logging statement as not changed, the Martingale residual has value 0. However, if there is a mismatch, i.e., the Cox model classifies the logging statement as not changed but the logging statement is changed, the Martingale residual has values lower than 0. For Cox models to be a good fit to the dataset, the Martingale residuals should have an asymmetric distribution between (-1,1) with minimal mismatches.

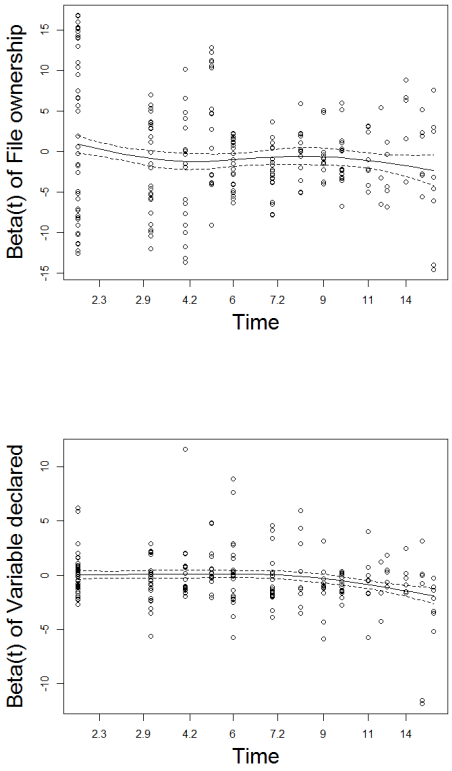


Fig. 16: Transformed scaled Schoenfeld residuals for file ownership and variables declared in the Camel application

7.2 Results

Our Cox models satisfy the linear relation between each collected metric and the change likelihood of logging statements. The scaled Schoenfeld residuals in Figure 13 show a random pattern against time except for *file ownership* and *declared variables* (similar results were found for metrics in other applications). This result suggests that *file ownership* and *declared variables* do not satisfy the linear relation between each collected metric. Hence, we transform those metrics by making them time dependent in our Cox model (i.e., we multiply *file ownership* and *declared variables* by the number of releases that a logging statement has existed) as proposed by prior research [9] and re-plot the scaled Schoenfeld residuals. Figure 16 shows the scaled Schoenfeld residuals after we transformed these two metrics and we observe that the Cox model assumptions are now validate for both metrics. Therefore both *file ownership* and *declared variables* can be used to build Cox models.

The built Cox models are a good fit for the studied applications. The Martingale residuals shown in Figure 15 are asymmetrically distributed in between (-1,1). These residuals plots show that our Cox models can accurately determine the change likelihood of long-lived logging statements and that our Cox models are a good fit for our survival data.

8 Important Metrics from Random Forest Classifiers and Cox Models

In our previous sections (i.e., Section 6 and 7), we find that a random forest classifier can accurately determine the change likelihood of a just-introduced logging statement and that Cox models are able to model the change likelihood of long-lived logging statements. Hence, in this section we analyze the random forest classifier and Cox models to better understand the important metrics which help in determining the likelihood of a logging statement changing. By analyzing the impact of the important metrics, developers can better understand what drives the random forest classifiers and Cox models and get a better understanding of the phenomenon of changing logging statements in their own projects.

8.1 Important Metrics in our Random Forest Classifier

To find the importance of each metric in a random forest classifier, we use a permutation test [42]. In this test, the classifier built using the training data is applied to the test data during bootstrapping (see Section 6.1). Then, in order to find the importance of the X_i^{th} metric, the values of the metric are randomly permuted in the test dataset and the accuracy of the classifier is recomputed [15]. The change in the accuracy 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 the *RandomForest* package of `R`, to calculate the importance of each metric. We call the *importance* function during the bootstrapping process explained in Section 6.1 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 metrics based on their effect size [45]. The SK-ESD algorithm groups metrics based on their importance in determining the change likelihood of a just-introduced logging statement. The SK-ESD algorithm uses effect sizes that are calculated using *Cohen's* delta [22], 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 as seen in Table 8.

8.2 Selecting Statistically Significant Metrics in a Cox Model

After validating the Cox model assumptions and goodness of fit, we finally identify the statistically significant metrics in our Cox models. We use the backward selection method [14] for selecting the statistically significant metrics in our Cox model since the backward selection method outperforms forward selection methods [14]. In backward selection, multiple Cox models are built and in each successive model the statistically insignificant metrics are eliminated until only statistically significant metrics are left in the final Cox model. If a metric has a p-value that is greater than 0.05, the metric is excluded from the next model while metrics that have a p-value that is less than 0.05 are retained in successive models. We use the *validate.cph()* method from the *rms* package to identify the statistically significant metrics in our Cox models.

Important Metrics in a Cox Model

To determine the importance of each metric in Cox models, we use the chunk test/ANOVA test [16]. In this test, the Cox model is constructed using the statistically significant metrics after backward selection and the chi-squared values [13] are computed using the residuals obtained from the Cox model for each metric. By ordering the metrics based on their chi-squared values, we determine the importance of each metric with respect to our Cox model, i.e., the metric with the highest chi-squared value is the most important metric in our Cox model. As chi-squared values are computed from residuals obtained from statistical models, this method is not applicable for random forest classifiers. We use the *anova* function defined in the *stat* package of `R`, to calculate the chi-squared values of each metric.

8.3 Plotting the Important Metrics

After identifying the important metrics in both random forest classifiers and Cox models, it is crucial to plot these important metrics to determine how the metrics affect the change likelihood of logging statements, i.e., to determine the direction of the effect of the metrics. To identify how a metric affects the change likelihood of a logging statement we plot each metric against the predicted likelihood of whether a just-introduced logging statement will change for random forest classifiers and against the relative risk (i.e., likelihood of logging statement changing) for Cox models.

To plot the predicted value of whether a just-introduced logging statement will change, we first construct a random forest classifier as explained in Section 6 with training data (i.e., two-thirds of entire data). Next, using the *predict* function in `R`, we determine whether a just-introduced logging statement will change for our training data set (i.e., remaining one-thirds of the data). We then plot the predicted value of whether a just-introduced logging

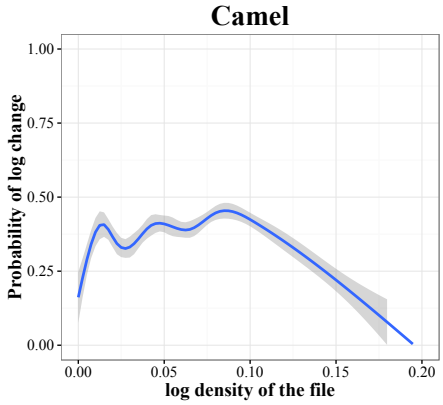


Fig. 17: Showing the change likelihood of logging statement change against log density for random forest classifiers. Change likelihood varies between 0 to 1, where '1' implies that the change likelihood of a logging statement is 100% and '0' implies that the change likelihood of a logging statement is 0%. Note that there were no files with a log density larger than 0.20

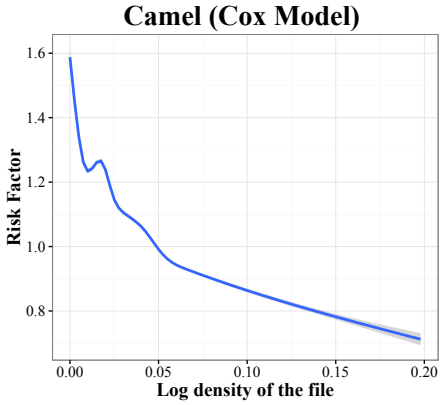


Fig. 18: Showing the relative risk of depending on a logging statement against the log density for Cox models. A relative risk value over '1' implies that these logging statements have a higher risk of being changed than those logging statements with a relative risk lower than '1'. Note that there were no files with a log density larger than 0.20

statement will change against the significant metrics for our random forest classifier. Using the *qplot* function in `R`, we fit a curve through the plotted data points by setting the *geom* variable and set the confidence interval to 95% by configuring the *span* variable in *qplot* function as seen in Figure 17.

For the Cox models we use a similar approach and use the *predict.cph* function from *rms* package in `R`, instead of *predict* as used for the random forest

Table 7: Percentage of all logging statements in an application that are 1) introduced by the top 3 developers, 2) changed by the top 3 developers and 3) the total number of developers that introduce logs in an application.

	Introduced logging statements	**Changed logging statements**	**Total # of developers**
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

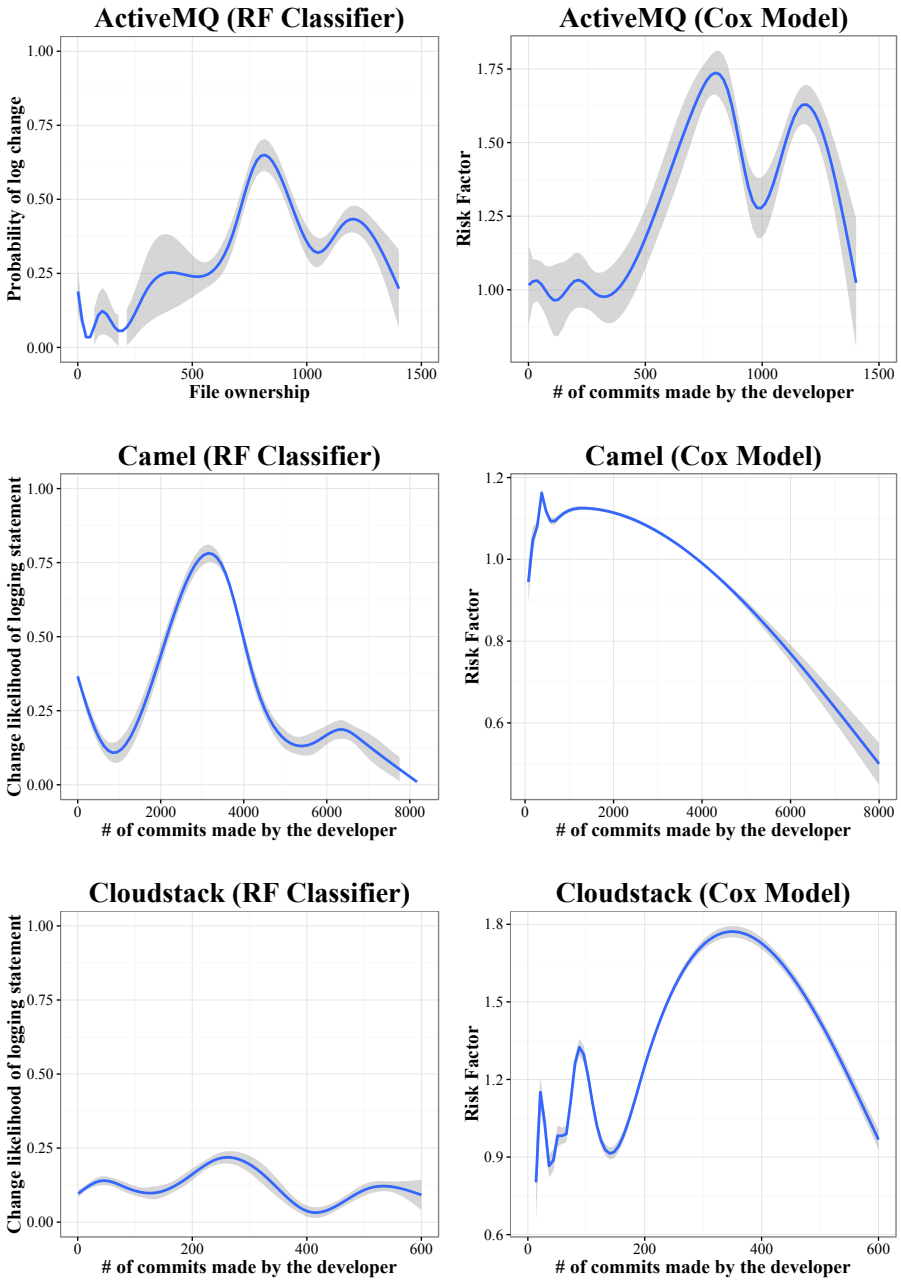
classifiers. The *predict.cph* function returns the relative risk (i.e., likelihood of logging statement changing) for each row in our test dataset and we plot the risk against each metric. Next, we fit a curve through the plotted data points using *qplot* function the same as for random forest classifiers. An example of this is seen in Figure 18, where we observe the that relative risk of a logging statement changing is higher for those logging statements present in files having lower logging density (i.e., the relative risk is higher than 1), than for logging statements in well logged files.

8.4 Results

> *Developer experience plays an important role in determining the change likelihood of both just-introduced and long-lived logging statements.*

From Table 8 and Table 9 we see that developer experience is one of the four most important metrics in our studied applications for both the random forest classifiers and Cox models for determining the change likelihood of just-introduced and long-lived logging statements. From our analysis we find that logging statements introduced by more experienced developers are more likely to be changed in both random forest classifiers and Cox models for the ActiveMQ, Cloudstack and Liferay applications. From Figure 19 we observe that with increase in developer experience (i.e., number of commits) in ActiveMQ and Cloudstack there is a drastic increase in the change likelihood of a logging statement.

However, we observe that logging statements introduced by the top developers are less likely to be changed as seen by the downward trend in Figure 19 for the ActiveMQ, Camel and Cloudstack applications. This downward trend may be explained by the fact that in the studied applications the top developers are responsible for introducing more than 59% of the logging statements as seen in Table 7 and up to 70% of the logging statements introduced by these top developers never change.


ActiveMQ (RF Classifier)
Probability of log change
1.00
0.75
0.50
0.25
0.00
0
500
1000
1500
File ownership
ActiveMQ (Cox Model)
Risk Factor
1.75
1.50
1.25
1.00
0
500
1000
1500
# of commits made by the developer
Camel (RF Classifier)
Change likelihood of logging statement
1.00
0.75
0.50
0.25
0.00
0
2000
4000
6000
8000
# of commits made by the developer
Camel (Cox Model)
Risk Factor
1.2
1.0
0.8
0.6
0
2000
4000
6000
8000
# of commits made by the developer
Cloudstack (RF Classifier)
Change likelihood of logging statement
1.00
0.75
0.50
0.25
0.00
0
200
400
600
# of commits made by the developer
Cloudstack (Cox Model)
Risk Factor
1.8
1.5
1.2
0.9
0.6
0
200
400
600
# of commits made by the developer

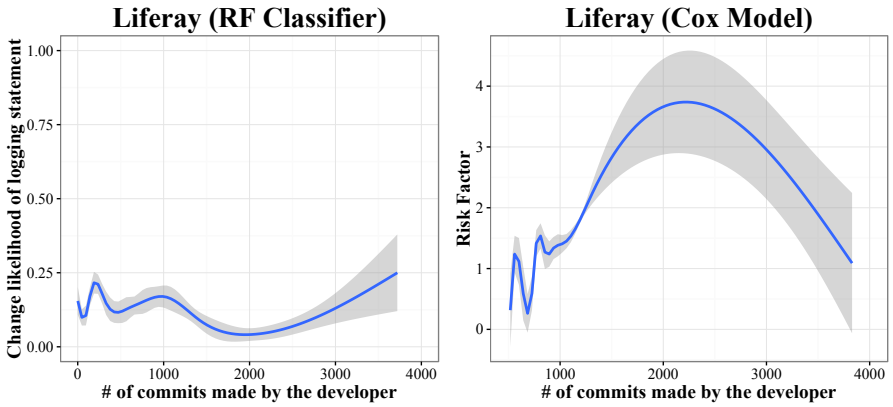
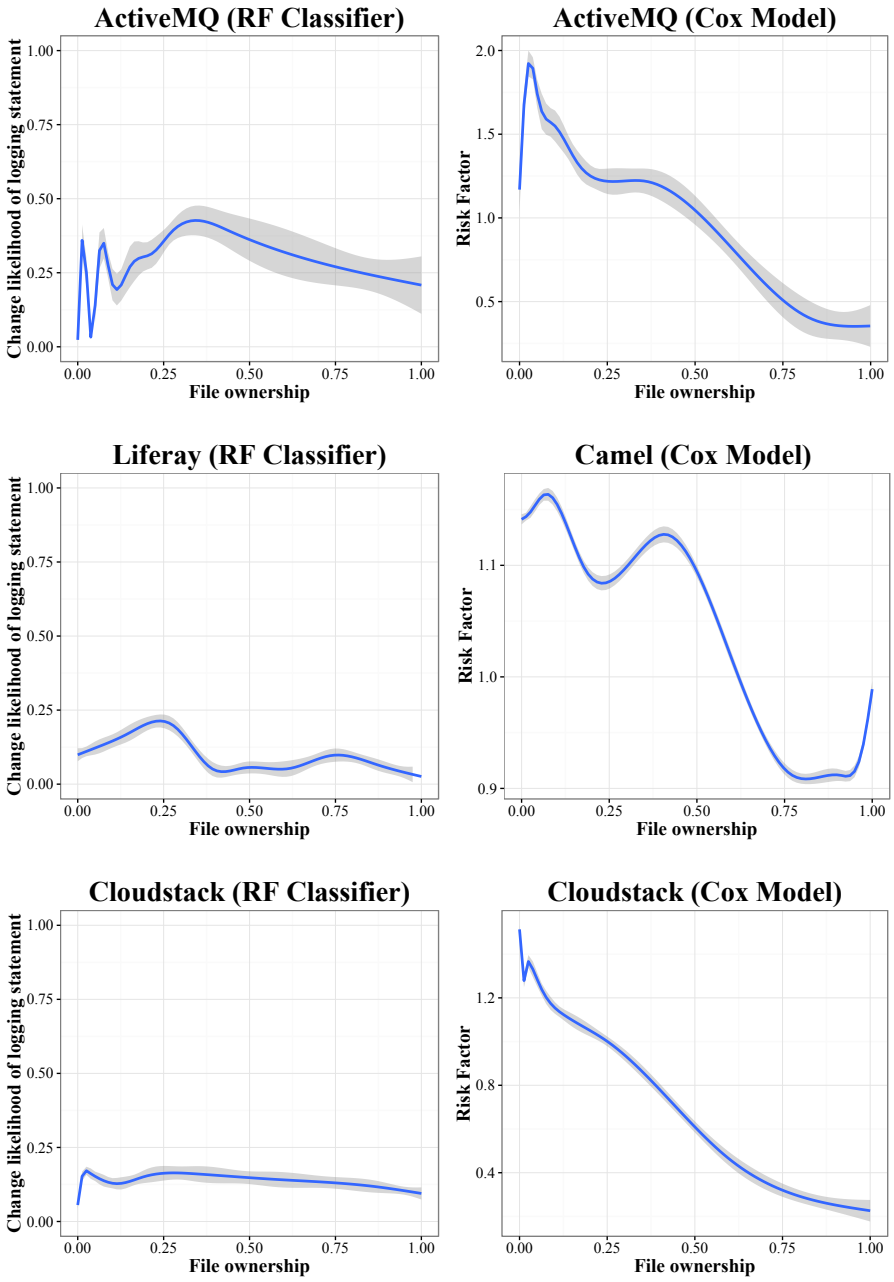


Fig. 19: Comparing the probability of changes to just-introduced and long-lived logging statement against the experience of the developer who introduces that logging statement

Table 8: The most important metrics in random forest classifier, divided into homogeneous rank groups using the Scott-Knott Effect Size clustering

	ActiveMQ			**Camel**	
Rank	**Metric**	**Importance**	**Rank**	**Metric**	**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 context	0.069	5	Log addition	0.090
6	Log level	0.063		Log density	0.088
7	Declared variables	0.048	6	Log variable count	0.063
8	Log text length	0.022	7	Log context	0.052
			8	Declared variables	0.051

	CloudStack			**Liferay**	
Rank	**Metric**	**Importance**	**Rank**	**Metric**	**Importance**
1	Log density	0.224	1	Developer experience	0.195
2	Ownership of file	0.215		Log density	0.192
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	Declared variables	0.092	6	Declared variables	0.080
9	Log context	0.061	7	Log text length	0.071


ActiveMQ (RF Classifier)
Change likelihood of logging statement
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00
File ownership
ActiveMQ (Cox Model)
Risk Factor
2.0
1.5
1.0
0.5
0.00
0.25
0.50
0.75
1.00
File ownership
Liferay (RF Classifier)
Change likelihood of logging statement
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00
File ownership
Camel (Cox Model)
Risk Factor
1.1
1.0
0.9
0.00
0.25
0.50
0.75
1.00
File ownership
Cloudstack (RF Classifier)
Change likelihood of logging statement
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00
File ownership
Cloudstack (Cox Model)
Risk Factor
1.2
0.8
0.4
0.00
0.25
0.50
0.75
1.00
File ownership

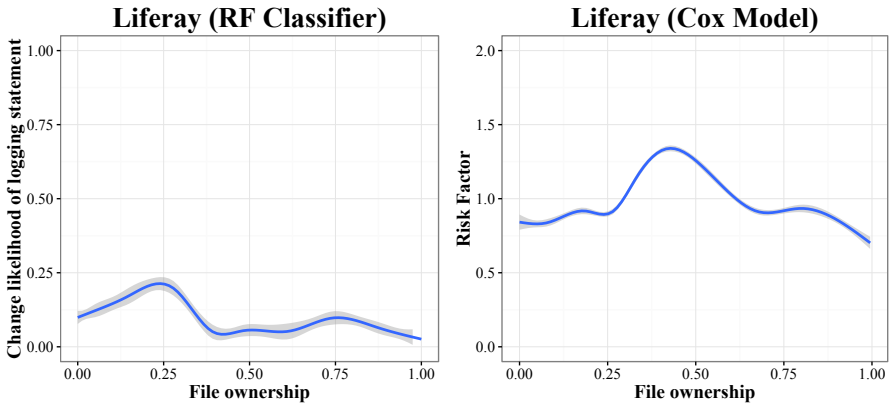


Fig. 20: Comparing the probability of changes to just-introduced and long-lived logging statement against ownership of the file

Table 9: The most important metrics in the Cox model ranked using the chunk test

	ActiveMQ			Camel	
Rank	**Metric**	**Importance**	**Rank**	**Metric**	**Importance**
1	Log variable count	12.55	1	Log density	4.05
2	File Ownership	11.55	2	Declared variables	3.37
3	Declared variables	8.73	3	File Ownership	3.10
4	Developer experience	4.82	4	Log variable count	0.29

	CloudStack			Liferay	
Rank	**Metric**	**Importance**	**Rank**	**Metric**	**Importance**
1	Log variable count	85.31	1	Log text length	41.01
2	SLOC	85.12	2	SLOC	20.79
3	File ownership	65.02	3	Developer experience	8.38
4	Developer experience	47.96	4	Total revision count	4.85
5	Code churn in commit	20.96	5	Log density	4.59
6	Log text length	8.73			
7	Log density	0.63			

Logging statements that are introduced by owners of the file are unlikely to be changed in the future.

From Table 8 and Table 9, we see that file ownership (i.e., the number of lines of code in a file that are introduced by a developer) is one of the top four metrics after developer experience in determining the change likelihood of just-introduced and long-lived logging statements. From Figure 20 we observe in three of the studied applications, 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 15% of the file are responsible

for 27%-67% of the changes to logging statements in the studied applications, which is seen as an upward trend from 0 to 0.15 in Figure 20 for the ActiveMQ, Camel and Cloudstack applications. 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 15% of the file.

> *Logging statements in files with a low log density (i.e., files having a lesser number of logging statements per line of code) are more likely to change than logging statements in files with a high log density.*

Log density is defined as the number of logging statements to the total lines of code within a file. From Table 8 and Table 9, we observe that log density has the highest importance in the Liferay and Cloudstack applications for just-introduced logging statements and in the Camel application for long-lived logging statements. We find that in random forests, changes to just-introduced 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 with a higher change likelihood are present in files with a significantly higher SLOC (2×-3× higher) than logging statements with a lower change likelihood. We find similar results for the Cox models where long-lived logging statements have a higher change likelihood in files where log density reduces over time than in files where log density increases over time.

Though there are common patterns observed across different applications, our analysis reveals that some metrics are important only in few of the studied applications. For example, from Table 8 and Table 9, we observe that the number of variables logged is ranked in the top 3 for determining the change likelihood of long-lived logging statements. However, for just-introduced logging statements the same metric is ranked in the bottom 3 of the table. This discrepancy suggests that, developers of log processing tools should analyze their respective applications and produce risk tables from their analysis at each release of the application.

9 Threats to Validity

External Validity. Our empirical study is performed on Liferay, ActiveMQ, Camel and CloudStack. Though these studied applications have years of history and a large number of developers, these applications are all Java-based. Other languages may not use logging statements as extensively. More studies on other domains, with other programming languages are needed to see whether our findings can be generalized.

Our approach works well on projects that maintain a source code history, and explicitly specify logging statements in their code that can be extracted using a regular expression (as explained in Section 4.1). To employ our approach

on projects that dynamically generate logging statements, a more advanced data collection method (e.g., using an abstract syntax tree) is needed.

We extract the necessary metrics at every official release of an application. The official release excludes minor releases, release candidates and any other 'hotfixes'. This precautionary measure is taken to ensure that we only consider logging statement changes which occur in different releases for our Cox models. However, if a logging statement is introduced and changed within the same release, the change is not collected as such a change would not affect log processing tools.

Construct Validity. In our study, we only explore the first change after the introduction of a logging statement. While the first change is sufficient for deciding 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 study to give more specific details about stability of logs (i.e., how likely will a changed log be changed again and why do some logging statements exhibit a large number of changes in their lifetime).

Our heuristic for matching logging statements in code also matches logging statements that are inside code comments. We verified the matched logging statements and found that less than 1% of the logging statements in each studied application was inside code comments. Therefore, such logging statements will not affect the results of our study.

Internal Validity. Our study is based on the data that is collected from the Git repositories of all the studied applications. The quality of the data that is contained in the repositories can impact the internal validity of our study. For example, rewriting the history of the repository (i.e., by *rebasing* the history) may affect our results [2].

Our results are not impacted by any threshold of Levenshtein distance used in identifying the modification of logging statements.

We collect and study only a subset of the available metrics in our random forest classifiers and Cox models. More metrics should be leveraged and studied in depth in future studies.

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

Our study utilizes two-thirds of the training data for drawing the plots for risk factors for Cox models and change likelihood for random forest classifiers. More exhaustive methods should be used in future studies to use all the training data for drawing the plots.

10 Conclusion

Logging statements are snippets of code, introduced by developers to yield valuable information about the execution of an application. Logging state-

ments 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 select more stable logging statements by providing early advice about the stability of a logging statement. We consider our work an important first step towards helping developers to construct 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 (or at least factor such instability into the maintenance costs of their log processing tools). The highlights of our work are:

- We find that 20%-45% of the logging statements are changed at least once.
- We find that our random forest classifier for determining the change likelihood of a just-introduced logging statement achieves a precision of 83%-91%, a recall of 65%-85% and an AUC of 0.95-0.96.
- We find that just-introduced and long-lived logging statements added by a developer who owns more than 75% of a file are less likely to change in the future in our studied applications.
- Well-logged files are less likely to have changes to both just-introduced and long-lived logging statements in our studied applications.
- We find that our random forest classifiers and Cox models show that developer experience, file ownership, log density and SLOC play an important role in determining the change likelihood of both just-introduced and long-lived logging statements in our studied applications.

Our findings help in determining the likelihood of whether a logging statement will change and calculate the relative stability of using a particular logging statement. Developers of tools that process logs of proprietary software often do not have access to the source code history of that software. Hence, they cannot directly apply our approach as described in this paper. A possible solution is that proprietary software vendors use our approach as described and publish the risk factors, i.e., the change likelihood, for each log line with every release of their software. These risk factors can then be used to build robust log processing tools.

Developers of log processing tools can use the knowledge about the change likelihood of logging statements for conducting *preventative* analysis by calculating the relative stability of each logging statement in a release and selecting the most stable logging statements when building their log processing tools. If developers have no alternative choices but have to use unstable logging statements, they can leverage the risk factors calculated from survival analysis techniques for *proactive analysis*, such that they are more aware of the risks of using that logging statement in their log processing tools.

Table 10: Data for survival analysis

ID	Start	Stop	Log change (event)	Number of logged variables
$Log-1$	0	1	0	3
$Log-1$	1	2	1	1
$Log-2$	0	1	0	3
$Log-2$	1	2	0	4

Appendix A: Background on Survival Analysis

Survival analysis comprises a set of statistical modeling techniques that model the time taken for an event to occur [31]. These modeling techniques can be parametric, semi-parametric or non-parametric in form. However, they share the common goal of modeling the time that it takes between the start of an observation period (i.e., logging statement introduction) and an event (i.e., logging statement change) to occur i.e., they model the survival time of a logging statement. Survival analysis also helps in identifying the important metrics that affect the survival time of a logging statement. The following section discusses the crucial aspect of survival analysis as described in [43]: survival analysis data and measuring time to event.

Survival Analysis Data and Measuring Time to Event

Survival analysis uses the data that is collected at specific time intervals to observe the relation between how a subject changes over time and the occurrence of an event of interest (e.g., whether a log statement changes). We explain survival analysis using the stability of logging statements as an example.To model the time to change of a logging statement, we collect the data about content, context and developers (metric) for each release (observation period) after a logging statement (subject) is introduced into the application. Each observation in the survival data contains the following fields:

1. UID: Unique number of each logging statements.
2. Start: Time of introduction of a logging statement.
3. Stop: the time at which the logging statement changes.
4. Event: (1) if the logging statement was changed or (0) if the logging statement was not changed at the end of observation period.
5. Metrics: The content, context and developer metrics.

Table 10 shows the survival data for a logging statement (Log-1), where the observations are recorded at the beginning of a release. If a logging statement is changed (event occurs), the logging statement is not tracked and the study halted for that particular logging statement. However, some logging statements may never be changed and in such cases it is impractical to track them. Hence,

the logging statements are tracked for a certain period of time (e.g., 3 years), during which they may or may not be changed.

To conduct the survival analysis we need to define how we measure the *introducing event* (i.e., the first release after introduction of a logging statement), the *censored event* (i.e., the subsequent months where the logging statement is not changed) and the *terminating event* (i.e., month the logging statement is changed). From Table 10, we find that the logging statement is changed in the second release, which makes it the *terminating event*. In the prior releases, the event of interest does not occur which makes the observations *censored events*. In addition, when a logging statement is not changed during the period of study (i.e., 3 years), their survival is considered equal to the period of study. We include both *censored* and *terminating* events for our survival analysis as the models can handle both *censored* and *terminating* events and can produce effective survival models without bias [17].

References

1. Luca Bigliardi, Mario Lanza, Alberto Bacchelli, Marco D'Ambros, and Andrea Mocci. Quantitatively exploring non-code software artifacts. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 286–295. IEEE, 2014.
2. Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE, 2009.
3. Jerome Boulon, Andy Konwinski, Runping Qi, Ariel Rabkin, Eric Yang, and Mac Yang. Chukwa, a large-scale monitoring system. In *Proceedings of Cloud Computing and its Applications*, volume 8, pages 1–5, 2008.
4. David Carasso. Exploring splunk. *published by CITO Research, New York, USA, ISBN*, pages 978–0, 2012.
5. Jacob Cohen, Patricia Cohen, Stephen G. West, and Leona S. Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013.
6. David Collett. *Modelling survival data in medical research*. CRC press, 2015.
7. Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 139–150, 2015.
8. Chris Elbers and Geert Ridder. True and spurious duration dependence: The identifiability of the proportional hazard model. *The Review of Economic Studies*, 49(3):403–409, 1982.
9. Lloyd D. Fisher and Danyu Y. Lin. Time-dependent covariates in the cox proportional-hazards regression model. *Annual review of public health*, 20 (1):145–157, 1999.

10. Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Louand Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao 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,*, pages Pages 24–33.
11. Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the ICDM 2009, Ninth IEEE International Conference on Data Mining*, pages 149–158. IEEE, 2009.
12. Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.
13. Priscilla E. Greenwood and Michael S. Nikulin. *A guide to chi-squared testing*, volume 280. John Wiley & Sons, 1996.
14. Frank Harrell. *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer, 2015.
15. Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.
16. David C. Hoaglin and Roy E. Welsch. The hat matrix in regression and anova. *The American Statistician*, 32(1):17–22, 1978.
17. David W. Hosmer Jr. and Stanley Lemeshow. Applied survival analysis: Regression modelling of time to event data (1999), 1999.
18. George Hripcsak and Adam S. Rothschild. Agreement, the f-measure, and reliability in information retrieval. *Journal of the American Medical Informatics Association*, 12(3):296–298, 2005.
19. Ross Ihaka and Robert Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.
20. Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E. Hassan. Examining the stability of logging statements. In *SANER 2016: Proceedings of IEEE International Conference on the Software Analysis, Evolution and Re-engineering,*. IEEE, 2016.
21. Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E. Hassan Hassan. Logging library migrations: A case study for the apache software foundation projects. *Mining Software Repositories*, page To appear, 2016.
22. Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag IK. Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11):1073–1086, 2007.
23. Maurice George Kendall. Rank correlation methods. 1948.
24. A. Gunes Koru, Dongsong Zhang, and Hongfang Liu. Modeling the effect of size on defect proneness for open-source software. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 10. IEEE Computer Society, 2007.

25. Heng Li, Weiyi Shang, and Ahmed E. Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, page To appear, 2016.
26. Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, page To appear, 2016.
27. Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association.
28. Haroon Malik, Hadi Hemmati, and Ahmed E. 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*, pages 1012–1021, May 2013.
29. Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 2015.
30. Martins Mednis and Maike K. Aurich. Application of string similarity ratio and edit distance in automatic metabolite reconciliation comparing reconstructions and models. *Biosystems and Information technology*, 1(1): 14–18, 2012.
31. Rupert G. Miller Jr. *Survival analysis*, volume 66. John Wiley & Sons, 2011.
32. Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. Industry practices and event logging: Assessment of a critical software development process. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 169–178. IEEE Press, 2015.
33. Hening Ren, Ximing Tang, J. Jack Lee, Lei Feng, Allen D. Everett, Waun Ki Hong, Fadlo R. Khuri, and Li Mao. Expression of hepatoma-derived growth factor is a strong prognostic predictor for patients with early-stage non–small-cell lung cancer. *Journal of Clinical Oncology*, 22 (16):3230–3237, 2004.
34. C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979. ISBN 0408709294.
35. Robert J. Serfling. *Approximation theorems of mathematical statistics*, volume 162. John Wiley & Sons, 2009.
36. Weiyi Shang. Bridging the divide between software developers and operators using logs. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1583–1586. IEEE, 2012.
37. Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.

38. Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.
39. Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.
40. Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan, and Zhen Ming Jiang. Understanding log lines using development knowledge. In *Proceedings of ICSME 2014, The International Conference on Software Maintenance and Evolution,*, pages 21–30. IEEE, 2014.
41. Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.
42. Carolin Strobl, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis. Conditional variable importance for random forests. *BMC bioinformatics*, 9(1):307, 2008.
43. Mark Syer, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. Replicating and re-evaluating the theory of relative defect-proneness. *IEEE Transactions on Software Engineering,*, 41(2):176–197, Feb 2015.
44. Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. In *WASL'08: Proceedings of the 1st USENIX Conference on Analysis of System Logs*, pages 6–6. USENIX Association, 2008.
45. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi 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).
46. Terry M. Therneau, Patricia M. Grambsch, and Thomas R. Fleming. Martingale-based residuals for survival models. *Biometrika*, 77(1):147–160, 1990.
47. Log4j Last visited March'16. URL `http://logging.apache.org/log4j/2.x/`.
48. Xpolog. URL `http://www.xpolog.com/`.
49. Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SOPS 2009, 22nd symposium on Operating systems principle*, pages 117–132.
50. Xiwei Xu, Ingo Weber, Len Bass, Liming Zhu, Hiroshi Wada, and Fei 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*, page 5. ACM, 2013.

51. Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *in OSDI 2012, USENIX Symposium on Operating Systems Design and Implementation*, pages 293–306.
52. Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *Proceedings of ASPLOS 2011, The 16th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–14, 2011.
53. Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of ICSE 2012, The 34th International Conference on Software Engineering*, pages 102–112. IEEE Press, 2012.
54. Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering,*, 35(2):293–304, 2009.
55. Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of ICSE 2015, The 37th International Conference on Software Engineering - Volume 1*, pages 415–425, Piscataway, NJ, USA, 2015. IEEE Press.