A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives

Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan

Abstract—Software developers insert logging statements in their source code to collect important runtime information of software systems. In practice, logging appropriately is a challenge for developers. Prior studies aimed to improve logging by proactively inserting logging statements in certain code snippets or by learning *where to log* from existing logging code. However, there exists no work that systematically studies developers' logging considerations, i.e., the benefits and costs of logging from developers' perspectives. Without understanding developers' logging considerations, automated approaches for logging decisions are based primarily on researchers' intuition which may not be convincing to developers. In order to fill the gap between developers' logging considerations and researchers' intuition, we performed a qualitative study that combines a survey of 66 developers and a case study of 223 logging-related issue reports. The findings of our qualitative study draw a comprehensive picture of the benefits and costs of logging from developers' perspectives. We observe that developers consider a wide range of logging benefits and costs, while most of the uncovered benefits and costs have never been observed nor discussed in prior work. We also observe that developers use *ad hoc* strategies to balance the benefits and costs of logging. Developers need to be fully aware of the benefits and costs of logging, in order to better benefit from logging (e.g., leveraging logging to enable users to solve problems by themselves) and avoid unnecessary negative impact (e.g., exposing users' sensitive information). Future research needs to consider such a wide range of logging benefits and costs when developing automated logging strategies. Our findings also inspire opportunities for researchers and logging library providers to help developers balance the benefits and costs of logging, for example, to support different log levels for different parts of a logging statement, or to help developers estimate and reduce the negative impact of logging statements.

Index Terms—software logging, issue reports, developer survey, qualitative analysis.

1 Introduction

Developers insert logging statements in the source code to collect valuable runtime information of software systems. Logging statements produce execution logs at runtime, which are usually appended onto log files or the standard output. Logs play important roles in the daily tasks of developers and other software practitioners [2]. For example, logs are usually the only available resource for diagnosing field failures [38].

Prior studies find that developers face challenges when making logging decisions [3, 40]. Therefore, prior studies have proposed automated approaches to help developers improve their logging decisions, through *proactive logging* [39, 41] and *learning to log* [12, 20, 21, 22, 44]. *Proactive logging* approaches use static analysis to automatically add more logged information to the existing code, in order to improve software failure diagnosis. *Learning to log* approaches, on the other hand, learn statistical models from existing logging practices and further leverage the models to provide logging suggestions.

However, there exists no work that systematically studies developers' logging considerations, i.e., the benefits and costs of logging that developers consider when making logging decisions. Without a clear understanding of developers' logging considerations, automated approaches for logging improvement might not be convincing to developers. On the other hand, developers may not be fully aware of the benefits and costs of logging, and in some cases raise conflicting views (see the discussions in Section 5). Therefore, this work performs a qualitative study to understand the benefits and costs of logging from developers' perspectives, as well as how developers balance such benefits and costs.

Our qualitative study combines a survey of developers and a case study of logging-related issue reports. In our survey, we asked developers open-ended questions regarding their logging considerations and how they balance the benefits and costs of logging. In our case study, we derive developers' logging considerations that are communicated in the logging-related issue reports. While the survey provides us with developers' general opinions regarding their logging considerations, the case study reveals developers' logging considerations in the context of specific scenarios (i.e., logging-related issues). For example, issue report HADOOP-13693[1] raises a concern that logging an error message for a successful operation is confusing and misleading.

In particular, our qualitative study aims to find answers to the following three research questions (RQs).

RQ1: *What are the benefits of logging from developers' perspectives?*

Without understanding the benefits of logging from developers' perspectives, automated approaches for

- *Heng Li, Mohammed Sayagh, and Ahmed E. Hassan are with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada.*
  E-mail: {hengli, msayagh, ahmed}@cs.queensu.ca
- *Weiyi Shang is with the Department of Computer Science and Software Engineering, Concordia University, Canada*
  E-mail: shang@encs.concordia.ca
- *Bram Adams is with MCIS, Polytechnique Montreal, Canada.*
  E-mail: bram.adams@polymtl.ca

1. https://issues.apache.org/jira/browse/HADOOP-13693

logging improvement may not meet developers' real needs. We observe that developers consider a wide range of logging benefits. While some logging benefits (e.g., *diagnosing runtime failures* and *using logs as a debugger*) are discussed in prior studies, most of our newly uncovered benefits of logging (e.g., *user/customer support* and *system comprehension*) have never been observed nor discussed in prior studies.

RQ2: *What are the costs of logging from developers' perspectives?*
Prior work proposed automated approaches to enrich logging information in the source code, which usually only considered the *performance overhead* to evaluate the costs of the added logging information. However, we observe that developers are actually concerned about a much wider range of logging costs. In particular, some important costs of logging (e.g., *exposing sensitive information*) are ignored by both prior work and most developers.

RQ3: *How do developers balance the benefits and costs of logging?*
Prior work proposed automated approaches to help developers insert logging statements in a systematic manner, e.g., proactively adding logging information into certain code snippets (e.g., exception *catch* blocks) or using statistical models to learn *where to log*. However, we observe that developers use *ad hoc* strategies to balance the benefits and costs of logging. For example, in addition to proactively determining *where to log*, developers may add logging statements over time on demand. In addition to deciding on the most appropriate log levels in the first place, developers may evaluate the impact of logging and refactor log levels afterward.

Our work helps developers and researchers understand a wide range of logging benefits and costs. Developers need such a comprehensive understanding in order to make better use of logging (e.g., to enable customers to solve problems themselves using logs) and avoid unnecessary logging costs (e.g., exposing users' sensitive information). Future research needs to consider our newly uncovered benefits and costs of logging when developing automated logging strategies. Our work also encourages logging library providers and researchers to put efforts into improving current *ad hoc* logging practices. For example, logging library providers could improve their logging libraries by providing more flexible logging options (e.g., to support different log levels for different parts of a logging statement). Future research could help developers leverage the benefits of logging while minimizing logging costs (e.g., to help developers estimate and reduce the negative impact of logging, or to help developers improve the quality of logging).

Paper organization. The remainder of the paper is organized as follows. Section 2 surveys prior studies that are relevant to our study of developers' logging considerations. Section 3 describes our qualitative study methodology. Section 4 presents our findings that answer our research questions. Section 5 discusses the implications of our findings. The threats to the validity of our findings are discussed Section 6. Finally, Section 7 concludes the paper.

2 Related Work

2.1 Automated Logging Improvement

Prior work on logging improvement does not fully consider the benefits and costs of logging from developers' perspectives. Prior studies propose *LogEnhancer* [41], *Errlog* [39] and *Log20* [43] that proactively add additional log information in the source code to maximize the debugging capability of logging. *LogEnhancer* [41] automatically adds causally-related information on existing logging statements to aid in future failure diagnosis. *Errlog* [39] analyzes the source code to detect unlogged exceptions (abnormal or unusual conditions) and automatically insert the missing logging statements. *Log20* [43] automates the placement of logging statements such that the informativeness of the placed logging statements are maximized under a performance overhead threshold. In comparison, *Log*^{2} [6] and *Log4Perf* [37] proactively insert logging statements in the source code for performance monitoring and diagnosis. Although these studies claim to add minimal performance overhead, they fail to consider other logging costs that are discussed in this work. Future proactive logging tools should fully consider the benefits and costs of logging that are discussed in this work.

Instead of proactively inserting logging information to the source code, prior studies also leverage statistical models to learn *where to log*. *LogAdvisor* [44], *SmartLog* [12] and Li *et al.* [20] extract contextual features of a code snippet (e.g., an exception catch snippet), then learn statistical models to suggest whether a logging statement should be added to such a code snippet. Li *et al.* [21] also propose an approach to automatically suggest the most appropriate log level for a logging statement, based on its contextual features. Liu *et al.* [23] use representative learning to rank the variables in the source code based on their likelihood of being logged. These approaches aim to help developers make informed logging decisions by providing logging suggestions. However, without a clear understanding of developers' logging considerations, automated logging suggestions might not be convincing to developers. We believe that the logging benefits and costs uncovered in this paper can help researchers better understand the logging considerations from developers' perspective and provide guidance for future work on learning *where to log*.

2.2 Studying Logging Practices

Prior work on logging practices studies the artifacts and the maintenance efforts of logging instead of considering why developers make their logging decisions in the first place. Prior work studies the characteristics of current logging practices. Fu *et al.* [8] study the logging practices in two industrial software projects. They investigate what categories of code snippets (e.g., exception *catch* blocks) are logged. Another study of industrial logging practices [28] observes that logging behaviors are strongly developer dependent, and highlights the need to establish standard company-wide logging policies. Yuan *et al.* [40], Chen *et al.* [3], Shang *et al.* [32], and Kabinna *et al.* [14, 15] study

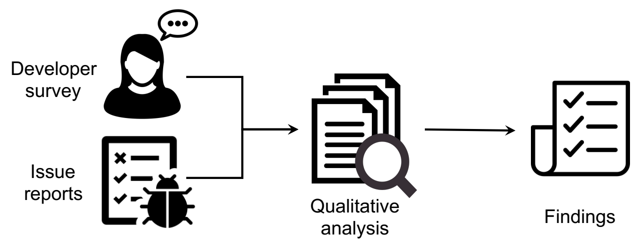

Fig. 1. Overview of our qualitative study.

the evolution of logging code in open source projects. They observe that developers spend much effort on updating their logging code (e.g., modifying logging statements or upgrading logging libraries). He *et al.* [11] find that developers use repetitive text in their logging statements and demonstrate the flexibility of automatically generating logging text. While these studies focus on developers' logging behaviors, our work goes deeper and study developers' considerations of the benefits and costs of logging behind their logging behaviors.

Barik et al. [2] study the activities of using event data (logs and telemetry) that are performed by different roles in Microsoft. Cito et al. [5] study how developers utilize data and tools (including log data and log management & processing tools) in the software development process in a cloud environment. While these studies focus on how developers and other roles utilize log data, our work studies the benefits and costs of logging from developers' perspectives and how developers balance the benefits and costs of logging.

Chen *et al.* [4] and Hassani *et al.* [9] investigate logging-related issues in open source projects and propose automated solutions to detect such issues. Chen et al. [4] study log-fixing patterns in open source projects and characterize logging anti-patterns (e.g., *nullable objects*). Hassani et al. [9] study the characteristics of logging-related issues and summarize the root causes of logging-related issues (e.g., *inappropriate log level*). In comparison, our work provides a comprehensive understanding of developers' logging considerations (i.e., the benefits and costs of logging from developers' perspectives).

3 Research Methodology

As shown in Figure 1, we combine a survey of developers and a case study of logging-related issue reports in our qualitative study. In the survey, we directly collect answers from developers about their logging considerations, which provides us with a comprehensive understanding of developers' logging considerations from a general perspective. In comparison, the case study provides us a more detailed context of real-life scenarios that are described in logging-related issue reports. As discussed in Section 4, the findings derived from the survey and those derived from the case study of issue reports complement each other.

3.1 Qualitative Study Part I (Developer Survey)

3.1.1 Survey design

In order to answer our research questions (see Section 1), we asked developers the following three *open-ended* questions:

Q1: Based on your experience, what are the benefits of logging that you consider when making logging decisions?
Q2: Based on your experience, what are the costs (negative impact) of logging that you consider when making logging decisions?
Q3: How do you usually balance the benefits and costs of logging?

We asked these questions in an open-ended way to get free opinions from developers. We did not want the results to be impacted by the limitations of our understandings by asking closed-ended questions. We also asked an additional question about developers' interest in the outcomes of the survey:

Q4: Would you like to keep in touch about the outcomes of this survey?

We sent these questions to each of our survey participants in an email. In the next sub-section, we describe how we selected participants in our survey.

3.1.2 Participants

We selected participating developers who had considerable logging experiences in top-rated open source projects, for two reasons: 1) developers who are experienced in logging are more likely to have a better understanding of the benefits and costs of logging; 2) the top-rated projects are likely to have a higher requirement for logging quality, which is desired for our study of logging.

Specifically, we selected open source software projects under the Apache Software Foundation[2] (ASF) that received the most starts on GitHub[3]. We selected software projects under ASF because ASF incubated many popular software projects (e.g., *Hadoop*, *Maven*, *Spark*, *JMeter*, *Kafka*, *NetBeans*, *Tomcat*, etc.) that are widely used in various domains of today's software industry. Besides, ASF projects usually use standard logging libraries (e.g., SLF4J[4]), which allows us to track developers' logging experience in a large number of projects.

For each selected open source project, we analyzed its code change history and tracked each developer's log changes (i.e., adding/deleting/modifying logging statements). Then, we selected those developers who changed at least five logging statements in the past years (i.e., March 1, 2016 to March 1, 2019) as our participants.

We did not pre-determine the number of projects and the number of participants at the beginning of the survey. Instead, we first sent our emails to the selected developers of a few most top-rated projects. Based on the responses from the surveyed developers, we gradually added more projects and more participants until we received enough responses. At the end, we sent our questions to 1,398 developers across 55 different projects. We received responses from 66 participants across 31 different projects, with a response rate

2. https://www.apache.org
3. https://github.com/apache
4. https://www.slf4j.org

Answer to Q1 (Benefits of logging):
* Being able to inspect the current/recent of the program without interrupting it
* Being able to inspect previous program executions (e.g., for comparison purposes)
* Being able to establish the temporal order of things when it's hard to do so in the IDE, e.g., due to breakpoints messing up timings in a multithreaded execution context
* Communicate liveness and progress, which may be otherwise hard to indicate (i.e., what is my process doing now?)

Answer to Q2 (Costs of logging):
* Logging often equals printouts, so there are performance costs involved making it inappropriate for hot paths.
* Excessive logging clutters the log file /screen, making it difficult to actually identify anything of value.
* Large logs may take ages to analyze (load in text editors) and even just to grep over.
* Large logs have a tendency to make disks run out of space, causing all kinds of cascading (and sometimes fatal) operational issues

Answer to Q3 (Balancing logging benefits and costs):
* Personally I'd say I'm not very generous with logging by default. I tend add them only when I see a direct value, e.g. after having dealt with an issue which these logs helped resolve and may help resolve/identify in the future.
* If I really have to log a hot path, I usually do so by logging once in T time / once in N invocations.
* I also use log levels (DEBUG, INFO, etc) to control what gets logged when. Some logging infrastructures even allow changing these log levels for various components at run time, though this often requires tempering with log configurations which I don't think anyone particularly likes to do.

Fig. 2. An example response to our survey invitations.

of 5%. The 31 projects cover a variety of domains, including big data (e.g., *Hadoop*), cloud computing (e.g., *CloudStack*), database (e.g., *HBase*), network client/server (e.g., *Camel*), testing (e.g., *JMeter*), build management (e.g., *Maven*), web framework (e.g., *Nutch*), content (e.g., *PDFBox*), and library (e.g., *Mahout*).

3.1.3 Survey responses

Most of the developers who responded to our survey invitations provided high-quality responses. Such responses usually contain valuable opinions of developers regarding their logging experience and logging considerations. Figure 2 shows an example response provided by a participating developer. In this response, the developer talked about multiple benefits (e.g., *communicating liveness and progress*) and costs (e.g., *performance costs*) of logging, as well as approaches to balance the benefits and costs of logging (e.g., *aggregating logging in hot paths*), based on his own experience. Table 1 shows some statistics of the length of the responses. An average response has more than 900 characters.

3.1.4 Qualitative analysis of survey data

We used an open card sorting approach [29, 33, 45] to code the survey data according to our research questions. 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) [24, 26, 31]. In our open card sorting approach, we aimed to derive high-level answers to our

TABLE 1
Statistics about the length (number of characters) of the survey responses.

Statistics	Q1 answer	Q2 answer	Q3 answer	All answers
Max length	1,140	2,063	1,910	4,578
Avg length	334	331	270	935
Median length	271	203	203	725
Min length	39	11	6	66

research questions from the responses provided by the participating developers. For each answer to each survey question, we can assign multiple labels. For example, for the Q2 answer in the response shown in Figure 2, we can assign multiple labels for logging costs (e.g., performance cost, and hiding valuable information). We did not print our content (i.e., survey responses) on physical cards. Instead, we use the Emacs Org Mode[5] to organize our codes and quotes. We also use a GitHub repository to track the history of our coding process. 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 responses to the three coders. Each coder coded one-third of the responses separately, which took several hours up to one day for each coder to finish their portion.

Step 2: Discussions after round-1 coding. The goal of the discussions is to reach the same codes among the coders. We had a few meetings to discuss our resulting codes and reached consensus. Each meeting took one to two hours.

Step 3: Revisiting round-1 coding. Based the updated codes from our discussions, we revisited our separate round-1 coding results, which took one to two hours for each of us.

Step 4: Round-2 coding. Each coder coded another one-third of the responses separately, based on the codes resulting from round-1. Each portion of the survey responses 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 response was coded by two different coders in the two rounds. Coders could add new codes in round-2. Round-2 coding took several hours for each of us to finish our separate portion.

Step 5: Discussions after round-2 coding. We had one more meeting to discuss our separate codes updated in round-2 and reached consensus. The meeting took two 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 one to two hours for us to finish our respective portions. We measured our inter-coder agreement (see Section 3.3) after this step.

Step 7: Resolving disagreement. We discussed every conflict in our coding results and reached an agreement.

5. https://orgmode.org

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.

3.2 Qualitative Study Part II (Issue Reports)

3.2.1 Subject Projects

In order to study developers' logging considerations, we manually investigated the logging-related issue reports from three large and successful open source software projects, namely *Hadoop Common*[6], *Hive*[7], and *Kafka*[8]. We focused only on three projects as we needed to have a good understanding of the source code and the runtime behaviors of these projects, in order to better understand the context of the logging-related issue reports. *Hadoop Common* implements the common utilities for Hadoop, a distributed computing platform. *Hive* is a data warehouse that supports accessing big data sets residing in distributed storage using SQL. *Kafka* is a streaming platform for messaging, storing and processing real-time records. All of these projects are widely used by today's tech giants, such as Google, Amazon, and Facebook. We selected these three subject projects because their logging code has been actively developed and maintained. For example, they have many logging-related issue reports that are dedicated to adding, removing, and improving logging code, besides fixing logging-related bugs. As the log messages that are generated by these projects are exposed to the aforementioned tech giants as well as a much wider audience, the quality of their logging code might be critical to their success. Besides, all of these projects use JIRA[9] as their issue tracking systems, making it convenient for us to extract issue report data.

We study the logging-related issue reports that were created from June 2012 to June 2017. We extracted the logging issue data in December 2017 (at least six months after the creation of any studied issue report), to ensure that the status of the studied logging issues is relatively stable after a long time since their creation.

3.2.2 Collecting logging-related issue reports

We extracted our logging issues from the Apache JIRA issue tracking system[10]. Figure 3 demonstrates our data extraction process. First, we used the JIRA Query Language (JQL) to automatically search for the JIRA issues reports that are related to logging (i.e., issue reports with logging-related keywords in their summaries). We used the JQL query in Figure 4 to search for the logging issue reports of each of the studied projects. The "Project Name" is replaced by "Hadoop Common", "Hive", and "Kafka" for our respective projects. This JQL query searches for all the issue reports of the specified project that have "log", "logger" , "print" , or their variations (e.g., "logging"), but don't have "log in", "log out", "blue print", or "print" and "command" together,

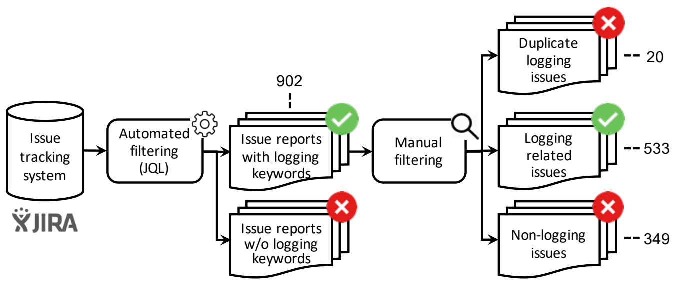

Fig. 3. Our process of collecting logging-related issue reports.

```
project in ("Project Name") AND summary ~ "(log || logger || print)
      NOT (\"log in\" || \"log out\" || \"blue print\" || \"print
      command\"~10)" ORDER BY created DESC
```

Fig. 4. The JQL query that we used to search for the logging-related issue reports.

in its summary, sorted by their creation time using a reverse-chronological order.

The resulting issue reports from the automated filtering process may falsely include some non-logging issue reports. For example, issue report HADOOP-14060[11] has "log" in its summary but it is about the access control for the "logs" folder instead of a logging issue. In order to remove these non-logging issue reports, we manually examined all the resulting issue reports from the automated filtering process. For each issue report, we first checked its summary to determine if it is a logging issue. If we could not decide it from the summary, we further checked the description of the issue report. We only kept the issue reports that deal with logging issues. We also removed duplicated logging issue reports and kept only one issue report for each duplication. We ended up with 533 logging-related issue reports.

Table 2 shows the number of issue reports that we obtained from the automated filtering process and the number of remaining issue reports after the manual filtering process (i.e., the number of logging issue reports that are studied in the rest of the paper). Using our query criterion (i.e., Figure 4), we get 193, 395 and 314 issue reports for *Hadoop Common*, *Hive* and *Kafka*, respectively. 74% and 68% of the JQL-queried issue reports are concerned with logging for the *Hadoop Common* and *Hive* projects, respectively. However, only 39% of the JQL-queried issue reports are concerned with logging for the *Kafka* project. As the *Kafka* project deals with messaging, storing and processing of log messages, it has a large number of issue reports with the keyword "log" (or its variations) in their summaries but they are not necessarily related to the logging aspect of the project.

3.2.3 Qualitative analysis of issue reports

We used a similar approach as the one we used for the qualitative analysis of the survey data (Section 3.1.4) to code the logging-related issue reports. Our goal is not to find the root causes of the reported issues. Instead, we aim to understand developers' considerations of the benefits and costs of logging that are communicated during the

6. http://hadoop.apache.org
7. https://hive.apache.org
8. https://kafka.apache.org
9. https://www.atlassian.com/software/jira
10. https://issues.apache.org/jira
11. https://issues.apache.org/jira/browse/HADOOP-14060

TABLE 2
Number of studied logging issue reports per project.

Project	# JQL-queried issues	# Logging issues
Hadoop Common	193	143 (74%)
Hive	395	268 (68%)
Kafka	314	122 (39%)
Total	902	533 (59%)

reporting, discussion, and fixing processes of the issues, as well as developers' strategies of balancing logging benefits and costs that are reflected in the issue reports. For each logging-related issue report, we examined the summary, description, comments, patches, and the associated code review comments to extract useful information that answers our research questions. Investigating an issue report usually involves much effort to understand the context and developers' considerations that are reflected in their interactions with the issue report. Therefore, instead of analyzing all the 533 logging-related issue reports, we randomly chose a statistically representative sample of 223 issue reports, which ensures a 95% confidence level on a 5% confidence interval.

We coded the issue reports using the existing codes that we derived from the survey responses as seeding codes. We also added new codes when we coded the issue reports. In other words, we used a hybrid card sorting approach [45] to code the issue reports. We followed similar steps as described in Section 3.1.4 to code our issue reports, except that we only performed one round of coding as we reused the codes derived from the survey responses. The first four authors of the paper (i.e., coders) jointly performed the coding process, following the steps below:

Step 1: Initial coding of issue reports. Each issue report was randomly assigned to two coders. Coders code the issue reports using the codes derived from the survey responses, while being allowed to add new codes. This step took a few days for each coder to finish their separate portion.

Step 2: Discussions after the initial coding. We had two meetings to discuss our separate codes updated in the initial coding of the issue reports and reached consensus. Each meeting took one to two hours. We finalized the codes for the issue reports after this step.

Step 3: Revisiting the issue-report coding. Based on the updated codes from our discussions, we revisited our coding for the issue report data, which took a few days for each of us to finish our separate portion. We measured our inter-coder agreement (see Section 3.3) after this step.

Step 4: 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 issue report discussed and tried to resolve it; if an agreement could not be reached, a third coder was involved and voting was conducted if necessary. We resolved the disagreement after a three-hour meeting followed by a one-hour meeting.

TABLE 3
Reliability of our qualitative study measured by Krippendorff's α.

	Krippendorff's α
Coding the survey responses	0.825
Coding the issue reports	0.815

3.3 Measuring the Reliability of our Qualitative Study

Reliability is a prerequisite for ensuring the validity of the coding results [1, 18]. 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, 18].

Krippendorff's α. In this work, we use Krippendorff's α [10, 18] 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, 10], which takes the form of:

$$\alpha = 1 - \frac{D_o}{D_e} \qquad (1)$$

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 agreement [17]. The detailed methodology for calculating Krippendorff's α is described in prior work [17].

As the coded categories are non-exclusive in our coding scheme (i.e., we can assign multiple categories to each survey response or issue report), we cannot directly use Krippendorff's α which requires a single value to be assigned to each coded item. Instead, we treat each category for each coded item as a coding unit and measure the coders' agreement on the coding units. In other words, we measure the coders' agreement on whether a particular category (e.g., a logging cost) should be assigned to a coded item (e.g., a survey response).

4 FINDINGS

In this section, we present the findings of our qualitative study. We first provide a high-level overview of our findings in Section 4.1. Then, we present more detailed results for answering our research questions in Section 4.2 (**RQ1**), Section 4.3 (**RQ2**), and Section 4.4 (**RQ3**).

4.1 Overview

86% of the survey respondents indicated their interest in the outcomes of our survey. Among the 66 survey respondents, 57 (86%) of them provided positive answers to our survey question Q4 (see Section 3.1.1), for example:

> *"Feel free to send your results to the Hadoop Common Dev list. Others in the community may find it interesting."*

The results of our qualitative study are reliable. As shown in Table 3, our coding of the survey response data achieves a Krippendorff's α of 0.825, and our coding of the issue report data achieves a Krippendorff's α of 0.815. Krippendorff [18] suggests that $\alpha \geq 0.800$ indicates a reliable agreement.

While logging-related research usually considers a single benefit or cost of logging, developers consider a much wider range of logging benefits and costs. For example, prior studies [39, 41, 43] only consider the benefit of supporting debugging and the cost of performance overhead when proactively adding extra logging information in the source code. In our survey, 97% (64 out of 66) and 100% of the respondents mentioned the benefits and costs of logging, respectively. There are a median of two types of logging benefits and two types of logging costs mentioned by each developer in our survey. Overall, developers consider a variety of logging benefits (see Table 4) and logging costs (see Table 5). In fact, some respondents do not take into consideration the performance overhead which was used as the only criterion to evaluate logging costs in prior work [39, 41, 42, 43].

> *"There is virtually no overhead and it makes sense to log as much as possible. Of course there is a problem of runaway log length which should be considered." [S]*[12]

Developers balance the benefits and costs of logging in *ad hoc* ways. In our survey, all the respondents shared their approaches for balancing logging benefits and costs. Each developer mentioned a median of two strategies for balancing the benefits and costs of logging. Overall, developers consider a variety of strategies (Table 6) to balance the benefits and costs of logging. They usually find it difficult to determine appropriate logging in the first place. Instead, they maintain their logging code by continuous trials-and-errors.

> *"Besides constant arguing whether certain log-lines need to be INFO, DEBUG or whatever with colleagues, the log-format produced is also changed from time to time due to changes in the tooling support or services used to analyse the logs automatically (i.e. create issue tickets via Sentry)." [S]*

As an important step, this work aims to understand the benefits and costs of logging from developers' perspectives. In the rest of this section, we present the detailed findings of our qualitative study, in particular, as answers to our research questions.

4.2 Benefits of Logging

Table 4 summarizes the benefits of logging that we derived from our collected survey responses and issue reports. In total, we derived eight main categories of logging benefits, each of which may include several second-level categories and third-level categories. These categories fall into a few broader themes: **assisting in troubleshooting**, **tracking execution status**, **assisting in comprehension**, and **bookkeeping**. In the rest of this sub-section, we present the main categories of logging benefits under each theme.

4.2.1 Assisting in troubleshooting

Diagnosing runtime failures. When there is a system failure, the first action is to figure out *what is wrong*, for example, whether it's a user configuration error or a software bug. Logs are usually an important or even the only source for diagnosing a system runtime failure (e.g., a field failure) [2, 5, 6]. This category does not include the benefit of using logs as a debugger to trace down execution paths.

> *"The upmost priority in our case is to be able to state the reasons for a request failing and whether it was our fault or the user's fault by sending an erroneous request." [S]*

Using logs as a debugger. When the diagnosis finds there is a bug in the software, logs can be used as a debugger to help developers narrow down the execution paths and find the root cause of the software failure [2, 5, 38, 39, 41, 43]. In particular, logging specific information instead of general information (e.g., HIVE-11163[13]), logging the causes of an error in addition to the error itself (e.g., KAFKA-4164[14]), and logging the stack trace of an unexpected exception (e.g., HADOOP-13682[15]) in addition to the exception message can effectively help developers narrow down the root causes.

User/customer support. Logs are directly exposed to users and customers. Well-developed logs can provide actionable suggestions for users to solve their problems by themselves when using the software:

> *"If you do it well enough, your customers will be able to solve their own issues, or at worst be well informed when filing a support case. " [S]*

4.2.2 Tracking execution status

Tracking execution progress. Logs help track the status or progress of an execution, such as the start or end of an event (e.g., HIVE-11314[16]), a status change (e.g., flip over a flag, HADOOP-10046[17]), an ongoing action (e.g., retrying, HADOOP-10657[18]), or the status of waiting for some resources (e.g., waiting for a lock, HIVE-14263[19]). In particular, printing the progress information for a process that takes a long time is important for figuring out what's going on in the process. For example, issue report KAFKA-5000[20] requests regular progress information to be logged for a long process so that one can know whether the process is progressing or stuck.

> *"Communicate liveness and progress, which may be otherwise hard to indicate (i.e., what is my process doing now?)" [S]*

Monitoring & Alerting. Logs can alert developers and users for problems or anomalies during system execution, in real-time [2] or postmoterm analysis [2, 7, 36]. For example, issue report HADOOP-12901[21] requests to log a "warn" message when a client fails to connect to a server, so that users can easily identify and fix the connection problem. Anomaly detection tools [7, 13, 36] automatically analyze large amounts of log messages (that are hard for human beings to investigate manually) and alert anomalies that are indicated in the log messages.

Logging statements are also used to record the performance information of a system at runtime (*a.k.a.*, per-

12. "[S]" indicates a quote from the survey of developers; "[I]" indicates a quote from the issue reports.

13. https://issues.apache.org/jira/browse/HIVE-11163
14. https://issues.apache.org/jira/browse/KAFKA-4164
15. https://issues.apache.org/jira/browse/HADOOP-13682
16. https://issues.apache.org/jira/browse/HIVE-11314
17. https://issues.apache.org/jira/browse/HADOOP-10046
18. https://issues.apache.org/jira/browse/HADOOP-10657
19. https://issues.apache.org/jira/browse/HIVE-14263
20. https://issues.apache.org/jira/browse/KAFKA-5000
21. https://issues.apache.org/jira/browse/HADOOP-12901

TABLE 4
The categories of logging benefits derived from the survey responses and the issue reports. The column "Quote" shows an example quote for the corresponding category which is extracted from the survey responses ("[S]") or the issue reports ("[I]"); the column "Freq(S/I)" shows the frequencies of the corresponding category in the survey responses ("S") and the issue reports ("I"); the column "Ref./New" lists literature that observes or discusses a related concept, or indicates a new category that has never been observed nor discussed in prior work.

Benefit	**Quote**	**Freq(S/I)**	**Ref./new**
Diagnosing runtime failures		**35/28**	
Configuration error diagnosis	"Logging application configurations reveals errors caused by misconfigurations"[S]	2/2	New
Field failure diagnosis	"Easier to find the root cause if there is a bug or system failure"[S]	28/18	[2, 5]
Test failure diagnosis	"During integration and end-to-end testing or manual testing the logged exception stack traces often immediately reveals the cause of a test failure"[S]	2/2	New
Diagnosing performance issues	"See performance impact"[S]	1/4	[5, 6]
Diagnosing security access issues	"During Dev stage, logs help mainly on UT stages, integration with other systems, identifying access issues"[S]	1/2	New
Triaging failures	"To be able to figure out the responsibility if something goes wrong"[S]	1/0	[2]
Using logs as a debugger		**39/38**	
Generic debugging	"Help identify the root cause of the problems"[S]	19/30	New
Production debugging	"When users report bugs, we want to collect more info about the state in which bugs happens"[S]	12/3	[2, 5] [38, 39] [41, 43]
Development-time debugging	"During Dev stage, logs help mainly on UT stages"[S]	8/5	New
Users/customer support		**8/8**	
Enabling user troubleshooting	"Enable users troubleshooting by themselves"[S]	6/6	New
Increasing usability	"It increases the framework usability via feeding user with important information"[S]	2/2	New
Tracking execution progress		**13/9**	
Tracking execution progress (immediate feedback)	"They are very important to track the workflow"[S]	8/5	New
Providing ongoing event status	"It often makes sense to periodically log some ongoing event status"[S]	3/4	New
Checking liveness of the system	"If users ever feel 'why is it taking so long, is it even doing something?' , they could check log to see if it is doing something"[S]	2/0	New
Monitoring & Alerting		**11/16**	
Real time alerting	"Things that are fatal should always be logged to help with quick debugging"[S]	4/0	[2]
Anomaly indicator	"An analyzing of logs can you discover bugs before your soft will be deployed"[S]	5/16	[2, 7, 36]
Enabling tool monitoring	"Monitoring logs by external tools allows to integrate alerts based on presence of WARN/ERROR statements at logs"[S]	2/0	New
Performance monitoring	"We should log a warning message if group resolution takes too long"[I]	0/5	[2, 13] [19, 37]
System comprehension		**45/17**	
Code familiarization	"Logging statements provide additional insights when reading the code, and are equally or even more useful than code comments"[S]	2/0	New
System runtime familiarization			
Understanding the dynamics of a scenario	"Being able to establish the temporal order of things when it's hard to do so in the IDE"[S]	*12/1*	New
Understanding the major events at run-time	"I use logging show when 'key' events happen in a system"[S]	*6/3*	New
Providing context of events	"I add this one for the information why decision was made"[S]	*10/13*	New
Communicating internal state	"Main benefit is to provide information about system state, i.e. printing values of fields of a class, printing the result of a computation"[S]	*10/0*	New
Verifying behaviors vs. expectations	"I want to know that software executes the path that I intended/assumed"[S]	5/0	New
Assisting in developing software		**4/2**	
Easing developing software	"During POC logs allows you to make better decisions when and how to use alternative components or alternative flows"[S]	3/1	New
Better coding practice	"Some developers just use STDOUT and STDERR. I don't think I need to tell you why that is a bad idea"[S]	1/1	New
Bookkeeping		**9/4**	
Gathering statistics of run-time events	"The main benefit of logging I see is an availability to log any needed data in a free form for subsequent analyzing"[S]	4/1	[2, 16] [27, 35]
Auditing	"Auditing ('WHO?' - and 'WHAT?') - it's sometimes a bit outside of the logging context, but many cases data can be gathered through a log parser/shipper"[S]	5/3	[30, 42]

formance monitoring) [2, 19, 37]. Such performance information is usually related to the execution time or resource usage of a process. For example, issue report HIVE-14922[22] requests the logging of the time spent in several performance-critical tasks, and issue report KAFKA-4044 requests the logging of the actually used buffer size. Such performance information helps in understanding system health (e.g., HIVE-8210[23]), tuning system performance (e.g., HADOOP-13301[24]), and adjusting resource allocation (e.g.,

22. https://issues.apache.org/jira/browse/HIVE-14922
23. https://issues.apache.org/jira/browse/HIVE-8210
24. https://issues.apache.org/jira/browse/HADOOP-13301

KAFKA-4044[25]).

4.2.3 Assisting in comprehension

System comprehension. In addition to providing clues for resolving problems and tracking execution status, logs also help developers get familiar with the source code (i.e., functioning like comments) and the runtime behaviors of a system when there are no failures (e.g., the ordering of events). Besides, logs can help developers verify the system runtime behaviors against the expected behaviors.

> *"The logs need to tell a story." [S]*

> *"Logging statements provide additional insights when reading the code, and are equally or even more useful than code comments." [S]*

Assisting in developing software. Logs can actually make developing software easier. During the development process, developers can get insights from logs to evaluate the execution flows and make better software development decisions.

> *"These might give valuable insights on the lack of quality within the own code logic or documentation or give hints on a lack of considerations when designing (or implementing) the application to start with." [S]*

4.2.4 Bookkeeping

Bookkeeping. Developers can use log messages to record (i.e., bookkeep) important transactions or operations in a system execution, such as user login/logout, database operations, and remote queries and requests. Such bookkeeping log information can be later processed for various analysis activities, such as security analysis [27], performance analysis [35], and capacity planning [16]. The Sarbanes-Oxley Act of 2002 [30] requires all telecommunication and financial applications to log some mandatory log events, such as user activities, network activities and database activities[26].

> *"Logging may act as business-operation metering provider." [S]*

Summary of RQ1

Developers consider a wide range of logging benefits, covering the dimensions of troubleshooting, tracking, comprehension, and bookkeeping. While some logging benefits are discussed in prior studies, most of our newly uncovered benefits of logging have never been observed nor discussed in prior studies. Fully understanding the benefits of logging can help developers make better use of logging (e.g., to enable customers to solve problems themselves using logs). The uncovered benefits of logging also inspire new research opportunities on logging improvement (e.g., to improve logging for system comprehension).

25. https://issues.apache.org/jira/browse/KAFKA-4044
26. https://sarbanes-oxley-101.com/sarbanes-oxley-audits.htm

4.3 Costs of Logging

Table 5 summarizes the costs of logging that we derived from the survey responses and the issue reports. In total, we derived nine main categories of logging costs, which fall into four broader themes: **costs of managing and processing large log data**, **impacting system behaviors**, **direct negative impact on users**, and **increasing development efforts**. In the rest of this sub-section, we present the main categories of logging costs under each theme.

4.3.1 Costs of managing and processing large log data

Storage cost. Excessive logging can lead to expensive storage costs [2, 6]. Excessive log information is usually caused by repetitive log messages, such as logging database operations on every row of a table (e.g., HIVE-8153), logging every entry (e.g., KAFKA-3792[27]), logging every request (e.g., KAFKA-3737[28]), or logging every user (e.g., HADOOP-12450[29]). In particular, repetitive logging of stack traces usually grows the log files very fast and frustrates the end-users. For example, issue report HADOOP-11868[30] raises a major concern about the excessive logging of stack traces for invalid user logins.

> *"Also because there are already too many log-generating in Hadoop, we can easily have hundreds of GB logs single cluster per day." [S]*

Producing noise that hides important information. Excessive logging can also make it hard to "sift through the *noise* to find out what is actually important when diagnosing a failure". In particular, logging normal events or properly handled problems at the "warn" or "error" levels can spam the log files with "warn" or "error" messages and make it difficult to identify real problems (e.g., HIVE-8382[31]).

> *"Too much log kills the logs." [S]*

Effort on log collection, processing and management. For large software systems, excessive logging also increases the effort for collecting, processing, and managing logs [2, 5, 6]. In particular, consuming a large volume of logs from different sources is difficult [2].

> *"Aggregation, cross-referencing, and searching through logs is hard." [S]*

4.3.2 Impacting system behaviors

Performance overhead. Performance overhead is considered as a major cost of logging [6, 39, 41, 42, 43], as printing a log message into a log file involves expensive IO operations, string concatenations, and possible method invocations for generating the log strings. One cause of performance overhead is overwhelmingly repetitive printing of similar log messages. For example, issue report HIVE-12312[32] complains that the compilation of a complex query significantly slowed down as a code snippet with a logging statement is called thousands of times. The execution process was speeded up by 20% after disabling the logging statement.

27. https://issues.apache.org/jira/browse/KAFKA-3792
28. https://issues.apache.org/jira/browse/KAFKA-3737
29. https://issues.apache.org/jira/browse/HADOOP-12450
30. https://issues.apache.org/jira/browse/HADOOP-11868
31. https://issues.apache.org/jira/browse/HIVE-8382
32. https://issues.apache.org/jira/browse/HIVE-12312

TABLE 5
The categories of logging costs derived from the survey responses and the issue reports.

Cost	Quote	Freq(S/I)	Ref./new
Storage cost	"Log can fill your disk *very* quickly"[S]	**27/11**	[2, 6]
Producing noise that hides important information	"Too much verbose logging makes it difficult to find useful information"[S]	**40/38**	New
Effort on log collection, processing and management		**10/16**	
Effort for log processing	"Consumption is difficult. Aggregation, cross-referencing, and searching through logs is hard"[S]	7/1	[2, 6]
Effort on log collection & management	"There is the management overhead of keeping and archiving log files etc."[S]	3/13	[5, 6]
Missing logging info that causes extra effort	"Currently we don't log exceptions raised when reading from the local log which makes tracking down the cause of problems a bit tricky"[I]	0/2	New
Performance overhead	"Affects running speed of cpu and memory"[S]	**44/10**	[6, 39, 41] [42, 43]
Perturbing system behaviors	"When you observe a system, you are perturbing it"[S]	**2/5**	[4, 9]
Confusing users		**8/88**	
Incorrect severity level that causes confusion	"Logging with the incorrect severity level can result in confused users thinking there is something terribly wrong when in actuality things are normal"[S]	4/27	[4, 9, 21]
Unnecessary exceptions for normal situations	"This causes the error stream handling thread to log an exception backtrace for a 'normal' situation"[I]	0/7	New
Not actionable info	"Some costs of logging are the parsing of un-meaningful statements"[S]	3/12	New
Unnecessary support cost	"We eliminated some logging output just because people were scared. Sometimes we even get support calls on DEBUG output"[S]	1/0	New
Incorrect logging info	"When printing out a Struct, we don't print out the content wrapped in an array properly"[I]	0/19	[4, 9, 22]
Incorrect documentation of logging	"Description of hive.server2.logging.operation.level states that this property is available to set at session level. This is not correct"[I]	0/4	New
Inconsistent logging info that causes confusion	"If 'closing connection' is logged, it is better to also log the 'creating connection' event, otherwise the 'closing connection' log would not be very useful"[I]	0/4	New
Missing logging info that causes confusion	"When the filter on DelegationTokenAuthenticationFilter is called it hits an exception there and there is no log message there. This leads to the confusion that we have had a success while the exception happens in the next class"[I]	0/15	New
Exposing sensitive information	"Logs may ... expose some sensitive data"[S]	**2/9**	New
Logging code development and maintenance cost		**13/30**	
Effort on maintenance of logging code	"Logs can create additional bugs"[S]	7/24	[3, 6, 14] [15, 22] [32, 40]
Slow down development / effort of using logging API & management	"The biggest cost of logging is probably setting up an implementation and configuring something like Log4j"[S]	6/6	New
Decreasing code readability	"I think it decreases code readability and quality when there are a lot of log statement between the main logic"[S]	**6/1**	New

Another cause of performance overhead is the invocation of expensive methods in logging statements. For example, issue report HADOOP-14369[33] complains that including some method calls in logging statements is "pretty expensive". Surprisingly, even disabled lower level logging can cause serious performance overhead, because the parameters of a logging statement are evaluated before the check for the log level. For example, issue KAFKA-2992[34] reports that "trace" logs in tight loops cause significant performance issues even when "trace" logs are not enabled.

Perturbing system behaviors. Logging code is usually considered as "side code" (i.e., the code that won't impact the normal behaviors of a software system). However, our survey respondents also mentioned that logging can critically impact the functional behaviors of a software system. Prior work also observed that logging issues can cause system runtime failures (e.g., triggering a *NullPointerException*) [4, 9].

> *"There is a real possibility that logging effects the core operation, a logging bug that only shows up when trying to troubleshoot another issue can be critical." [S]*

4.3.3 Direct negative impact on users

Confusing users. As log messages are directly exposed to end-users, inappropriate log information can be confusing and misleading [4, 9, 21, 22]. In particular, logging "warn" or "error" messages for successful operations is the most frequent cause for this concern. For example, HADOOP-13693[35] complains that a warning in a successful operation confuses end-users. Even worse, sometimes inappropriate log messages can annoy or frustrate end-users. For example, HADOOP-13552[36] complains that there are too many

33. https://issues.apache.org/jira/browse/HADOOP-14369
34. https://issues.apache.org/jira/browse/KAFKA-2992
35. https://issues.apache.org/jira/browse/HADOOP-13693
36. https://issues.apache.org/jira/browse/HADOOP-13552

"scary-looking" stack traces being printed out in the log files, but in fact those exceptions can be handled automatically. Such large amount of repetitive stack traces can frustrate (e.g., HIVE-11062[37]) or annoy (e.g., HIVE-7737[38]) end-users.

Inconsistent logging can also cause user confusion. For example, when the creation of a certain object (e.g., a table) is logged, the deletion of the object should also be logged. Otherwise one cannot confirm if the object still exists (e.g., HIVE-13058[39]). Similarly, "waiting for lock" should be followed by "lock acquired" (HIVE-14263[40]), while "start of process" should be followed by "end of process" (HIVE-12787[41]).

Exposing sensitive information. Sensitive information (e.g., usernames and passwords) should not be printed in log files. Once such sensitive information is logged, it might be archived for years and cannot be tampered with due to legal regulations. However, sometimes such sensitive information might end up logged by mistakes. For example, issue reports HIVE-14098[42] complains that users' passwords are logged in cleartext. In particular, developers have difficulties to avoid logging sensitive information that is contained in an URL, e.g., user names and passwords (HIVE-13091[43]), or a user configuration field, e.g., cloud storage keys (HADOOP-13494[44]). Developers may log the content of a URL or a configuration field without noticing the contained sensitive information. In particular, users may put their sensitive information in an unknown configuration field that is caused by a typo. In such a case, developers are likely to log the unknown configuration field (i.e., to alert the unknown configuration) and unintentionally expose users' sensitive information (e.g., KAFKA-4056[45]).

4.3.4 Increasing development efforts

Logging code development and maintenance cost. First, the costs for developing and maintaining logging code come from "setting up an implementation and configuring something like Log4j" and "constant arguing whether certain log-lines need to be INFO, DEBUG or whatever with colleagues". Second, developers need to constantly evolve their logging code to keep it updated with the evolving business logic of the source code [15]. In fact, prior studies [3, 14, 15, 22, 32, 40] find that developers spend much effort updating their logging code.

Decreasing code readability. In Section 4.2, we mentioned that logging (functioning like comments) can help developers better understand the source code. However, it is interesting that other survey respondents complained that logging can also decrease the readability of other code.

> *"Apart from those, I find it harder to read the code when too many log statements are sprinkled into the code." [S]*

37. https://issues.apache.org/jira/browse/HIVE-11062
38. https://issues.apache.org/jira/browse/HIVE-7737
39. https://issues.apache.org/jira/browse/HIVE-13058
40. https://issues.apache.org/jira/browse/HIVE-14263
41. https://issues.apache.org/jira/browse/HIVE-12787
42. https://issues.apache.org/jira/browse/HIVE-14098
43. https://issues.apache.org/jira/browse/HIVE-13091
44. https://issues.apache.org/jira/browse/HADOOP-13494
45. https://issues.apache.org/jira/browse/KAFKA-4056

Summary of RQ2

While *too much logging* can increase the cost of managing & processing large log data and impact system runtime behaviors (e.g., performance), logging has a more direct impact on developers (i.e., increasing development and maintenance efforts) and users (i.e., confusing users and exposing users' sensitive information). Developers need to fully understand the costs of logging in order to avoid unnecessary negative impact (e.g., exposing users' sensitive information). Future research also needs to be aware of the wide range of logging costs when developing automated logging strategies.

4.4 Balancing the Benefits and Costs of Logging

Table 6 summarizes the approaches for balancing the benefits and costs of logging that we derived from the survey responses and the issue reports. In total, we derived nine main categories of approaches, which fall into four broader themes: **differentiating logging**, **where to log**, **reducing logging impact**, and **improving logging quality**. In the rest of this sub-section, we present the main categories of the approaches under each theme.

4.4.1 Differentiating logging

Appropriate log levels. The most common approach for balancing the benefits and costs of logging is to assign appropriate log levels for logging statements with different levels of importance [4, 6, 9, 21, 27, 40], which also comes with configuring the appropriate log levels when running a software system. In particular, supporting dynamic configuration of log levels (e.g., using JMX) provides better flexibility for developers and users to tradeoff between logging benefits and costs (i.e., on the fly).

> *"Use different log levels: Log exceptions, missing configuration properties as error or warning, api calls at server side and some relevant parameter values (like ids) as info, state of internal (package level) objects as debug or trace." [S]*

There is a strong need for the support of logging different parts (in particular, stack traces) of a logging statement at different log levels, which is not supported by modern logging libraries. Many issue reports suggest, for a single event, to log the regular log text at a higher level (e.g., "error") and the stack trace at a lower level (e.g., "debug"), such that the stack traces are hidden in normal cases and only printed out when needed (e.g., HADOOP-13669[46]). It is also suggested to log the important information of an event at a higher level, while logging the detailed information of the same event at a lower level (e.g., KAFKA-1199[47]).

Differentiate different logging purposes. In addition to using different log levels, developers also suggested other approaches to differentiate different logging purposes, e.g., using different loggers. For example, it is suggested to log performance information using standardized performance

46. https://issues.apache.org/jira/browse/HADOOP-13669
47. https://issues.apache.org/jira/browse/KAFKA-1199

TABLE 6
The categories of approaches for balancing the benefits and costs of logging (derived from the survey responses and the issue reports).

Approach	Quote	Freq(S/I)	Ref./new
Appropriate log levels		**30/53**	
Developing right log levels	"Use the proper levels of logs, make sure this information would be relevant for either monitoring or debugging"[S]	14/26	[4, 9, 21] [27, 40]
Evaluating and refactoring log levels	"We tend to leave most of logging statements added when initially writing a code, and maybe just decrease logging level when merging the final version of code"[S]	6/16	[6]
Configuring right levels in system deployment	"Running the application with the right log levels"[S]	4/9	New
Supporting dynamic configuration of log levels	"By utilizing the power of JMX we are able to update the log configuration at runtime for certain classes and thus modify the output produced by logs"[S]	4/2	New
More debug logs and less production logs	"Lots of debug level logs, because it helps when in dev. Way, way less log for production, just the minimum needed to understand where an error is coming from."[S]	2/0	New
Differentiating different logging purposes		**10/8**	
Failing loudly	"When something crashes out- it should be apparent and easy to find"[S]	1/1	New
Adding format to help post filter	"Adding some meaningful prefixes makes it easier to grep logs"[S]	2/3	New
Considering user experience of logs	"Maintain a human-processable picture of the scenario played through with both INFO and DEBUG level activated"[S]	7/1	New
Using different loggers for different purposes	"Perflogger should log under single class, so that it could be turned on without mass logging spew"[I]	0/3	New
Proactively determining appropriate scope/focus of logging		**33/17**	
Logging uncommon/unexpected situations	"I put logs only to unusual and interesting cases, when some special event can happen"[S]	11/5	[8, 12, 44]
Logging important events / state changes	"I only log key events that I feel are important to applications/systems (usually some type of state change)"[S]	8/10	New
Project/team specific style/rules	"I tend to go with the flow (of the existing project) and try to be a Good Scout"[S]	1/2	[2, 28]
Other tips for where (not) to log	"Over-logging is way better than no logging at all"[S]	13/0	New
Reactive logging		**14/22**	
Adding logs over time on demand	"In cases where the traceability of logs for a certain error case isn't given or optimal we introduce new logs over time or adapt the logging level of existing logs"[S]	6/11	New
Logging buggy places	"In places were we had some bugs, we add additional logging when performing the fix"[S]	6/2	New
Following a logging budget	"Adjust the amount of log by trial and error"[S]	2/1	[6, 43]
Removing unnecessary logs	"Remove unnecessary log line in common join operator"[I]	0/8	[6]
Minimizing repeated logging		**10/14**	
Avoiding logging in frequent operations	"Avoid logging for every iteration in case of loops to ensure it doesn't slow down the executioNewpplication"[S]	3/5	[6]
Avoiding duplicate logging statements	"I try to log every exception only once: do not log and rethrow the same exception in the same catch block but log it if it is not rethrown"[S]	5/5	New
Aggregating log messages	"For repeated actions, such as in a loop, aggregate together and log"[S]	2/4	New
Considering logging impact		**10/7**	
Do not change program behaviour	"Logging shouldn't give a chance to fail application"[S]	1/1	New
Keeping performance impact in mind	"The impact from a performance point of view needs to be fully understood"[S]	7/6	[6, 39] [41, 43]
Performance is not a problem	"Slowdowns is not so important"[S]	2/0	New
Ensuring quality of logging code		**6/58**	
Manually testing the logging code before deploying system-wide	"Performance testing and simply observing and thinking about noise level in the logs are two things that are done while testing and in code reviews"[S]	4/1	New
Using unit tests for logging code	"Unit tests can reveal logging statement issues"[S]	1/1	New
Redacting/masking sensitive info	"Anonymize passwords before logging"[I]	0/8	New
Providing appropriate context			
Correlation IDs	"It would be useful to log the correlation id of cancelled inflight requests"[I]	0/17	[28]
Exceptions	"This is not useful to understand the underlying cause. The WARN entry should additionally include the exception text"[I]	0/8	New
Error details	"We should log a better message, to include more details (e.g. token type, username, tokenid) for trouble-shooting purpose"[I]	0/13	New
Other context	"Logging minimally, but providing as much context as possible"[S]	1/7	New
Ensuring logging consistency			
Symmetric logging	"When connecting to HS2, we can confirm that some directories were created... But when closing from HS2, we cannot confirm that these directories were deleted. So I change it so that some messages about these directories deletion may be output as follows."[I]	0/2	New
Consistency between logging guard and statement	"Fix inconsistency between log-level guards and statements"[I]	0/1	New
Using advanced tooling to support logging		**23/20**	
Leveraging advanced logging libraries	"Use clever and efficient loggers to skip log statements that are disabled"[S]	6/8	[9, 14]
Runtime throttling	"Throttling is probably the most powerful trick. Instead of logging a message directly, we add a helper class to track how much log are generated during the time, do some in-place aggregation and log only the aggregate results"[S]	2/1	New
Using NOP loggers	"I don't use logging:-) Or rather I use nop loggers"[S]	1/0	New
Rotating/cleaning logs	"Rotating the logs to keep only the most relevant logs"[S]	4/5	[2, 6]
Using log processing tools	"Using tools like Splunk to store and search logs"[S]	3/0	[2]
Leveraging structured logging	"Structured log messages e.g. JSON can help counteract the complexity, making logs amenable to automated analysis"[S]	3/0	New
Leveraging logging guard	"Do not overuse log statements, use ifLoggable() to keep performance impact down"[S]	4/6	New
High configurability of logging		**0/18**	
Enabling turning on/off certain logs independently	"Provide a way to to independently configure perflogger and root logger levels"[I]	0/7	[2]
Enabling turning on/off stack traces	"Provide an option for IPC server users to avoid printing stack information for certain exceptions"[I]	0/1	New
Simplicity of configurations	"Our customer always want simple log4j configurations"[I]	0/1	New
Configurability of log locations	"Add KAFKA_LOG_DIR to allow LOG_DIR to be outside of code dir"[I]	0/9	New

loggers to separate performance logging from event logging, for better performance analysis (e.g., HIVE-11891[48]).

> *"Sometimes we configure loggers to write statements with level higher then DEBUG into one file, and all statements to another file, with different log rotation settings. Audit log is at a separate file." [S]*

4.4.2 *Where to log*

Developers combine proactive and reactive strategies to determine *where to log*.

Proactively determining appropriate scope/focus of logging. Prior work proactively adds additional log information in the source code [39, 41, 43] or learns statistical models to suggest *where to log* [12, 20, 44]. In our qualitative study, developers highlighted the importance of logging uncommon/unexpected situations (e.g., when an error happens) [8, 12, 44] and logging important events/state changes (e.g., when a connection is created). Besides, developers also suggested that logging should follow project or team-wise styles [2, 28].

> *"Make sure it has value (such as an error happens, a connection is created, etc.)" [S]*

Reactive logging. In addition to proactively determining *where to log*, developers also highlighted the need to update logging over time (i.e., reactively) as appropriate logging is difficult to achieve in the first place. Developers gradually add logging on demand or at buggy places. However, logging should follow a budget [6, 43], and unnecessary logs need to be removed [6].

> *"You should do improvement your logging step by step just like you do refactoring of code." [S]*

4.4.3 *Reducing logging impact*

Minimizing repeated logging. Developers suggested approaches to reduce logging impact by minimizing repeated logging. First of all, inserting logging statements in tight loops is considered as a bad logging practice [6]. It is also suggested to aggregate highly repetitive log messages, for example, by logging aggregated information at a higher log level and detailed information at a lower log level (e.g., HIVE-10214[49] and KAFKA-4829[50]). Developers also need to be cautious when throwing and logging an exception at the same time[51], because it may lead to duplicated logging as the handler of the exception may log the exception again [25, 34] (e.g., KAFKA-1591[52]).

Considering logging impact. Developers suggested to always consider the impact of logging, in particular, performance impact, when making logging decisions. Prior work [6, 39, 41, 43] also considered the performance impact of logging when automatically adding logging information in the source code. However, logging can cause performance problems for some software systems but not for others.

> *"The impact from a performance point of view needs to be fully understood." [S]*

4.4.4 *Improving logging quality*

Ensuring quality of logging code. The quality of the logging code impacts the benefits and costs of logging. Logging statements without enough context information (e.g., correlation IDs [28]) may hinder their usefulness. For example, developers suggested to add more context information (e.g., thread ID, session ID, query ID and user ID) in the logging statements of a multi-task program (e.g., HIVE-13517[53], HIVE-11488[54], KAFKA-3816[55], HIVE-15631[56] and HIVE-6876[57]).

As discussed in Section 4.3, logging statements with inappropriate content may confuse end-users. Logging statements even can cause additional bugs [4, 9]. Therefore, it is suggested to carefully test the logging code before releases.

Using advanced tooling to support logging. Another direction to improve logging quality is to leverage advanced logging tooling, such as advanced logging libraries (e.g., SLF4J) [9, 14] and logging processing tools (e.g., Splunk) [2]. As discussed in Section 4.3, even when a logging statement with a lower level is disabled, it still can have some performance overhead that is caused by the evaluation of the logging statement's parameters (e.g., concatenation of logging strings), as the parameters are evaluated before the logging method checks the log level. Advanced logging libraries (e.g., SLF4J) have recently started to support *parameterized logging statements*[58]. Parameterized logging statements delay the concatenation of logging strings until after checking if the log level is enabled, thereby avoiding unnecessary string concatenation when a log level is disabled. As a result, developers start to replace logging guards with parameterized logging (e.g., issue report HADOOP-13317[59]).

> *"We recently tansition to slf4j so can do parameterized logging removing if LOG.LEVEL.X gateway checks." [S]*

High configurability of logging. Developers also suggested enhancing the configurability of logging, such as enabling turning on/off certain logs independently and enabling turning on/off stack traces. For example, issue report HADOOP-8711[60] suggests to provide an option for users to suppress the stack traces triggered by certain exceptions. Prior work [2] also pointed out the need for the support of turning on/off logs at a fine granular level.

48. https://issues.apache.org/jira/browse/HIVE-11891
49. https://issues.apache.org/jira/browse/HIVE-10214
50. https://issues.apache.org/jira/browse/KAFKA-4829
51. https://www.loggly.com/blog/logging-exceptions-in-java/
52. https://issues.apache.org/jira/browse/KAFKA-1591
53. https://issues.apache.org/jira/browse/HIVE-13517
54. https://issues.apache.org/jira/browse/HIVE-11488
55. https://issues.apache.org/jira/browse/KAFKA-3816
56. https://issues.apache.org/jira/browse/HIVE-15631
57. https://issues.apache.org/jira/browse/HIVE-6876
58. https://logging.apache.org/log4j/2.x/performance.html
59. https://issues.apache.org/jira/browse/HADOOP-13317
60. https://issues.apache.org/jira/browse/HADOOP-8711

Summary of RQ3

Developers balance the benefits and costs of logging in an *ad hoc* manner. In addition to proactively determining *where to log*, developers highlight the need for logging reactively on demand. Besides, developers consider various *ad hoc* strategies that aim to differentiate logging purposes, reduce logging impact, and improve logging quality. We encourage logging library providers and researchers to put efforts into improving current *ad hoc* logging practices. For example, logging library providers could improve their logging libraries by providing more flexible logging options (e.g., to support different log levels for different parts of a logging statement).

5 Implications

Developers consider a diverse range of benefits and costs when making their logging decisions. Prior work only considers a few developers' logging considerations (e.g., the benefits of *field failure diagnosis* [2, 5], or the costs of *performance overhead* [6, 39, 41, 42, 43]). In comparison, this work provides a full picture of developers' logging considerations. As discussed in sections 4.2 and 4.3, we observe that developers consider a wide range of logging benefits and costs. Some of the logging benefits (e.g., *user/customer support*) and costs (e.g., *exposing sensitive information*) are considered by neither prior studies nor most of the survey respondents. Developers need to be fully aware of the benefits and costs of logging, in order to leverage logging better (e.g., to improve user/customer support with logs) and avoid unnecessary negative impact (e.g., exposing users' sensitive information). Future research needs to consider the wide range of logging benefits and costs when developing automated logging strategies. Our uncovered logging benefits and costs also inspire future research opportunities for filtering logging statements by their different benefits and costs, for example, to keep only the logging statements that are necessary for diagnosing runtime failures.

Developers have conflicting views about the benefits and costs of logging. As discussed in Section 4.3, *performance overhead* is the logging cost that is mentioned by most developers in our survey. However, as discussed in Section 4.1, some developers are not concerned about the performance overhead that is caused by logging and argue that "it makes sense to log as much as possible". Some developers mention that logging statements can help understand the source code (i.e., *code familiarization* in Table 4), while others argue that logging statements *decrease code readability* (see Table 5). Some developers claim that logging can *assist in developing software* (see Table 4), while others complain about the *logging code development and maintenance cost*. Therefore, the benefits and costs of logging need to be considered within the specific context of applications.

Developers balance the benefits and costs of logging in an *ad hoc* manner. It is challenging for developers to write appropriate logging code in the first place [3, 21, 22, 40]. Thus, developers usually maintain their logging code by continuous trials-and-errors. For example, developers usually reactively add logging statements on demand (i.e., *reactive logging* in Table 6). They also constantly re-evaluate and refactor log levels over time (i.e., *evaluating and refactoring log levels* in Table 6). Besides, developers' strategies for balancing the benefits and costs of logging are usually based on subjective judgment. For example, *considering logging impact* (see Table 4) is subject to each developer's judgment of *logging impact*. It is still a big challenge that lies in front of researchers and developers to derive systematic guidelines for logging. Future research is encouraged to help developers improve current *ad hoc* logging practices, for example, to help developers estimate and reduce the negative impact of logging, or to help developers improve the quality of logging). Future work is also encouraged to explore the idea of integrating logging considerations in early software design, in order to reduce the *ad hoc*ness of logging. For example, logging budget can be allocated for each subsystem/component during architecture design.

Developers need higher flexibility from logging libraries. As discussed in Section 4.4, developers need support for logging different parts (in particular, stack traces) of a logging statement at different log levels, which is not supported by modern logging libraries. Developers also highlight the importance of *supporting dynamic configuration of log levels* and *high configurability of logging* in general. Besides, in order to reduce logging impact, developers need supports for *aggregating log messages* and *runtime throttling* (see Table 6). Currently, developers use workarounds to mitigate the lack of support from logging libraries. For example, in order to log different parts of a logging statement at different levels, developers usually need to insert two separate logging statements at different log levels for a single event (e.g., HADOOP-11868[61]). In order to save developers' logging efforts, we suggest that logging library providers consider such developer needs for higher flexibility.

6 Threats to Validity

External Validity. This paper studies the benefits and costs of logging from the perspectives of the developers from 31 open source projects. The 31 open source projects are mainly Java projects, as Java is well supported by many good logging libraries and logging is a common practice in Java projects [3]. Developers of other software projects (e.g., closed source projects) may consider different aspects of logging. In particular, the numerical representations of the coding results (i.e., the frequencies of the codes) only indicate the distributions in the received survey responses and the studied issue reports. The 31 projects cover a variety of domains, including big data, cloud computing, database, network client/server, testing, build management, web framework, content, and library. Therefore, our findings may generalize to a much broader set of software projects. However, findings from additional case studies on other projects can benefit our study.

When investigating logging-related issue reports, we consider three open source projects that have active development and maintenance of logging code (e.g., with

61. https://issues.apache.org/jira/browse/HADOOP-11868

many logging-related issue reports). Having active logging development and maintenance does not necessarily mean these projects follow good logging practices. Nevertheless, as logs generated by these projects are exposed to a wide audience including many big tech companies, the quality of logging might be critical to these projects. We consider all the logging-related issue reports even though some of them remain unfixed by developers. However, our goal is not to find the root causes of the reported issues. Instead, we aim to understand developers' considerations of the benefits and costs of logging that are communicated in the issue reports. Future work is encouraged to study why some logging-related issue reports are not fixed by developers.

Internal Validity. In this paper, we study developers' logging considerations through a developer survey and a case study of logging-related issue reports. However, developers' logging considerations may also be expressed in other forms, such as requirement specifications or mailing-lists. Therefore, our findings may not provide a complete view of developers' logging considerations. In our study, the developer survey provides us with developers' general opinions about their logging considerations, while the case study reveals developers' logging considerations within the context of specific issue reports. Besides, in the three subject projects that we used for studying logging-related issue reports, an issue report is always needed for every code commit. Thus, we expect that our findings are representative of developers' logging considerations that involve code changes.

Construct Validity. In order to obtain high-quality survey responses, we selected those developers who changed at least five logging statements as our survey participants. We assumed that developers who changed a number of logging statements are experienced in logging. Our selection might miss developers who did not change as many logging statements in the studied projects but in other projects. Besides, our survey results may be biased by the 5% of the surveyed developers who responded to our surveys. For example, the developers who responded to our survey may be more positive towards logging. Future surveys at a larger scale could enhance our study.

We use qualitative analysis to study developers' logging considerations and how developers balance the benefits and costs of logging. Like other qualitative studies, our results may be biased by the individuals who conduct the qualitative analysis. In order to reduce the bias, three and four authors of the paper jointly performed the qualitative analysis to derive high-level concepts from the survey responses and the logging-related issue reports, respectively.

7 Conclusion

In order to understand developers' considerations of the benefits and costs of logging and how they balance such benefits and costs, we performed a qualitative study that combines a survey of 66 developers and a case study of 223 logging-related issue reports. Our study uncovers a wide range of logging benefits and costs. We also observe that developers balance the benefits and costs of logging in an *ad hoc* manner. Our findings provide insights for developers to improve their logging practices: to make better use of logging (e.g., to enable customers to solve problems themselves using logs) and avoid unnecessary logging costs (e.g., exposing users' sensitive information). Future work on automated logging improvement should also consider the wide range of logging benefits and costs. We encourage logging libraries to support more flexible logging capabilities, e.g., to support different log levels for different parts of a logging statement. Our findings also shed light on future research opportunities that help developers leverage the benefits of logging while minimizing logging costs (e.g., to help developers estimate and reduce the negative impact of logging).

References

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

[2] T. Barik, R. DeLine, S. Drucker, and D. Fisher, "The bones of the system: A case study of logging and telemetry at microsoft," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, ser. ICSE Companion '16, 2016, pp. 92–101.

[3] 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 Software Engineering*, vol. 22, no. 1, pp. 330–374, 2017.

[4] B. Chen and Z. M. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 71–81.

[5] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: An empirical study on software development for the cloud," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, pp. 393–403.

[6] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *2015 USENIX Annual Technical Conference*, ser. ATC '15, 2015, pp. 139–150.

[7] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proceedings of the 9th IEEE International Conference on Data Mining*, ser. ICDM '09, 2009, pp. 149–158.

[8] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? An empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion '14, 2014, pp. 24–33.

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

[10] A. F. Hayes and K. Krippendorff, "Answering the call for a standard reliability measure for coding data," *Communication methods and measures*, vol. 1, no. 1, pp. 77–89, 2007.

[11] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018, pp. 178–189.

[12] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "Smartlog: Place error log statement by deep understanding of log intention," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER '18, 2018, pp. 61–71.

[13] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of the 24th IEEE International Conference on Software Maintenance*, ser. ICSM '08, 2008, pp. 307–316.

[14] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, “Logging library migrations: A case study for the apache software foundation projects,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16, 2016, pp. 154–164.

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

[16] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, “An analysis of traces from a production mapreduce cluster,” in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID ’10, 2010, pp. 94–103.

[17] K. Krippendorff, “Computing krippendorff’s alpha-reliability,” Retrieved from http://repository.upenn.edu/asc_papers/43, 2011, accessed 28 August 2019.

[18] ——, “Reliability,” in *Content analysis: An introduction to its methodology, fourth edition*. Sage publications, 2019, ch. 12, pp. 277–356.

[19] H. Li, T.-H. P. Chen, A. E. Hassan, M. Nasser, and P. Flora, “Adopting autonomic computing capabilities in existing large-scale systems: An industrial experience report,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’18, 2018, pp. 1–10.

[20] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, “Studying software logging using topic models,” *Empirical Software Engineering*, vol. 23, no. 5, pp. 2655–2694, 2018.

[21] H. Li, W. Shang, and A. E. Hassan, “Which log level should developers choose for a new logging statement?” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, 2017.

[22] H. Li, W. Shang, Y. Zou, and A. E. Hassan, “Towards just-in-time suggestions for log changes,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, 2017.

[23] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, “Which variables should i log?” *IEEE Transactions on Software Engineering*, 2019.

[24] D. Lo, N. Nagappan, and T. Zimmermann, “How practitioners perceive the relevance of software engineering research,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 415–425.

[25] A. Newman, “Logging exceptions in java: Don’t log and throw,” Retrieved from https://www.loggly.com/blog/logging-exceptions-in-java/, 2015, accessed 28 August 2019.

[26] N. Nurmuliani, D. Zowghi, and S. P. Williams, “Using card sorting technique to classify requirements change,” in *Proceedings of the 12th IEEE International Requirements Engineering Conference*, ser. RE ’04, 2004, pp. 240–248.

[27] A. Oliner, A. Ganapathi, and W. Xu, “Advances and challenges in log analysis,” *Commun. ACM*, vol. 55, no. 2, pp. 55–61, 2012.

[28] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, “Industry practices and event logging: Assessment of a critical software development process,” in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE ’15, 2015, pp. 169–178.

[29] G. Rugg and P. McGeorge, “The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts,” *Expert Systems*, vol. 22, no. 3, pp. 94–107, 2005.

[30] P. Sarbanes, “Sarbanes-Oxley Act of 2002,” in *The Public Company Accounting Reform and Investor Protection Act*, 2002.

[31] M. Sayagh, N. Kerzazi, and B. Adams, “On cross-stack configuration errors,” in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE ’17, 2017, pp. 255–265.

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

[33] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[34] Stack Overflow Community, “Why is ”log and throw” considered an anti-pattern?” Retrieved from https://stackoverflow.com/questions/6639963/why-is-log-and-throw-considered-an-anti-pattern, 2011, accessed 28 August 2019.

[35] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, “Leveraging performance counters and execution logs to diagnose memory-related performance issues,” in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, ser. ICSM ’13, 2013, pp. 110–119.

[36] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP ’09, 2009, pp. 117–132.

[37] K. Yao, G. B. de Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, “Log4perf: Suggesting logging locations for web-based systems’ performance monitoring,” in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’18, 2018, pp. 127–138.

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

[39] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, “Be conservative: Enhancing failure diagnosis with proactive logging,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI ’12, 2012, pp. 293–306.

[40] D. Yuan, S. Park, and Y. Zhou, “Characterizing logging practices in open-source software,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12, 2012, pp. 102–112.

[41] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, “Improving software diagnosability via log enhancement,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’11, 2011, pp. 3–14.

[42] L. Zeng, Y. Xiao, and H. Chen, “Linux auditing: Overhead and adaptation,” in *2015 IEEE International Conference on Communications*, ser. ICC ’15, 2015, pp. 7168–7173.

[43] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, “Log20: Fully automated optimal placement of log printing statements under specified overhead threshold,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17, 2017, pp. 565–581.

[44] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, “Learning to log: Helping developers make informed logging decisions,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15, 2015, pp. 415–425.

[45] T. Zimmermann, “Card-sorting: From text to themes,” in *Perspectives on Data Science for Software Engineering*, L. W. Tim Menzies and T. Zimmermann, Eds. Burlington, Massachusetts: Morgan Kaufmann, 2016, pp. 137–141.

Heng Li is a postdoctoral fellow in the Software Analysis and Intelligence Lab (SAIL) at Queen's University, Canada. He obtained his Ph.D. from the School of Computing, Queen's University (Canada), M.Sc. from Fudan University (China), and B.Eng. from Sun Yat-sen University (China). He worked at Synopsys as a full-time Software Engineer before he started his PhD. During his PhD and postdoc fellowship, he kept close collaborations with the industry (e.g., Alibaba, BlackBerry). His research interests lie within Software Engineering, in particular, intelligent operations of software systems, software log mining, software performance engineering, and mining software repositories. Contact him at: hengli@cs.queensu.ca; https://www.hengli.org.

Weiyi Shang is an Assistant Professor and Concordia University Research Chair in Ultra-large-scale Systems at the Department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his Ph.D. and M.Sc. degrees from Queens University (Canada) and he obtained B.Eng. from Harbin Institute of Technology. His research interests include big data software engineering, software engineering for ultra-large-scale systems, software log mining, empirical software engineering, and software performance engineering. His industrial experience includes helping improve the quality and performance of ultra-large-scale systems in BlackBerry. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/˜shang.

Bram Adams is an associate professor at Polytechnique Montreal, where he heads the Lab on Maintenance, Construction, and Intelligence of Software. His research interests include release engineering in general, as well as software integration, software build systems, and infrastructure as code. Adams obtained his PhD in computer science engineering from Ghent University. He is a steering committee member of the International Workshop on Release Engineering (RELENG) and program co-chair of SCAM 2013, SANER 2015, ICSME 2016 and MSR 2019.

Mohammed Sayagh is a postdoctoral fellow in the Software Analysis and Intelligence Lab (SAIL) in Queen's University. He obtained his PhD from the Lab on Maintenance, Construction, and Intelligence of Software (MCIS) in Ecole Polytechnique Montreal (Canada). He obtained his engineering degree in Software Engineering from the Faculty of Science and Techniques in Mohammedia (FSTM) - Morocco. His research interests include software configuration engineering, as well as multi-layer and components architectures source code analysis. More details about his work is available on "https://sailhome.cs.queensu.ca/˜msayagh".

Ahmed E. Hassan is an IEEE fellow and member, ACM influential educator, an NSERC Steacie Fellow, a Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queens University, Canada. His industrial experience includes helping architect the Blackberry wireless platform, and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. Dr. Hassan serves on the editorial board of the IEEE Transactions on Software Engineering, the Journal of Empirical Software Engineering, and PeerJ Computer Science. He spearheaded the organization and creation of the Mining Software Repositories (MSR) conference and its research community. More information at https://sail.cs.queensu.ca/