Studying the Practices of Logging Exception Stack Traces in Open-Source Software Projects

Heng Li, Haoxiang Zhang, Shaowei Wang, and Ahmed E. Hassan, *Fellow, IEEE*

Abstract—Logging the stack traces of runtime exceptions assists developers in diagnosing runtime failures. However, unnecessary logging of exception stack traces can have many negative impacts such as polluting log files. Unfortunately, there exist no guidelines for the logging of exception stack traces and developers usually practice it in an *ad hoc* manner. In this work, we perform a comprehensive study of the source code, code change history, and issue reports of ten open-source Java projects, combining quantitative and qualitative analysis, in order to understand how developers log and modify the logging of exception stack traces, their rationale for logging or not logging exception stack traces, and the factors that impact their logging of exception stack traces. We observe that logging of exception stack traces is a popular practice in open-source projects, while developers have difficulties making appropriate logging of exception stack traces in the first place. Through a qualitative analysis of 385 related issue reports, we derived recommendations for the logging of exception stack traces, such as logging of stack traces should be avoided or downgraded for user errors, normal execution, expected exceptions, in user interfaces, or when there is a security concern. Finally, based on our empirical observations, we design and extract a set of code metrics and construct models to explain the likelihood of logging an exception stack trace. Our analysis of the models indicates the important factors (e.g., the exception type and the method that throws the exception) for determining the logging of exception stack traces. Our study helps developers and researchers understand the current practices of logging exception stack traces, provides recommendations for developers to consider when determining whether to log the stack trace of an exception, and provides insights for future research and practices to derive global or company-wide guidelines for the logging of exception stack traces.

Index Terms—Software maintenance, software logging, exception logging, stack traces, random forest

1 Introduction

The stack trace of an exception provides the trace of method calls from the start of the application (or the thread in a multi-threaded context) execution to the point where the exception is triggered. Thus, the logged stack trace of an exception is an important tool for diagnosing runtime failures [20], [52]. Fig. 1 shows a log message with a stack trace as an example. On the other hand, as a stack trace is usually much longer (e.g., more than ten times longer) than a regular log message, stack traces can usually grow log files very quickly and result in excessive logging, which in turn would cause performance bottlenecks [14], [60], raise the storage costs of logs [5], and lead to increased effort on log management and analysis [34].

Modern logging libraries (e.g., Log4j[1] and SLF4J[2]) support convenient ways to log the stack trace of an exception in a logging statement within the scope of an exception (i.e., inside the *catch* block that catches the exception). In this paper, we refer to a logging statement inside a *catch* block as an *exception logging statement*. For example, in the code snippet shown in Fig. 2, the stack trace of an exception is logged by specifying the exception object as the last parameter of an exception logging statement. In particular, the logging statement in this example produces the log message shown in Fig. 1.

However, we observe that developers sometimes have issues making appropriate decisions for logging the stack traces of exceptions [34]. For example, issue report HADOOP-10571[3] proposes the addition of stack traces to many exception logging statements across several modules. However, other developers raise concerns that the logging of stack traces should be avoided for some of these exception logging statements. As a result, it takes significant efforts (e.g., seven developers involved, 30 comments, and 10 patches) to resolve the raised concerns. Table 1 lists examples of another six issue reports that are concerned with whether to log the stack trace of an exception. Three of the issue report examples request the addition of missing stack traces in exception logging statements, while the other three issue report examples request the removal of existing stack traces from exception logging statements. Most of the issues can be fixed by a few lines of code changes. However, it often takes a long time (e.g., years) from

- *Heng Li is with the Department of Computer Engineering and Software Engineering, Polytechnique Montreal, Montreal, QC H3T 1J4, Canada. E-mail: heng.li@polymtl.ca.*
- *Haoxiang Zhang is with the Centre for Software Excellence at Huawei, Markham, ON L3R 5A4, Canada. E-mail: haoxiang.zhang@acm.org.*
- *Shaowei Wang is with the Department of Computer Science, University of Manitoba, Winnipeg, MB R3T 2N2, Canada. E-mail: shaowei@cs.umanitoba.ca.*
- *Ahmed E. Hassan is with Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: ahmed@cs.queensu.ca.*

Manuscript received 21 May 2021; revised 12 Oct. 2021; accepted 12 Nov. 2021. Date of publication 22 Nov. 2021; date of current version 12 Dec. 2022. (Corresponding author: Heng Li.) Recommended for acceptance by L. Mariani. Digital Object Identifier no. 10.1109/TSE.2021.3129688

1. https://logging.apache.org/log4j/2.x/
2. https://www.slf4j.org
3. https://issues.apache.org/jira/browse/HADOOP-10571

0098-5589 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

```
2016−02−10 02:15:43,175 [WARN] Exception when evaluating predicate. Skipping ORC PPD. Exception:   Regular log message
java.lang.IllegalArgumentException: ORC SARGS could not convert from String to TIMESTAMP
    at org.apache.hadoop.hive.ql.io.orc.RecordReaderImpl.getBaseObjectForComparison(RecordReaderImpl.java:659)
    at org.apache.hadoop.hive.ql.io.orc.RecordReaderImpl.evaluatePredicateRange(RecordReaderImpl.java:373)
    at org.apache.hadoop.hive.ql.io.orc.RecordReaderImpl.evaluatePredicateProto(RecordReaderImpl.java:338)
    at org.apache.hadoop.hive.ql.io.orc.RecordReaderImpl$SargApplier.pickRowGroups(RecordReaderImpl.java:710)
    at org.apache.hadoop.hive.ql.io.orc.RecordReaderImpl.pickRowGroups(RecordReaderImpl.java:751)
    at org.apache.hadoop.hive.ql.io.orc.RecordReaderImpl.readStripe(RecordReaderImpl.java:777)
                                                                                              Stack
    ... (The stack trace is truncated to save space. The original stack trace is 43 lines long.)  trace
```

Fig. 1. An example of a log message which outputs a stack trace (check https://issues.apache.org/jira/browse/HIVE-13325 for the full log message). The log message is produced by the logging statement shown in Fig. 2.

when a logging statement was originally introduced in the source code until when the stack trace issue was fixed, which can pose long-term impact to developers and users. These examples motivate our study to help developers understand and improve their practices of logging stack traces.

Prior work performed extensive studies on software logging practices, including studying where to log [25], [32], [33], [60], [62], [63], what to log [23], [40], how to choose log levels [35], [37], the characteristics and evolution of logging [8], [27], [28], [55], [61], and logging anti-patterns [9], [21]. However, there exists no work that studies the logging of exception stack traces, which we believe is equally important as the previously studied aspects of logging. In addition, there exists no guidelines or standards for developers to follow for the logging of exception stack traces.

Therefore, in this work, we study the practices of logging exception stack traces in ten open-source projects across different domains. We perform a comprehensive investigation of the source code, code change history, and issue reports of these studied projects, combining quantitative and qualitative analysis, in order to understand how developers log and modify the logging of exception stack traces, their rationale for logging or not logging exception stack traces, and the factors that impact their logging of exception stack traces. Specifically, we structure our investigation along the following three research questions (RQs):

RQ1: *How do developers log and modify the logging of exception stack traces?* We analyze the source code and code change history to understand how developers log and modify the logging of exception stack traces. We observe that logging of exception stack traces is a popular practice in open-source projects, while developers have difficulties making appropriate logging of exception stack traces in the first place. In particular, some exceptions (e.g., generic exceptions) are more likely than other exceptions to be logged with a stack trace, while developers make the most changes to adjust the logging of stack traces at the *warn* and *error* levels.

RQ2: *Why do developers log or not log exception stack traces?* In order to understand developers' rationale for making their decisions of logging exception stack traces or not, we qualitatively examine 385 related issue reports raised in the development history of the studied projects. Our analysis derives several recommendations for the logging of exception stack traces, including that logging of stack traces should be avoided or downgraded for user errors, normal execution, expected exceptions, in user interfaces, or when there is a security concern. On the other hand, the logging of stack traces is recommended for generic exceptions, severe problems, and unexpected exceptions.

RQ3: *Which factors impact the logging of exception stack traces?* Based on our insights from RQ1 an RQ2, we design a set of code metrics and build a model to explain the likelihood of logging exception stack traces. Our results suggest that the exception type and the method that throws the exception should be considered as the primary factors for determining the logging of exception stack traces. The log level and how well the exception is handled should also be considered when making such decisions.

This work provides a comprehensive picture of developers' practices of logging exception stack traces, through examining the source code, code change history, and issue reports of ten open-source projects. Our findings make four important contributions:

- Our study helps developers and researchers understand the current practices of logging exception stack traces. In particular, as the first study in this under-investigated area, our work will raise the research community's attention in helping developers improve their practices of logging exception stack traces.
- Our findings provide recommendations for developers to consider when determining whether to log the stack trace of an exception.
- We developed models to guide developers' decisions of logging exception stack traces and understand what factors impact their decisions.
- Our findings provide insights for future research and practices to derive global or company-wide guidelines for the logging of exception stack traces.

Paper Organization. The remainder of the paper is organized as follows. Section 2 describes the setting of our

```
try {
  // Code in the try block omitted.
} catch (Exception e) {
  LOG.warn("Exception when evaluating predicate. Skipping ORC
      PPD. Exception:", e);
}
```

Fig. 2. Example of logging the stack trace of an exception by specifying the exception (e) as the last parameter of a logging statement.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Examples of Issue Reports of the *Hadoop* Project That are Concerned With Whether to Log the Stack Trace of an Exception

Issue ID[1]	Issue Report Title	Issue Request Type	Date Log Introduced	Date Stack Trace Fixed	LOC Changed for Fixing
HADOOP-14481	Print stack trace when native bzip2 library does not load	Add stack trace	06/Mar/13	02/Jun/17	2
HADOOP-13458	LoadBalancingKMSClientProvider#doOp should log IOException stacktrace	Add stack trace	26/Feb/15	03/Aug/16	4
HADOOP-10562	Namenode exits on exception without printing stack trace in AbstractDelegationTokenSecretManager	Add stack trace	09/Feb/10	01/May/14	3
HADOOP-14418	Confusing failure stack trace when codec fallback is happend	Remove stack trace	27/Apr/17	14/May/17	3
HADOOP-13710	Supress CachingGetSpaceUsed from logging interrupted exception stacktrace	Remove stack trace	12/Apr/16	12/Oct/16	3
HADOOP-11868	Invalid user logins trigger large backtraces in server log	Remove stack trace	04/May/12	21/Apr/15	6

[1]*For more details about each issue, one can refer to its web link which is "https://issues.apache.org/jira/browse/" followed by the issue ID. For example, the link for the first issue is "https://issues.apache.org/jira/browse/HADOOP-HADOOP-13711".*

experiments, including our studied projects and our overall approach for studying the practices of logging exception stack traces. Section 3 presents our experimental results for answering each of our research questions. Section 4 discusses the threats to the validity of our findings. A survey of related work is presented in Section 5. Finally, Section 6 draws conclusions based on our presented findings.

2 Case Study Setup

2.1 Studied Projects

In this work, we study ten open-source Java projects. We selected our subject projects based on three criterion: 1) the project should make extensive use of logging, so that we have enough data to study the logging practices in the project; 2) the project should be successful and mature (i.e., with years of development history), so that we can capture state-of-the-art logging practices in successful long-lived projects; 3) the selected projects should cover a variety of types, such that our findings are not limited to a particular project type. Table 2 provides an overview of our studied projects. These projects are of different types, including *distributed computing*, *network server*, *distributed data storage*, *data streaming platform*, *message broker*, *integration framework*, *cloud computing framework*, and *testing framework*. From the user interface point of view, our studied projects include desktop applications (e.g., JMeter), web applications (e.g., CloudStack), and frameworks (e.g, Hadoop). Our selected projects are also extensively studied in prior work on logging (e.g., [8], [23], [34], [40]).

TABLE 2
Overview of Our Studied Projects

Project	Type	#Rel.[1]	Studied rel. (Rel. time)	SLOC[2]
Hadoop	Distributed Computing	297	3.1.1 (2018.08)	2,906 K
Directory Server	Network Server	50	2.0.0.AM25 (2018.08)	245 K
Hive	Distributed Data Storage	45	3.1.0 (2018.07)	1,721 K
ZooKeeper	Distributed Data Storage	88	3.4.13 (2018.07)	127 K
Kafka	Data Streaming Platform	92	2.0.0 (2018.07)	327 K
Qpid Broker	Message Broker	24	6.1.7 (2018.08)	415 K
ActiveMQ	Message Broker	59	5.15.5 (2018.08)	465 K
Camel	Integration Framework	127	2.20.4 (2018.07)	1,345 K
CloudStack	Cloud Computing Platform	153	4.11.1.0 (2018.07)	1,195 K
JMeter	Testing Framework	31	5.0 (2018.09)	282 K

[1]*All releases (major, minor and patches) are counted and analyzed in our change history analysis (see Section 2.4).*
[2]*SLOC is calculated by the CLOC tool (http://cloc.sourceforge.net).*

Table 2 shows the SLOC (Source Lines of Code) of the studied projects. The SLOC of our studied projects range between 127 K to 2,906 K. *Hadoop* is the largest studied project while *ZooKeeper* is the smallest. All of our studied projects are primarily developed in Java. In this work, we only consider the Java source code of the studied projects. We also exclude the testing code, because developers follow different logging practices in their testing code [33].

2.2 Overview of Our Study

Fig. 3 shows an overview of our study for investigating the practices of logging exception stack traces in open-source software projects. Overall, we study the source code, change history, and issue reports of the studied projects to understand the practices of logging exception stack traces. Table 2 shows the studied release and the releasing time of each of the studied project. From the version control system (i.e., GitHub[4] in our case), we obtain the source code and the code change history of the specific release of each of the studied projects (i.e., using `git clone <repository>` and `git checkout <release>`). From the source code of each studied project, we employ static code analysis (see Section 2.3) to extract the exception logging statements and the context information (e.g., the exception type) from the source code. From the code change history of each studied project, we perform change history analysis (see Section 2.4) to extract the changes to the exception logging statements. From the issue tracking system of each studied project, we collect the issue reports that are related to the logging of exception stack traces (see Section 2.5).

The results of the static code analysis (i.e., the exception logging statements and the context information) and the change history analysis (i.e., the changes to the exception logging statements) are used to answer our RQ1 (Section 3.1). The results of our issue report analysis are used to answer our RQ2 (Section 3.2). Finally, based on the the results of the static code analysis and the insights provided in RQ1 and RQ2, we design and extract a set of code metrics and build a machine learning model to understand the factors that impact the logging of exception stack traces, i.e., RQ3 (Section 3.3).

2.3 Static Code Analysis

We analyze the source code of the studied projects to extract the exception logging statements and the context information.

4. https://github.com

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

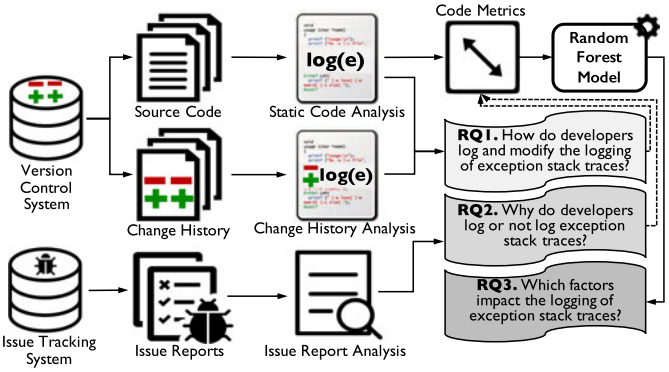

Fig. 3. Overview of our study for investigating the practices of logging exception stack traces.

We developed an IntelliJ plugin[5] based on the IntelliJ Platform SDK to perform our static code analysis. The plugin automatically extracts the context information (e.g., exception type) of each exception logging statement in a project. Specifically, for each studied project, the plugin first locates all the *catch* blocks in the Java source files. For each *catch* block, the plugin then locates the logging statements within the *catch* block (i.e., exception logging statements) by searching all the method calls that match a predefined regular expression for logging statements (shown in Fig. 4). For each located exception logging statement, the plugin then extracts detailed information about the logging statement (e.g., the log level, and whether a stack trace is logged) and the contextual information about the logging statement (e.g., the type of the caught exception, and whether the logging statement resides inside a loop).

2.4 Change History Analysis

We analyze the studied projects' code change history to extract the changes to the logging of exception stack traces.[6] For each studied project, we use the *git diff*[7] command to extract the detailed code changes of each commit in the history of that project. We consider the entire development history of each studied project that is available in its GitHub repository until the release specified in Table 2. Table 3 shows the statistics of the code changes of the studied projects. In the studied projects, 2.85% to 11.79% of the commits contain log changes that involve exception stack traces. For every thousand lines of code changes, there are 0.3 to 2.5 log changes that involve exception stack traces; although the percentages are small, the impact of such changes can be important to the maintenance and operations of software systems, as the issue report examples in Table 1 have shown.

We use the *word-diff* option to extract the code changes at the word level instead of at the line level. Fig. 5 shows an example of the results of applying the *word-diff* on a log change. In the example, the log level of a logging statement was changed from *warn* (i.e., "[-warn-]") to *debug* (i.e., "{+debug+}"), and an exception stack trace was added to the logging statement (i.e., "{+, e+}").

Regex: ".*log .*\\.(trace|debug|info|warn|error|fatal)"
Matched example: Logger.warn(...);

Fig. 4. Our regular expression (case-insensitive) used to match logging statements.

To understand developers' challenges of logging exception stack traces, we focus on the log changes that modify the logging of exception stack traces. Based on the *word-diff* results, we use predefined regular expressions to match the code changes that modify the logging of stack traces. For example, we use the regular expression shown in Fig. 6 to match the scenario of adding a stack trace to a logging statement. We consider all the scenarios of changing the logging of exception stack traces, including adding or removing stack traces from existing logging statements and changing the log level of the logging statements with stack traces. Table 4 provides an example for each scenario of changing the logging of exception stack traces.

2.5 Issue Report Collection

In order to understand developers' reasons for logging or not logging exception stack traces, we manually investigated the issue reports from the subject projects that are related to the logging of stack traces. Nine out of the ten subject projects use JIRA[8] as their issue tracking systems, except that JMeter uses Bugzilla[9] as its issue tracking system. Besides, we only find two issue reports of JMeter that are related to the logging of stack traces. Therefore, for the convenience of extracting and analyzing the issue report data, we exclude JMeter from our analysis of issue reports.

We extracted the issue reports from the Apache JIRA issue tracking system.[10] We study the issue reports that were created until May 17th, 2020, the time when we extracted the data. We used the JIRA Query Language (JQL) to automatically search for the JIRA issues reports that are related to the logging of exception stack traces. Specifically, we used the JQL query in Fig. 7 to search for the related issue reports of each of the studied projects. The "<Project Name>" is replaced with the specific project name (e.g., Hive) for each of the studied projects. This JQL query searches for all the issue reports of the specified project that have "log", "print" , "output" , or "write", and "exception" in its summary, or have "stack trace", "exception trace", or "stacktrace" in its summary, sorted by their creation time using a reverse-chronological order.

We detail our qualitative analysis of the collected issue reports in RQ2 (Section 3.2).

3 Research Questions and Results

In this section, we present the results for our RQs. For each RQ, we discuss our motivation, our approaches, and the detailed experimental results.

5. We share the source code repo of the plugin at: https://github.com/mooselab/logging-observer (GitHub username: loggingobserverguest, password: guest-passwd-21). We will make the repo public once the paper is accepted for publication.
6. We share our code for the change history analysis at: https://github.com/mooselab/logging-history-analyzer (GitHub username: loggingobserverguest, password: guest-passwd-21). We will make the repo public once the paper is accepted for publication.
7. https://git-scm.com/docs/git-diff
8. https://www.atlassian.com/software/jira
9. https://www.bugzilla.org
10. https://issues.apache.org/jira

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Statistics of the Code Changes of the Subject Projects in the Studied Periods

Project	Comm.	Comm.Log	Comm.ST	CodeChg.	LogChange	STChange
Hadoop	18.5K	4.8K(26.06%)	1,308(7.06%)	4,966.6K	33.1K(0.67%)	3,677(0.07%)
Directory Server	9.5K	1.2K(12.48%)	309(3.24%)	2,143.2K	11.7K(0.55%)	1,212(0.06%)
Hive	12.2K	2.8K(23.01%)	930(7.61%)	12,917.7K	21K(0.16%)	3,801(0.03%)
ZooKeeper	1.3K	0.4K(26.88%)	157(11.79%)	375.1K	5K(1.34%)	921(0.25%)
Kafka	5.2K	0.6K(11.03%)	183(3.5%)	1,087.6K	4.2K(0.38%)	663(0.06%)
QpidBroker	7.2K	1.4K(19.82%)	561(7.74%)	2,253K	14.3K(0.64%)	2,348(0.1%)
ActiveMQ	10K	2.2K(21.94%)	692(6.93%)	1,688.3K	13K(0.77%)	2,287(0.14%)
Camel	30.3K	4.5K(14.92%)	864(2.85%)	3,817.4K	21.4K(0.56%)	1,739(0.05%)
CloudStack	31.3K	4.5K(14.37%)	1,496(4.77%)	4,532.2K	43.8K(0.97%)	7,370(0.16%)
JMeter	15.7K	1.8K(11.7%)	685(4.36%)	1,451.9K	11.7K(0.8%)	2,498(0.17%)

Comm.: number of commits; Comm.Log: number and percentage of commits with log changes; Comm.ST: number and percentage of commits with log changes that involve stack traces; CodeChg.: number of lines of code changes; LogChange: number of log changes and its rate to code changes; STChange: number of log changes involving stack traces and its rate to code changes.

3.1 RQ1. How do Developers Log and Modify the Logging of Exception Stack Traces?

3.1.1 Motivation

Stack traces of exceptions help developers identify the causes of runtime issues. However, the logging of too many stack traces would grow the log files very fast and hide other important information. Prior work on logging focused on the characteristics and evolution of logging, where to log, what to log, how to choose log levels, and logging anti-patterns. However, there exists no work that studies the logging of exception stack traces. We believe that appropriate logging of exception stack traces is as important as the logging aspects studied in prior work. As the first step, in this research question, we study how developers log and modify the logging of exception stack traces. Our results can help developers and researchers understand the current practices of logging exception stack traces and provide insights for future research to improve the logging practices of exception stack traces.

3.1.2 Approach

We first study how often developers log exception stack traces and their associated code context (e.g, exception type). Second, we examine the code change history of the studied projects to understand how developers modify the logging of exception stack traces.

1) Studying the context under which exception stack traces are logged. We examine the source code of the studied projects to understand the context of logging exception stack traces in the studied projects. We use the described approach in Section 2.3 to extract the characteristics of the logging statements (i.e., the log level and whether a stack trace is logged) and the contextual information (e.g., the exception types). Then, we examine the relationship between stack trace logging and log levels, exception types, and exception sources, as described below:

- *Log level* is the assigned severity level to a logging statement, which is usually one of *trace*, *debug*, *info*, *warn*, *error*, and *fatal*, ordered from the lowest severity level to the highest.
- *Exception type* is the type of the exception caught by the containing *catch* block, such as *IOException* and *FileNotFoundException*. We only consider the exception types that appear in more than half of the studied projects (i.e., common exception types).
- *Exception source* indicates the source where an exception class is defined. An exception class can be defined by the application *project*, by the programming language (i.e., *JDK* for Java), or by third-party *libraries*.

LOG.[−warn−]{+debug+}("Ignoring socket shutdown exception"{+, e+});

Fig. 5. Example of applying *word-diff* on a log change.

Regex: "{\+[\t]*,[\t]*\b(e|ex|t|e\d)\b(\.getCause\(\))?(?!\.)[\t]*\+}"
Matched example: log.info(...{+, ex+});

Fig. 6. Our regular expression (case-insensitive) used to match the scenario of adding a stack trace to a logging statement.

Chi-Square Test. We use Pearson's Chi-square test to evaluate the statistical relationship between the aforementioned characteristics (e.g., exception types) and stack trace logging (i.e., the logging of an exception stack trace). The Chi-square test is used to test the strength of association between two categorical variables [17], [26]. For example, we use the Chi-square test to test whether there is a statistically significant association between exception types and stack trace logging. In our Chi-square test, the samples are independent and they are not paired or matched, and the categories (e.g., log levels) are exclusive from each other (e.g., one logging statement can not have two log levels). Thus we choose the Chi-square tests instead of a test for dependent samples. A *p-value* resulting from the Chi-square test that is less than 0.05 indicates the two categorical variables have a statistically significant association. As we perform multiple Chi-square tests, in order to address the multiple testing problem [6] that may lead to over-estimation of statistical significance, we use the Bonferroni correction [59] to adjust the *p-value* threshold to $\frac{0.05}{3}$ where 3 is the number of Chi-square tests we perform. In each Chi-square test, we consider all the data of the studied projects together.

Stack Trace Ratio. For each group of exception logging statements (e.g., with a certain log level), we measure the ratio of these exception logging statements that are with stack traces (i.e., the stack trace ratio). The stack trace ratio indicates how likely the exception logging statements in a certain group (e.g., with a certain log level) are with stack traces.

Odds Ratio. While the stack trace ratio indicates how likely a certain group (e.g., with a certain log level) is associated with the logging of stack traces, it does not indicate whether the characteristic of the group increases the likelihood of logging a stack trace. Thus, for each group, we measure the odds ratio of logging exception stack traces. The odds ratio is the ratio between the odds that exception logging statements in a group are with stack traces and the odds that exception logging statements in other groups are with stack traces

$$OR = \frac{STR_{group}/(1 - STR_{group})}{STR_{other-groups}/(1 - STR_{other-groups})}, \quad (1)$$

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
Scenarios of Changing the Logging of Exception Stack Traces

Scenario	Example
Logging an additional stack trace	LOG.error("Failed to send last message."{+, e+});
Removing a previously-logged stack trace	LOG.error("Lifecycle start issue"[-, e-]);
Increasing the level of a logged stack trace	LOG[-.debug-]{+.error+}("Could not stop tez dags: ", e);
Decreasing the level of a logged stack trace	LOG.[-info-]{+debug+}("Failed to vectorize", e);

A pair of {+ and +} indicates an added code unit, while a pair of [- and -] indicates a deleted code unit.

where STR_{group} is the stack trace ratio of a certain group (e.g., with a certain log level) and $STR_{other-groups}$ is stack trace ratio of other groups (e.g., with other log levels). An odds ratio greater than 1 indicates increased likelihood of logging exception stack traces while an odds ratio smaller than 1 indicates decreased likelihood of logging exception stack traces.

*2) Studying how developers modify the logging of exception stack traces.*We examine the code change history of the studied projects to understand how developers modify the logging of exception stack traces. We use the approach that is described in Section 2.4 to analyze the code change history of the studied projects. Theoretically, modifying the logging of a stack trace includes logging additional stack traces, removing previously-logged stack traces, and changing the log level of the logging statements with a stack trace. We analyze the distribution of these types of modifications to the logging of exception stack traces. In particular, we examine the patterns of how developers modify the logging of exception stack traces and how they change their log levels.

3.1.3 Results

Developers are more likely than not to log the stack trace in an exception logging statement. Overall, 65% of the exception logging statements log a stack trace. Table 5 summarizes the number of exception logging statements and the percentage that log a stack trace. The studied projects have a range of 279 to 3,548 exception logging statements, and 51% to 77% of them log a stack trace.

Developers are less likely to log a stack trace in info-level logging statements than in higher-level (e.g., error) or lower-level (e.g., debug) logging statements. Our Chi-square test indicates that the log level of a logging statement and the likelihood of logging a stack trace in that logging statement have a statistically significant association (i.e., *p-value* $< \frac{0.05}{3}$). Fig. 8 shows the distributions of stack trace logging over log levels in the studied projects. When the log level increases along *trace*, *debug*, to *info*, the likelihood of logging a stack trace drops and reaches the lowest at the *info* level; while when the log level increases along *info*, *warn*, *error*, to *fatal*, the likelihood of logging a stack trace increases and reaches the highest at the *fatal* level. Showing a similar trend as the stack trace ratios, the odds ratios indicate that the *fatal* and *error* levels increase the likelihood of logging exception stack traces the most, while the *info* level decreases the likelihood of logging exception stack traces the most. A low-level (i.e., *trace* and *debug*) logging statement usually indicates debugging purposes. A high-level (i.e., *warn*, *error*, and *fatal*) logging statement is usually concerned with the handling of a runtime issue. Stack traces can support debugging for both types of logging statements. In comparison, an *info* level logging statement usually indicates a normal (i.e., non-error) event, and logging a stack trace in such a case could be confusing to the users. For example, issue report AMQ-2902[11] argues that a stack trace should not be logged in an *info* level logging statement, because "the cause for the message is actually harmless - hence the INFO level - but the messages are confusing and annoying nonetheless". Similarly, in RQ2, we find that "normal executions" is one of the reasons for not logging exception stack traces.

However, in some cases, the *info* level is used for situations where developers want the exception stack traces to be delivered to users to indicate some runtime issues even though such issues may not impact the normal execution of the system. For example, in issue report CAMEL-2710,[12] the log level of an exception stack trace was changed from *warn* to *info*, as "the end user has configured it to ignore" a certain runtime issue (i.e., invalid endpoint), while the user may "still need to know about" the runtime issue.

Some exception types, in particular, generic exceptions (e.g., Exception*),*[13] *are more likely than other exception types (e.g.,* FileNotFoundException*) to be logged with a stack trace.* Our Chi-square test results show that the type of an exception has a statistically significant association with the likelihood of logging a stack trace about it (i.e., *p-value* $< \frac{0.05}{3}$). Fig. 9 shows the distribution of stack trace logging over the ten most popular exception types in the studied projects. In general, generic exceptions (e.g., *Throwable*, *Exception*, *IOException*, and *RuntimeException*) are more likely than an average exception type to be logged with a stack trace. The odds ratios also indicate that these generic exceptions increase the likelihood of logging exception stack traces. In RQ2, we also find that "generic exceptions" is one of the reasons for logging exception stack traces. When a generic exception is caught in a *catch* block, logging the stack trace of such a generic exception is important as it helps developers locate more specific information about an exception (e.g., the specific exception type). Prior work suggested always catching a specific exception instead of a generic exception [45], while it

```
project in (<Project Name>) AND summary ~ "((log || print ||
    output || write) AND (exception)) OR (\"stack trace\") OR
    (\"exception trace\") OR (stacktrace)" ORDER BY created DESC
```

Fig. 7. The JQL query that we used to search for the logging-related issue reports.

11. https://issues.apache.org/jira/browse/AMQ-2902
12. https://issues.apache.org/jira/browse/CAMEL-2710
13. Generic exceptions refer to how an exception is caught in a *catch* block, i.e., a superclass (e.g., *IOException*) is declared in a *catch* statement to catch a subclass exception (e.g., *FileNotFoundException*).

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

TABLE 5
Summary of the Exception Logging Statements in the Studied Projects

	Hadoop	Dir. Server	Hive	ZooKeeper	Kafka	Qpid Broker	ActiveMQ	Camel	CloudStack	JMeter
# Exception log. stmts	3,548	363	1,636	394	279	381	758	862	3,454	647
% with a stack trace	63%	51%	66%	77%	75%	75%	76%	63%	62%	65%

was found that only a small portion of exceptions are handled in a *specific* way, while the majority of exceptions are handled with a *generic* strategy [11].

The JDK and third-party-defined exceptions are more likely to be logged with a stack trace than project-defined exceptions. Fig. 10 shows the distribution of stack trace logging over exception sources in the studied projects. Our Chi-square test results show that the exception source (i.e., JDK, third-party libraries, or application projects) has a statistically significant association with the likelihood of logging a stack trace (i.e., $p\text{-}value < \frac{0.05}{3}$). The project-defined exceptions are least likely to be logged with a stack trace, which might be explained by the intuition that these exceptions are expected by developers and they understand the causes of such project-defined exceptions. In contrast, exceptions defined outside of their own project may be unexpected by developers or they may not understand the causes of these exceptions, thus they are more likely to need stack traces when investigating such exceptions to explain their causes. As discussed in RQ2, "unexpected exceptions" is one of the reasons for logging exception stack traces while "expected exceptions" is one of the reasons for not doing so.

Developers are more likely to log additional stack traces to existing logging statements rather than remove already-logged stack traces. In contrast, developers are more likely to reduce than increase the log level for the logged stack traces. Table 6 shows the numbers of different types of stack trace modifications for all the studied projects. Fig. 11 indicates that the distributions of the different types of stack trace modifications are relatively consistent in the studied projects. We observed some instances when the logging of a stack trace was noted as being inappropriate, developers then often opted to reduce the log level of the corresponding logging statement rather than removing the logged stack trace. For example, issue report HDFS-3454[14] claims that a stack trace should not be logged for a normal situation. As the result, the developer changed the log level of the corresponding logging statement from *info* to *debug* instead of removing the logged stack trace.

*Developers are most likely to adjust the logging of stack traces in the warn and error-level logging statements.*As shown in Fig. 12, compared to other log levels, developers are more likely to log additional stack traces to existing *warn* and *error*-level logging statements and remove previously-logged stack traces from existing *warn* and *error* logging statements. On the one hand, stack traces can help developers diagnose runtime failures (e.g., indicated by *warn* or *error*-level log messages). For example, issue report HADOOP-12840[15] proposed to add a stack trace to an existing logging statement to help debugging. On the other hand, logging too many unnecessary stack traces in *warn* or *error* logging statements can "easily pollute the log files".[16] Therefore, developers should carefully consider whether to log the stack trace of an exception in a *warn* or *error*-level logging statement.

Developers appear to be confused between the warn and error log levels when logging stack traces. Fig. 13 illustrates the pattern of the changes to the levels of logged stack traces. The log levels of logged stack traces are mostly being changed between the *warn* and *error* levels. For example, issue report AMQ-4801[17] changed the log level of a logged stack trace from *error* to *warn*, because the exception could be ignored. The *info*, *warn* and *error* level for logged stack traces also tend to be changed to the *debug* level. For example, issue report AMQ-2902[18] changed the log level of a logged stack trace from *info* to *debug*, in order to avoid logging stack traces for a successful operation. On the other hand, some *debug* level stack traces were changed to the *warn* level. For example, issue report YARN-6815[19] changes the log level of a logged stack trace from *debug* to *warn*, in order to avoid the hiding of an exception. Finally, some *fatal* level stack traces were changed to *error*, due to the migration of these projects to SLF4J which does not support the *fatal* level.

Developers sometimes make back-and-forth changes to the stack traces of the same logging statements. In order to understand how the logging of stack traces evolve, we identified the stack trace changes that are for the same logging statements. Specifically, we first grouped the 915 changes by their containing files, then manually examined whether the changes in the same files are for the same logging statements. We

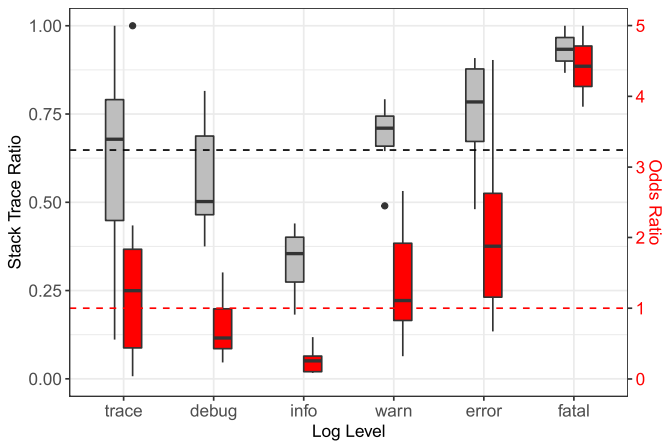

Fig. 8. The likelihood of logging stack traces for each log level. The grey box-plots show the distribution of the stack trace ratios across the studied projects; the black dashed line shows the average stack trace ratio of the studied projects across all the log levels. The red box-plots show the distribution of odds ratios across the studied projects; the red dashed line marks the odds ratio of 1.

14. https://issues.apache.org/jira/browse/HDFS-3454
15. https://issues.apache.org/jira/browse/HADOOP-12840
16. https://issues.apache.org/jira/browse/KAFKA-1591
17. https://issues.apache.org/jira/browse/AMQ-4801
18. https://issues.apache.org/activemq/browse/AMQ-2902
19. https://issues.apache.org/jira/browse/YARN-6815

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

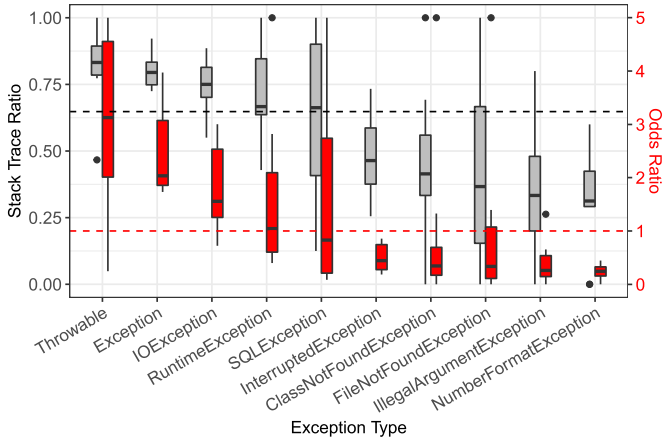

Fig. 9. The likelihood of logging stack traces for each of the ten most popular exception types. The grey box-plots show the distribution of the stack trace ratios across the studied projects; the black dashed line shows the average stack trace ratio of the studied projects across all exception types. The red box-plots show the distribution of odds ratios across the studied projects; the red dashed line marks the odds ratio of 1.

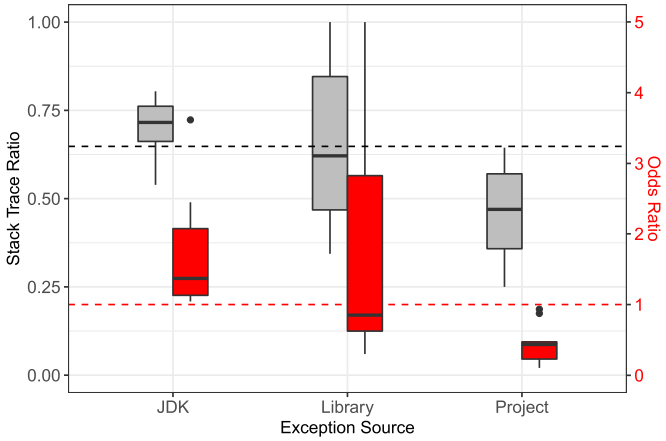

Fig. 10. The likelihood of logging stack traces for each type of exception sources. The grey box-plots show the distribution of the stack trace ratios across the studied projects; the black dashed line shows the average stack trace ratio of the studied projects across all types of exception sources. The red box-plots show the distribution of odds ratios across the studied projects; the red dashed line marks the odds ratio of 1.

found that the stack traces of 43 exception logging statements were changed two to three times (the 915 changes actually changed 863 original logging statements). For example, issue report AMQ-2119[20] changed the level of an exception stack trace from *debug* to *info* as "network problems are imho important". However, a later issue report *AMQ-2902*[21] changed its level back to *debug*, as the stack traces are "confusing and annoying". This phenomenon further indicates that developers face challenges when deciding whether to log the stack trace in an exception logging statement.

Summary of RQ1. Logging of exception stack traces is a popular practice in open-source projects while developers have difficulties making appropriate logging of exception stack traces in the first place. Developers are more likely to log stack traces in low-level (e.g., *debug*) and high-level (e.g., *error*) logging statements than in median-level (e.g., *info*) logging statements. Generic exceptions, as well as JDK and third-party-defined exceptions are more likely than other exceptions to be logged with a stack trace. Developers make the most changes to adjust the logging of stack traces at the *warn* and *error* levels, by adding or removing a stack trace in *warn* and *error*-level logging statements, or changing the level of a logged stack trace between the *warn* and *error* levels. In addition, they sometimes make back-and-forth changes to the stack traces of the same logging statements. Our findings indicate that developers face challenges when deciding whether to log the stack trace in an exception logging statement.

3.2 RQ2. Why do Developers Log or not Log Exception Stack Traces?

3.2.1 Motivation

In the previous research question, we find that logging exception stack traces is a popular practice in software development while developers have difficulties making appropriate logging of exception stack traces in the first place. In this research question, we aim to understand why developers log or not log exception stack traces. As issue reports raised in the development history of the studied projects usually communicate developers' reasons for making specific code changes (e.g., adding or removing the logging of a stack trace), we examine the issue reports that are related to the logging of exception stack traces. The results may provide insights for developers on whether they should log the stack trace of an exception. The results may also provide insights for future research to help developers balance the benefits and costs of logging exception stack traces, e.g., by helping developers identify the inappropriate logging of stack traces.

3.2.2 Approach

Table 7 shows the number of issue reports that we obtained from each subject project. Below, we describe the process of our qualitative analysis for studying developers' rationale for logging or not logging the stack trace of an exception.

Qualitative analysis of issue reports. We used an open card sorting approach [50], [56], [64] to code the issue reports related to the logging of stack traces. Open card sorting is widely used in the software engineering community to deduce a higher level of abstraction (i.e., categories or themes) from lower level descriptions of data (e.g., survey responses) [41], [48], [51]. For each issue report, we examined the summary, description, comments, and patches, aiming to understand the reason why developers log or not log the

TABLE 6
The Numbers of Stack Trace Modifications in the Studied Projects

Logging an additional stack trace	Removing a previously-logged stack trace	Increasing the level of a logged stack trace	Decreasing the level of a logged stack trace	Total[1]
347	85	207	352	915

[1]*There are 76 changes that add/remove stack traces and increase/decrease the log levels of the same logging statements.*

20. https://issues.apache.org/jira/browse/AMQ-2119
21. https://issues.apache.org/jira/browse/AMQ-2902

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

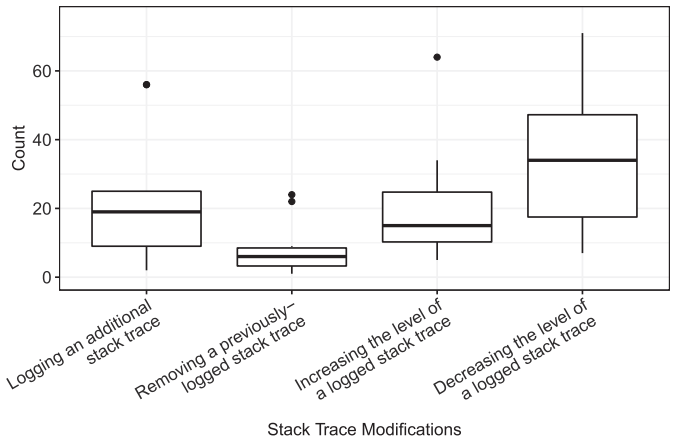

Fig. 11. The distributions of stack trace modifications in the studied projects.

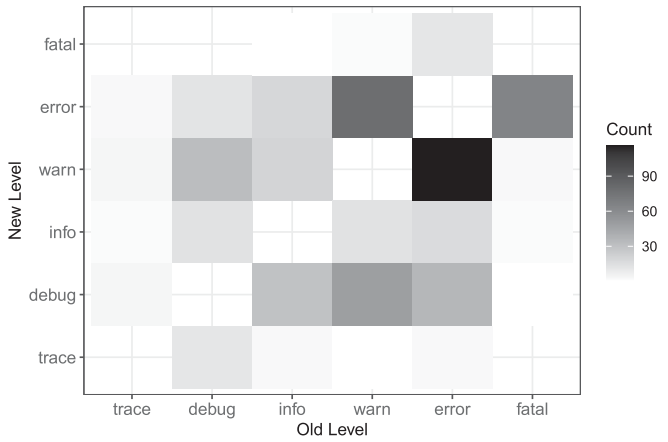

Fig. 13. The pattern of the changes of the stack trace log levels.

stack trace of an exception For each issue report, we assign one label (the reason). When there are multiple reasons, we choose the primary one. We also identify the issue reports that are not related to the logging of exceptions stack traces and label them as "irrelevant". The first three authors of the paper (i.e., coders) jointly performed the coding process, following the steps listed below:

Step 1: *Round-1 coding.* We randomly and evenly distributed all the issue reports to the three coders. Each coder coded one-third of the issue reports separately, which took a few days for each coder to finish their portion.

Step 2: *Discussion after round-1 coding.* The goal of the discussion was to reach the same codes among the coders. We had a meeting to discuss our resulting codes and reached consensus. The meeting took about two hours.

Step 3: *Revisiting round-1 coding.* Based the updated codes from our discussion, we revisited our separate round-1 coding results, which took a few hours for each of us.

Step 4: *Round-2 coding.* Each coder coded another one-third of the issue reports separately, based on the codes resulting from round-1. Each portion of the issue reports assigned to a coder in round-1 were randomly and evenly distributed to the other two coders in round-2. In this way, we made sure each issue report was coded by two different coders in the two rounds. Coders could add new codes in round-2. Round-2 coding took a few days for each of us to finish our separate portion.

Step 5: *Discussion after round-2 coding.* We had one more meeting to discuss our separate codes updated in round-2 and reached consensus. The meeting took about 1.5 hours. We finalized the codes after this step.

Step 6: *Revisiting round-1 and round-2 coding.* Based on the updated codes from our discussion, we revisited our round-1 and round-2 coding results, which took a few hours for us to finish our respective portions. We measured our inter-coder agreement after this step.

Step 7: *Resolving disagreement.* We discussed every conflict in our coding results and reached an agreement. Whenever there was a conflict, the two coders who coded that particular response discussed and tried to resolve it; if an agreement could not be reached, the third coder was involved and voting was conducted if necessary. We resolved the disagreement in a two-hour meeting.

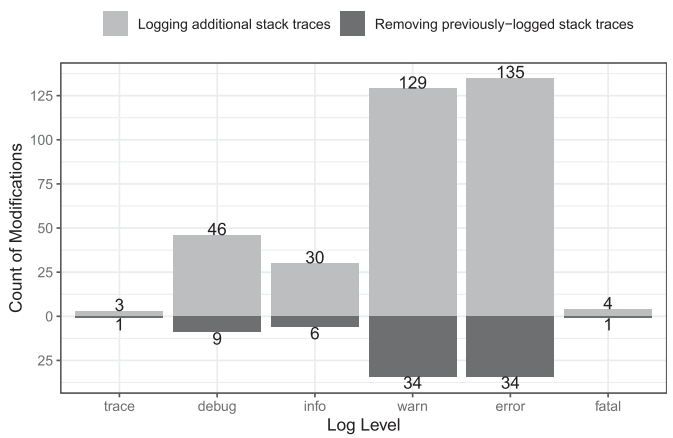

Fig. 12. The occurrences that developers log additional stack traces to existing logging statements and remove previously-logged stack traces from existing logging statements, across different log levels.

Measuring the Reliability of Our Coding Results. Reliability is a prerequisite for ensuring the validity of the coding results [1], [31]. The coding results are reliable if the coders show a certain level of agreement on the categories assigned to the coded instances (*a.k.a.*, inter-coder agreement) [1], [31].

In this work, we use Krippendorff's α [22], [31] to measure the inter-coder agreement of our coding results. Krippendorff's α is a standard and flexible coefficient for measuring inter-coder agreement [1], [22], which takes the form of

$$\alpha = 1 - \frac{D_o}{D_e}, \tag{2}$$

where D_o is the observed disagreement between coders and D_e is the disagreement expected by chance. When coders agree perfectly, $\alpha = 1$; when coders agree as if chance had produced the results, $\alpha = 0$, which indicates the absence of

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

TABLE 7
Number of Studied Issue Reports per Project

	Hadoop	Dir. Server	Hive	ZooKeeper	Kafka	Qpid Broker	ActiveMQ	Camel	CloudStack	JMeter	Total *
# issue reports	179	2	54	14	31	18	31	33	23	-	385

** The resulting issue reports from the automated querying may falsely include some issue reports that are not related to the logging of stack traces. We manually identify such irrelevant issue reports in our manual analysis.*

TABLE 8
The Rationale for Logging or not Logging Exception Stack Traces, Derived From Manually Studying 385 Issue Reports That Concern the Logging of Exception Stack Traces

Rationale[1]	Description (D) — Quote (Q)	Freq.
Avoiding/downgrading stack trace for user errors ↓	**D**: The logging of stack traces should be avoided or downgraded when the failures are caused by users' usage problems (e.g., invalid authentication or invalid syntax) instead of programming bugs. **Q**: [HIVE-7737] "Table not found is generally user error, the call stack is annoying and unnecessary."	33
Avoiding stack trace in user interfaces ↓	**D**: Stack traces should not be printed in user interfaces (e.g., console, STDOUT, STDERR) as they can pollute other important information in the user interfaces. **Q**: [HADOOP-15] "Stack trace should not be printed out when running hadoop key commands."	30
Avoiding/downgrading stack trace for normal executions ↓	**D**: Logging stack traces for normal executions can confuse users as stack traces usually suggest runtime issues **Q**: [CAMEL-6247] "We should avoid ugly stacktraces for osgi blueprint shutdown which we don't care about."	29
Avoiding flooding the logs ↓	**D**: Too much logging of stack traces can pollute the log files. **Q**: [KAFKA-1591] "Some of the unnecessary stack traces in error / warning log entries can easily pollute the log files."	17
Avoiding/downgrading stack trace for expected exceptions ↓	**D**: Expected exceptions or well-handled exceptions may not need detailed stack traces **Q**: "Ensure handled ioexceptions are not propagated back to client"	10
Avoiding duplicated stack traces ↓	**D**: Duplicated logging of the stack trace of the same exception (e.g., caused by throwing and logging an exception at the same time) should be avoided. **Q**: [HADOOP-14] "When there's no bucket ends up double-listing the stack trace, which is somewhat confusing."	6
Avoiding stack trace for security concerns ↓	**D**: The logging of stack traces may leak information about security-sensitive code. **Q**: [AMQ-7209] "For security exceptions this can leak some information about the implementation."	3
Reducing performance overhead ↓	**D**: Excessive logging of stack traces can slow down system performance. **Q**: [HDFS-4714] "The major contributing factor in slow down is the long namenode log message, which includes full stack trace."	1
Assisting in general debugging ↑	**D**: Stack traces can in general help developers and users find the root cause of an exception. **Q**: [HIVE-14368] "The stack trace will be really helpful for client to debug failed queries."	76
Logging stack trace for generic exceptions ↑	**D**: Logging the stack trace of a generic exception (e.g., *Throwable*, *Exception*, *Error*) can provide the concrete exception information. **Q**: [KAFKA-6757] "Log any 'Throwable' in order to catch these unpredictable errors."	12
Reminding/alerting severe runtime problems ↑	**D**: Severe exceptions are those that may break normal executions of the system and that need user attention or actions. However, users may not be aware of a severe exception if the stack trace is not printed out in log files. **Q**: [HDFS-12683] "The ZKFC should log fatal exceptions before closing the connections and terminating server."	10
Logging stack trace for unexpected exceptions ↑	**D**: Stack traces can help understand the cause of an unexpected exception. **Q**: [AMQ-5879] "The full stack trace should be logged when a transport fails and the exception is unexpected."	9
Refactoring logging code	**D**: Developers sometimes review and change the logging code to improve the logging quality in general. **Q**: "Improve the error loggings for printing the stack trace."	7
Enhancing the configurability of stack traces	**D**: Enabling users to turn on / off the logging of stack traces on demand. **Q**: [HADOOP-87] "Provide an option for IPC server users to avoid printing stack information for certain exceptions."	5
Irrelevant	**D**: Some issue reports resulted from the keyword searching are not related to the logging of exception stack traces. For example, they may be related to using existing stack traces for diagnosing a bug.	137

[1] *A ↑ symbol indicates a rationale for logging an additional stack trace or increasing the log level of a logged stack trace; a ↓ symbol indicates a rationale for removing a previously-logged stack trace or reducing the log level of a logged stack trace.*

agreement [30]. The detailed methodology for calculating Krippendorff's α is described in prior work [30].

Our coding of the issue report data achieves a Krippendorff's α of 0.826. Krippendorff [31] suggests that $\alpha \geq 0.800$ indicates a reliable agreement. Thus, *the result of our manual coding is reliable*.

3.2.3 Results

Table 8 shows the results of qualitative analysis: the rationale for logging or not logging exception stack traces. We derived eight reasons for not logging exception stack traces (i.e., the ones followed by a ↓ symbol), including *avoiding/*

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

downgrading stack trace for user errors, *avoiding stack trace in user interfaces*, *avoiding/downgrading stack trace for normal executions*, *avoiding flooding the logs*, *avoiding/downgrading stack trace for expected exceptions*, *avoiding duplicated stack traces*, *avoiding stack traces for security concerns*, and *reducing performance overhead*, ordered by their frequencies in our studied issue reports. In addition, we derived four reasons for logging exception stack traces (i.e., the ones followed by a ↑ symbol), including *assisting in general debugging*, *logging stack trace for generic exceptions*, *reminding/alerting severe runtime problems*, and *logging stack trace for unexpected exceptions*, ordered by their frequencies. Two reasons are related to modifying the logging of exception stack traces in general, including *refactoring logging code* and *enhancing the configurability of stack traces*. Finally, there are 137 issue reports that are not related to the logging exception stack traces, though they are in the results of the keyword-based filtering.

Exception stack traces should be avoided or logged at low levels for user errors, normal executions, and expected exceptions. The failure of an user operation can be caused by either a software bug or a user error (e.g., providing the wrong input or configuration). When the failure is caused by a software bug, a stack trace can help developers find the root cause of the bug. However, when the failure is cased by a user error, what the user really needs is a message alerting the usage problem. In this case, a stack trace can not really help and it may negatively impact the user's experience. For example, in issue report HIVE-7737, it is argued that the logging of stack traces when tables cannot be found is "annoying" and "unnecessary", because the problem is usually caused by user errors. In addition, developers argue that the logging of stack traces should be avoided or downgraded when the exception is expected or handled, or when the exception does not interrupt the normal execution of software. For example, in issue report CAMEL-6247, it is argued that the "ugly stack traces' should be avoided for "osgi blueprint shutdown" which does not impact the normal execution.

Exception stack traces should be avoided in user interfaces or when there is a security concern. Exception stack traces are usually leveraged by developers to diagnose the root cause of a software bug. However, such stack traces are undesirable in user interfaces (i.e., console, STDOUT, STDERR) as the long stack traces can hide other important information in the user interfaces. For example, issue report HADOOP-15 suggests that stack traces "should not be printed out" in the console when running Hadoop key commands. In addition, as stack traces can expose the implementation details in the source code, developers should carefully avoid logging the stack traces when there is a security concern. For example, in issue report AMQ-7209, developers recommend that the stack traces for some security-related exceptions should not be logged as they can "leak some information about the implementation'.

Excessive logging of stack traces should be avoided as it may pollute the logs and increase performance overhead. As stack traces are usually much longer than regular log messages, excessive logging of stack traces can grow log files very fast, which can only only pollute the logs and hide important log information, but also negatively impact the performance of the software. For example, it is reported in issue report KAFKA-1591 that unnecessary stack traces can "easily pollute the log files"; issue report HDFS-4714 reported that the "major contributing factor" of a performance slowdown is the long log messages that include full stack traces. In addition, duplication of stack traces (e.g., caused by throwing and logging an exception at the same time) should be carefully avoided as it leads to redundant information.

Logging of stack traces for generic exceptions, severe problems, and unexpected exceptions is recommended. In general, developers add exception stack traces for debugging purposes. In particular, it is recommended stack traces be logged for generic exceptions (e.g., *Throwable*, *Exception*, *Error*). When an exception is caught by a catch block that declares a generic exception class (i.e., a superclass), it is not clear which specific exception (i.e., a subclass of the generic exception or the superclass itself) has caused the exception. Thus, logging the stack trace can provide information about the specific exception information. For example, in issue report KAFKA-6757, it is suggested to log any "Throwable" exceptions (i.e., one type of generic exceptions) as the specific causes of such exceptions are unpredictable. In addition, logging of stack traces for severe problems is recommended as as they can alert users of such severe problems. For example, issue report HDFS-12683 suggests to log the stack traces of "fatal exceptions" that cause the closing of connections and termination of the server. Finally, developers suggest the logging of stack traces for unexpected exceptions, as opposed to the suggestion of avoiding/downgrading stack traces for expected exceptions. For instance, issue report AMQ-5879 suggests that the full stack trace should be logged when a transport fails unexpectedly.

In order to balance the benefits and costs of logging exception stack traces, it is recommended to allow developers and users to turn on or off the logging of exception stack traces on demand. As the logging of exception stack traces come with many costs (e.g., flooding the logs), developers recommend to enhance the configurability of stack traces by providing users the flexibility to turn on/off the stack traces. For example, issue report HADOOP-87 demands "an option for IPC server users to avoid printing stack information for certain exceptions". However, existing logging frameworks (e.g., SLF4j) do not provide such an option. We recommend that future logging frameworks improve their flexibility by allowing developers and users to configure the logging of stack traces.

Summary of RQ2. Stack traces can help developers debug the cause of an exception. However, developers should pay extra attention when deciding whether to log the stack trace of an exception. In particular, logging of stack traces should be avoided or downgraded for user errors, normal execution, expected exceptions, in user interfaces, or when there is a security concern. On the other hand, logging of stack traces for generic exceptions, severe problems, and unexpected exceptions is recommended. In addition, logging frameworks should improve their flexibility by allowing developers and users to configure the logging of stack traces.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

TABLE 9
Selected Code Metrics That are Relevant to the Likelihood of Logging the Stack Trace of an Exception in an Exception Logging Statement

Dimension	Metric	Definition (D) — Rationale (R)
Logging statement metrics	Log level	**D**: The verbosity level of the logging statement **R**: Logging statements at certain levels (e.g., *info*) are less likely than other levels to log stack traces (see Section 3.1)
	Log in loop	**D**: Is the logging statement contained in a loop? **R**: Logging statements in loops are more likely to produce excessive logging [35] thus might be less likely to log stack traces
	Log in branch	**D**: Is the logging statement contained in a branch inside the containing catch block? **R**: Logging statements in a branch might be related to a branching condition instead of the exception [18]
Exception metrics	Exception type	**D**: Type of the exception that is caught by the containing *catch* block **R**: Some exception types are more likely than others to be logged with a stack trace (see Section 3.1)
	Exception source	**D**: Source of the exception (i.e., JDK, libraries, or the studied project) **R**: Problems related to different sources might be handled differently (e.g., logging the stack trace or not)
	Exception package	**D**: The containing package of the exception class **R**: Exceptions that are defined in the same package might indicate similar problems thus being handled similarly
Catch block metrics	LOC before logging	**D**: Number of lines of code in the containing *catch* block that are prior to the logging statement **R**: A logging statement that is further from the start of a *catch* block might be less related to the caught exception
	LOC after logging	**D**: Number of lines of code in the containing catch block that are after the logging statement **R**: Indicates how well the exception is handled. Well-handled exceptions are less likely to need stack traces for debugging
	Method calls before logging	**D**: Number of method calls in the containing catch block that are prior to the logging statement **R**: A logging statement that is more operations from the start of a *catch* block might be less related to the caught exception
	Method calls after logging	**D**: Number of method calls in the containing catch block that are after the logging statement **R**: Indicates how well the exception is handled
Exception-throwing method metrics	Exception method	**d**: The method that may have thrown the caught exception, or the containing method if the exception is thrown by the try block **r**: The exceptions that are triggered by the same method call might be logged in similar ways
	Exception method source	**d**: The source of the exception-throwing method (i.e., from the JDK, libraries, or the studied project) **r**: Exceptions that are triggered by different sources might have different level of severity
	Exception method package	**d**: The containing package of the method that may have thrown the exception **r**: Exceptions that are triggered by methods from the same package might be logged in similar ways
Containing code metrics	Containing file	**D**: The containing file of the logging statement **R**: Logging statements in the same file might share similar logging patterns [35]
	Containing package	**D**: The containing package of the logging statement **R**: Logging statements in the same package might share similar logging patterns

3.3 RQ3. Which Factors Impact the Logging of Exception Stack Traces?

3.3.1 Motivation

Logging the stack traces of exceptions is beneficial to developers in failure diagnosis. However, stack traces can easily cause excessive logging and hide other important information in logs. As discussed in RQ1, developers have difficulties making appropriate logging of exception stack traces in the first place and they make many changes to adjust the logging of exception stack traces. As discussed in RQ2, developers raise many concerns (in the form of issue reports) regarding the logging of exception stack traces, which indicates the logging of exception stack traces is an important yet challenging task for developers. Therefore, in this research question, we propose an automated solution for suggesting the logging of exception stack traces and study the factors that have the most important influence in determining the logging of exception stack traces. We build machine learning models to explain the logging of exception stack traces using a set of metrics extracted from the source code. However, the logging of exception stack traces in the source code of the studied projects may not be optimal, which can impact our results. We emphasize that our models and findings are derived from the common exception logging practices exhibited in the existing source code of the studied projects. Developers can leverage such an automated solution to guide their logging practices. The understanding of the influential factors can provide insights for future work to derive guidelines for the logging of exception stack traces.

3.3.2 Approach

Based on our insights obtained from the results of RQ1 and RQ2, we design a set of code metrics to explain the likelihood of logging the stack trace of an exception. For each studied project, we use the static code analysis approach that is described in Section 2.3 to extract all the exception logging statements in the source code and their contextual information. For each exception logging statement, we extract a set of code metrics that capture the contextual information of the logging statement. Based on the extracted metrics, we then build a machine learning model to explain the relationship between the the likelihood of logging an exception stack trace and the set of code metrics. Below, we explain the code metrics that we extract for each exception logging statement.

Code Metrics. We extracted a number of code metrics for each exception logging statement to explain the likelihood of logging a stack trace in the logging statement. Table 9 lists our code metrics for each exception logging statement and explains our rationale for choosing each of these metrics. These metrics fall into five dimensions:

- *Logging statement metrics* capture the characteristics of an exception logging statement (e.g., log level) and its local code context (e.g., whether the logging

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

statement is inside a loop). As discussed in RQ1, the log level of an exception logging statement impacts the likelihood of logging an exception stack trace. More importantly, the impact of logging an exception stack trace or not largely depends on the log level, for example, logging an exception stack trace at the trace level may have a negligible impact as the stack trace is usually not printed (i.e., the trace level logging statements are usually suppressed at runtime), while logging an exception stack trace at the info level could have a much larger impact. Besides, the log level can be useful in the models for automated suggestions of exception stack trace logging, as the log level is typically specified earlier than the exception stack trace in a logging statement.

In addition, the local context of exception logging statements may impact the likelihood of logging the stack trace. For example, those within a loop may be less likely to log the stack traces as they may flood the logs (as discussed in RQ2).

- *Exception metrics* capture the characteristics of the exception that is caught by the containing *catch* block of an exception logging statement. As discussed in RQ1, the characteristics of the exception (e.g., exception type and exception source) statistically significantly impact the likelihood of logging the stack trace. If there are multiple exceptions caught by the same *catch* block (which only accounts for 2% of the cases), we use the first one.
- *Catch block metrics* capture the characteristics of the containing *catch* block of an exception logging statement. As discussed in RQ2, developers suggest that expected or well-handled exceptions should avoid or downgrade the logging of stack traces. We measure the number of lines of code and the number of method calls before and after the exception logging statement, assuming that more lines of code or method calls indicate that an exception is expected and handled.
- *Exception-throwing method metrics* capture the characteristics of the method in the *try* block that can throw the caught exception (i.e., declared in the *throws* clause). As discussed in RQ2, the logging of exception stack traces should be avoided or downgraded for user errors, normal executions, security concerns, and it is recommended for severe runtime problems. We assume that the characteristics of the method that throws the exception can capture some of these information. For example, the method may indicate the severity of the exception thrown by the method. If there are multiple methods that can throw the caught exception (which accounts for 28% of the cases), we use the first one.
- *Containing code metrics* capture the characteristics of the code units that contain the exception logging statement, including the containing file and the containing package. We assume that exception logging statements in the same files or packages may exhibit similar patterns of logging exception stack traces no not.

Model Fitting. Using these code metrics as explanatory variables, we train random forest models to suggest the likelihood of logging an exception stack trace in an exception logging statement. We choose random forest models for three reasons: 1) many of our code metrics are categorical variables (e.g., the exception type), and a tree-based model such as random forest can handle such categorical variables naturally; 2) random forest is naturally robust against overfitting [7] and it usually perform very well in software engineering tasks [19], [38]; and 3) random forest provides us a way to do sensitivity analysis on the metrics so that we can understand the most influential factors for explaining the likelihood of logging exception stack traces [7], [39]. We use a 10-time repeated 10-fold cross-validation to estimate the efficacy of our models. In each repetition of a 10-fold cross-validation, the whole data set is randomly partitioned into 10 sets of roughly equal size. One subset is used as the testing set (i.e., the held-out set) and the other nine subsets are used as the training set. We train our models using the training set and evaluate the performance of our models on the held-out set. We use precision, recall, and AUC (Area Under the ROC Curve) to measure the performance of our models. The process repeats 10 times until all subsets are used as testing set once. We repeat the 10-fold cross-validation 10 times. In total, 100 different held-out sets are used to estimate the efficacy of our models.

Variable Importance. We measure the importance of a variable by permuting the values of the variable while keeping the values of the other variables unchanged in the testing data (i.e., the so-called "OOB" data) [7], [39]. The importance score of a variable measures the impact of such a permutation of the variable on the classification error rate. For each of the 100 folds in our repeated 10×10-fold cross-validation, we measure the importance score of each of our metrics (i.e., variables in the model). As a result, we get 100 importance scores for each metric.

Double Scott-Knott Clustering. In order to understand the important factors that explain the likelihood of logging an exception stack trace, we compare the average importance of our metrics in the random forest models. However, the differences among the importance of some metrics might actually be due to random variability. Therefore, for each studied project, we use a Scott-Knott (SK) algorithm [53] to partition all the metrics into statistically ranked groups. The SK algorithm hierarchically cluster the metrics into groups and uses the likelihood ratio test to judge the significance of the difference of the importance scores among the metric groups [24]. Specifically, the SK algorithm first sorts the metrics by their means and creates k-1 (k is the number of metrics) candidate 2-group partitions formed by dividing the metrics between two successive ones. Then, the partition with the maximum between-groups-sum-of-squares is chosen and a likelihood ration test is performed to determine if the two groups resulted from the partition are statistically significantly different. The two groups are kept if the test shows statistical significance. The process repeats for the partitioned groups in a top-down manner until statistically distinct group of metrics are produced, i.e., the importance scores of the metrics in two different groups are significantly different (i.e., p-value < 0.05), while the importance scores of the metrics within the same group are not significantly different (i.e, p-value ≥ 0.05).

Each metric might have different importance ranks in different projects. Therefore, we use another SK clustering

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

TABLE 10
The Performance of Our Random Forest Models for Suggesting the Likelihood of Logging a Stack Trace in an Exception Logging Statement

Project	With all metrics			With 5 top-ranked metrics		
	Precision	Recall	AUC	Precision	Recall	AUC
Hadoop	0.83	0.83	0.86	0.82	0.79	0.83
DirectoryServer	0.85	0.84	0.93	0.85	0.79	0.90
Hive	0.86	0.86	0.89	0.85	0.80	0.85
ZooKeeper	0.88	0.88	0.84	0.86	0.83	0.79
Kafka	0.87	0.88	0.86	0.86	0.79	0.78
QpidBroker	0.87	0.89	0.83	0.88	0.81	0.80
ActiveMQ	0.90	0.91	0.87	0.89	0.88	0.84
Camel	0.90	0.89	0.93	0.88	0.87	0.89
CloudStack	0.88	0.87	0.90	0.85	0.81	0.86
JMeter	0.88	0.89	0.90	0.84	0.83	0.86
Average	**0.87**	**0.87**	**0.88**	**0.86**	**0.82**	**0.84**

to further group the metrics into ranked groups by statistically comparing their ranks in different projects (*a.k.a.*, double Scott-Knott clustering [19], [57]). As a result, we get an overall rank of each metric's importance across all the studied projects for determining the likelihood of logging an exception stack trace.

3.3.3 Results

Our code metrics can accurately explain the likelihood of logging a stack trace in an exception logging statement, with an average precision of 0.87, an average recall of 0.87, and an average AUC of 0.88. Table 10 shows the performance of our random forest models for the ten studied projects. The results indicate that our models can accurately distinguish exception logging statements that need to log stack traces and that do not. Our models also achieve fairly consistent performance on the studied projects (i.e., a precision of 0.83 to 0.90, a recall of 0.83 to 0.91, and an AUC of 0.83 to 0.93). Table 10 also shows the performance of our random forest models when only considering the top-ranked metrics (i.e., the metrics that are ranked in the first group from our double Scott-Knott results, as shown in Fig. 14). Using only the five top-ranked metrics, the models achieve an average precision, recall, and AUC of 0.86, 0.82, and 0.84, respectively. The results indicate that using only a few metrics can achieve a relatively accurate suggestion of whether to log a stack trace in an exception logging statement. Developers can leverage the accurate models to guide their logging practices. For example, the models can be applied directly to data from new source code to provide automated suggestions on whether an exception logging statement needs to log a stack trace. In the future, we plan to integrate the models into an IDE plugin to provide real-time coding suggestions.

The type of an exception and the method that throws the exception are among the most important factors for explaining the likelihood of logging the stack trace of the exception. Fig. 14 shows the ranks of the importance of our metrics in our random forest models. The *exception type* and the *exception package* metrics from the *exception metrics* dimension are ranked in the first and the second groups by their importance, respectively. Similarly, the *exception method package* and the *exception method* metrics from the *exception-throwing method metrics* dimension are ranked in the first and the second groups, respectively. The type of an exception and the method that throws the exception capture the nature and the severity of the exception. We recommend that organization-wide or global guidelines for logging exception stack traces should be derived primarily based on the exception type and the method that throws the exception.

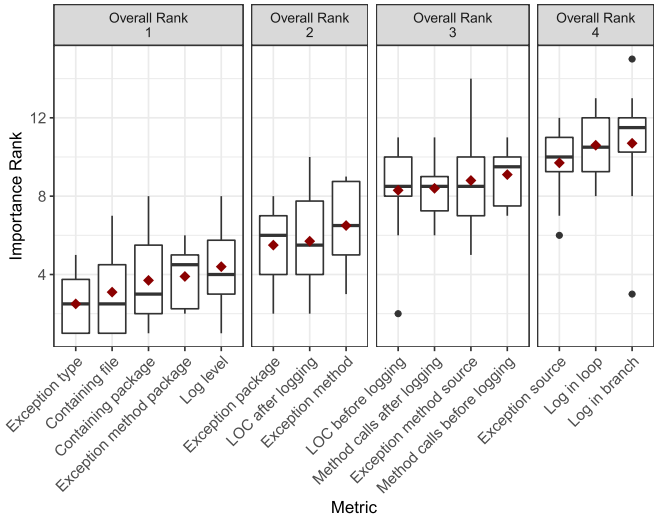

Fig. 14. The distribution of the importance ranks of the metrics in the studied projects. A box-plot shows the distribution of the importance ranks of a metric in the studied projects. A red diamond indicates the average rank of a metric in the studied projects. The *overall rank* indicates the overall importance rank of a metric across all the studied projects (i.e., the double Scott-Knott ranking results).

The containing files and packages also play one of the most important roles in explaining the likelihood of logging exception stack traces. As shown in Fig. 14, the *containing file* and *containing package* metrics from the dimension of *containing code metrics* are ranked in the first group for determining whether an exception logging statement need to log a stack trace or not. In other words, these projects tend to follow similar logging practices (i.e., in terms of logging stack traces) within the same files and packages. However, the consistent logging practices within the same files or packages may be due to the lack of company-wide or global guidelines, thus developers tend to follow similar practices in the same files or packages.

The log level of an exception logging statement is also a top influential factor for explaining the likelihood of logging a stack trace in the logging statement. As shown in Fig. 14, the *log level* metric from the dimension of *logging statement metrics* is ranked as one of the most important metrics (i.e., ranked in the first group) in our random forest models. As discussed in Section 3.1, developers are less likely to add stack traces in logging statements with some levels (e.g., *info*) than in logging statements with other levels (e.g., *warn*). Therefore, guidelines for logging exception stack traces should consider the log level as an important factor.

The lines of code in the exception catch block after the exception logging statement is an influential factor for the logging of exception stack traces. As shown in Fig. 14, the *LOC after logging* metric from the dimension of *catch block metrics* is ranked in the second group in our random forest models. *LOC after logging* can be considered as an indicator of how well the exception is handled. As discussed in RQ2, the logging of stack traces should be avoided or downgraded for expected or well-handled exceptions. Therefore, guidelines for logging exception stack traces should also consider the exception handling code.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

Summary of RQ3. As it is challenging for developers to make appropriate decisions for the logging of exception stack traces, they can leverage our random forest models to provide automated suggestions. Developers tend to follow similar practices for logging exception stack traces within the same files or packages, which may be due to the lack of company-wide or global guidelines. Our results suggest that the exception type and the method that throws the exception should be considered as the primary factors for determining the logging of exception stack traces. The log level and how well the exception is handled should also be considered when making such decisions. Such influential factors offer insights for future research and practices to derive global or company-wide guidelines for the logging of exception stack traces.

4 Threats to Validity

External Validity. This work performs a case study on ten open-source Java projects. Our findings may not generalize to other projects. In particular, as the number of stack trace modifications is relatively small across the studied projects, our findings in RQ1 regarding how developers modify the logging of exception stack traces may not generalize to other projects. Considering a wider range of projects, in particular, closed-sourced projects and non-Java projects, could benefit the results of our work. To mitigate the generalizability issue, we selected eight different types of projects (e.g., distributed computing and network server) as our subject projects. We also ensured that our selected projects are mature projects with many years of development history, such that our findings represent state-of-the-art logging practices of successful long-lived projects.

This work studies the practices of logging exception stack traces through examining the source code, code change history, and issue reports of open-source software projects. In the future, we will validate our findings through communications with developers. Specifically, we will share our findings with developers in the open-source community and perform a survey of developers to validate our findings, as well as further understand their challenges and identify opportunities to help them improve their practices.

Internal Validity. The internal validity of our findings is concerned with correlation versus causation. In RQ1, we study the correlation between the likelihood of logging a stack trace in a logging statement and the context information of that logging statement (e.g., the log level, and the exception type). However, the correlation does not necessarily suggest causation. For example, developers may not necessarily consider log levels or exception types when logging stack traces. In RQ3, we analyze the important factors in our models that impact the logging of exception stack traces. However, the important factors in the models are not necessarily the actual drivers behind developers' decisions of whether to log the stack trace of an exception.

In RQ3, we build machine learning models to explain the logging of exception stack traces using a set of metrics extracted from the source code. Although we studied mature projects with many years of development history, the logging of exception stack traces in the source code of these projects may not be optimal, which can impact our models and our findings derived from them. This is a common issue in empirical studies of logging that are based on the source code. We call for future collaborative efforts among researchers and practitioners to derive polished versions of logging code to improve the studies of software logging, including our work.

Construct Validity. In RQ1, we derive our findings from quantitatively analyzing the code and the code change history. We explained some of our quantitative findings (e.g., generic exceptions, JDK and third-party-defined exceptions are more likely than other exceptions to be logged with a stack trace) based on our expertise and intuition. Other researchers may have different explanations for the qualitative results. Nevertheless, in RQ2, we performed a large-scale qualitative study of related issue reports, in order to understand why developers log or not log exception stack traces.

In RQ2, we performed a qualitative analysis on 385 issue reports that are related to the logging of exception stack traces, in order to understand why developers log or not log exception stack traces. Other research methods (e.g., surveys or interviews) may also be used to study developers' rationale for logging or not logging exception stack traces. We opted for studying the issue reports instead of conducting surveys or interviews as issue reports provide better context about developers' rationale for logging or not logging `specific` exception stack traces. In order words, issue reports communicate developers' concerns on their familiar code in a real development scenario. In contrast, surveys or interviews may not provide such a real and specific context and developers may not pay their full attention in a survey or interview.

In RQ3, we extract a set of code metrics and construct random forest models to explain the likelihood of logging an exception stack trace. Selecting other code metrics or other models may lead to different results. Nevertheless, using the selected code metrics and the random forest models, we achieve an average precision, recall and AUC of 0.87, 0.87 and 0.88, respectively, which suggest that our analysis of the important factors is based on accurate models. Some of the code metrics (e.g., containing file and containing package) used for building the models cannot apply to different projects, thus, a model trained from one project cannot be applied to a different project. However, the insights derived from the models are general among the studied projects, e.g., the exception type and the method that throws the exception are among the most important factors for explaining the likelihood of logging the stack trace of the exception. Such general insights can benefit future research and practices in deriving global or company-wide guidelines for the logging of exception stack traces.

Other Limitations. Some of our findings may be known to the researchers and practitioners in the community. Nevertheless, there exists no prior work that studied the problem or collected data to demonstrate these findings. Besides, developers' faced challenges in the logging of exception stack traces indicate the need to help them better understand the "seemly obvious" observations and improve their practices. As we discussed in RQ1 and RQ2, developers still

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

have difficulties making appropriate logging of exception stack traces and they make many changes to adjust their logging of exception stack traces, sometimes back-and-forth, and they raised many concerns (in the form of issue reports) regarding the logging of exception stack traces. Thus this work makes the first effort to study the practices of logging exception stack traces, aiming to raise awareness of the importance of the problem and help developers and researchers better understand the practices.

5 Related Work

In this section, we discuss prior studies that examined software logging practices, proposed automated solutions for logging improvement, and that studied exception handling in the source code.

Empirical Studies of Logging Practices. Fu *et al.* [18] and Pecchia *et al.* [49] studied the logging practices in industrial software projects. Fu *et al.* [18] investigated what types of code snippets were logged and analyzed the factors that impact developers' logging decisions. Pecchia *et al.* [49] observed that the logging practices are strongly developer dependent, and highlights the need to establish standard company-wide logging practices. Yuan *et al.* [61], Chen *et al.* [8], Kabinna *et al.* [27], [28] and Shang *et al.* [55] studied the evolution of logging code in open-source projects. They observed that developers spend many efforts on updating their logging code (e.g., modifying logging statements or upgrading logging libraries). Chen *et al.* [9] and Hassani *et al.* [21] characterized logging-related issues in open-source projects and proposed automated solutions to detect logging-related issues. Recently, Li *et al.* [34] performed a qualitative study on the benefits and costs of logging in general, which found that developers consider a wide range of benefits and costs when making their logging decisions. Our work complements the existing studies by studying the practices of logging exception stack traces, which is an important yet under-investigated area.

Automated Solutions for Logging Improvements. Yuan *et al.* proposed *Errlog* [60] and *LogEnhancer* [62] that proactively log additional log information to the source code. *Errlog* [60] detects unlogged exceptions (abnormal or unusual conditions) and automatically insert the missing logging statements. *LogEnhancer* [62] automatically adds causally-related information on existing logging statements to aid in future failure diagnosis. Prior studies also leverage statistical models to learn *where to log*. *LogAdvisor* [63], *LogOpt* [32], *SmartLog*[25] and Li *et al.* [33] extracted contextual features of a code snippet (e.g., a exception catch snippet), then learned statistical models to suggest whether a logging statement should be added to such a code snippet. Our work is different from the "where to log" papers from two important aspects. First, our work suggests whether an exception logging statement needs a stack trace, whereas the "where to log" approaches suggest which code snippet needs a logging statement. Second, although some of our features (e.g., exception type) are similar to the ones used in the "where to log" approaches, most of our features are very different. Li *et al.* [35] also proposed an approach to automatically suggest the most appropriate log level for a logging statement, based on its contextual features. Similarly, a recent study by Li *et al.* [37] leverages deep neural networks to suggest log levels. In addition, prior work proposed approaches to automatically suggest the content of logging (e.g., log variables [40] and log text [23]). In addition to prior work, our work proposes an automated solution to help developers make informed decisions for logging the stack trace of an exception. Our work encourages future work on automated solutions for logging improvements to consider the aspect of logging exception stack traces, as stack traces can usually grow log files much faster than regular log messages, which can leverage the insights provided in this work (e.g., the factors that impact the logging of exception stack traces).

Studies on Exception Handling Practices. There are many studies on the handling of exceptions in the source code. Prior studies examined exception handling practices and patterns in the source code [11], [12], [13], [42], [44], [54], [58]. For example, Nakshatri *et al.* [44] studied exception handling patterns in open-source Java projects. They found that logging and printing stack traces are two of the top operations for exception handling, which reflects the importance of studying the logging of exception stack traces. De Pádua *et al.* [11] studied exception handling practices in Java and C# projects. They found that only a small portion of exceptions are handled in a *specific* way, while the majority of exceptions are handled with a *generic* strategy (i.e., handling generic exceptions instead of specific exceptions). Our work recommends that stack traces should be logged for such *generic* handling strategy. Sena *et al.* and De Pádua *et al.* studied exception handling anti-patterns [54] and the prevalence of the anti-patterns [12]. Prior work also studied program bugs that are related to exception handling [10], [15], [16], [29], [47]. For example, Ebert *et al.* [15], [16] studied exception handling bugs in Java programs and proposed a classification of exception handling bugs (e.g., general catch block). In addition, prior work proposed approaches to improve the adoption of exception handling policies [4], [43] and recommend exception handling strategies [36] or code [2], [3], [46]. While these studies focus on exception handling practices in the source code and their impact on program behaviors, we studied the handling of exceptions from the logging point of view, which has a minimum impact on the source code but poses a large impact on the generation, analysis, and management of log data.

6 Conclusion

Logging of exception stack traces is a popular practice in open-source projects. However, due to the lack of guidelines, developers log the exception stack traces in an *ad hoc* manner and they make many changes to modify their logging of exception stack traces. Through a comprehensive study of the source code, code change history, and issue reports of ten open-source Java projects, we present a first picture of developers' practices of logging exception stack traces. We recommend that developers should pay extra attention when deciding whether to log the stack trace of an exception. In particular, logging of stack traces should be avoided or downgraded for user errors, normal execution, expected exceptions, in user interfaces, or when there is a security concern. On the other hand, logging of stack traces for generic exceptions, severe problems, and unexpected

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

exceptions is recommended. We encourage that guidelines for the logging of exception stack traces be developed company-wide or globally, which should consider several important factors such as the exception type, the method that throws the exception, and the log level. We also recommend that logging frameworks improve their flexibility by allowing developers and users to configure the logging of stack traces on demand.

REFERENCES

[1] R. Artstein and M. Poesio, "Inter-coder agreement for computational linguistics," *Comput. Linguistics*, vol. 34, no. 4, pp. 555–596, 2008.

[2] E. A. Barbosa and A. Garcia, "Global-aware recommendations for repairing violations in exception handling," *IEEE Trans. Softw. Eng.*, vol. 44, vol. 9, pp. 855–873, Sep. 2017.

[3] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *Proc. 26th Braz. Symp. Softw. Eng.*, 2012, pp. 171–180.

[4] E. A. Barbosa, A. Garcia, M. P. Robillard, and B. Jakobus, "Enforcing exception handling policies with a domain-specific language," *IEEE Trans. Softw. Eng.*, vol. 42, no. 6, pp. 559–584, Jun. 2016.

[5] T. Barik, R. DeLine, S. Drucker, and D. Fisher, "The bones of the system: A case study of logging and telemetry at microsoft," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 92–101.

[6] R. Bender and S. Lange, "Adjusting for multiple testing - When and how?," *J. Clin. Epidemiol.*, vol. 54, no. 4, pp. 343–349, 2001.

[7] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[8] B. Chen and Z. M. (Jack) Jiang, "Characterizing logging practices in java-based open source software projects – A replication study in apache software foundation," *Empirical Softw. Eng.*, vol. 22, no. 1, pp. 330–374, Feb 2017.

[9] B. Chen and Z. M. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 71–81.

[10] R. Coelho, L. Almeida, G. Gousios, and A. Van Deursen , "Unveiling exception handling bug hazards in android based on github and google code issues," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 134–145.

[11] G. B. De Padua and W. Shang, "Revisiting exception handling practices with exception flow analysis," In *Proc. 17th IEEE Int. Work. Conf. Source Code Anal. Manipulation*, 2017, pp. 11–20.

[12] G. B. De Padua and W. Shang, "Studying the prevalence of exception handling anti-patterns," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension*, 2017, pp. 328–331.

[13] G. B. de Pádua and W. Shang, "Studying the relationship between exception handling practices and post-release defects," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 564–575.

[14] R. Ding *et al.*, "Log2: A cost-aware logging mechanism for performance diagnosis," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 139–150.

[15] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *J. Syst. Softw.*, vol. 106, pp. 82–101, 2015.

[16] F. Ebert, F. Castor, and A. Serebrenik, "A reflection on "an exploratory study on exception handling bugs in java program," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng.*, 2020, pp. 552–556.

[17] D. Freeman, R. Pisani, and R. Purves., *Statistics*, 4th ed., New York, NY, USA: W.W. Norton & Company, 2007.

[18] Q. Fu *et al.*, "Where do developers log? An empirical study on logging practices in industry," in *Proc. Companion 36th Int. Conf. Softw. Eng.*, 2014, pp. 24–33.

[19] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 789–800.

[20] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 145–155.

[21] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Softw. Eng.*, vol. 23, no. 6, pp. 3248–3280, 2018.

[22] A. F. Hayes and K. Krippendorff, "Answering the call for a standard reliability measure for coding data," *Commun. Methods Measures*, vol. 1, no. 1, pp. 77–89, 2007.

[23] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 178–189.

[24] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman, "Scottknott: A package for performing the scott-knott clustering algorithm in r," *TeMA - J. Land Use, Mobility Environ.*, vol. 15, no. 1, pp. 3–17, 2014.

[25] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "Smartlog: Place error log statement by deep understanding of log intention," in *Proc. 25th IEEE Int. Conf. Softw. Anal., Evol. Reeng.*, 2018, pp. 61–71.

[26] R. Kabacoff., *R in Action*. Shelter Island, NY, USA: Manning Publications Co., 2011.

[27] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the apache software foundation projects," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 154–164.

[28] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Softw. Eng.*, vol. 23, no. 1, pp. 290–333, 2018.

[29] M. Kechagia, M. Fragkoulis, P. Louridas, and D. Spinellis, "The exception handling riddle: An empirical study on the android API," *J. Syst. Softw.*, vol. 142, pp. 248–270, 2018.

[30] K. Krippendorff, "Computing krippendorff's alpha-reliability," Accessed: Aug. 28, 2019. [Online]. Available: http://repository.upenn.edu/asc_papers/43, 2011

[31] K. Krippendorff, "Reliability," in *Content analysis: An Introduction to its Methodology*, 4th ed., Thousand Oaks, CA, USA: Sage Publications, 2019, pp. 277–356.

[32] S. Lal and A. Sureka, "Logopt: Static feature extraction from source code for automated catch block logging prediction," in *Proc. 9th India Softw. Eng. Conf.*, 2016, pp. 151–155.

[33] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 2655–2694, 2018.

[34] H. Li, W. Shang, B. Adams, M. Sayagh, and A. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2020.2970422.

[35] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?," *Empirical Softw. Eng.*, vol. 22, no. 4, pp. 1684–1716, 2017.

[36] Y. Li *et al.*, "Eh-recommender: Recommending exception handling strategies based on program context," in *Proc. 23rd Int. Conf. Eng. Complex Comput. Syst.*, 2018, pp. 104–114.

[37] Z. Li, H. Li, T.-H. P. Chen, and W. Shang, "DeepLV: Suggesting log levels using ordinal based neural networks," in *Proc. 43rd Int. Conf. Softw. Eng.*, 2021, pp. 1461–1472.

[38] L. Liao *et al.*, Using black-box performance models to detect performance regressions under varying workloads: An empirical study," *Empirical Softw. Eng.*, page to appear, vol. 25, pp. 4130–4160, 2020.

[39] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.

[40] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 2012–2031, Sep. 2021.

[41] D. Lo, N. Nagappan, and T. Zimmermann, "How practitioners perceive the relevance of software engineering research," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 415–425.

[42] H. Melo, R. Coelho, and C. Treude, "Unveiling exception handling guidelines adopted by java developers," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng.*, 2019, pp. 128–139.

[43] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa, "Improving developers awareness of the exception handling policy," in *Proc. IEEE 25th Int. Conf. Softw. Analy., Evol. Reeng.*, 2018, pp. 431–422.

[44] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in java projects: An empirical study," in *Proc. 13th Work. Conf. Mining Softw. Repositories*, 2016, pp. 500–503.

[45] A. Newman, "Logging exceptions in java: Don't catch general java exceptions," Accessed: Aug. 28, 2019. [Online]. Available: https://www.loggly.com/blog/logging-exceptions-in-java/

[46] T. Nguyen, P. Vu, and T. Nguyen, "Code recommendation for exception handling," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1027–1038.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.

[47] T. T. Nguyen, P. M. Vu, and T. T. Nguyen, "An empirical study of exception handling bugs and fixes," in *Proc. 2019 ACM Southeast Conf.*, 2019, pp. 257–260.
[48] N. Nurmuliani, D. Zowghi, and S. P. Williams, "Using card sorting technique to classify requirements change," in *Proc. 12th IEEE Int. Requirements Eng. Conf.*, 2004, pp. 240–248.
[49] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 169–178.
[50] G. Rugg and P. McGeorge , "The sorting techniques: A tutorial paper on card sorts, picture sorts and item sorts," *Expert Syst.*, vol. 22, no, 3, pp. 94–107, 2005.
[51] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 255–265.
[52] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories*, 2010, pp. 118–121.
[53] A. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, vol. 30, no. 3, pp. 507–512, 1974.
[54] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of java libraries: An empirical study," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 212–222.
[55] W. Shang *et al.*, "An exploratory study of the evolution of communicated information about the execution of large software systems," *J. Softw., Evol. Process*, vol. 26, no. 1, pp. 3–26, 2014.
[56] D. Spencer, *Card Sorting: Designing Usable Categories*. New York, NY, USA: Rosenfeld Media, 2009.
[57] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.
[58] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 496–506.
[59] E. W. Weisstein. Bonferroni correction. 2004. [Online], Available: https://mathworld. wolfram. com/
[60] D. Yuan *et al.*, "Be conservative: Enhancing failure diagnosis with proactive logging," in *Proc. 10th USENIX Conf. Oper. Syst. Des. Implementation*, 2012, pp. 293–306.
[61] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 102–112.
[62] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2011, pp. 3–14.
[63] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proc. 37th Int. Conf. Softw. Eng.* , 2015, pp. 415–425.
[64] T. Zimmermann, "Card-sorting: From text to themes," in *Proc. Perspectives Data Sci. Softw. Eng.*, L. W. Tim Menzies and T. Zimmermann, Ed, 2016, pp. 137–141.

Heng Li received the BEng the degree from Sun Yat-sen University, China, the MSc degree from Fudan University, China, and the PhD degree in computing from Queen's University, Canada. He is currently an assistant professor with the Department of Computer and Software Engineering, Polytechnique Montreal, Canada, where he leads the Maintenance, Operations and Observation of Software with intelligence Lab. He was with Synopsys as a software engineer and with BlackBerry as a software performance engineer for several years. His research interests include software observability, intelligent operations of software systems, software log mining, software performance engineering, and mining software repositories.

Haoxiang Zhang recevied the BSc degree in physics from the University of Science and Technology of China, the MSc and PhD degrees in electrical engineering from Lehigh University, and the PhD degree in computer science from Queen's University, Canada. He is currently a senior researcher with the Centre for Software Excellence, Huawei, Canada. His research interests include empirical software engineering, mining software repositories, and intelligent software analytics.

Shaowei Wang recevied the BSc degree from Zhejiang University and the PhD degree from Singapore Management University. He is currently an assistant professor with the Department of Computer Science, University of Manitoba. He leads Software Management, Maintenance, and Reliability Lab (Mamba). His research interests include software engineering, machine learning, data analytics for software engineering, automated debugging, and secure software development. He authored or coauthored in flagship conferences and journals, including *Frontiers in Science and Engineering*, *Automated Software Engineering*, *IEEE Transactions on Software Engineering*, *ACM Transactions on Software Engineering and Methodology*, and *Empirical Software Engineering*. He is currently a program committee member of international conferences in the field of software engineering, including ASE, ICSME, SANER, and ICPC, and a regular reviewer for software engineering journals, including *Journal of Systems and Software*, *Empirical Software Engineering*, and *IEEE Transactions on Software Engineering*. He was the recipient of the 2018 Distinguished Reviewer Award for the Springer *Empirical Software Engineering* (SE's highest impact journal).

Ahmed E. Hassan (Fellow, IEEE) received the PhD degree in computer science from the University of Waterloo. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He was or is currently on the editorial boards of *IEEE Transactions on Software Engineering*, Springer *Journal of Empirical Software Engineering*, and *PeerJ Computer Science*. He is currently an ACM SIGSOFT influential educator, a TCSE distinguished educator, an NSERC Steacie fellow, Canada research chair in software analytics, and NSERC/BlackBerry software engineering chair with the School of Computing, Queen's University, Canada.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:50:05 UTC from IEEE Xplore. Restrictions apply.