Towards Providing Automated Supports to Developers on Making Logging Decisions and Log Analysis

Zhenhao Li

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montréal, Québec, Canada

September 2022

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Zhenhao Li**

Entitled: **Towards Providing Automated Supports to Developers on Making Logging Decisions and Log Analysis**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. Ahmed Kishk

_____ External Examiner
Dr. Zhenchang Xing

_____ Examiner
Dr. Abdelwahab Hamou-Lhadj

_____ Examiner
Dr. Yann-Gael Gueheneuc

_____ Examiner
Dr. Eugene Belilovsky

_____ Supervisor
Dr. Tse-Hsun Chen, Dr. Weiyi Shang

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

_____ 2022 _____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Towards Providing Automated Supports to Developers on Making Logging Decisions and Log Analysis

Zhenhao Li, Ph.D.

Concordia University, 2022

Due to the lack of practical guidelines on how to write logging statements and large volume of logs routinely generated by software products, how to make proper logging decisions and efficiently analyze the logs are challenging in practice. In this thesis, we focus on these two main challenges and propose a series of approaches to address the problem and help developers on logging practices in two aspects: (1) assist in making logging decisions and (2) assist in log analysis.

For logging decisions, we tackle the challenge by providing suggestions on writing logging statements. We first provide suggestions for logging locations. We find that our models are effective in suggesting logging locations at the block level. We then study the verbosity levels in the logging statements. We propose a deep learning based approach that can leverage the ordinal nature of log levels to make suggestions on choosing log levels. Our approach outperforms the baseline approaches and are effective at suggesting log levels. Finally, we investigate practitioners' expectation on the readability of log messages by conducting a series of semi-structured interviews with industrial practitioners. We derive three aspects that are related to the readability of log messages. We also explore the potential of automatically classifying the readability of log messages and find that both deep learning and machine learning approaches is effective at such classifications.

For log analysis, we focus on studying log abstraction, which is a crucial step for automated log analysis. We find that different categories of dynamic variables in logs record valuable information that can be important for different tasks, such information is abstracted by prior log abstraction techniques. We propose a deep learning based log abstraction approach, which can identify different categories of dynamic variables and abstract specified categories. Our approach outperforms state-of-the-art log abstraction techniques on general log abstraction and also achieves promising results on variable-aware log abstraction. We also find that variable-aware log abstraction can help improve the performance of log-based anomaly detection.

Statement of Originality

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Acknowledgements

Foremost, I would like to express my greatest gratitude to my supervisors Dr. Tse-Hsun Chen and Dr. Weiyi Shang for their patient guidance and encouragement on my research and life. Without their supervision and invaluable support, nothing of this thesis and my research life would have been possible.

Apart from my supervisors, I would like to sincerely thank my thesis examiners, Dr. Hamou-Lhadj, Dr. Guéhéneuc, Dr. Belilovsky, and Dr. Xing for their extremely valuable and constructive suggestions.

I am very lucky to have lively communications and fruitful discussions with all the members of SPEAR and SENSE. I learned so much from all of you, it is my honor and pleasure to work with you all.

Last but not least, I would like to express my special thanks to my parents. Words can hardly express my gratitude and feelings towards you. Your unconditional support sustains me thus far and keeps me going.

Related Publications

Zhenhao Li, "Towards Providing Automated Supports to Developers on Writing Logging Statements", in IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-DS 2020), Seoul, Korea, July 6-11, 2020, pp. 198–201. This work is discussed in Chapter 1.

Zhenhao Li, Tse-Hsun (Peter) Chen, and Weiyi Shang, "Where Shall We log? Studying and Suggesting Logging Locations in Code Blocks," in IEEE/ACM 35th International Conference on Automated Software Engineering (ASE 2020), Melbourne, Australia, September 21-25, 2020, pp. 361–372. This work is discussed in Chapter 3.

Zhenhao Li, "Studying and Suggesting Logging Locations in Code Blocks", in IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-SRC 2020), Seoul, Korea, July 6-11, 2020, pp. 125–127. This work is discussed in Chapter 3.

Zhenhao Li, Heng Li, Tse-Hsun (Peter) Chen, and Weiyi Shang, "DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks", in IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021), Madrid, Spain, May 25-28, 2021, pp. 1461–1472. This work is discussed in Chapter 4.

Zhenhao Li, Tse-Hsun (Peter) Chen, and Weiyi Shang, "Studying Practitioners' Expectation on the Readability of Log Messages", under review[1]. This work is discussed in Chapter 5.

Zhenhao Li, Tse-Hsun (Peter) Chen, and Weiyi Shang, "Studying and Exploring Variable-aware Log Abstraction", under review[1]. This work is discussed in Chapter 6.

[1]These works are currently under the double-blind review process in a conference, we re-phrased the titles to be different from the submissions and omitted the authors other than the student and his supervisors (the student is the first author for both of the submissions).

The following publication is not directly related to the materials presented in this thesis but it is the result of a work conducted in parallel to the research presented in the thesis

Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang, "Studying Duplicate Logging Statements and Their Relationships with Code Clones", IEEE Transactions on Software Engineering (TSE), pp. 2476-2494, 2021.

Contents

List of Figures **xii**

List of Tables **xiv**

I Introduction, Background, and Literature Review **1**

1 Introduction **2**

1.1 Introduction 2

1.2 Research Hypothesis 3

1.3 Thesis Overview and Contributions 5

1.3.1 Chapter 2: Background and Literature Review 5

1.3.2 Chapter 3: Studying and Suggesting Logging Locations in Code Blocks . . . 5

1.3.3 Chapter 4: Suggesting Log Levels Using Ordinal Based Neural Networks . . . 5

1.3.4 Chapter 5: Studying Practitioners' Expectation on the Readability of Log Messages 6

1.3.5 Chapter 6: Studying and Exploring Variable-Aware Log Abstraction 6

1.3.6 Chapter 7: Conclusion and Future Work 6

1.4 Thesis Organization 6

2 Background and Literature Review **8**

2.1 Background 8

2.2 Literature Review 10

2.2.1 Paper Selection 10

2.2.2 Empirical studies on logging practices 10

2.2.3 Improving logging practices 11

2.2.4 Research on log abstraction. 14

II Assist in Making Logging Decisions **15**

3 Studying and Suggesting Logging Locations in Code Blocks **16**

3.1 Introduction . . . 17
3.2 Background and Related Work . . . 18
3.3 Studying the Characteristics of Logging Location in Code Blocks . . . 20
3.4 Automatically Suggesting Logging Locations at the Code Block Level . . . 25
3.4.1 Extracting Block Features . . . 25
3.4.2 Deep Learning Framework and Implementation . . . 28
3.5 Evaluation . . . 30
3.5.1 Evaluation Metrics . . . 30
3.5.2 Case Study Results . . . 31
3.6 Discussion . . . 38
3.7 Threats to Validity . . . 39
3.8 Conclusion . . . 40

4 Suggesting Log Levels Using Ordinal Based Neural Networks **41**

4.1 Introduction . . . 42
4.2 Preliminary study on log levels . . . 44
4.2.1 An Overview of the Studied Systems . . . 44
4.2.2 Investigating Log-level-related Issues . . . 45
4.2.3 Manually Studying the Characteristics of Log Levels . . . 45
4.3 Automatically Suggesting Log Levels . . . 50
4.3.1 Feature Extraction . . . 50
4.3.2 Deep Learning Framework and Implementation . . . 52
4.4 Evaluation . . . 54
4.4.1 Evaluation Metrics . . . 54
4.4.2 Case Study Results . . . 55
4.5 Threats to Validity . . . 62
4.6 Related Work . . . 62
4.7 Conclusion . . . 63

5 Studying Practitioners' Expectation on the Readability of Log Messages **64**

5.1 Introduction . . . 64
5.2 Related Work . . . 66
5.3 Research Methodology . . . 67

5.3.1 Stage 1: Interviews . . . 68
5.3.2 Stage 2: Manual Investigation . . . 69
5.3.3 Stage 3: Survey . . . 70
5.3.4 Stage 4: Automatic Classification . . . 71
5.4 Results . . . 72
5.4.1 RQ1: What are Practitioners' Expectation on the Readability of Log Messages and How to Improve It? . . . 72
5.4.2 RQ2: How is the Readability of Log Messages in large-scale Open Source Software Systems? . . . 81
5.4.3 RQ3: What is the Potential of Automatically Classifying the Readability of Log Messages? . . . 84
5.5 Implications . . . 84
5.6 Threats to Validity . . . 86
5.7 Conclusion . . . 87

III Assist in Log Analysis 88

6 Studying and Exploring Variable-Aware Log Abstraction 89
6.1 Introduction . . . 90
6.2 Related Works . . . 92
6.3 Motivating Examples . . . 93
6.4 Studying the Dynamic Variables in Logs . . . 94
6.4.1 Manually Studying and Characterizing the Dynamic Variables in Logs . . . 94
6.4.2 A Survey on Log Analysis and Dynamic Variables . . . 97
6.5 An automated Approach for Variable-aware Log Abstraction . . . 100
6.5.1 Data Annotation . . . 101
6.5.2 Deep Learning Framework and Implementation . . . 101
6.6 Evaluation of VALB . . . 104
6.6.1 Experimental Setup. . . . 104
6.6.2 Research Questions . . . 104
6.7 Discussion . . . 109
6.8 Threats to Validity . . . 111
6.9 Conclusion . . . 112

IV Conclusion and Future Work **113**

7 Conclusion and Future Work **114**

7.1 Thesis Contribution 114

7.2 Future work 115

Bibliography **117**

List of Figures

1 An overall view of this thesis. 4
2 An example of the process from a logging statement to a parsed log template. . . . 9
3 An example of how we label code blocks and extract the tokens for generating the features. We illustrate the tokens extracted from Code Block B0. 27
4 The overall architecture of our approach. 28
5 Venn diagrams of TP, TN, FP and FN of the three block features. Each number represents the percentage of the corresponding intersecting set out of the union set. . 34
6 The (a) Balanced Accuracy, (b) Precision, (c) Recall, and (d) F1 of the models trained from three block features when applied on different types of blocks. 35
7 An example of the syntactic, log message, and combined features we extracted for each logging statement 51
8 Overall framework of our approach 52
9 The accuracy of our approach on each log level 58
10 Overview of our research methodology and corresponding research questions. 67
11 Survey participants' rating for the importance of the three aspects. 73
12 Survey participants' rating for each improvement practice. 81
13 Percentage of log messages with adequate or inadequate readability for different lengths. Length refers to the number of words of a log message. 83
14 An example of the log abstraction process. 90
15 An example of log parsing before analyzing the sequences of logs. The dynamic variables in the raw logs and abstracted variables in the parsed log templates are marked in red. 93
16 An example of our log annotation process. Static words are annotated with *O*, object ID is annotated with *B-OID*, and object name is annotated with *B-OBN*. 101
17 Overall diagram of our framework. Areas surrounded by dashed lines on the right illustrate the detailed structure for the character-level representation. 102

18 Average variable-aware accuracy of models fine-tuned with different number of logs in the target data set comparing with the original results in RQ2-A. 109
19 F1 score achieved by different anomaly detection techniques using sequences of log templates without variables (Original), and using sequences of log templates with corresponding category of variables. 111

List of Tables

1 An overview of the studied systems. 21
2 The results of suggesting logging locations using syntactic (Syn.), semantic (Sem.), and fused (Fus.) block features. 31
3 The results of cross-system logging locations suggestion using syntactic block features. 37
4 An overview of the studied systems and their log level distributions (%) 44
5 The distribution of the categories of logging locations and log messages for each log level 46
6 A comparison between the vectors of log levels that are ordinally-encoded and one-hot encoded 54
7 The results of suggesting logging levels using syntactic context (Syn), log message (Msg), and a combination of both (Comb), compared with Ordinal Regression (OR) and One-hot Encoding Neurual Network (OEN) 56
8 The distribution of incorrectly suggested log levels for each actual log level (the first column) 58
9 The results of comparing enlarging training data (RQ3-A) on syntactic (S-enlarge.) and combined feature (C-enlarge.) and cross-project prediction (RQ3-B) on syntactic (S-cross.) and combined feature (C-cross.) with the within project prediction in RQ1 60
10 An overview of the studied systems. LOC: Lines of code, NOL: Number of logging statements. 69
11 Percentage (%) of log messages in each system that have adequate readability for all the three aspects, or inadequate in each of the aspect. 82
12 Balanced Accuracy (%) of different approaches on classifying the readability for each aspect. 85
13 Precision, Recall, and F1 score (%) of Classifying each aspect of readability using Bi-LSTM. 85
14 An overview of the log abstraction benchmark data sets. NOL: Number of log templates, TWV (%): Percentage of log templates with variables, NOV: number of variables 96

15 The manually-derived categories of dynamic variables with their corresponding abbreviations (Abbrev.). Dynamic variable in the example is marked in **bold and underline** 96
16 The survey results of "Opinions on the Categories of Dynamic Variables in Logs". UI: usually important, CBI: can be important in some situations, UNI: usually not important. 97
17 List of questions and results for "Follow-up Questions on Dynamic Variables and Log Analysis", the answers are in a scale from 1 (very low extent) to 5 (very high extent). 99
18 Accuracy (%) of VALB on general log abstraction compared with other log parsers and the baseline (Base). **bold** numbers: higher than 90, star mark (*): highest accuracy in each row. 107
19 Variable-aware Accuracy (%) of VALB and the baseline (Base) discussed in RQ2, and Fine-tuning models with 50 logs from the target data set (F-50) discussed in RQ3. . 107
20 The results of identifying different categories of dynamic variables by our approach (VALB) and the baseline (Base). 108

Part I

Introduction, Background, and Literature Review

Chapter 1

Introduction

1.1 Introduction

Logging is a common practice in software development and is widely used in large-scale systems to record system execution behaviors. Developers use the generated logs to assist in various tasks, such as debugging [156, 43], program comprehension [123], and performance analysis [24, 153]. Developers write logging statements (e.g., *logger.info("Got successful response {} from URL {}", response, url);*) to generate logs. In the example above, the logging statement is at the *info* level, which is the level for recording execution information. The static text messages include *"Got successful response"* and *"from URL "*, and the dynamic messages record the values for variables response and url.

Unlike general code, which can be verified through a series of testing practices, currently there are no frameworks or systematic approaches for verifying the the logging statements. Thus, developers usually rely on their intuition and experience to compose, review, and update logging statements, which makes the logging decision a challenging task. Developers need to revisit the logging statements (e.g., change log message or move the logging statement to another place in the source code) to correctly record execution behaviors [19, 157, 75]. Even though logs are often the most important source of information for debugging and maintaining large-scale software systems, there exists no industrial standard on how to write logging statements or help developers make logging decisions [43]. Therefore, making proper logging decisions is a very challenging task [157, 122]. Writing logging statements improperly may result in ambiguous or inefficient logs that mask important run-time information.

Hence, *C1*: **Making proper logging decisions** is challenging for developers. For example, it is difficult to decide where to write logging statements (i.e., *where-to-log*). Although logs provide rich information for system diagnosis, logs come with a cost. Adding too much logging may

cause significant performance overhead (e.g., affect disk I/O, bandwidth, CPU, and memory consumption) [36, 153]. Inserting too many logging statements might also result in generating too many trivial logs, which may be redundant or useless; thus, masking real problems [43]. Prior studies [168, 75] only provide limited suggestions on logging locations (e.g., only at the method-level or for specific code snippets such as exception handling code). A finer-grained suggestion is needed to assist developers in making logging decisions.

Apart from the logging locations, it is also difficult to log sufficient information that might be useful for the downstream tasks (i.e., *what-to-log*) and also succinctly and accurately record execution behaviors in a good manner (i.e., *how-to-log*). Developers rely on log messages to record and understand system execution [123, 157]. However, the lack of guidelines on writing logging statements often results in insufficient or even incorrect information recorded in the logs generated [158, 18](e.g., outdated static messages or incorrect dynamic variables). Prior studies [158, 18, 51] often focus on improving *existing logging statements* by recording additional dynamic variables or fixing typos in log messages. However, these studies do not help developers mitigate the uncertainty while composing the contents of logging statements from scratch. Writing unambiguous and informative logging statements still remains a challenging task.

On top of the challenges in writing logging statements, how to efficiently analyze the generated logs is also an important yet challenging task in practice. On the one hand, software systems regularly generate a large amount of logs in a daily routine, the huge volume of logs makes it difficult to manually analyze the logs. On the other hand, the descriptive texts in logs often have various format and in free form, which further increases the difficulty of log analysis. Hence, *C2*: **How to efficiently analyze logs** remains challenging in practice.

Motivated by the importance of logs and the above-mentioned challenges, in this thesis, we present studies and propose approaches to provide automated supports to developers on making logging decisions and log analysis. We tackle the above challenges in logging practices from two aspects, respectively: *A1*: **Assist in making logging decisions**, and *A2*: **Assist in log analysis**. We then propose a series of approaches on these two aspects to address the challenges and provide supports to developers.

1.2 Research Hypothesis

Thesis Statement: Making logging decisions remains challenging yet crucial task for developers. Due to the large volume of logs, log abstraction is a crucial first step for log analysis. By mining software development data and leveraging practitioners' knowledge, we can provide systematic supports to developers on making logging decisions and log analysis.

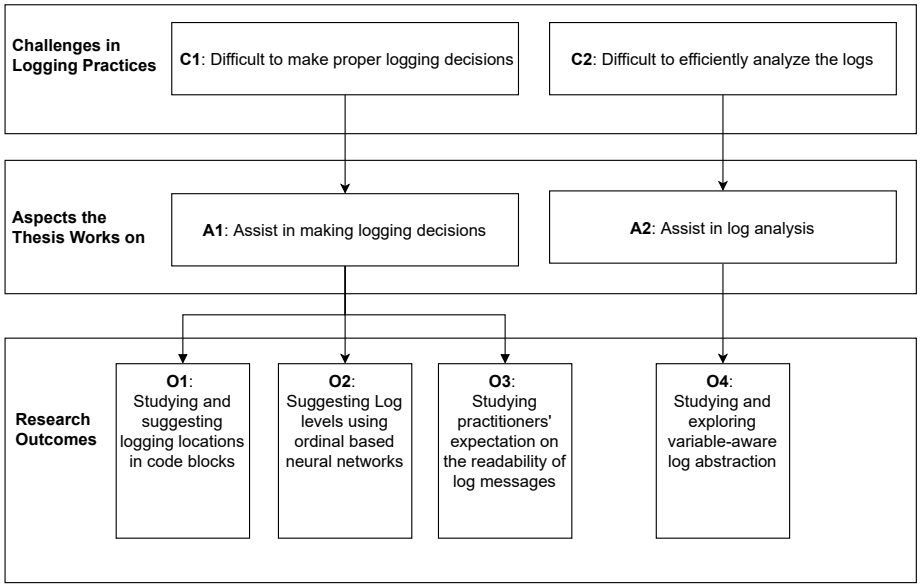


Figure 1: An overall view of this thesis.

As shown in Figure 1, we tackle the two challenges that developers are facing in two aspects by mining software development data and leveraging practitioners' knowledge. Below, we describe the two aspects and the corresponding studies for each aspect.

A1: Assist in making logging decisions. Proper logging decision has a great impact on system maintenance [43]. We propose three approaches to address the first challenge (*C1*). *O1):* We conduct a comprehensive study to uncover guidelines on logging locations (i.e., where do developers log) by analyzing logging statements and their surrounding code, and propose an automated approach to provide suggestions for where to write logging statements at the code block level. *O2):* We propose an automated approach to suggest the verbosity level for a given logging statement. *O3):* We investigate practitioners' expectation on the readability of log messages by conducting a multi-method study including interview, manual study, online survey, and automated classification.

A2: Assist in log analysis. Log abstraction is a crucial step for automated log analysis. We propose one approach to address the second challenge (*C2*) we discussed above. *O4):* We conduct a comprehensive study to investigate what roles do dynamic variables play in log analysis and propose a deep learning based log abstraction approach, which can identify different categories of dynamic variables and abstract specified categories. Because the dynamic variables in logs are usually completely abstracted by prior log abstraction techniques. These abstracted dynamic variables may also contain important information that is useful based on the given tasks.

1.3 Thesis Overview and Contributions

This section presents an overview of the thesis and the contributions, including a brief summary of each chapter.

1.3.1 Chapter 2: Background and Literature Review

In this chapter, we first present the background related to this thesis. We then present the related works of our research, including empirical studies on logging practices, improving logging practices, and research on log abstraction. We also discuss the limitations of prior studies and the points that this thesis may complement the corresponding research.

1.3.2 Chapter 3: Studying and Suggesting Logging Locations in Code Blocks

In this chapter, we tackle the challenge by first conducting a comprehensive manual study on the characteristics of logging locations in seven open-source systems. We uncover six categories of logging locations and find that developers usually insert logging statements to record execution information in certain types of code blocks. Based on the observed patterns, we propose a deep learning approach to automatically suggest logging locations at the block level. We model the source code at the code block level using syntactic and semantic information. We find that our approach is effective in suggesting logging locations at the block level. Our cross-system logging suggestion results also reveal that there might be an implicit logging guideline across systems.

1.3.3 Chapter 4: Suggesting Log Levels Using Ordinal Based Neural Networks

In this chapter, we tackle the challenge by first conducting a preliminary manual study on the characteristics of log levels. We find that the syntactic context of the logging statement and the message to be logged are related to the decision of log levels, and log levels that are further apart (e.g., *trace* and *error*) tend to have more differences in their characteristics. We propose a deep-learning based approach that can leverage the ordinal nature of log levels to make suggestions on choosing log levels, by using the syntactic context and message features of the logging statements extracted from the source code. Through an evaluation on nine large-scale open source projects, we find that our approach outperforms the state-of-the-art baseline approaches and our study highlights the potentials in suggesting log levels to help developers make informed logging decisions.

1.3.4 Chapter 5: Studying Practitioners' Expectation on the Readability of Log Messages

In this chapter, we investigate practitioners' expectation on the readability of log messages by conducting a series of interviews with industrial practitioners. We derive three aspects related to the readability of log messages along with several improvement practices for each aspect. Our findings receive encouraging feedback from subsequent online questionnaire surveys. We also find that a considerable proportion of the log messages in large-scale open-source systems have inadequate readability. Therefore, we further explore the potential of automatically classifying the readability of log messages and find that both deep learning and machine learning approaches can effectively perform such classifications. The findings of our study provide a systematic understanding of the readability of log messages and shed light for future studies on providing comprehensive and automated supports for practitioners' logging practices.

1.3.5 Chapter 6: Studying and Exploring Variable-Aware Log Abstraction

Through an empirical study and a survey with industrial practitioners, we find that different categories of dynamic variables in logs can be important for different tasks, and the distinction of dynamic variables in the process of log abstraction may help log analysis. We then propose a deep learning based approach that can identify the category of dynamic variables in the process of log abstraction. Our approach outperforms state-of-the-art log abstraction techniques on general log abstraction (i.e., abstracts all the identified dynamic variables), and also achieves promising results on variable-aware log abstraction (i.e., also identifies the category of dynamic variables). Through an exploratory study, we also find that variable-aware log abstraction can help improve the performance of log-based anomaly detection.

1.3.6 Chapter 7: Conclusion and Future Work

In this chapter, we summarize the contributions of this thesis and discuss several potential directions for future work.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses the background and literature review of this thesis. Chapter 3 presents our study on logging locations at the code block level. Chapter 4 describes our study on suggesting the logging level using ordinal based neural networks.

Chapter 5 discusses our study on the readability of log messages. Chapter 6 presents our study on investigating the dynamic variables in log abstractions. Chapter 7 concludes the thesis and discuss the potential directions for future work.

Chapter 2

Background and Literature Review

In this chapter, we first present the background related to our research proposal. We then review the literature related to this thesis, including empirical studies on logging practices, improving logging practices, and research on log abstraction.

2.1 Background

Background of Logging and Log Analysis. Developers insert logging statements into the source code to record the run-time behavior of the software systems. Typically, a logging statement consists of a verbosity level (e.g., *trace*, *debug*, *info*, *warn*, *error*), static words, and parameters to record dynamic run-time information. When the logging statement is executed, a corresponding log message (i.e., log) will be generated. Developers can then collect and use the generated logs to assist in a variety of tasks. However, due to the semi-structured format of logs, it is usually difficult to directly use the raw logs to conduct log analysis. Therefore, one of the first and most important steps of automated log analysis is log abstraction [25, 54], which parses the raw logs into a more structured format (i.e., log event).

Figure 2 shows an example of the process from a logging statement to log abstraction. First, developers insert logging statements into the code during the process of software development or maintenance. The logging statement in the example consists of a verbosity level (i.e., *info* level), some static words (i.e., *"Upper limit on the thread pool size is "*), and a parameter to record dynamic run-time information (i.e., *this.limitOnPoolSize*). The logging statement then generates a log message when it is executed during system running. The generated log is composed of a message header (e.g., timestamp) that can be configured via the logging library, the static words that always remain constant, and the dynamic variables that may vary depending on the run-time behaviors. Developers

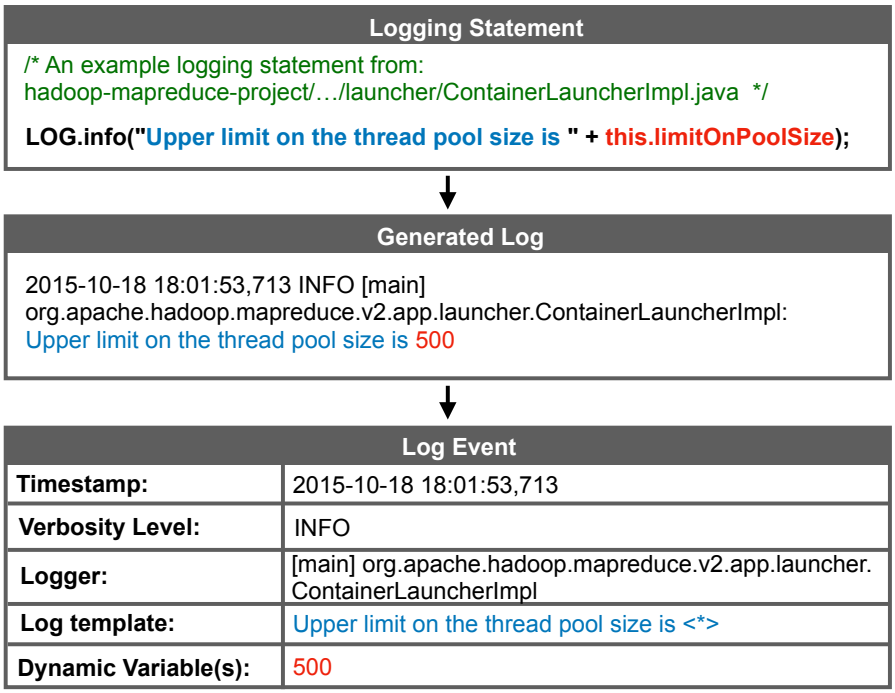


Figure 2: An example of the process from a logging statement to a parsed log template.

may then use the generated logs to assist in various tasks, such as failure diagnosis [158]. However, software systems routinely generate a very large number of logs (e.g., tens of gigabytes), and developers usually have to rely on their intuitions and experiences to write the logging statements in an ad-hoc manner [19], which results in the unstructured nature of the generated logs.

Due to the large size of routinely generated logs, it is still challenging and difficult to directly use or manually analyze the logs in practice. Hence, log abstraction techniques (also called log parsers) are used to automatically identify the static parts and the dynamic parts from the raw logs to output the logs in a structured format (i.e., log events). The static parts form the log template, the dynamic variable is abstracted as a mark of placeholder in the log template. The sequences of log template can then be analyzed in an automated manner for various tasks (e.g., anomaly detection [162]).

Prior studies propose various log abstraction techniques using different algorithms or methodologies. For example, some prior studies [135, 103, 136, 33] use frequent pattern mining to identify the words that occur frequently in logs. The frequent words are more likely to be part of a log template (i.e., static words), and the less frequent words are more likely to be values of a dynamic variable. Some other studies apply clustering algorithms to cluster logs that are textually similar [42, 130, 101, 124, 49]. Logs that are clustered in the same group are considered to have the same log template. There are also some prior studies that use different heuristics to identify the

static and dynamic parts of logs (e.g., Drain [54], AEL [64], and IPLoM [90]). Nevertheless, prior studies only focus on abstracting all the dynamic parts of logs and output the static words, without considering the potentially useful information that is recorded in the dynamic variables.

2.2 Literature Review

2.2.1 Paper Selection

We mainly review the papers related to this thesis in three aspects: 1) empirical studies on logging practices; 2) improving logging practices; 3) research on log abstraction. We select research papers related to the above aspects from renowned research venues that publish log-related studies (e.g., ICSE, FSE, ASE, TSE, TOSEM, EMSE, ICSME).

The research papers related to empirical studies on logging practices mainly study the characteristics of logging practices in different languages (e.g., C/C++, Java) of platforms (e.g., F-Droid, Linux). Such findings may help developers better understand the logging practices in the corresponding domain. There are also many research papers work on improving logging practices. In particular, such works can be summarized into three sub-aspects based on the goal of improvement: 1) Where to insert the logging statements (i.e., *Where-to-log*); 2) What information to include in the logging statements (i.e., *What-to-log*); and 3) How to develop and maintain high quality logging code (i.e., *How-to-log*). Log abstraction is an important first step towards automated log analysis. We also summarize the state-of-the-art research works related to log abstraction.

2.2.2 Empirical studies on logging practices

There are several studies on characterizing the logging practices in software systems. Yuan et al. [157] conducted a quantitative characteristics study on log messages in large-scale open source C/C++ systems.

Chen et al. [17] replicated the study by Yuan et al. [157] on Java open-source projects. Both of their studies found that log message is crucial for system understanding and maintenance, and developers often change log messages to help debugging.

Zeng et al. [159] investigated the logging practices in Android applications. Their findings show that logs are essential for debugging and maintenance purposes in different domains and platforms. They find that the a large portion of the logging statements are used for debugging purposes in those applications, but such debugging logging statements are usually still generated in the released version of the apps, which may cause potential performance overhead.

Patel et al. [110] studied the logging practices in Linux kernel, they particularly focus on three

main aspects related to the logging practices: the pervasiveness of logging in Linux, the types of changes made to logging statements, and the rationale behind these changes. For the pervasiveness of logging in Linux, they find that logging statements account for a small portion of the total code (3.73%) but distribute in more than 70% of the Linux files. For the types of changes made on logging and the rationale behind, they find that a majority of the changes in logging statements are made for improving the precision and consistency of the logs, such as fixing language issues and modify log levels.

Hassani et al. [51] identified seven root-causes of the log-related issues from log-related bug reports. They found that inappropriate log messages and missing log statements are the most common issues.

Fu et al. [43] studied where do Microsoft developers add logging statements in the code and summarized a few logging strategies. They found that developers often add logging statements to check the returned value of a method.

Prior studies identify the importance of logs and motivate future studies to assist developers in writing logging statements. The outcome of this research proposal will help developers make better logging decisions; hence, reduce the cost of maintaining logging statements and help developers record more accurate logging information.

2.2.3 Improving logging practices

The related work on improving logging practices can be summarized into three sub-aspects: 1) Where to insert the logging statements (i.e., *Where-to-log*); 2) What information to include in the logging statements *What-to-log*; and 3) How to develop and maintain high quality logging code *How-to-log*. Below, we summarize the related work of each aspect.

Where-to-log. The great value of logs results from proper logging decisions that are made by practitioners during software development. The decisions of where to insert logging statements are often made in order to balance the benefit and cost from logs. On one hand, inserting too few logging statements may increase the maintenance difficulty due to the lack of important system execution information for debugging and analysis. On the other hand, inserting too many logging statements may result in system overhead (e.g., CPU and storage consumption), and may also hide the truly crucial system event due to the excessive amount of redundant logs. In order to avoid the problem of logging too little or logging too much, practitioners need to make more informed decisions on where to insert logging statements. However, such a crucial task of making logging decisions remains challenging due to the lack of concrete logging specification and guidelines. Practitioners have to rely on their intuitions and experiences to compose, review, update and even fix logging statements in an ad-hoc manner.

Zhu et al. [168] provided a tool for suggesting log placement, specifically, in exception handling code blocks using machine learning techniques. Zhao et al. [165] proposed a tool that determines how to optimally record system execution without exceeding a pre-defined value of logging performance overhead.

Fu et al. [43] conduct a systematical study to investigate how do developers in Microsoft make the decisions on logging locations. They find that around half of the logging statements record unexpected situations, and the other half record normal execution information in both the the two systems.

Li et al. [75] proposed an approach that can suggest whether a method should be logged or not when code changes are committed. They further investigate the most influential factors in their random forest classifiers to better understand the reasons for log changes. They find that the change measures are the most influencial factors for the changes of log addition, product measures are the most influencial factors for the changes of log

What-to-log. A logging statement has three main components: static message, dynamic variables, and a verbosity level. The static messages and dynamic variables describe the specific runtime behaviours and provides valuable information for the downstream tasks using logs. Elaborate message can facilitate the process of quality assurance and failure diagnosis. However, confusing message may decrease the efficiency of log analysis or even mislead the effort of developers to a wrong direction. Due to the lack of concrete guidelines on logging practice, it is difficult to compose the messages with good quality.

He et al. [53] conducted a comprehensive study to investigate the natural language descriptions in logging statements (i.e., how do developers write the static messages). They conduct the study on 10 Java projects and 7 C# projects. They find that that static messages in logging statements are locally endemic, and the static messages are locally specific in a few source files. Motivated by the findings, this paper further explores the potential of automatically generating static messages in logging statements by proposing a retrieval-based approach. Their approach achieves reasonable BLEU and ROUGH scores and highlights the research opportunies on future researches for towards automatically generating static messages with high reliability and accuracy.

Apart from static messages, Yuan et al. [158] proposed an approach that can automatically insert additional variables into logging statements to enhance the error diagnostic information. Liu et al. [87] proposed a deep learning framework to suggest the variables that should be recorded in logging statements.

Moreover, when developers are writing a logging statement, they should typically specify a verbosity level for the logging statement to indicate the extent of severity of this log message. For common logging libraries such as SLF4j [7], the verbosity levels include *trace*, *debug*, *info*, *warn*, and

error, while *trace* is the most verbose level and *error* usually indicates severe system events. When the logging statement is executed at runtime, the verbosity level will be attached to the header of the log message to give developers a insight of whether this message needs instant attention or not. Developers can set the verbosity level to be printed at runtime. For example, if developers set the level as *info*, then only the logging statements with *info* level and more severe levels (i.e., *warn*, and *error*) will generete logs. However, it is challenging for developers to choose a proper log level for the logging statements. On one hand, choosing a relatively non-severe log level (e.g., trace) for an critical event can hide important runtime information and make it difficult to diagnose runtime failures. On the other hand, choosing a relatively severe log level (e.g., error) for a trivial event can confuse end users and increase the overhead of log management and analysis. Too many false alarms may also become a burden for developers during the process of debugging and failure diagnosis. Despite the importance of log levels, there exists no practical guidelines on what log levels to choose while logging statements.

Li et al. [74] proposed an approach that leverages ordinal regression models to suggest log levels for a logging statement and achieves a Brier score of 0.44 to 0.66 and an AUC score of 0.75 to 0.81 in the studies projects and outperforms the baseline approaches.

How-to-log. Since there are no well-established logging gudelines available, developers have to rely on their experience and intuition to compose or revise logging statements, which may result in "bad" logging statements that generate confusing logs. Moreover, unlike general code, of which the quality can be verified via software testing, currently there are no exising frameworks for verifying the quality of the logging statements. How to write "good" logging statements is still challenging for practitioners.

Chen et al. [18] conducted a study by characterizing and detecting the anti-patterns in the logging code. They found that developers commonly make some mistakes when writing logging statements (e.g., logging objects whose values may or null). They concluded five categories of logging anti-patterns from code changes, and implemented a tool to detect the anti-patterns. They further proposed an automated tool that can detect the above anti-patterns in the logging statements. This tool leverages static analysis to analyze the source code and define different rules for each anti-pattern.

Hassani et al. [51] identified seven root-causes of the log-related issues from log-related bug reports and found that inappropriate log messages and missing log statements are the most common issues.

Li et al. [81] studies duplicate logging statements which are logging statements with identical static messages. They uncover five patterns of duplicate logging code smells and implement a tool to detect such code smells. They report the detected instances to developers and they are fixed by

the developers.

Prior studies mainly focus on improving existing logging statements by adding additional variables or detecting logging anti-patterns. We study both of the logging contents and logging locations in the purpose of providing suggestions and guidelines on making logging decisions. Our research outcome will provide benefits to enhance existing logging statements and suggest better logging locations.

2.2.4 Research on log abstraction.

There are many prior studies that propose log abstraction techniques (i.e., log parsers) to assist in log analysis. Some prior studies use frequent pattern mining (e.g., SLCT [135], LFA [103], LogCluster [136], Logram [33]) to identify the static words that occur frequently in logs.

Some studies leverage clustering algorithms to cluster similar logs (e.g., LKE [42], LogSig [130], SHISO [101], LenMa [124], and LogMine [49]), since logs in the same cluster then tend to have the same log template.

There are also some prior studies that use heuristics or a combined approach to identify the static and dynamic parts of logs. For example, Drain [54] uses a fixed-depth tree to maintain log groups with the same log template. AEL [64] compares the occurrences between the static words and the dynamic words to separate logs into different groups. IPLoM [90] leverages an iterative partitioning strategy to partition logs into different groups. ULP [121] combines string matching and local frequency analysis to parse large log files, their evaluation results show that ULP can effectively and efficiently parse large log files.

Zhu et al. [169] conducted a comprehensive study to survey the state-of-the-art log parsers and evaluate their performance on log parsing. Specifically, this paper collects 13 existing log parsers using different categories of methodologies. This paper summarizes the characteristics of those log parsers from six aspects: mode (i.e., online or offline), efficiency, coverage (i.e., the capability to parse all the logs), preprocessing (i.e., requires manual pre-processing), open-source, and industrial use. They further evaluate the performance of the 13 log parsers on 16 benchmark data sets in three aspects: accuracy, robustness, and efficiency. Based on their results, they find that Drain [54] achieves the best overall accuracy among the 13 log parsers.

In addition to prior log abstraction techniques that aim to identify and abstract the dynamic parts in logs, the outcome of our research proposal can also distinguish different categories of dynamic variables. Developers can specify the categories to keep their values based on the tasks and needs.

Part II

Assist in Making Logging Decisions

Chapter 3

Studying and Suggesting Logging Locations in Code Blocks

Developers write logging statements to generate logs and record system execution behaviors to assist in debugging and software maintenance. However, deciding where to insert logging statements is a crucial yet challenging task. On one hand, logging too little may increase the maintenance difficulty due to missing important system execution information. On the other hand, logging too much may introduce excessive logs that mask the real problems and cause significant performance overhead. Prior studies provide recommendations on logging locations, but such recommendations are only for limited situations (e.g., exception logging) or at a coarse-grained level (e.g., method level). Thus, properly helping developers decide finer-grained logging locations for different situations remains an unsolved challenge. In this chapter, we tackle the challenge by first conducting a comprehensive manual study on the characteristics of logging locations in seven open-source systems. We uncover six categories of logging locations and find that developers usually insert logging statements to record execution information in various types of code blocks. Based on the observed patterns, we then propose a deep learning framework to automatically suggest logging locations at the block level. We model the source code at the code block level using the syntactic and semantic information. We find that: 1) our models achieve an average of 80.1% balanced accuracy when suggesting logging locations in blocks; 2) our cross-system logging suggestion results reveal that there might be an implicit logging guideline across systems. Our results show that we may accurately provide finer-grained suggestions on logging locations, and such suggestions may be shared across systems.

3.1 Introduction

Logs play an important role in maintaining software systems and diagnosing issues that happen during runtime. Developers rely on logs for various maintenance activities, such as debugging [158, 156, 43], testing [21, 24, 84, 25], and system comprehension [105, 104]. Developers insert logging statements in the source code with different verbosity levels (e.g., *trace*, *debug*, *info*, *warn*, *error*, and *fatal*) to record system execution information and values of dynamic variables. For example, in the logging statement: *log.warn("Invalid groupingKey:", + key)*, the static text message is *"Invalid groupingKey:"*, and the dynamic message is the value of the variable *key*. The logging statement is at the *warn* level, which is the level for recording information that may potentially cause system oddities [4].

The great value of logs results from proper logging decisions that are made by practitioners during software development [73]. The logging decisions are often made in order to balance the benefit and cost from logs [73]. On one hand, inserting too few logging statements may increase the maintenance difficulty due to missing important system execution information for debugging and analysis [168]. On the other hand, inserting too many logging statements may increase system performance overhead and produce excessive trivial logs which increase the difficulty of log analysis [73, 152, 168]. However, such a crucial task of making logging decisions remains challenging due to the lack of concrete logging specification and guidelines [43]. As a result, developers have to rely on their intuitions and experiences to compose, review, update and even fix logging statements in an ad-hoc manner [19, 157, 75, 51].

To address the challenge of making logging decisions, prior studies [168, 43, 72, 36, 153] provide automated recommendations on logging locations. However, there exist two main limitations in prior research: 1) Such recommendations are often only for a very limited number of situations. Approaches from prior research may only provide suggestions for exception handling blocks and method return values [43, 168]. 2) The logging locations are recommended at a coarse-grained level, e.g., method level [72]; while practitioners still need to decide the specific location to place a logging statement inside a method. As a result, in many cases, practitioners often still face challenges when making decisions on logging locations, despite the advance from recent research outcomes.

In this chapter, we conduct a study to uncover guidelines and provide suggestions on logging locations (i.e., where do developers log) at a finer-grained level (i.e., block level) by analyzing logging statements and their surrounding code. Through a manual study on the logging statements from seven open source systems, we find that the decisions of logging location are often influenced by both the syntactic and semantic information in the source code. Moreover, the logging statements often record execution information related to the block in which they reside. Driven by our manual study results, we extract syntactic (e.g., nodes in abstract syntax trees) and semantic (e.g., variable

names) information from the source code and propose an automated deep learning based approach to suggest logging locations at the block level. We find that our deep learning models outperform the baseline, and the syntactic block feature achieves the best results (an average balanced accuracy of 80.1%) compared to semantic and fused (a fusion of syntactic and semantic) features. Moreover, syntactic information of blocks may be leveraged to provide general logging guidelines across different software systems. In summary, this study makes the following contributions:

- We uncover six categories of logging locations, which are exception logging in *catch* blocks, branch logging in blocks associated with decision-making statements, program iteration logging, logging the start or the end of a method, and function logging in domain-specific methods. We also discuss the common types of information that is recorded in each category.
- We propose a deep learning based approach to suggesting logging locations at the block level by leveraging syntactic, semantic and fused block features extracted from the source code. We find that models trained using the syntactic features have the highest balanced accuracy (80.1%) among the three types of features. Although there are some differences in the suggestion results among the three features, syntactic features can capture around 80% of all the suggested true positives. Our finding shows that most logging decisions may be related to the syntactic structure of the code.
- The cross-system suggestion results achieve an average balanced accuracy of 67.3%. We also find that there is a moderate to substantial agreement among the cross-system models trained using the syntactic features, which shows that developers of different systems may follow certain implicit guidelines on deciding logging locations.

Chapter Organization Section 3.2 discusses the background and related work of our study. Section 3.3 describes the setup of our manual study and the categories of logging locations we find. Section 3.4 discusses how do we extract block-level features and describes our deep-learning based approach. Section 3.5 presents the evaluation metrics and the results by answering two research questions. Section 3.6 discusses the False Positives and False Negatives in our suggestion results. Section 3.7 discusses the threats to validity of our study. Section 3.8 concludes the chapter.

3.2 Background and Related Work

Developers insert logging statements into the source code to record system runtime information and use the generated logs to assist in software debugging and maintenance. For example, as shown in the simplified code snippet from Zookeeper below, the logging statement is at the *error* level,

contains the static message *"Missing count node for stat"*, and records the dynamic value of the variable *statNode*.

```
DataNode node = nodes.get(statNode);
if (node == null) {
    // should not happen
    LOG.error("Missing count node for stat {}",
        statNode);
    return;
}
```

The logging statement is closely related to the specific value of the DataNode object and records an unexpected execution behavior in an *if* block when the value of the node is *null*. Helping developers decide where to log is an on-going research problem. Fu et al. [43] studied where do Microsoft developers add logging statements in their projects written in C# and focused on studying the characeristics of logging in some specific code snippets (i.e., catch blocks and return value checks). They found that developers often add logging statements to check the returned value of a method and record exceptions. Zhu et al. [168] further extended the work by providing a tool for suggesting log placement in the two above-mentioned cases. Li et al. [72, 75] provide suggestions on whether a method or commit requires a logging statement. In short, prior studies either only target a limited number of logging locations or provide a coarse-grained suggestion. Therefore, in this chapter, we explore the potential of providing a finer-grained support on deciding general logging locations through a manual study (Section 3.3) and propose an automated deep-learning based approach to suggest logging locations at the code block level (Section 3.4).

Below, we further discuss the related works of this chapter.

Studies on Logging Practices. There are several studies on characterizing the logging practices in software systems. Yuan et al. [157], Chen et al. [17], and Zeng et al. [159] conducted quantitative characteristics studies on log messages in large-scale open source C/C++, Java systems, and mobile applications. Chen et al. [20] studied the logging utilities, and Zhi et al. [166] studied the logging configurations in Java. They found that logs are essential for debugging and maintenance.

Given the importance of logs, other studies try to help developers improve logging practices. Chen et al. [18] found that developers commonly make some mistakes when writing logging statements (e.g., logging objects whose values may be null) and concluded five categories of logging anti-patterns from code changes. Hassani et al. [51] identified seven root-causes of the log-related issues from log-related bug reports and found that inappropriate log messages and missing log statements are the most common issues. Li et al. [81] uncovered potential problems with logging statements that have the same text message and developed an automated tool to detect the problems. Yuan et al. [158]

proposed an approach that can automatically insert additional variables into logging statements to enhance the error diagnostic information. Li et al. [74] propose the use of prediction models to suggest the log level of a newly added logging statement. Liu et al. [87] proposed a deep learning framework to suggest the variables that should be recorded in logging statements.

Different from prior studies, we focus on studying logging locations in the purpose of providing suggestions and guidelines on the decisions of logging locations. The findings and approaches in this study can complement prior studies in providing more comprehensive logging supports to developers.

Applying Deep Learning in Software Engineering Tasks. Due to the advances in deep learning, recent research starts to investigate source code representation and apply deep learning models in software engineering tasks. Zhang et al. [160] proposed an AST-based neural network for source code representation. They evaluated their approach on several software engineering tasks, such as source code classification and code clone detection, and the results outperformed existing approaches. Tufano et al. [133] evaluated different representation of source code (e.g., abstract syntax tree and control flow graph) and their effect on applying deep learning models in SE tasks. Hu et al. [60] proposed a deep learning based approach to automatically generate comments for Java methods. Nghi et al. [106] applied deep learning models to identify the programming language used in an algorithm. Different from prior studies, we focus on extracting source code features to suggest which blocks need to be logged. We conduct a comprehensive manual study on the characteristics of logging locations and propose a deep learning based approach to provide automated suggestions.

3.3 Studying the Characteristics of Logging Location in Code Blocks

To better understand developers' logging decisions and provide more concrete logging suggestions, in this section, we manually inspect the logging statements and their surrounding code. We examine if there exist finer-grained (e.g., at code block levels) implicit or explicit common characteristics of the locations where developers insert logging statements.

Studied Systems. We conduct a manual study on seven large-scale open source Java systems: Cassandra, Elasticsearch, Flink, HBase, Kafka, Wicket, and ZooKeeper. Table 1 shows an overview of the systems. The studied systems cover different domains (e.g., message broker, search engine, and database), have high quality logging code, and are commonly used in prior log-related studies [72, 18, 19, 81]. The size of the studied systems ranges from 97K to 1.5M LOC, and they contain from 0.4K to 5.5K logging statements.

Table 1: An overview of the studied systems.

System	Version	LOC	NOL	#LB	#NLB	%LB
Cassandra	3.11.4	432K	1.3K	1.0K	25.0K	3.8%
ElasticSearch	7.4.0	1.50M	2.5K	1.9K	54.0K	3.4%
Flink	1.8.2	177K	2.5K	2.4K	27.5K	8.0%
HBase	2.2.1	1.26M	5.5K	4.1K	81.1K	4.8%
Kafka	2.3.0	267K	1.5K	1.0K	9.0K	10.0%
Wicket	8.6.1	216K	0.4K	0.3K	9.1K	3.2%
Zookeeper	3.5.6	97K	1.2K	0.9K	5.2K	14.8%

Note: **LOC** refers to the lines of code, **NOL** refers to the number of logging statements, **#LB** and **#NLB** refers to the number of *logged* and *non-logged* blocks respectively, **%LB** refers to the percentage of *logged* blocks over all the blocks.

Manual Study Setup. Our goal is to manually inspect the logging statements and their surrounding code to study the characteristics of logging locations. To prepare the data for our manual study, we extract the logging statements from the source code by implementing a static code parser. Our parser identifies every logging statement that invokes common logging libraries (e.g., Log4j [4] and SLF4J [7]) in the code. Then, for each logging statement, we extract its static message and dynamic variables, its verbosity level, its location (i.e., the file and method that contains the logging statement), and its surrounding code (i.e., the method that contains the logging statement). After getting all the logging statements and the extracted information, we randomly sample 375 out of 14.9K logging statements based on a 95% confidence level and 5% confidence interval [13]. For each sampled logging statement, we study its structural information and data flow of the surrounding code, in order to see the potential factors taking part in the decision of inserting the logging statements in a block. Specifically, we follow an open coding-like process similar to prior studies [81], and involve in the following three phases to conduct the manual study:

Phase I : We study 100 randomly sampled logging statements and their extracted information, and record the characteristics of their data flow, structural, and semantic information (e.g., the dependency of variables, control flow, and the business logic of the code). We further derive a draft list of categories of logging locations based on the information recorded. Then we follow the draft list to label the 100 samples collaboratively. During this phase, the categories are revised and refined.

Phase II : We independently assign the categories derived in Phase I to the rest of the 375 sampled logging statements. There is no new category derived in this phase.

Phase III : We compare the assigned categories in Phase II. Any disagreement of the categorization is discussed until reaching a consensus. No new categories are introduced during the discussion. The results in this phase have a Cohen's Kappa of 0.86, which is a substantial-level of agreement [126].

Categories of Logging Locations. In our manual study, we uncover six categories of logging

locations that are associated with four different types of blocks (i.e., try-catch, branching, looping, and method declaration). In particular, we find that three categories are associated with try-catch, branching, and looping blocks; and three categories are associated with method declaration blocks that record method execution information. Below, we discuss each category in detail with an example.

Category 1 (Try-Catch Block): Exception information logging in catch blocks (122/375, 32.5%). Exceptions are widely used to capture errors. Developers rely on logs for debugging and error diagnostics when exceptions occur [155, 43]. The code snippet below shows an example of logging statements in this category. Similar to a prior study [43], we find that a large number of sampled logging statements reside in catch blocks. Most of them are at *error* (52/122, 42.6%) or *warn (46/122, 37.7%)* level. The logging statements often record messages or execution information related to the prior try block.

```
try {
    listener.onCache(shardId, fieldName, fieldData);
} catch (Exception e) {
    logger.error("Failed to call listener on atomic
        field data loading", e);
}
```

Category 2 (Branching Block): Branch logging in blocks associated with decision-making statements (139/375, 37.1%). We find that many sampled logging statements reside in blocks associated with decision-making statements [5] (e.g., if-else and switch) to record the execution information in different branches. The variable or the invoked method in the condition of the decision-making statement (e.g., the arguments in the if statement) are processed or defined in the prior code. Among the logging statements in this category, around half of them (68/139, 48.9%) record the occurrence of an unexpected execution behavior (e.g., an error or a failure) with a *warn* level or above (e.g. *error*), as shown in the code snippet below. The remaining cases record the occurrence of a normal execution behavior with an info, debug, or trace level for system comprehension or debugging purposes.

```
final TaskId id = partitionsToTaskId.get(tp);
...
if (id != null) {
    taskIds.add(id);
} else {
    log.error("Failed to lookup taskId for partition
        {}", tp);
}
```

Category 3 (Looping Block): Program iteration logging (25/375, 6.7%). We find that some sampled logging statements reside in blocks that are associated with looping statements [5, 148] (e.g., code blocks that are associated with for, while, and do-while statements). These logging statements often record the execution state during iterating (e.g., recording the i_{th} execution inside a for block) or variables that are processed or defined in prior blocks. We also find that no logging statements under this category are at *error* or *fatal* level. All logging statements are at the level of *info* (13/25, 52.0%), *debug* (6/25, 24.0%), or *trace* (6/25, 24.0%). In short, developers are more likely to add logging statements in such blocks for debugging and recording program execution.

```
IndexStatistics[] stats = getIndexStatistics(total);
...
for (IndexStatistics s : stats) {
    LOG.info(" Object size " + s.itemSize() + " used="
        + s.usedCount());
}
```

Category 4 (Method Declaration Block): Logging the start of a method (33/375, 8.8%). We find that some sampled logging statements reside at the beginning of a method, mostly for recording the program execution state or debugging purposes. These logging statements record the start of the method execution (e.g., *"Start to build the program from JAR file."*) at the *info* (21/33, 63.6%), *debug* (8/33, 24.3%), or *trace* (4/33, 12.1%) level. Different from other categories, we do not find the location of logging statements in this category depend on prior code in the method. However, we find that these logging statements record the execution of some methods of which the process is important to know and with some specific semantics in the code (e.g., recovery(), perform(), and queue()), as shown in the code snippet below.

```
public void perform() throws Exception
{
    LOG.info(String.format("Performing action: Rolling
        batch restarting {} of region servers",
        (int)(ratio * 100)));
    List<ServerName> selectedServers = selectServers();
    Queue<ServerName> serversToBeKilled = new
        LinkedList<>(selectedServers);
    ...
    //code for performing server-killing related tasks
    ...
}
```

Category 5 (Method Declaration Block): Logging the end of a method (27/375, 7.2%). In this category, the logging statements reside at the end of a method, recording the successful method execution (e.g., *"Removed job graph from ZooKeeper"*, as shown in the code snippet below). We find that most of them (22/27, 81.5%) are at the *info* level, and the rest are in *debug* (3/27, 11.1%), *trace* (1/27, 3.7%) and *warn* (1/27, 3.7%) level, which may show that such logs are mostly for debugging and recording program execution. The logging statement may record variable values that are declared or modified in prior blocks when the method execution finishes. Similar to *Category 5*, we find that the logging statements in this category might reside in semantically similar methods of which the execution is important to be recorded (e.g., shutdown(), delete(), and remove()).

```
public void removeJobGraph(JobID jobId) throws
    Exception
{
    checkNotNull(jobId, "Job ID");
    String path = getPathForJob(jobId);
    ...
    addedJobGraphs.remove(jobId);
    //code for removing ZooKeeper job graph
    ...
    LOG.info("Removed job graph {} from ZooKeeper.",
        jobId);
}
```

Category 6 (Method Declaration Block): Function logging in domain-specific methods (29/375, 7.7%). We find that developers sometimes insert logging statements in some domain-specific methods (e.g., handling a specific request) to record the execution of this method. We also find that these methods are usually very short (i.e., within 10 lines of code). As shown in the example below, in the method handleResponse() in Elasticsearch's *JoinHelper.java*, there are only a few lines of functional code statements but has a logging statement recording the execution behavior of this method. Among the logging statements in this category, 21/29 (72.4%) are at the *info* level or below (i.e., *debug* or *trace* level) to record the methods handling normal requests, and the rest 8/29 (27.6%) logging statements are at the *warn* or *error* level to record the methods handling abnormal situations (e.g., onFailure()). We also find that these short methods might be semantically similar based on our manual observation (e.g., share many common words such as *handle* and *execute*).

```
public void handleResponse(Empty response) {
    pendingOutgoingJoins.remove(dedupKey);
    logger.debug("successfully joined {} with {}",
        destination, joinRequest);
```

```
            lastFailedJoinAttempt.set(null);
        }
```

In summary, our findings show that there may be an implicit logging guideline that developers follow in the studied systems. Both **syntactic** and **semantic** information are important considerations in such logging guidelines. In particular, we find that 76.3% (286/375, combining *Category 1*, *Category 2* and *Category 3*) of the sampled logging statements are related to recording information in blocks associated with **syntactic** information of the source code (e.g., try-catch, branching, or looping blocks). These logging statements also often record information (e.g., variable values or execution states) that is related to prior blocks. We find that 23.7% (89/375, combining *Category 4*, *Category 5* and *Category 6*) of the sampled logging statements may be inserted based on the **semantic** information (i.e., business logic) of the method inside the method declaration block. These logging statements often record the start and end of method execution, or record the execution of some domain-specific methods (e.g., request handling or task execution).

> By uncovering six categories of logging locations, we find that both **syntactic** and **semantic** information are important considerations in such logging guidelines. 76% of the sampled logging statements are related to recording exception, branching, and program iteration; while 24% are related to recording the start, end, or execution of certain methods.

3.4 Automatically Suggesting Logging Locations at the Code Block Level

As we find in Section 3.3, developers usually insert logging statements to record the behavior or state of the program in blocks (e.g., exception handling in `catch` blocks or branch logging in `if`/`else` blocks). We also find that some logging locations may be related to the semantics of a method (e.g., recording the start of a certain method execution). Hence, such syntactic and semantic information may compose implicit logging guidelines that developers follow when deciding on logging locations.

In this section, we seek to explore the potential of automatically suggesting logging locations at the block level. Such an automated approach might further assist developers in making logging decisions and improving logging practice. Below, we describe our approaches that extract block features and build a deep learning model to suggesting logging locations.

3.4.1 Extracting Block Features

Identifying Logged Blocks. Our goal is to provide suggestions on deciding blocks that require logging statements. We choose to provide a suggestion at the block level because as we find in

Section 3.3 that many logging statements are recording the behaviour or state of the program in blocks. In addition, blocks provide a finer-grand suggestion which may be more actionable compared to coarse-grand suggestions (e.g., method or file level) [144, 143]. We analyze the source code by parsing the abstract syntax tree (AST) of every method in the studied systems. Then, we identify the AST nodes in a method that represent blocks, such as the block nodes that are associated with if, for, and catch. Hence, each method may contain multiple blocks. For the block nodes that we identified, we then label them as either *logged* block or *non-logged* block by analyzing if the block contains at least one logging statement. Specifically, only the block that directly contains a logging statement is labelled as a *logged* block. For example, as shown in Figure 3, block B0 (line 3 - 6) is labelled as a *logged* block because there is a logging statement in line 4. Block B1 is labelled as a *non-logged* block, because the logging statement in line 4 is not directly contained by block B1. Table 1 shows the statistics of blocks in the studied systems. #LB refers to the number of *logged* blocks, #NLB refers to the number of *non-logged* blocks, and %LB is the percentage of *logged* blocks over all the blocks. Note that there might be multiple logging statements in a block, so the number of *logged* block is smaller than the number of logging statements in each system. In general, we find that only a small portion, i.e., 3.2% to 14.8% of the blocks contain logging statements. Hence, **accurately suggesting logging locations at the block level is a challenging task.**

As we find in Section 3.3, the locations of logging statements may be influenced by either the syntactic, semantic, or both types of information in source code. In order to obtain the features for training deep learning models and to further study the effectiveness of these features in suggesting logging locations, we then extract the syntactic, semantic, and fused block-level features when we are analyzing the source code of each block.

Extracting Block Features. In our manual study, we find that logging statements often have dependencies with the preceding code in the same method. For example, the arguments in the if statement are processed or defined in the prior code (as shown in Section 3.3). As also found in prior studies [43, 53], developers may insert logging statements based on the execution flow prior to the logging point.

Therefore, for each identified block, we analyze the source code from the start of the method, in which the code block is located, to the end of the block. This could also reflect developers' sequential workflow by suggesting whether or not a block needs a logging statement when developers finish implementing the block [40]. For example, in Figure 3, for block B0, we consider the code statements from line 1 to line 6. Similarly, for block B1, we consider the code statements from line 1 to line 8. Specifically, for each code block in a studied system, we find all the AST nodes from the start of the method to the end of this block. Then for each AST node, we record its type (e.g., MethodInvocation, VariableDeclaration, or CatchClause), the associated semantic information (e.g., the name of the

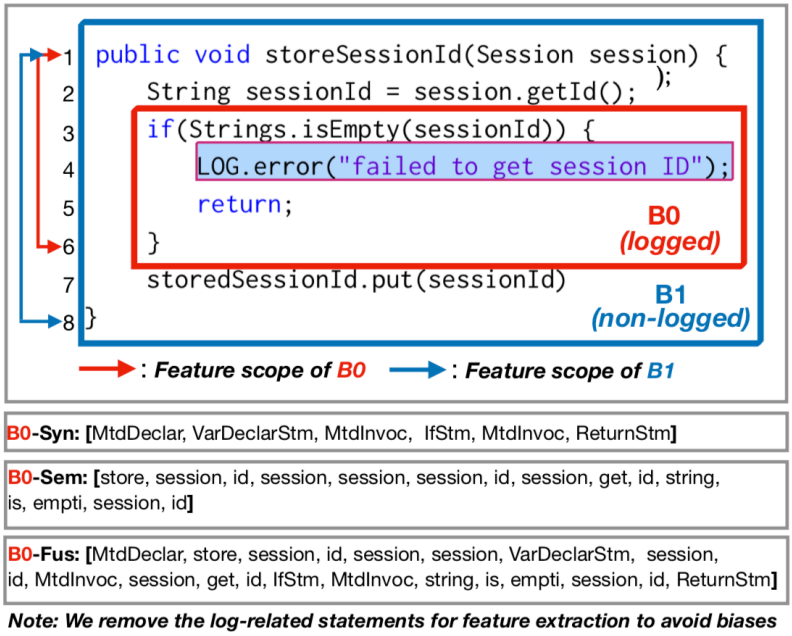


Figure 3: An example of how we label code blocks and extract the tokens for generating the features. We illustrate the tokens extracted from Code Block B0.

variable declared in the VariableDeclaration node) as well as the location (i.e., class and method, and the start line and end line). We then extract three types of code block features using the above-mentioned information from these AST nodes.

Below, we discuss the approaches that we use to extract syntactic, semantic, and fused block features, respectively.

Syntactic Block Features: We extract the syntactic features that represent the structural information from the AST nodes in code blocks. We capture the syntactic information by extracting the AST nodes that are related to the control flow of the code. We exclude AST nodes, such as SimpleName (i.e., identifier name) and SimpleType (i.e., identifier type), which do not contain structural information of the code. For each block, we count the occurrence of each AST node in the block and all preceding code in the same method. At the end of the syntactic feature extraction, for each block, we obtain a vector that represents the occurrence of each structural AST node in the block and its preceding code. We call each element in the vector as a token. Figure 3 shows an example of the syntactic features for the B0 block, where we extract the AST nodes from the feature scope.

Semantic Block Features: We extract the semantic features from the textual information inside the code blocks. Prior studies found that information such as variable names may capture the semantic information of the code [168, 27, 26, 61]. Therefore, we process variable names and invoked methods in the block as plain text. For each block, we consider all the semantic information in the

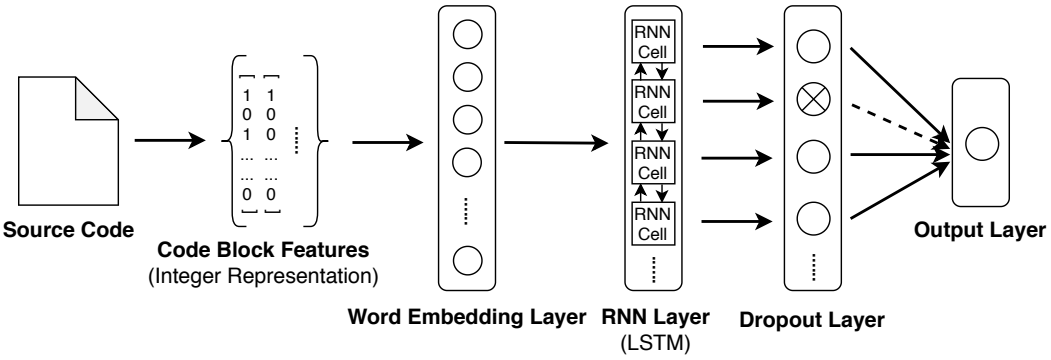


Figure 4: The overall architecture of our approach.

block and in the preceding code in the same method. Note that we exclude all reserved keywords in the programming languages, such as if, else, and for, to avoid capturing structural information. We follow common source code preprocessing techniques: splitting the words using camel case, converting all words to lower case, and applying stemming [87, 27]. Figure 3 shows an example of the semantic block features of the B0 block.

Fused Block Features: Developers may add logging statements based on both the syntactic and semantic of the code. Therefore, in addition to building separate models using the above-mentioned syntactic and semantic features, we also combine both types of information together (i.e., fused features). To obtain fused features of code blocks, we build an unified corpus containing both syntactic and semantic information of the source code by following a prior study [76]. Specifically, we merge the syntactic and semantic features in a block together, while keeping the original orders of those AST nodes in the source code. Then, for each fused code block, we obtain the vector representation similar to the process discussed in other types of features. Figure 3 shows an example of the fused code block features of the B0 block.

3.4.2 Deep Learning Framework and Implementation

We formulate the process of suggesting logging locations as a binary classification problem. Given a block, we apply deep learning models to suggest whether or not the block should contain a logging statement. In this subsection, we discuss the overall architecture and implementation of our deep learning model.

Overall Architecture. Figure 4 shows the overall architecture of our approach. We first map our input vectors (i.e., syntactic, semantic, and fused features) through an embedded layer. The embedded layer learns the relationship and similarity among the vectors in each block feature and processes each vector based on integer encoding to probabilistically distributed representations. We

then employ a recurrent neural network (RNN) layer to model the relationship between the logging decision of a block and the vectors returned from the embedding layer. Finally, the output layer of our deep learning model is a one-dimension dense layer with the sigmoid activation function to suggest whether a block should be logged or not. Below, we discuss the details of each layer.

Embedding Layer. After extracting the syntactic, semantic, and fused features (i.e., in the forms of vectors, as illustrated in Figure 3), we feed them to the embedding layer. The embedding layer captures the linear relationships among tokens in the input vector, and outputs a set of new vectors, called word embeddings [134, 100]. Compared to simple integer encoding or one-hot encoding which does not consider the relationship among the tokens, word embeddings can learn the similarities among tokens and return probabilistically distributed representations of the words (e.g., `run` and `execute` might be similar in vector space).

RNN Layer. Since source code provides instruction on system execution, there are dependencies between consecutive lines of source code. For example, as we discussed in our manual study, the condition variable in *IfStatement* may have dependency on prior source code, because the variable is defined or processed priorly. Hence, we follow prior studies [29, 46, 106, 160] and model source code as sequential data (i.e., we consider the order of the source code tokens in the data). We include a layer of Long Short Term Memory (LSTM) in the deep learning model, which is a variant of RNN that includes a memory cell and gate mechanisms in the recurrent unit to preserve long term dependencies of the code [59, 69, 47, 39].

Output Layer. After the previous layers, the block features are still high-dimensional vectors. In order to make a binary suggestion of whether a block is *logged* or *non-logged*, we use a one-dimensional dense layer with sigmoid activation function as the output layer of our approach. This layer takes all outputs from the previous layer to its unique neuron, then the neuron provides the final suggestion (i.e., *logged* or *non-logged*) of this block.

Implementation and Training We use Keras [3] to implement our deep learning model. For the embedding layer, we adopt Skip-gram from Word2vec [2] and set the dimension to 100 [87] to obtain the word embeddings of each type of the three features separately. For the RNN layer, we set the dimension of hidden states as 128 and attach a dropout layer with a 0.2 dropout rate in order to reduce the potential impact of overfitting and immoderate reliance on the trained system [128, 58, 161]. We train our model for 100 epochs on each studied system and set the batch size to 24. Because there is a noticeable imbalance between the number of *logged* blocks and *non-logged* blocks (overall only 3.2% to 14.8% blocks are *logged* blocks, as shown in Section 3.4), for each studied system and each type of code block features, we apply stratified random sampling [113] (i.e., ensure the random sample has the same distribution of classes as the original data) to split the block features into

training set (60%), validation set (20%) and testing set (20%) [160, 87]. **Note that we remove the log-related statements when we are generating the features to avoid biases in the suggestion results.** Finally, we upsample the *logged* block features in the training set after the splitting process to mitigate the impact of data imbalance [112, 11].

3.5 Evaluation

In this section, we evaluate our approach by introducing the evaluation metrics and answering two research questions.

3.5.1 Evaluation Metrics

Given the features of a code block as inputs, our deep learning model suggests if this block is *logged* or *non-logged*. To evaluate the performance of our model, we use Balanced Accuracy, Precision, Recall, and F-measure as our evaluation metrics.

Balanced Accuracy. Balanced accuracy is widely used by prior studies to evaluate model performance on imbalanced data [168, 75]. It calculates the average of True Positive Rate (i.e., how many suggested *logged* blocks are correct) and True Negative Rate (i.e., how many suggested *non-logged* blocks are correct). Balanced accuracy is computed as:

$$BalancedAccuracy = (\frac{TP}{TP+FN} + \frac{TN}{TN+FP})/2$$

where TP, TN, FP and FN refer to True Positive, True Negative, False Positive (i.e., suggested as a *logged* block but is actually a *non-logged* block) and False Negative (i.e., suggested as a *non-logged* block but is actually a *logged* block), respectively. A high balanced accuracy means both the majority class (i.e., *non-logged* block) and minority class (i.e., *logged* block) are accurately suggested.

Precision. In our study, precision represents the ability of our approach to correctly suggest *logged* blocks (i.e., how many *logged* blocks suggested by our model are correct). Specifically, precision is defined as:

$$Precision = \frac{TP}{TP+FP}$$

Note that only positive labels (i.e., *logged* block) are considered for this metric (i.e., the performance on *non-logged* data does not affect our calculation of precision). Hence, a high precision means that most of the suggested *logged* blocks are indeed logged.

Recall. Recall represents the ability of finding *logged* blocks from the data set (i.e., how many *logged* blocks can be suggested by our approach). It is computed as:

Table 2: The results of suggesting logging locations using syntactic (Syn.), semantic (Sem.), and fused (Fus.) block features.

	Balanced Accuracy				**Precision**				**Recall**				**F_1**			
Systems	Syn.	Sem.	Fus.	RG.	Syn.	Sem.	Fus.	RG.	Syn.	Sem.	Fus.	RG.	Syn.	Sem.	Fus.	RG.
Cassandra	**83.0**	65.8	65.2	49.8	**51.7**	37.1	31.8	3.0	**56.6**	33.7	33.2	3.5	**54.0**	35.3	32.5	3.2
Elasticsearch	**81.9**	67.7	69.9	50.1	**52.0**	29.7	24.3	3.6	**55.6**	38.6	44.7	3.7	**52.9**	33.6	31.5	3.6
Flink	**83.0**	74.2	75.0	50.0	**58.9**	36.0	37.6	5.6	**70.9**	54.4	55.6	8.7	**64.3**	43.3	44.9	6.8
HBase	**80.5**	69.7	72.9	49.9	**56.1**	45.2	43.2	4.8	**63.4**	41.9	49.1	5.0	**59.5**	43.5	45.9	4.9
Kafka	**74.4**	68.3	67.5	50.1	**41.5**	30.8	37.4	9.5	**58.2**	48.5	49.0	11.0	**47.3**	37.7	42.5	10.2
Wicket	**84.7**	76.6	72.2	50.0	**45.7**	28.1	26.9	3.7	**72.3**	58.5	49.2	3.2	**56.0**	37.9	34.8	3.4
Zookeeper	**72.9**	64.6	70.5	49.8	**48.3**	39.6	47.5	12.8	**55.6**	39.2	50.3	16.8	**51.7**	39.4	48.9	14.5
Average	**80.1**	69.6	70.5	50.0	**50.6**	35.2	35.6	6.1	**61.8**	45.0	47.3	7.4	**55.1**	38.7	40.2	6.7

Note: RG. represents the result of the baseline. For each system and for each evaluation metric, the block feature that yields the best performance is marked in bold. All the numbers represent percentage.

$$Recall = \frac{TP}{TP + FN}$$

Same as precision, only positive labels are considered for this metric. A higher recall means that we can identify more code blocks that need to be logged.

F1 Score. F1 score is a metric that considers both precision and recall. It is computed as:

$$F_1 = 2 * (\frac{Precision * Recall}{Precision + Recall})$$

F1 score balances the use of precision and recall and provides a more realistic measure of the performance by using both of them. A high F1 score means that we can both accurately and sufficiently suggest *logged* blocks.

3.5.2 Case Study Results

In this subsection, we present the results for our research questions (RQs). For each RQ, we describe the motivation, approach, and results and discussions.

RQ1: How effective are different block features when suggesting logging locations?

Motivation. Deciding where to log is a challenging practice [43, 168, 75]. As we find in our manual study, there exist some common characteristics of where developers insert logging statements. Logging location might be related to either the syntactic information, semantic information of the code, or both. In this RQ, we investigate the performance of our deep learning models and how each block feature performs in suggesting logging location. Our finding may help validate our manual

study results that there may be an implicit logging guideline that developers follow, and identify the important features in suggesting logging location. Specifically, we split this RQ into three sub-RQs:

RQ1.1: What is the performance of the three block features when suggesting logged blocks?

RQ1.2: Do different block features capture different information?

RQ1.3: What are the suggestion accuracies for different categories of *logged* blocks?

Approach. We train our deep learning framework on the training data by following the process discussed in Section 3.4. We conduct our experiments on the same systems that we used in our manual analysis. For each studied system, we train three models using different types of block features (i.e., syntactic, semantic, and fused). Finally, we evaluate the model performance on the testing set using the above-mentioned evaluation metrics. Note that we pre-determined the training (60%), validation (20%), and testing (20%) data set before extracting the block features. Hence, we use the same set of code blocks for each system when evaluating the syntactic, semantic and fused blocks features.

RQ 1.1: What is the performance of the three block features when suggesting logged blocks? To evaluate the effectiveness of our models, we compare the results of the models trained using three block features with a baseline. Since there is no prior study that suggests logging locations at the block level, we use Random Guess (RG) as our baseline, which is commonly used by prior studies [146, 145, 137, 81, 45, 28, 87]. Given a block in a studied system, Random Guess suggests whether this block should be a *logged* block or *non-logged* block based on the proportion of *logged* block in this system. For example, 10% of the code blocks in Kafka are *logged* block as shown in Table 1. Then, for each code block being tested, there is a 10% chance for Random Guess to suggest it as a *logged* block and a 90% chance to suggest it as an *non-logged* block. We repeat the Random Guess 30 times (as suggested by previous studies [45, 28]) for each system to reduce the biases. We report the average values of the four evaluation metrics computed based on the 30 times of iterations as the result of Random Guess.

RQ 1.2: Do different code block features capture different information? To further investigate if different block features capture different information in the source code, we examine the overlap and differences of the results generated from the models trained by using three block features. For each type of block feature, we collect the prediction results on the testing data of seven studied systems, analyze the True Positives, True Negatives, False Positives, and False Negatives, and compute the percentage of overlap among the syntactic, semantic and fused block features.

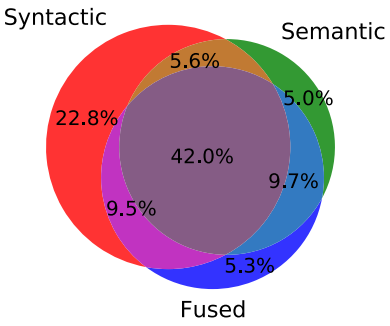
RQ 1.3: What are the suggestion accuracies for different categories of logged blocks? Since three code block features may capture different information in the source code, they might have varied performance when predicting different categories of *logged* blocks. Hence, we further evaluate the performance of the three block features on the different categories of logging statements (Section 3.3).

We report the suggestion results based on the type of blocks that the logging statement is associated with (i.e., try-catch block, branching block, looping block, and method declaration block).

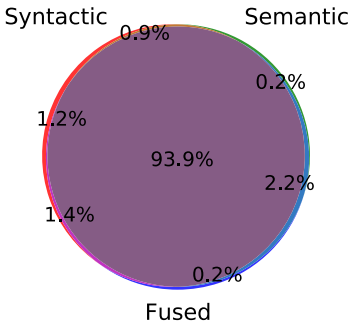
Results and Discussions.

RQ 1.1. Table 2 presents the results of the models built using the Syntactic (Syn.), Semantic (Sem.) and Fused (Fus.) code block features, and the baseline Random Guess (RG.). Overall, for all the evaluation metrics, models trained by using the block features outperform the baseline. The precision of RG ranges from 3.0% to 12.8%, recall ranges from 3.2% to 16.8%, and the balanced accuracy ranges from 49.8% to 50.1%. Note that RG makes suggestion based on the distribution of training data, the distribution of *logged* and *non-logged* blocks in the testing data is the same as the original data (as shown in Table 1). Therefore, given sufficient trails, the balanced accuracy of RG will be close to 50%. We find that models trained using the syntactic block features have the best performance compared to other block features across all studied systems. In particular, the balanced accuracy of semantic and fused features ranges from 64.6% to 76.6%, while for syntactic feature it is over 72.9% on all the studied systems (with an average of 80.1%). The average precision ranges from 24.3% to 47.5% when using semantic and fused block features, and the average recall ranges from 33.2% to 58.5%. In comparison, the average precision and recall on syntactic feature are 50.6% and 61.8%, respectively. The results show that syntactic information might play an important role in logging decisions and may be leveraged to suggest logging locations.

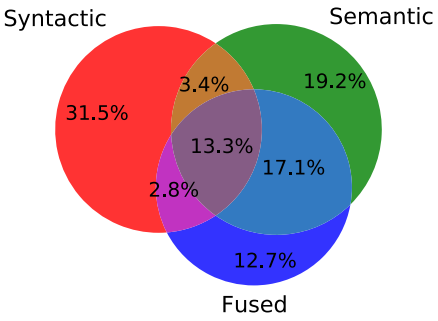
RQ 1.2. Figure 5 shows the percentage overlap on (a) True Positive, (b) True Negative, (c) False Positive, and (d) False Negative among the models trained using syntactic, semantic, and fused block features. Note that Red, Green, and Blue circle represents the suggestion results of syntactic, semantic, and fused block features, respectively. Each number represents the percentage of the corresponding intersecting data set (e.g., for TP, 42.0% represents the common set of True Positive among Syn., Sem. and Fus.) out of the entire set (e.g., for TP, the entire data set is the set that combines the TP from syntactic, semantic and fused altogether across all studied systems). There is a 42.0% overlap in TP among the three features, while syntactic covers most of the TPs (79.9% out of all the TPs) compared to semantic (62.3%) and fused (66.5%) block features. Only 20.1% of the TPs are missed by syntactic but captured by two other block features. For TNs (i.e., correctly suggest as *non-logged* block), almost all (93.9%) are overlapping among the three block features. The results show that there is a high level of agreement among the models when suggesting the *non-logged* blocks. For FPs, there is no considerable overlap among the three features (13.3%). For FNs, syntactic has the lowest number of FNs (63.2% of the FNs are covered by syntactic, compared to 80.2% covered by semantic and 76.2% covered by combined feature). The results show that different block features might capture different information from source code. As semantic and fused block features still capture TPs that are missed by syntactic block feature (20.1%), future work could



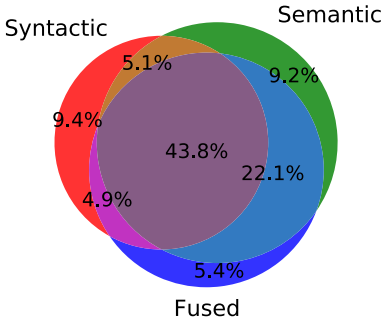
(a) True Positive



(b) True Negative

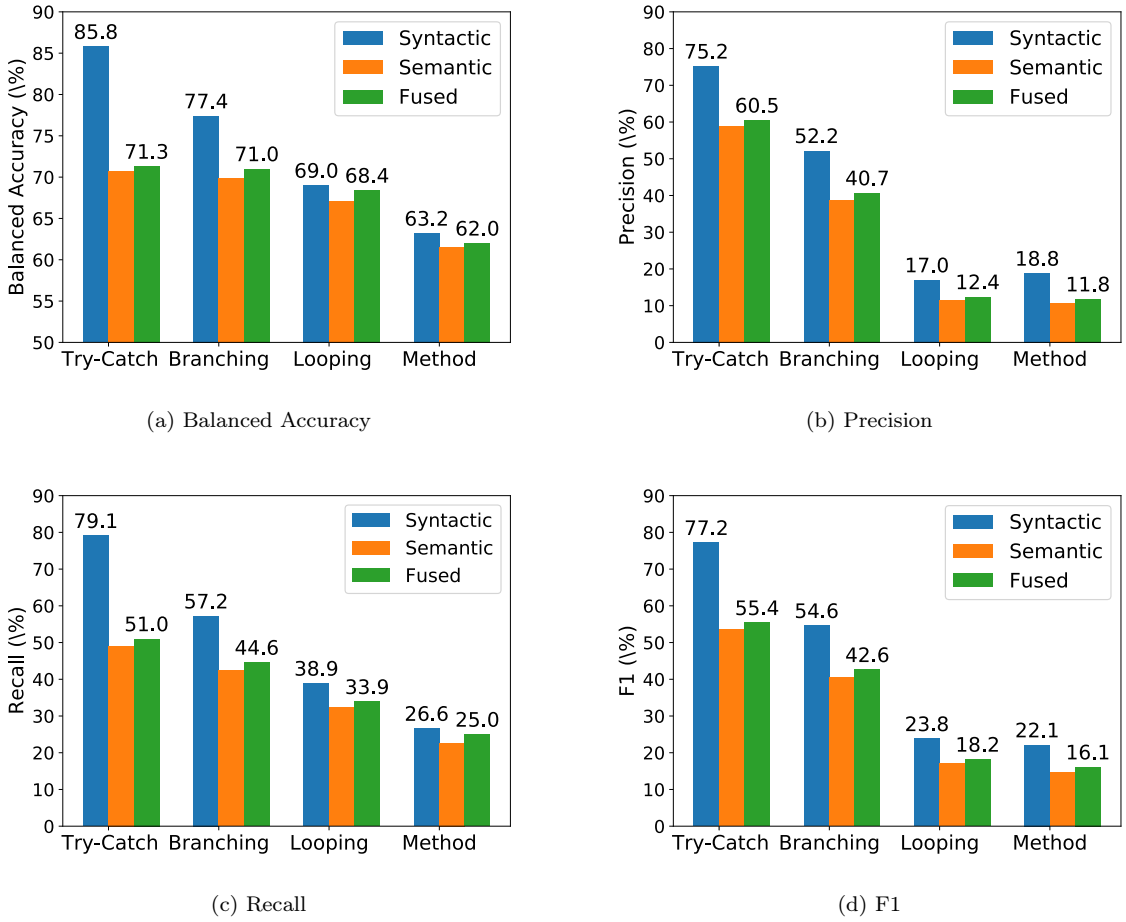


(c) False Positive



(d) False Negative

Figure 5: Venn diagrams of TP, TN, FP and FN of the three block features. Each number represents the percentage of the corresponding intersecting set out of the union set.



(a) Balanced Accuracy

(b) Precision

(c) Recall

(d) F1

Figure 6: The (a) Balanced Accuracy, (b) Precision, (c) Recall, and (d) F1 of the models trained from three block features when applied on different types of blocks.

further investigate how to better combine the two sources of information to provide a sufficient and accurate suggestion. Moreover, we manually investigate a sample of FPs and FNs. We identify their characteristics and find that many of them are not indeed FPs and FNs (details in Section 3.6).

RQ 1.3. Figure 6 shows the (a) Balanced Accuracy, (b) Precision, (c) Recall, and (d) F1 of the models when suggesting on different types of blocks associated with the categories in Section 3.3. Overall, the three block features have a similar trend for the results on different types of blocks. Syntactic features have the best results for all types of blocks on all the evaluation metrics. Among the four types of blocks, logging statements associated with try-catch blocks have the best results on all the evaluation metrics (85.8% balanced accuracy, 75.2% precision, 79.1% recall and 77.2% F1 for syntactic). As also found in prior studies [43, 168], logging statements in such blocks may be better defined. We also find that logging statements associated with branching blocks have a good overall suggestion result. In contrast, the results of suggesting logging statements associated with looping and method declaration blocks are relatively lower (balanced accuracy ranges from 63.2% to 69.0%, and F1 ranges from 22.1% to 23.8%). Although the three block features have a similar trend of results on different types of blocks, syntactic features are better than the other two for suggesting logging locations on all the studied types of blocks. Moreover, our study shows that there is a clearer pattern of inserting logging statements in try-catch and branching blocks (i.e., higher precision and recall). Practitioners may prioritize reviewing and deciding the given logging suggestions in such blocks. In addition, future research may investigate other sources of information in order to better assist in making logging decisions for looping and method declaration blocks.

> All the trained models noticeably outperform the baseline. Among the three types of block features, models trained using syntactic block features achieve the best results on all the evaluation metrics. The results show that syntactic information might be leveraged to suggest logging locations.

RQ2: Are the trained models transferable to other systems?

Motivation. When working on a new system, developers may encounter difficulties when deciding logging locations. Different from matured systems with a long period of development and maintenance history, developers working on new systems may not have sufficient knowledge on deciding where to log. Therefore, in this RQ, we investigate whether different systems share similar implicit guidelines of logging locations. Our findings may provide evidence on the existence of common logging characteristics across systems and help future research derive a universal logging guideline. In particular, we study two sub-RQs:

RQ2.1: What is the effectiveness of cross-system logging suggestion?

RQ2.2: What is the level of suggestion agreement on cross-system models?

Approach. In this RQ, we study if *logged* blocks share similar syntactic block features by doing a

Table 3: The results of cross-system logging locations suggestion using syntactic block features.

	Balanced Accuracy			**Precision**			**Recall**			**F_1**			**Fleiss' Kappa**	
Systems	Within	Cross	Ratio	Within	Cross	Ratio	Within	Cross	Ratio	Within	Cross	Ratio	logged	non-logged
Cassandra	83.0	67.8 (σ3.0)	81.7	51.7	37.5 (σ9.1)	72.6	56.6	41.9 (σ7.8)	74.1	54.0	39.1 (σ7.3)	72.5	0.47 (*Mod.*)	0.90 (*Sub.*)
Elasticsearch	81.9	65.5 (σ4.5)	80.0	52.0	36.2 (σ11.5)	69.7	55.6	42.0 (σ5.2)	75.6	52.9	37.8 (σ7.0)	71.5	0.45 (*Mod.*)	0.90 (*Sub.*)
Flink	83.0	70.2 (σ3.0)	84.6	58.9	30.5 (σ8.8)	51.8	70.9	49.2 (σ9.1)	69.4	64.3	36.7 (σ7.6)	57.1	0.46 (*Mod.*)	0.91 (*Sub.*)
HBase	80.5	67.5 (σ2.3)	83.9	56.1	37.5 (σ4.8)	66.9	63.4	41.8 (σ6.7)	66.0	59.5	40.5 (σ4.6)	68.1	0.49 (*Mod.*)	0.92 (*Sub.*)
Kafka	74.4	65.7 (σ4.1)	88.4	41.5	32.0 (σ4.2)	77.2	58.2	42.5 (σ6.8)	73.1	47.3	36.2 (σ3.9)	76.6	0.42 (*Mod.*)	0.88 (*Sub.*)
Wicket	84.7	67.8 (σ3.3)	80.1	45.7	40.3 (σ5.2)	88.2	72.3	42.1 (σ8.0)	58.3	56.0	40.8 (σ4.7)	72.9	0.43 (*Mod.*)	0.85 (*Sub.*)
Zookeeper	72.9	66.8 (σ2.7)	91.7	48.3	33.6 (σ6.2)	69.6	55.6	44.8 (σ5.6)	80.6	51.7	38.3 (σ5.8)	74.1	0.37 (*Fair*)	0.81 (*Sub.*)
Average	80.1	67.3	84.0	50.6	35.4	70.0	61.8	43.5	70.4	55.1	38.6	70.0	0.44 (*Mod.*)	0.88 (*Sub.*)

Note: Within shows the results of within-system suggestion. Cross shows the average results and the standard deviation (σ) when applying the models trained using other systems.

cross-system transferable learning. Namely, we study if a model that is trained using the syntactic features from one system can be used to suggest logging location in another system. We choose to study syntactic block features because they are extracted from the AST nodes in the source code, which are common across all Java systems, and they have the best performance compared to the other two block features as shown in RQ1. Moreover, since the syntactic block features capture the underlying code structure [61], a high cross-system suggestion accuracy may show the potential of deriving a logging guideline based on code structure in future studies.

RQ 2.1: What is the effectiveness of cross-system logging suggestion? For each studied system, we build a model using the syntactic block features and apply the model on each of the other systems. For example, we train a model using the syntactic block features in Cassandra, and apply the model on six other studied systems. Finally, we compute and report the average balanced accuracy, precision, recall, and F-measure of the cross-system logging suggestion.

RQ 2.2: What is the level of suggestion agreement on cross-system models? To study whether the models trained using different systems capture similar information (i.e., the relationship between the syntactic features and logging location), we analyze the agreement level of cross-system suggestion results. We separately examine the suggestion agreement of the cross-system models on *logged* blocks and *non-logged* blocks. Namely, for each studied system, we apply the models trained using other systems, and study the suggestion results of the cross-system models on the *true* logged blocks and the *true* non-logged blocks, respectively. In particular, we compute Fleiss's Kappa to study the agreement among the suggestion results from cross-system models [41]. Fleiss's Kappa computes the inter-rater agreement among a fixed set of raters (i.e., suggestion results from different cross-system models). A higher level of agreement may show that the syntactic block features have very similar relationships with logged or *non-logged* blocks across all studied systems.

Results and Discussions.

RQ 2.1. Table 3 shows the results of our cross-system suggestions using syntactic block features. In general, we find that the results of cross-system suggestions are lower than within-system suggestions using syntactic block features. However, the results are still comparable to within-system suggestions using semantics and fused block features. For balanced accuracy, the cross-system suggestions achieve over 80% (i.e., Ratio column in Table 3) of the corresponding within-system suggestion using syntactic block features. On average, the balanced accuracy ranges from 65.5% to 70.2%, with standard deviations range from 2.3 to 4.5. For precision, recall, and F_1, the cross-system suggestions achieve 51.8% to 88.2% ratio of the within-system suggestion results. In short, even though we find that the results of cross-system suggestion are slightly lower than those of within-system, we may still achieve a reasonable performance. Similar to RQ1, we also manually study a sample of FPs and FNs from the results of RQ2 (details in Section 3.6).

RQ 2.2. Table 3 also shows the Fleiss's Kappa [41] for each studied system. For *logged* blocks, the agreements are moderate in six studied systems. The agreement level is fair in Zookeeper, but the value is also close to the threshold of a moderate agreement (i.e., 0.41 [41]). Our results show that the models trained using the syntactic block features may share certain underlying properties. Namely, there are some commonalities in the code structure on how developers decide logging locations across the studied systems. For *non-logged* blocks, the agreements are substantial across all studied systems. In short, our findings show that developers are rather consistent on deciding which blocks *do not* need logging statements. Although there are some inconsistencies across the studied systems, we may still apply cross-system models to help suggest logging locations in other systems.

> We find that cross-system logging location suggestion achieves a reasonable performance compared to within-system suggestion (i.e., 84% of the within-system balanced accuracy). We also find that the cross-system models have moderate agreements on *logged* blocks and substantial agreements on *non-logged* blocks. Our results show that developers in different systems may follow certain implicit guidelines on deciding logging locations.

3.6 Discussion

As shown in the RQs, our models can provide promising results of suggesting logging locations. To further inspire future studies and better assist practitioners, we conduct a manual study to understand the FPs and FNs in the suggestion results. For each studied system in RQ1 and for each of the three models (i.e., Syntactic, Semantic, and Fused, simplified as Syn., Sem. and Fus.), we select the top-five FPs and FNs for our manual study, ranked by their suggested probabilities of being *logged* and *non-logged*, respectively (a total of 105 FPs and 105 FNs). For the cross-system models in RQ2, we also select the top-five FPs and FNs from each system (a total of 35 FPs and 35 FNs).

False Positives. For Syn. in RQ1, we find that 25/35 of the studied FPs are actually TP. The code block either contains some other types of print statements to record the execution information (e.g., System.out.print()), or contains only one child block and has no other code statements, and the child block contains a logging statement. For Sem., Fus., and cross-system models, we also find 15/35, 16/35, and 18/35 cases that belong to this category, respectively. For the remaining studied FPs, the suggestions are made when the code block is at the beginning of a method (9 cases for Syn., 12 for Sem., 10 for Fus., 16 for cross-system models), or in complex code with multiple nested blocks (1 case for Syn., 8 for Sem., 9 for Fus., 1 for cross-system models).

False Negatives. We find that 15/35 of the studied Syn. FNs may not truly be FN. Similar to the situation in FP that a code block only contains a *logged* child block and has no other code statements, the child block is suggested as a *non-logged* block and thus becomes an FN. For Sem., Fused. and cross-system models, we find 7/35, 6/35, and 15/35 cases that belong to this category. We also find that for 4/35, 3/35, 3/35 and 5/35 of the studied FNs from Syn., Sem., Fus. and cross-system models, they are blocks that have many very similar sibling blocks nearby (e.g., many similar if blocks having similar structures), while only the FN cases here contain logging statements. For the remaining studied FNs, similar to what we find in FPs, they locate at the beginning of a method (15 cases for Syn, 18 for Sem., 14 for Fus., 14 for cross-system models), or in complicated code structure with multiple nested code blocks (1 case for Syn., 7 for Sem., 12 for Fus., 1 for cross-system models).

Our findings show that the actual performance of our model may be even better due to the diverse nature of how developers write logging code. We also find that it may be more difficult to suggest a logging statement at the beginning of a method due to the lack of prior information in the code block.

3.7 Threats to Validity

Construct Validity. Our approach presumes that the training data has high-quality source code and follow good logging practice. However, there exist no industrial standards guiding developers to write logging statements. In this study, we choose seven large-scale, well-maintained systems with different sizes, across various domains to conduct the study. They are commonly used in prior log-related studies and are considered as following good logging practice [72, 18, 19, 81]. We evaluate our models on the test data set of each studied system. Different test data set might lead to very different results. To mitigate the fluctuation caused by different test data set, we apply stratified random sampling by following prior studies [113, 160, 87] to split the data set and ensure each randomly sampled data set has the same distribution of labels as the original data.

Internal Validity. We conduct manual studies to investigate the characteristics and uncover the

categories of logging locations. To avoid biases, we examine the data independently. For most of the cases, we reach an agreement. Any disagreement is discussed until a consensus is reached with a substantial-level agreement (Cohen's Kappa 0.86) [126]. Involving third-party logging experts to verify our results might further reduce this threat. Different parameters used in the neural networks might affect the effectiveness of the trained models. We follow prior studies [87, 160] to set the parameters for our deep learning models. The models trained using our approach might not be optimal on some of the evaluation metrics (e.g., an average F1 score of 66.7 on syntactic code block features). Future study may further improve the performance of our approach and provide a more comprehensive perspective of the suggestion results by surveying software engineering practitioners. We use word embeddings [134, 100], which is widely used by prior studies [87, 160] as the distributed representations of source code. Future study may consider other code representation approaches to examine the performance on suggesting logging locations.

External Validity. We conducted our study only on seven large-scale open source systems. However, we selected the studied systems in various domains and sizes (from 97K to 1.5M LOC as shown in Table 1) in order to improve the representativeness of our studied systems. Our studied systems are all implemented in Java. The results and models may not be transferable to systems in other programming languages. Future studies should validate the generalizability of our findings and the transferability of our models in systems that are implemented written in other programming languages.

3.8 Conclusion

In this chapter, we aim to tackle the challenges that developers might encounter when deciding logging locations by first conducting a comprehensive manual study. We uncover six categories of logging locations and find that developers usually insert logging statements to record execution information that happens in various types of code blocks. We propose a deep learning based approach to provide finer-grained (i.e., at the code block level) suggestions on logging locations. Our approach achieves promising results on suggesting logging locations in both within-project and cross-project predictions. Our results highlight the potential of providing finer-grained suggestions on logging locations by leveraging syntactic information in the source code, and such suggestions may be shared across systems. Future studies could explore a more advanced way of combining syntactic and semantic information in the source code, in order to provide better suggestions on logging locations.

Chapter 4

Suggesting Log Levels Using Ordinal Based Neural Networks

Developers write logging statements to generate logs that provide valuable runtime information for debugging and maintenance of software systems. Log level is an important component of a logging statement, which enables developers to control the information to be generated at system runtime. However, due to the complexity of software systems and their runtime behaviors, deciding a proper log level for a logging statement is a challenging task. For example, choosing a higher level (e.g., *error*) for a trivial event may confuse end users and increase system maintenance overhead, while choosing a lower level (e.g., *trace*) for a critical event may prevent the important execution information to be conveyed opportunely. In this study, we tackle the challenge by first conducting a preliminary manual study on the characteristics of log levels. We find that the syntactic context of the logging statement and the message to be logged might be related to the decision of log levels, and log levels that are further apart in order (e.g., *trace* and *error*) tend to have more differences in their characteristics. Based on this, we then propose a deep-learning based approach that can leverage the ordinal nature of log levels to make suggestions on choosing log levels, by using the syntactic context and message features of the logging statements extracted from the source code. Through an evaluation on nine large-scale open source projects, we find that: 1) our approach outperforms the state-of-the-art baseline approaches; 2) we can further improve the performance of our approach by enlarging the training data obtained from other systems; 3) our approach also achieves promising results on cross-system suggestions that are even better than the baseline approaches on within-system suggestions. Our study highlights the potentials in suggesting log levels to help developers make informed logging decisions.

4.1 Introduction

Software logs have been widely used in practice for various maintenance activities, such as testing [21, 24, 84, 22, 79], failure diagnosis [167, 158, 156, 119], and program comprehension [105, 104]. Developers insert logging statements in the source code with different verbosity levels (e.g., *trace*, *debug*, *info*, *warn*, *error*, and *fatal*) to record system execution information and values of dynamic variables. For example, in the logging statement: *LOG.info("stopping server ", + serverName)*, the static text message is *"stopping server "*, and the dynamic message is the value of the variable *serverName*. The logging statement is at the *info* level, which is the level for recording informational messages that highlight the progress of the application at a coarse-grained level [4].

Log levels enable developers to only print important log messages (e.g., error or warning information) at runtime while suppressing less important messages (e.g., debug messages). It is important for developers to choose the right log levels for their logging statements. On one hand, choosing a lower log level (e.g., *debug*) for a critical event can hide important runtime information and make it difficult to diagnose runtime failures [156]. On the other hand, choosing a higher level (e.g., *warn*) for a trivial event can confuse end users and increase the overhead of log management and analysis [73].

However, it is usually challenging for developers to choose a proper log level for the logging statements [157, 108, 73, 74]. Prior studies shows that developers may not have sufficient understanding of the runtime behaviors of their systems and the purposes of different log levels [108, 73], leading to suboptimial choices of log levels. In particular, prior work [157, 73, 74] observes that developers spend significant efforts in modifying the levels of existing logging statements, as it is challenging for them to make the right decisions in the first place.

In this study, we conduct a study to help developers make informed decisions on deciding proper log levels. Through a preliminary manual study on the logging statements from nine open source systems, we find that the decisions of log levels might be related to the locations of the logging statements and the messages to be recorded, and log levels that are further apart in order (e.g., *trace* and *error*) tend to have more differences in their characteristics of locations and messages. We then extract syntactic context features (to represent the location information) of the logging statements as well as their log messages, and propose a deep-learning based approach to automatically suggest log levels. Unlike other multi-class classification tasks which consider the classes as independent, log levels have an ordinal nature, i.e., the levels preserve an order among each other. Therefore, we ordinally encode the log levels to capture their ordinal nature.

We evaluate our approach on nine large-scale open source systems and compare the results with two baseline approaches: a state-of-the-art ordinal regression approach from a prior study [74]; and a deep-learning based approach with standard one-hot encoding. We find that, our models trained using the syntactic context feature achieve an average AUC of 80.8, outperforming our models

trained using the log message feature (i.e., with an average AUC of 71.5) in suggesting log levels. Combining both features in our approach would lead to the best performance (i.e., with an average AUC of 83.7). Trained from either the syntactic context feature (i.e., without log message feature) or the combined feature (i.e., with log message feature), our approach outperforms both baseline approaches in all the studied systems. By further studying the results of our approach, we find that the syntactic context and combined features have a similar capability of distinguishing different log levels; while the log message feature may only be useful for specific levels such as *error* and *warn*.

Finally, we evaluate the benefit and applicability of using data from other systems to enlarge the training data. We find that by carefully choosing the training dataset from other systems, the results of our approach can be further improved. In addition, our approach can achieve encouraging results on cross-system suggestions (e.g., on average 93.8% of the accuracy of within-system suggestions), which still outperform the baseline approaches on within-system suggestions.

The contributions of this study are as follows:

- We propose an automated deep-learning based approach that leverages the ordinal nature of log levels to make suggestions on choosing log levels. Our approach outperforms the existing state-of-the-art approaches in suggesting log levels.
- Our approach have encouraging cross-system suggestion results, which can benefit the systems without long development histories.
- Our manual study results can be leveraged as guidelines in future research on suggesting and improving log levels.

In short, our findings highlight the potentials of leveraging the characteristics of logging statements in suggesting log levels that can help developers make informed logging decisions. Our results also reveal the challenges and future research directions in assisting developers with logging.

Chapter Organization. Section 4.2 discusses the setup and results of manually studying the characteristics of log levels. Section 4.3 describes our deep learning approach on suggesting log levels. Section 4.4 presents the evaluation results of our approach by answering three research questions. Section 4.5 discusses the threats to the validity of our study. Section 4.6 summarizes the related work. Section 4.7 concludes the chapter.

Table 4: An overview of the studied systems and their log level distributions (%)

System	Version	LOC	NOL	Trace	Debug	Info	Warn	Error	Fatal
Cassandra	3.11.4	432K	1.3K	16.7%	10.9%	15.8%	16.8%	39.8%	0.0%
ElasticSearch	7.4.0	1.50M	2.5K	28.5%	32.4%	10.0%	19.2%	9.9%	0.0%
Flink	1.8.2	177K	2.5K	1.0%	30.8%	26.6%	23.7%	17.9%	0.0%
HBase	2.2.1	1.26M	5.5K	7.4%	17.3%	17.1%	24.4%	33.8%	0.0%
JMeter	5.3.0	143K	1.9K	0.7%	29.9%	16.9%	26.5%	26.0%	0.0%
Kafka	2.3.0	267K	1.5K	12.9%	28.5%	20.4%	15.3%	22.9%	0.0%
Karaf	4.2.9	133K	0.8K	0.9%	21.9%	23.1%	30.0%	23.6%	0.5%
Wicket	8.6.1	216K	0.4K	2.2%	39.3%	7.6%	28.5%	22.4%	0.0%
Zookeeper	3.5.6	97K	1.2K	2.2%	18.3%	19.3%	35.3%	24.9%	0.0%
Average	——	469K	2.0K	8.0%	25.5%	17.5%	24.5%	24.4%	0.1%

Note: LOC refers to the lines of code, NOL refers to the number of logging statements.

4.2 Preliminary study on log levels

4.2.1 An Overview of the Studied Systems

Studied Systems. We conduct the study on nine large-scale open source Java systems. Table 4 shows an overview of the systems. The studied systems are in various sizes (LOC from 97K to 1.5M, and NOL from 0.4K to 5.5K), have high quality logging code, are commonly used in prior log-related studies [18, 19, 81, 72], and cover various domains (e.g., database systems and search engines).

Log Level Distribution. Table 4 shows the distribution of the log levels in the studied systems. We find that many logging statements are used to show potential issues during system execution (i.e., on average 24.5% are at the *warn* level and 24.4% are at the *error* level). Note that, in modern logging frameworks such as SLF4J, *fatal* level is removed due to its redundancy with other log levels such as *error* [8]. As we found in the studied systems, only Karaf contains some logging statements with *fatal* level and the number is very small (only 0.5%). Therefore, we focus our study on the other log levels. We find that there is also a large proportion of the logging statements that are used for debugging (i.e., 25.5% for *debug*). As mentioned by the instruction of SLF4J, the *trace* level is not recommended since it has a high overlap with the *debug* level [8]. Hence, it may be the reason that some systems have noticeably fewer logging statements at the *trace* level. In general, there are fewer logging statements that show the general system execution (i.e., 17.5% are at the *info* level). Our preliminary findings show that the studied systems have a different distribution of log levels, and the levels are not evenly distributed. Therefore, suggesting log levels accurately either within the same or cross systems is a challenging task.

4.2.2 Investigating Log-level-related Issues

We collect the most recently *resolved* issue reports (from Jan. 2020 to Jul. 2020) in the bug tracking systems of our studied systems, and identify the log-related issue reports by examining if there are changes or patches on logging statements (106 issue reports in total). We then manually examine the changes and the discussions in those issue reports. We find that a large portion (45/106, 42.5%) of the issue reports have changes or discussions on log levels. Specifically, for 23/45 (51.1%) of the log-level-related issue reports, developers suggested changes of log levels on existing logging statements. For 22/45 (48.9%) of the log-level-related issue reports, developers suggested adding new logging statements and mentioned the reasons of the log levels of those newly added logging statements based on their execution point and the messages. In short, the proper choice of log levels is important and is actively considered by developers in both processes of improving existing logging statements and composing new ones. Both the locations and the messages of the logging statements might be important for deciding log levels.

4.2.3 Manually Studying the Characteristics of Log Levels

Prior work [157, 73, 74] found that developers spend significant efforts modifying the levels of existing logging statements that were inserted previously, and they tend to evaluate the impact of their logging statements and adjust their log levels over time [73]. Motivated by the prior studies and our investigation on log-level-related issue reports, we conduct a manual study to investigate the characteristics of different log levels, in order to better provide supports for developers on deciding log levels. In particular, we study the message and location of a logging statement to investigate if a log level is implicitly or explicitly related to the context information or the log message of the logging statement.

Manual Study Process. To prepare the data for our manual study, we first extract the logging statements from the source code using static analysis. We identify the method invocation statements that invoke common logging libraries (e.g., Log4j [4] and SLF4J [8]). Then, for each identified logging statement, we extract its log message (including static message and dynamic variables), verbosity level, and the method that contains the logging statement. In total, we extract 17.6K logging statements from the nine studied systems. Then, we randomly sample 376 out of 17.6K logging statements based on a 95% confidence level and a 5% confidence interval [13]. We apply stratified sampling to ensure the distribution of logging statements from different systems and their log levels in the sampled data is the same as the complete data [113]. Our manual study contains the following three phases:

Phase I : We leverage the categories of logging locations and messages that were derived in

Table 5: The distribution of the categories of logging locations and log messages for each log level

Category		Trace	Debug	Info	Warn	Error
Location	**CT**	2/39 (5.2%)	4/88 (4.6%)	12/70 (17.2%)	37/89 (41.6%)	52/90 (57.8%)
	LB	14/39 (35.9%)	33/88 (37.5%)	29/70 (41.4%)	50/89 (56.2%)	31/90 (34.5%)
	LP	3/39 (7.7%)	4/88 (4.6%)	1/70 (1.4%)	0/89 (0.0%)	0/90 (0.0%)
	MT	10/39 (25.6%)	14/88 (15.8%)	6/70 (8.6%)	1/89 (1.1%)	3/90 (3.3%)
	OP	10/39 (25.6%)	33/88 (37.5%)	22/70 (31.4%)	1/89 (1.1%)	4/90 (4.4%)
Message	**OD**	24/39 (61.5%)	42/88 (47.7%)	49/70 (70.0%)	27/89 (30.3%)	1/90 (1.1%)
	VD	15/39 (38.5%)	37/88 (42.1%)	6/70 (8.6%)	7/89 (7.9%)	1/90 (1.1%)
	ND	0/39 (0.0%)	9/88 (10.2%)	15/70 (21.4%)	55/89 (61.8%)	88/90 (97.8%)

prior studies [43, 53]. We use the categories to categorize 100 randomly sampled logging statements collaboratively. During this phase, the categories of logging locations and log messages are revised and refined. In the end, we reused and revised three categories of logging locations and three categories of log messages. We also derived two new categories of logging locations in this phase.

Phase II : We independently categorized the rest of the sampled logging statements (276 logging statements) by using the categories derived in Phase I.

Phase III : We compared the results from Phase II. Any disagreement of the categorization was discussed until reaching a consensus. No new categories were introduced during the discussion. The results in this phase have a substantial-level of agreement [126] for both of the categorizations of logging location and log message (Cohen's Kappa of 0.82 and 0.88 for logging location and log message, respectively).

Manual Study Results. Table 5 shows the distribution of the categories of logging locations and log messages for different log levels. Each row represents the number of logging statements that belong to each category, and each column represents the number of logging statements with each log level. The percentage in each cell shows the ratio between the logging statements out of the total sampled logging statements with the corresponding log level. Below, we discuss the results by each category.

Categories of Logging Locations v.s. Log Levels

Location 1: Catch Clause (CT). Catch clause is used for capturing the exceptions raised during the execution. As shown in the code snippet below, developers often log the exception information (e.g., the context information of the execution point) in catch clauses [43]. In our manual study, we find

that a large portion of the sampled *warn* (37/89, 41.6%) and *error* (52/90, 57.8%) logging statements are in this category. However, there are still a non-negligible number of logging statements that have different log levels. The percentage for the other three levels ranges from 4.6% (4/88 at the *debug* level) to 17.2% (12/70 at the *info* level).

```
/* Location Category 1: Catch Block (CT) */
   } catch (Exception ex) {
      LOG.error("Failed to stop infoServer", ex);
   }
```

Location 2: Logic Branch (LB). Logic branch is the code statement that leads to different system execution paths (e.g., *if-else and switch*) [43]. Developers may insert logging statements in the logic branches to help identify the execution path, or record the information in some critical branches. As shown in the code snippet below, developers added a *warn* logging statement to record an unexpected branch execution. We find that the distribution of the five log levels for the logging statements in LB are similar. Each log level has many logging statements in this category, the percentage ranges from 34.5% (31/90 at the *error* level) to 56.2% (50/89 at the *warn* level).

```
/* Location Category 2: Logic Branch (LB) */
   if (logFileReader == null) {
      LOG.warn("Nothing to split in WAL={}", logPath);
      return true;
   } else {
```

Location 3: Looping Block (LP). Logging statements in looping blocks (e.g., *for* and *while*) may record the execution state during iterating (e.g., recording the *i*th execution inside a *for* block) or recording variable values as shown in the code snippet below. We do not find any logging statements at the *warn* or *error* level that belong to this category. The logging statements that belong to this category generally have three log levels: 7.7% (3/39) at the *trace* level, 4.6% (4/88) at the *debug* level, and 1.4% (1/70) at the *info* level.

```
/* Location Category 3: Looping Block (LP) */
   while (active) {
      logger.trace("checking jobs [{}]",
          clock.instant().atZone(ZoneOffset.UTC));
      checkJobs();
```

Location 4: Method Start or End (MT). Logging statements might reside at the beginning or the end of a method, mostly for recording the program execution state or debugging purposes. For example, the code snippet below logs the event execution time whenever the method is executed. We find that 25.6% (10/39) of the logging statements are at the *trace* level, 15.8% (14/88) are at

the *debug* level, and 8.6% (6/70) are at the *info* level. However, logging statements with *warn* and *error* level only have a small portion: 1/89 (1.1%) and 3/90 (3.3%), respectively.

```
/* Location Category 4: Method Start or End (MT) */
public void onEventTime(long timerTimestamp) {
   logger.trace("onEventTime @ {}", timerTimestamp);
```

Location 5: Observation Point (OP). We categorize the rest logging locations that do not belong to any of the above-mentioned categories as Observation Point [43]. Logging statements in this category may have various characteristics of logging locations, such as locating before the entry point or after the exit point of a code block to record the execution status (as shown in the code snippet below). We find that a large portion of logging statements that belong to this category (from 25.6% to 37.5%) is at the *trace*, *debug*, and *info* level; while only 1.1% (1/89) and 4.4% 4/90 are at the *warn* and *error* level, respectively.

```
/* Location Category 5: Observation Point (OP) */
   final BinaryInMemorySortBuffer buffer =
       currWriteBuffer.buffer;
   LOG.debug("Retrieved empty read buffer " +
       currWriteBuffer.id + ".");
   long occupancy = buffer.getOccupancy();
   if (!buffer.write(current)) {
```

Categories of Log Messages v.s. Log Levels

Message 1: Operation Description (OD). Log messages in this category summarize the actions or intentions of its surrounding code [53]. Logging statements with this kind of log message could be placed before, inside, or after the execution point to record the status of an upcoming, ongoing, or a completed operation. As shown in the example below, an *info* logging statement logs the closing of a connection. We find that most of the *info* logging statements (49/70, 70.0%) are in this category. There are also a large portion of *trace* (24/39, 61.5%) and *debug* (42/88, 47.7%) logging statements in this category. For *warn* and *error* level, 30.3% (27/89) and 1.1% (1/90) of the logging statements belong to this category.

```
/* Message Category 1: Operation Descripion (OD) */
   connectionTracker.closeAll();
   logger.info("Stop listening for CQL clients");
```

Message 2: Variable Description (VD). Variable description records the value of a variable during execution [53]. As shown in the example below, a *trace* logging statement is placed after defining the

variable *parameterMap* to record its value. We find that many logging statements at the *trace* (15/39, 38.5%) and *debug* (37/88, 42.1%) level belong to this category. For other levels, the percentage is noticeably smaller (from 1.1% at the *error* level to 8.6% at the *info* level).

```
/* Message Category 2: Variable Description (VD) */
   Map<String, List<String>> parameterMap =
       request.getParameterMap();
   LOG.trace("parameterMap: {}", parameterMap);
   if (parameterMap != null) {
```

Message 3: Negative Execution Behavior Description (ND). During the system runtime, some unexpected execution behaviors may happen (e.g., an exception, or a failure). Logging statements are often inserted into these unexpected execution points to record the related information. Hence, developers can then be aware of the problem and fix the issue. We consider log messages as this category if they describe an unsuccessful attempt or an unexpected situation, with some specific negative words (e.g., fail, exception, unable). We find that most of the *error* (88/90, 97.8%) and a large number of *warn* (55/89, 61.8%) logging statements are in this category. For *info* and *debug* level, there are 21.4% (15/70) and 10.2% (9/88) of the logging statements in this category, respectively. We do not find logging statements at the *trace* level that belong to this category.

```
/* Message Category 3: Negative Execution Behavior
    Description (ND) */
   if (tokensIndex.isAvailable() == false) {
      logger.warn("failed to get access token [{}]",
          tokenId);
      listener.onResponse(null);
   } else {
```

Summary of the Manual Study Findings

As we found in our manual study, the information of logging locations and log messages may be related to the decision of log levels. For example, we find that developers are more likely to set the log level to *warn* or *error* if the logging statement resides in catch blocks (category CT). Moreover, if the logging statements reside at the beginning or end of a method (category MT), the log levels are more likely to be *trace* or *debug*. Similarly, logging statements with certain types of log messages, such as the category VD (variable description), are more often set to *trace* or *debug* level. Log levels that are further apart in order (e.g., *trace* and *error*) tend to have more different characteristics to distinguish. Our findings shed light on the relationship between log levels and the categories of

logging location and log messages, as well as the ordinal nature of log levels, that may be further leveraged to assist developers in determining log levels.

> We find that log levels that are further apart in order tend to have more different characteristics of logging locations and log messages. Locations and messages of logging statements, as well as the ordinal nature of log levels might be leveraged to help decide log levels.

4.3 Automatically Suggesting Log Levels

Inspired by our manual study findings, in this section, we propose an approach that automatically suggests log levels. We formulate the process of suggesting log levels as a multi-class classification problem. Given the information of an existing or a potential new logging statement (i.e., the structural information, or the log message, or both), we apply deep learning models to suggest which level to use. Below, we discuss how we extract the features and the framework of our deep learning approach for suggesting log levels.

4.3.1 Feature Extraction

For each logging statement, we extract three types of features: syntactic context features (simplified as Syn in the rest of chapter), log message features (Msg), and combined features of syntactic context and log messages (Comb).

Syntactic Context Features. We extract the syntactic context feature that represents the location information of a logging statement. Specifically, we parse the Abstract Syntax Tree (AST) of the source code and extract the AST nodes that are related to the control flow of the code to capture the structural information (e.g., IfStatement and CatchClause). We exclude the AST nodes that do not contain structural information of the code, such as SimpleName (i.e., identifier name) and SimpleType (i.e., identifier type). We also exclude AST nodes that are related to log guards (e.g., `if(isTraceEnabled)`). For each logging statement, we count the occurrence of each AST node from the start of the method, to the end of the basic block in which the logging statement resides. We analyze the AST nodes from the beginning of a method since the nodes represent the syntactic context of the logging statement (e.g., the logical flow of the method). As we found in Section 4.2, such syntactic context have a certain relationships with log levels. We choose to extract the features based on basic blocks since they represent a sequence of code statements where there is no branching in between (i.e., no other structural information that can affect the decision of the level of a logging statement in the block). Finally, we obtain a set of tokens (i.e., AST nodes) for each logging statement that represents the syntactic feature of the logging statement. Figure 7 shows an example of the syntactic context feature that we extract for the logging statement on line 4.

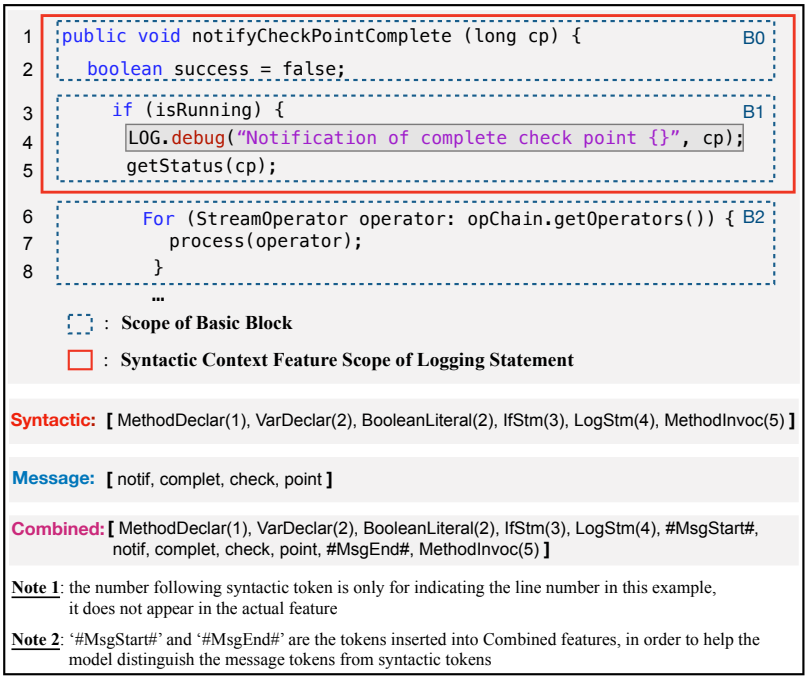


Figure 7: An example of the syntactic, log message, and combined features we extracted for each logging statement

Log Message Features. We extract the log message features from the textual information inside the logging statements. We exclude the dynamic variables, since many variable names in logging statements are not composed of natural language words [87] (e.g., variable *cp* in the logging statement in Figure 7, which is abbreviated from "check point"). For the static message in each logging statement, we first split the words using space and camel cases. We then follow common text pre-processing techniques [27]: remove the punctuation, convert the words into lower case, filter the common English words [1] and apply stemming [115] on the filtered words [61, 87, 72]. At the end of this process, we obtain a set of log message tokens, which represents its log message feature, for each logging statement.

Combined Features. As we found in Section 4.2, both the logging locations and log messages may have a certain relationship with the log levels, as they capture different aspects of a logging statement. Therefore, we combine both the syntactic information and the log message, by following an approach that is similar to prior studies [71, 131]. For each logging statement, we add the log message feature to the syntactic feature and preserve their actual order in the source code (i.e., the log message feature is added to the place that the logging statement appears in the source code). We then add a special token at the beginning and the end of the log message feature to help the model distinguish it with syntactic information. Finally, we obtain a set of tokens for each logging

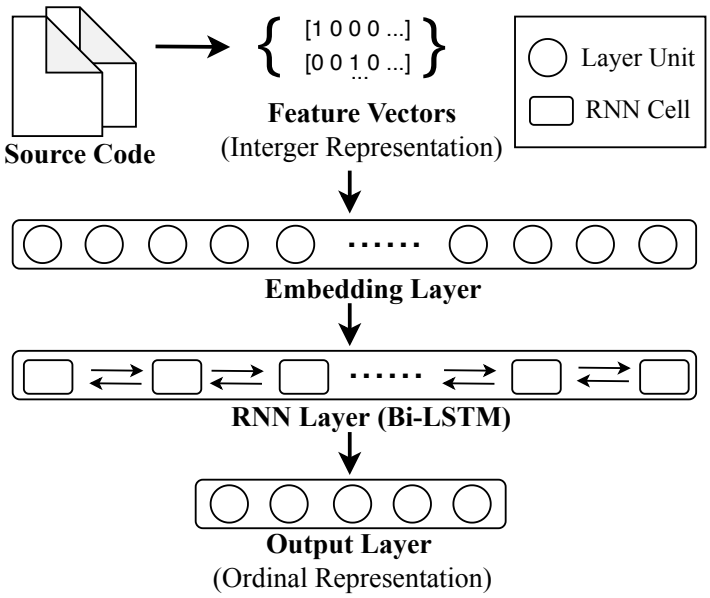


Figure 8: Overall framework of our approach

statement that represents the combined feature of the logging statement. Figure 7 shows an example of how do we combine the syntactic context feature and log message feature for the logging statement in line 4.

Ordinally Encoding Log Levels. One-hot encoding is widely used by prior studies for multi-class classification problems [87, 102, 138]. However, log levels, by nature, have an ordinal relationship. For example, if the system is configured to run and record *debug* logs, the system would also enable logging statements that are at the *info*, *warn*, and *error* levels and record logs in those levels. Therefore, we ordinally encode the log levels to preserve such ordinal relationship when suggesting log levels. Table 6 shows the comparison between the vectors of each log level that are ordinally encoded and encoded by standard one-hot encoding. Our encoding preserves the ordinal characteristics of log levels, where when a system is configured to record a certain log level (e.g., *info*), the system would also record all logs that have a higher log level (e.g., *warn* and *error*).

4.3.2 Deep Learning Framework and Implementation

Overall Architecture. Figure 8 shows the overall architecture of our approach. The deep learning framework contains an embedding layer, an RNN layer, and an output layer. Given the syntactic features, log message features, or combined features of logging statements, the embedding layer learns the relationship among the input vectors and transform each vector to a distributed

representation based on probability. We then use a recurrent neural network (RNN) layer to learn the relationship between the log level and the embedded vectors returned from the embedding layer. After that, the output layer gives an ordinal vector as the suggestion result. Finally, we map the ordinal vector returned from the output layer to a real log level as the final result. Below, we discuss the details of each component of our approach.

Embedding Layer. Our extracted features (i.e., Syn, Msg, and Comb) are represented in the form of vectors. Each dimension represents the unique tokens of the corresponding feature (e.g., the types of AST node in Syn), and each element represents the number of occurrences of the token for each logging statement. We then feed the feature vectors into the embedding layer. The embedding layer captures the linear relationships among the tokens in the feature vectors, and outputs the probabilistic representations of the vectors (i.e., word embeddings [100]). In other words, word embeddings learn the similarities among the tokens to create a more concise representation of the features [134, 76, 57].

RNN Layer. We model the source code and log message as sequential data (i.e., the order of the tokens that appear in the source code is preserved) by following prior studies [46, 29, 160, 106]. We employ a layer of Bidirectional Long Short Term Memory (Bi-LSTM) in the deep learning model, which is widely used by prior studies to process source code and natural language [61, 149]. Bi-LSTM is a variant of RNN that concatenates the outputs of two RNNs, one processing the sequence of input vector from the beginning to the end, the other one from the end to the beginning. Each RNN is composed of recurrent units including a memory cell and gate mechanisms to preserve long term dependencies of the given input. While training the model, we encode the log level of a logging statement into its ordinal representation (as discussed in Subsection 4.3.1).

Output Layer. We then use a five-dimension dense layer as the output layer. Specifically, the output layer takes the high-dimensional output vectors from the previous layer (i.e., the RNN layer) to the five neurons in this layer. Each of the five neuron represents one number in our ordinally encoded vector of log level. Then each neuron gives the result of the corresponding number (i.e., the probability of this digit to be 1) in the vector. After that, we accept the output vector and map the vector into an actual log level as the final suggestion result. For example, if the returned vector from the output layer is [1.0, 0.8, 0.6, 0.3, 0.1], we check each probability value from the start to the end of the vector. If a probability is larger than 0.5, the number is mapped to 1. If a probability that is smaller than 0.5 is encountered, the rest numbers will be mapped to 0. In the above-mentioned example, the output vector will be mapped to [1, 1, 1, 0, 0], as discussed in Subsection 4.3.1, which is an *info* level.

Implementation and Training We use Keras [3] to implement our deep learning framework. We

Table 6: A comparison between the vectors of log levels that are ordinally-encoded and one-hot encoded

	Ordinally Encoded	One-hot Encoded
Trace	[1, 0, 0, 0, 0]	[1, 0, 0, 0, 0]
Debug	[1, 1, 0, 0, 0]	[0, 1, 0, 0, 0]
Info	[1, 1, 1, 0, 0]	[0, 0, 1, 0, 0]
Warn	[1, 1, 1, 1, 0]	[0, 0, 0, 1, 0]
Error	[1, 1, 1, 1, 1]	[0, 0, 0, 0, 1]

use Skip-gram from Word2vec [2] in the embedding layer and set the dimension to 100 by following prior work [87]. We obtain the word embeddings for each type of features (i.e., syntactic context, log message, and combined features) separately. For the RNN layer, we set the number of units (i.e., the dimension of hidden states) as 128 and attach a dropout layer with a 0.2 dropout rate, in order to reduce the potential impact of overfitting on the trained system [128, 58, 161]. For each training process, we set the number of epochs as 100 and the batch size as 24 [87]. Since the model learns and predicts on each digit of the ordinally encoded vector, we use sigmoid as the activation function and use binary cross entropy as the loss function. Note that the distribution of log levels is noticeably different (e.g., on average, only 8.0% of the logging statements are in *trace* while 24.4% of the logging statement are in *error* level), as discussed in Section 4.2. Hence, we apply stratified random sampling [113] while splitting the training, validation, and testing data to ensure the sampled data set has the same distribution of log levels as the original data.

4.4 Evaluation

4.4.1 Evaluation Metrics

We use Accuracy and Area Under the Curve (AUC), which are widely used by prior multi-class classification studies, to evaluate our approach [74, 87]. According to the ordinal nature of log levels, we also propose a new metric, Average Ordinal Distance Score (AOD), to measure the average distance between the actual level and the suggested level.

Accuracy. Similar to the usage in prior classification studies [160, 87], Accuracy in our study is the percentage of correctly suggested log levels out of all the suggestion results. A higher accuracy means a model can correctly suggest the log levels for more logging statements. As a reference, the accuracy of a 5-category random guess is around 20%.

Area Under the Curve (AUC). AUC is the area under the ROC (receiver operating characteristic) curve that plots the true positive rate against the false positive rate, which evaluates the ability

of a model in discriminating different classes. AUC ranges between 0 and 1: a high value for the AUC indicates a high discriminative ability of a model; an AUC lower than 0.5 indicates a performance that is not better than random guessing. Following prior work [74], we use a multiple-class version of the AUC defined by Hand et al. [50]. The AUC gives us the insight about how well the model can discriminate different log levels, e.g., how likely a model is able to predict an actual *info* level as *info* (i.e., true positive), rather than predict an actual *debug* level as *info* (i.e., false positive).

Average Ordinal Distance Score (AOD). The prior two metrics consider different log levels as independent classes (i.e., the ordinal nature of log levels, as discussed in Section 4.3, is not considered). Hence, we propose Average Ordinal Distance Score (AOD) which measures the average distance between the *actual* log level and the *suggested* log level for each logging statement. It is computed as:

$$AOD = \frac{\sum_{i=1}^{N} (1 - Dis(a_i, s_i)/MaxDis(a_i))}{N},$$

where N is the total number of logging statements in the results. For each logging statement and its suggested log level, *Dis(a, s)* is the distance between the *actual* log level a_i and the *suggested* log level s_i (e.g., the distance between *error* and *info* is 2). *MaxDis(a)* is the maximum possible distance of the *actual* log level a_i. For example, the maximum possible distance for *trace* is 4 (i.e., from *trace* to *error*), for *info* is 2 (i.e., from *info* to *trace* or *error*). A higher AOD indicates *suggested* log levels are closer to their *actual* log levels.

4.4.2 Case Study Results

RQ1: How effective is our approach in suggesting log levels?

Motivation. As we found in the manual study, the decision of log level may be related to the syntactic information in the code and the log message. In this RQ, we want to evaluate the performance of our deep learning models trained using each of the three features (i.e., syntactic context, log message, and combined, as described in Section 4.3.1).

Approach. We first apply stratified random sampling [113] to split the input data into training set (60%), validation set (20%), testing set (20%) [160, 87], and ensure each of the sampled datasets has the same distribution of log levels as the original data. We compare our approach with two baselines described below. We then train our deep learning framework and the two baselines on the training data using each of the three features. Below, we describe the two baselines in details.

Baseline 1: Ordinal Regression (OR) model. We use machine learning based ordinal regression (OR) models [96] to suggest log levels by following a prior study [74]. OR considers the orders of the log levels (e.g., *error* is more severe than *info*) when training the model and predicting the log level given a new logging statement. In this work, we migrate the OR approach to our problem context:

Table 7: The results of suggesting logging levels using syntactic context (Syn), log message (Msg), and a combination of both (Comb), compared with Ordinal Regression (OR) and One-hot Encoding Neurual Network (OEN)

	Accuracy					**AUC**					**AOD**				
Systems	Syn	Msg	Comb	OR	OEN	Syn	Msg	Comb	OR	OEN	Syn	Msg	Comb	OR	OEN
Cassandra	**53.7**	**52.6**	***60.6***	43.2	39.9	**78.8**	**77.0**	***84.2***	75.6	70.9	**78.9**	**77.9**	***80.5***	70.3	65.6
Elasticsearch	**51.9**	40.4	***57.7***	49.8	37.8	**77.9**	66.4	***81.3***	75.8	72.6	**77.6**	69.1	***80.2***	76.8	41.7
Flink	**52.5**	35.5	***65.2***	50.1	42.0	**78.2**	69.9	***85.1***	74.1	73.5	**78.7**	72.4	***83.8***	78.5	43.9
HBase	**55.9**	50.7	***60.3***	51.0	49.5	**83.1**	74.3	***84.2***	79.4	78.3	**81.4**	72.8	***81.7***	78.9	62.7
JMeter	**55.1**	52.1	***62.3***	53.9	47.2	**83.5**	79.3	***83.9***	80.7	76.5	**82.3**	77.2	***80.9***	78.9	56.2
Kafka	**50.7**	38.5	***51.8***	42.3	41.8	**78.7**	71.5	***79.5***	74.1	69.2	**76.9**	68.6	***77.5***	72.2	59.4
Karaf	**56.5**	30.3	***67.2***	49.0	30.3	**84.3**	67.2	***85.6***	83.2	68.1	**80.6**	67.2	***81.6***	76.8	30.9
Wicket	**57.3**	28.1	***63.8***	46.1	39.0	**83.1**	61.6	***85.0***	80.7	76.4	**78.9**	62.1	***79.3***	67.8	54.5
Zookeeper	**52.8**	50.1	***60.9***	41.3	35.1	**79.6**	76.1	***84.8***	79.1	69.2	**78.8**	73.0	***82.0***	77.0	36.8
Average	**54.0**	42.0	***61.1***	47.4	40.3	**80.8**	71.5	***83.7***	78.8	72.7	**79.3**	71.1	***80.8***	75.2	50.2

Note: The number that is higher than both of the baselines is marked in **bold**, the best result is marked in ***italic-bold***.

suggesting the log level of each logging statement in the static code. We consider all the metrics used in the prior work [74] except those related to code changes, as the code change related metrics are irrelevant in our context: we suggest the log level of each logging statement in the static code. Besides, prior work finds that the influence of the metrics related to the code changes is negligible [74].

Baseline 2: One-hot Encoding RNN (OEN). As discussed in Section 4.3, the standard one-hot encoding treats all the classes as independent classes without considering the ordinal relation among them. In order to understand the effectiveness of our encoding on log levels, we would like to compare the performance of the models using our ordinally encoded log level vectors with the models using standard one-hot encoded log level vectors. Different from our approach that uses sigmoid as the activation function and binary cross entropy as the loss function to predict the value of each number in the ordinally encoded vector (as discussed in Section 4.3), to adopt standard one-hot encoding, we change the activation function to softmax and the loss function to categorical cross entropy [102, 138]. Hence, the goal of the baseline model is to predict the one-hot encoded vector (as shown in Table 6) which can be mapped back to a log level. Similar to our approach, we train the baseline on the Syn, Sem, and Comb features, respectively.

Results and Discussions.

Our approach can effectively suggest log levels for the studied systems. Our best models (i.e., using the combined feature) achieve an average AUC of 83.7. Table 7 presents the

results of our models trained using the syntactic feature (Syn), the log message feature (Msg), and the combined feature (Comb). Table 7 shows that the models trained using the Syn feature perform better than the models using the Msg feature in terms of all the three evaluation metrics. Specifically, the average accuracy, AUC, and AOD of the models trained using the Syn feature are 54.0, 80.8, and 79.3, respectively; while the average accuracy, AUC, and AOD of the models trained using the Msg feature are only 42.0, 71.5, and 71.1. More importantly, for all the three evaluation metrics, the models trained using the Comb feature have better results than the models trained only using Syn or Msg. Specifically, on average, the accuracy, AUC, and AOD of the models trained using Comb are 61.1, 83.7, and 80.8, respectively. Our results show that the Syn and Msg features both provide valuable information that can complement each other in our models.

Our approach outperforms the two baseline approaches (OR and OEN). For the baselines, due to the limitation of space, we only discuss the results of the models trained using the comb feature, which lead to the best results among the three features. For the the results of two baselines, the average accuracy are 47.4 for OR and 40.3 for OEN, the average AUC are 78.8 for OR and 72.7 for OEN, and the average AOD are 75.2 and 50.2, respectively. For every system, our models using Syn or Comb features always outperform the two baselines in the three evaluation metrics (as shown in Table 7). The results demonstrate the higher capability of our neural networks with ordinally encoded log levels than the ordinal regression and the standard one-hot encoding in suggesting log levels.

> Our approach outperforms the two baseline approaches in suggesting log levels. In particular, our approach achieves the best performance when both the syntactic and log message features are considered.

RQ2: What is the performance of our approach on different log levels?

Motivation. In RQ1, we find that our approach can effectively suggest the log level of a logging statement, and that the models trained using the syntactic, message, and combined information show different performance. However, for the logging statements with different log levels, choosing an inappropriate log level may have different costs. For example, choosing the *error* level for an *info* message may be worse (i.e., cause user confusion [73]) than choosing the *info* level for a *debug* message. Besides, different stakeholders may be more interested in certain log levels. For example, operators may be most interested in the *warn* and *error* levels which need their immediate actions; while developers doing debugging activities may be most interested in the *debug* level. Therefore, in this RQ, we further investigate the performance of our approach in providing suggestion for each log level.

Approach. We first analyze the overall performance of our models for each log level. We group the logging statements in our test datasets by their *actual* log level. Then, for each group of *actual* log

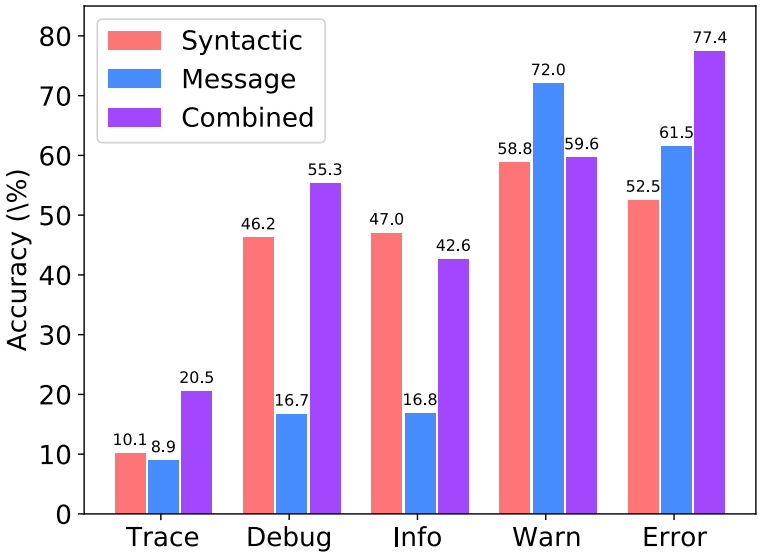


Figure 9: The accuracy of our approach on each log level

Table 8: The distribution of incorrectly suggested log levels for each actual log level (the first column)

	Syntactic Context					**Log Message**					**Combined**				
	Trace	Debug	Info	Warn	Error	Trace	Debug	Info	Warn	Error	Trace	Debug	Info	Warn	Error
Trace	—	**53.8**	25.4	16.7	4.1	—	**47.6**	30.1	20.7	1.6	—	**52.4**	21.1	18.7	7.8
Debug	5.4	—	**53.7**	33.6	7.3	7.3	—	**48.1**	41.7	2.9	7.5	—	**43.6**	32.6	16.3
Info	3.6	38.8	—	**48.9**	8.7	6.6	37.9	—	**52.2**	3.3	8.6	**43.0**	—	38.1	10.3
Warn	1.6	27.8	28.7	—	**41.9**	3.2	36.4	22.4	—	**38.0**	4.1	20.5	27.0	—	**48.4**
Error	0.9	7.7	20.7	**70.7**	—	0.4	8.2	12.9	**78.5**	—	3.9	12.3	10.4	**73.4**	—

Note: For each feature and each actual log level, the highest percentage of incorrectly suggested log level is marked in **bold**.

level, we measure the accuracy of our approach for suggesting the log levels. We then investigate how our models *mis-classify* each log level by computing the distribution of the incorrectly suggested log levels. In this RQ, we train the models and analyze the results of the three features (i.e., Syn, Msg, or Comb), respectively.

Results and Discussions.

The syntactic and combined features show more consistent performance than the log message feature among suggesting different log levels. For each log level and each feature, we present the results by showing the average accuracy of the models trained using different systems. Figure 9 shows the accuracy of the trained models using syntactic context feature (red bar), log message feature (blue bar), and combined features (purple bar) for each log level. Overall, the syntactic context and combined features have a similar trend on the results for different levels,

while log message features have a notable difference. The log message feature has a relatively high accuracy on suggesting the *warn* and *error* levels, but has a very low accuracy on other levels (ranges from 8.9% to 16.8%). The potential reason might be that, *warn* and *error* level might contain some specific words that can be used to distinguish them from other levels. As we found in Section 4.2, 61.8% and 97.8% of the log messages at *warn* and *error* level describe negative execution behaviors. However, syntactic and combined feature also achieve relatively good results on these two levels (range from 58.8% to 59.6% for *warn* level, and from 52.5% to 77.4% for *error* level). Both the syntactic and combined features also have reasonable results on suggesting *debug* and *info* levels (range from 42.6 to 55.3 accuracy).

Most of the incorrectly suggested log levels provided by our approach are close to their *actual* log levels. Table 8 presents the distribution of incorrectly *suggested* log levels for each *actual* log level (the first column, marked in bold). All the numbers are the percentage of an incorrectly *suggested* log level over all the incorrect suggestions for each *actual* log level. Overall, there is only a small portion of logging statements that are incorrectly suggested as *trace* level (range from 0.4% to 8.6% across all the three features). In comparison, most of the incorrect suggestions on *error* logging statements are suggested as *warn* level (over 70% for all the three features). Reversely, many *warn* logging statements are incorrectly suggested as *error* level (which is also the most common incorrectly *suggested* log level). We find that for each feature and each *actual* log level, the most common incorrect suggestions are one of their neighbouring log levels (i.e., the closest log levels). In particular, some log levels (e.g., *warn* and *error* levels) might be hard to distinguish. Future studies could conduct in-depth investigations on more characteristics of different log levels and help provide a more accurate suggestion correspondingly.

> The syntactic context and combined features show more consistent capability in making suggestion among different log levels, while the log message feature may only provide helpful suggestion on specific levels (e.g., *warn* and *error*). Many of the incorrectly suggested log levels are close to their *actual* log levels. Future work could investigate opportunities that leverage the characteristics of different log levels to distinguish log levels that are close in order.

RQ3: How is the generalizability of our approach?

Motivation. The success of deep neural networks often requires a large dataset in order to provide sufficient information for training [48, 132]. However, as presented in Section 4.2, the amount of logging statements in the studied systems ranges from 0.4K to 5.5K, i.e., small datasets compared to other areas, such as computer vision, where these deep neural networks are extensively leveraged [34]. Moreover, different from mature systems with a long period of development and maintenance history, new software systems may not have enough existing logging statements to train a deep neural network. Enlarging the training data from other source is often used to address the challenge of

Table 9: The results of comparing enlarging training data (RQ3-A) on syntactic (S-enlarge.) and combined feature (C-enlarge.) and cross-project prediction (RQ3-B) on syntactic (S-cross.) and combined feature (C-cross.) with the within project prediction in RQ1

	Accuracy				**AUC**				**AOD**			
Systems	S-enlarge	C-enlarge	S-cross	C-cross	S-enlarge	C-enlarge	S-cross	C-cross	S-enlarge	C-enlarge	S-cross	C-cross
Cassandra	58.5 (+4.8)	63.5 (+2.9)	45.9 (85.5%)	58.9 (97.2%)	79.7 (+0.9)	84.9 (+0.7)	74.6 (94.7%)	82.7 (98.2%)	82.1 (+3.3)	85.2 (+4.7)	76.3 (96.7%)	80.1 (99.5%)
Elasticsearch	41.7 (-10.2)	49.1 (-8.6)	47.3 (91.1%)	55.7 (96.5%)	75.2 (-2.7)	76.9 (-4.4)	75.1 (96.4%)	80.2 (98.6%)	73.3 (-4.3)	78.1 (-2.2)	77.5 (99.8%)	78.4 (97.8%)
Flink	54.4 (+1.9)	66.1 (+0.9)	45.2 (86.1%)	63.7 (97.7%)	81.5 (+3.3)	85.5 (+0.4)	74.4 (95.1%)	83.5 (98.1%)	78.8 (+0.1)	83.9 (+0.1)	76.8 (97.6%)	82.3 (98.2%)
HBase	57.3 (+1.4)	64.0 (+3.7)	40.3 (72.1%)	55.2 (91.5%)	84.2 (+1.1)	85.3 (+0.9)	72.5 (87.2%)	80.2 (95.2%)	82.0 (+0.6)	84.1 (+2.4)	73.9 (90.8%)	76.7 (93.9%)
JMeter	56.5 (+1.4)	63.7 (+1.4)	44.3 (80.4%)	53.6 (86.0%)	84.0 (+0.5)	84.6 (+0.7)	73.8 (88.4%)	76.8 (91.5%)	82.9 (+0.6)	83.8 (+2.9)	75.6 (91.9%)	76.3 (94.3%)
Kafka	51.1 (+0.4)	52.8 (+1.0)	45.7 (90.1%)	50.8 (98.1%)	79.3 (+0.6)	80.2 (+0.7)	74.8 (95.0%)	76.8 (96.6%)	77.5 (+0.6)	78.9 (+1.4)	75.2 (97.8%)	75.9 (97.9%)
Karaf	57.9 (+1.4)	68.9 (+1.7)	45.3 (80.2%)	62.1 (92.4%)	85.1 (+0.8)	86.2 (+0.6)	74.7 (88.6%)	83.8 (97.9%)	82.6 (+2.0)	83.5 (+1.9)	77.1 (95.7%)	81.2 (99.5%)
Wicket	58.9 (+1.6)	65.9 (+2.1)	45.1 (78.7%)	58.3 (91.4%)	83.8 (+0.7)	86.1 (+1.1)	74.6 (89.8%)	81.0 (95.3%)	80.0 (+1.1)	84.3 (+5.0)	76.7 (97.2%)	78.8 (99.4%)
Zookeeper	54.5 (+1.7)	61.8 (+0.9)	45.0 (85.2%)	57.8 (94.9%)	80.3 (+0.7)	85.6 (+0.8)	73.5 (92.3%)	79.7 (94.0%)	79.9 (+1.1)	83.3 (+1.3)	76.1 (96.6%)	79.8 (97.3%)
Average	54.5 (+0.5)	61.8 (+0.7)	44.9 (83.1%)	57.3 (93.8%)	81.5 (+0.7)	83.9 (+0.2)	74.2 (91.8%)	80.5 (96.2%)	79.9 (+0.6)	82.8 (+2.0)	76.1 (96.0%)	78.8 (97.5%)

Note: The +/- number after each data in the columns of B-enlarge and C-enlarge indicates the improve or decrease compared with within-system prediction in RQ1. The percentage in the columns of B-cross and C-cross represents the ratio against the results of within-system prediction in RQ1.

limited dataset [154]. In particular, one may use data from other projects to complement the existing dataset to train a better model. In this RQ, we investigate whether our approach is still generalizable when training and testing on different data sets. In particular, we study two sub-RQs:

RQ3-A: Can we improve the performance of our approach by including more training data from other studied systems?

RQ3-B: How accurate is our approach in cross-system suggestions?

Approach. We choose to use of the Syn and Comb features of our approach, since both outperform the baselines, as discussed in RQ1. Below, we describe the approach of each sub-RQ.

RQ3-A: We enlarge the dataset by combing the data from all the studied systems. For each system, we follow stratified sampling to split the data into training data (60%), validation data (20%), and testing data (20%). We then merge the training data from all studied systems and train a deep learning model, while using the 20% validation data set (combined from every studied system) to validate the model during the training process. Finally, we apply the model trained using the enlarged dataset separately on the testing data of each studied system.

RQ3-B: For each target system, we combine the data from the remaining eight systems together and apply stratified sampling to split 80% of the data combined from the eight systems as training data, and 20% as validation data. We then use the complete data of the target system as the testing data and apply the model trained using the combined data from the other eight studied systems.

Results and Discussions.

RQ3-A: **Our approach can benefit from the enlarged training data from other systems.** Table 9 shows the results of enlarging the training set using the syntactic context features (S-enlarge) and the combined features (C-enlarge). The +/- number after each data indicates the improve or

decrease compared to within-system suggestion in RQ1. Overall, for both of the two features, the performance is improved in eight of the studied systems on all of the evaluation metrics. Specifically, for syntactic context features (i.e., S-enlarge in Table 9), the improvement of accuracy ranges from 0.4 in Kafka to 4.8 in Cassandra. The average AUC and AOD also improve by 0.7 and 0.6, respectively. For combined features (i.e., C-enlarge in Table 9), the improvement of accuracy ranges from 0.9 in Flink and Zookeeper, to 3.7 in HBase. The average AUC and AOD also improve by 0.2 and 2.0, respectively. On the other hand, the performance in Elasticsearch is decreased after enlarging the training data from other systems (accuracy decreases by 10.2 on S-enlarge, and by 8.6 on C-enlarge). As shown in Table 4, Elasticsearch has considerably different log level distribution compared to other systems. In particular, there exist considerably more *trace* level logging statements than other systems (28.5% versus 5.5%); while much fewer *error* level logging statements (9.9% versus 26%). Hence, the data from other systems may not be able to complement the data from Elasticsearch in the model training. Our finding shows that, while enlarging the training data may improve suggestion performance, practitioners should carefully and tactically choose the data when enlarging the training set.

RQ3-B: **Our approach achieves encouraging results in cross-system log level suggestions.** Table 9 shows the results of cross-system suggestions using the syntactic context features (S-cross) and the combined features (C-cross). The percentage following each number represents the ratio of the corresponding evaluation metric against the results of within-system prediction in RQ1. For example, the accuracy of C-cross in Cassandra is 58.9. Compared with the original within-system accuracy of Comb in Cassandra, i.e., 60.6, the accuracy ratio of C-cross against Comb in Cassandra is 97.2% (58.9/60.6).

Overall, the cross-system suggestions achieve 83.1% accuracy on average for S-cross compared to Syn in RQ1, and achieve 93.8% accuracy on average for C-cross compared to Comb in RQ1. We also find that the results of cross-system suggestions on combined features are still higher than the results of the two baselines in RQ1. In other words, even with cross-system suggestions, our approach can still outperform the two baseline approaches that are trained and tested with data from the same system.

> By enlarging the training set, the performance of our approach can be improved in eight out of nine studied systems. Our approach also has an encouraging performance for cross-system log level suggestions, which still outperforms the within-system suggestions by the baseline approaches.

4.5 Threats to Validity

Construct Validity. To mitigate the fluctuation caused by different testing data set, we follow prior studies to split the training, validation, and testing data [160, 87] and apply stratified random sampling [113, 160, 87] to ensure each randomly sampled data set has the same distribution of log levels as the original data. Our approach presumes that the training data has high-quality source code and follows good logging practice. However, there is no "golden rule" for how to write logging statements, which may affect the stability of logging statements [65, 162]. To mitigate this threat, we choose nine well-maintained, large-scale systems across various domains, with different sizes to conduct our study. They are commonly used in prior log-related studies and are considered as following good logging practice [18, 19, 81, 73, 72].

Internal Validity. Different hyper-parameters used in the neural networks might affect the effectiveness of the trained models. We follow the advanced practices from prior studies [160, 87, 63] to set the hyper-parameters for our deep learning framework. We conduct manual studies to investigate whether log level is implicitly or explicitly related to log message or the structural information of the logging statement. To avoid biases, we examine the data independently. For most of the cases, we reach an agreement. Any disagreement is discussed until a consensus is reached with a substantial-level agreement (Cohen's Kappa of 0.82 and 0.88 for logging location and log message, respectively) [126]. Involving third-party logging experts to verify our manual study results may further mitigate this threat.

External Validity. Our studied systems are all implemented in Java, the results and models may not be transferable to systems in other programming languages. We conducted our study on nine large-scale open source systems only. However, we selected the studied systems that are across various domains, different sizes, and different amount of logging statements in order to improve the representativeness of our studied systems. Future studies should validate the generalizability of our findings and the transferability of our models in systems that are implemented in other programming languages.

4.6 Related Work

Studies on Logging Practices. Chen et al. [17] and Yuan et al. [157] conducted quantitative studies on logging statements in large-scale open source C/C++ and Java systems, respectively. They found that logs are essential for debugging and maintenance purposes. Fu et al. [43] studied the logging practices in Microsoft software systems. They investigated what categories of code blocks (e.g., catch blocks) are logged. Li et al. [73] summarized the benefits and costs of logging

through a qualitative study. Zhi et al. [166] studied how logging configurations are used in practice with respect to logging management, storage, and formatting. In this study, we focus on studying the characteristics of log levels, specifically, their explicit or implicit relationship with the syntactic context or message of a logging statement. The findings of our study could complement prior studies in providing more comprehensive logging supports to developers.

Improving Logging Practices. Given the importance of logging, some studies try to help developers improve logging practices. Yuan et al. [158] proposed an approach that can automatically insert additional variables into logging statements to enhance the error diagnostic information. Zhu et al. [168] proposed an automated tool for suggesting logging locations. Li et al. [80, 78] proposed a deep learning framework for suggesting logging locations at the code block level. Liu et al. [87] proposed a deep learning framework to suggest the variables that should be recorded in logging statements. Chen et al. [18] found that developers commonly make some mistakes when writing logging statements (e.g., logging objects whose values may be null) and concluded five categories of logging anti-patterns from code changes. Li et al. [81, 77, 82] uncovered potential problems with logging statements that have the same text message and developed an automated tool to detect the problems. Hassani et al. [51] identified seven root-causes of the log-related issues from log-related bug reports and found that inappropriate log messages and missing log statements are the most common issues. Different from prior studies, we focus on suggesting log levels by using features extracted from the source code. We conduct a manual study on the characteristics of log levels and propose a deep learning based approach to provide automated suggestions.

4.7 Conclusion

Deciding proper log levels for logging statements is a challenging task. In this chapter, we tackle the challenges in two steps. First, we conduct a manual study on the characteristics of log levels. We find that the syntactic context of logging statements and their messages, as well as the ordinal nature of log levels might be leveraged to help determine proper log levels. We then propose a deep-learning based approach to automatically suggest log levels for logging statements. Our approach ordinally encodes log levels and leverages the syntactic context information and the log message information of each logging statement to provide log level suggestions. Our approach outperforms the baseline approaches and are effective at suggesting log levels in both within-system and cross-system scenarios. Our results also highlight future research opportunities on improving logging decisions, for example, by leveraging the characteristics of different log levels to help distinguish similar log levels. Practitioners may also benefit from our findings to make better logging decisions.

Chapter 5

Studying Practitioners' Expectation on the Readability of Log Messages

Developers write logging statements to generate logs that provide run-time information for various tasks. The readability of log messages in the logging statements (i.e., the descriptive text) is crucial to the value of the generated logs. Immature log messages may slow down or even obstruct the process of log analysis. Despite the importance of log messages, there is still a lack of standards on what constitutes good readability in log messages and how to write them. In this chapter, we conduct a series of interviews with 17 industrial practitioners to investigate their expectations on the readability of log messages. Through the interviews, we derive three aspects related to the readability of log messages, including *Structure*, *Information*, and *Wording*, along with several specific practices to improve each aspect. We validate our findings through a series of online questionnaire surveys and receive positive feedback from the participants. We then manually investigate the readability of log messages in large-scale open source systems and find that a considerable percentage (38.1%) of the log messages have inadequate readability. Motivated by such observation, we further explore the potential of automatically classifying the readability of log messages using deep learning and machine learning approaches. We find that deep learning approach achieves the best results with a balanced accuracy of 84.4% on average. Our study provides comprehensive guidelines for composing log messages to further improve practitioners' logging practices.

5.1 Introduction

Developers can leverage the valuable information in logs to assist in many tasks, such as program comprehension [89, 104], anomaly detection [162, 150], and failure diagnosis [167, 91, 158, 84, 119,

118]. The value of logs highly relies on the quality of log messages (i.e., the part of *"Successfully updated remote job [{}]"* in the example above). Developers leverage the information in log messages as clues for debugging and failure diagnosis, unclear log messages may confuse developers and further slow down or even obstruct the process of log analysis [73]. For example, if the log message is only "*Shutting down.*", it is still difficult to know what is shutting down.

Prior studies provide some supports on composing logging statements. For example, where to insert logging statements [168, 43, 80, 16, 75], how to choose the verbosity level [74, 83, 86], and generate logging statements by learning from existing data [94, 37]. However, to the best of our knowledge, there is a lack of practical standards or systematic investigation on what are "good" log messages that record valuable information and are easy to comprehend. Therefore, how to compose log messages with "good" readability that can clearly and sufficiently record system runtime behaviors is still an on-going challenge. The reliability of automated recommendations learned from log messages with inadequate readability might also be decreased.

We conduct a comprehensive study to investigate practitioners' expectation on the readability of log messages and seek possible improvements: 1) We first conduct a series of semi-structured interviews with 17 industrial practitioners from 11 companies worldwide to gain insights on their perspectives of log messages' readability; 2) We manually study the readability of log messages in nine large-scale open source software systems; 3) We validate our findings from the interviews and manual studies through an online questionnaire survey with 56 participants; 4) We further explore the potential of automatically classifying the readability of log messages using deep learning and machine learning approaches.

In particular, we study the following three research questions:

RQ1: What are practitioners' expectation on the readability of log messages and how to improve it?

By analyzing the interview records, we derive three aspects that are related to the readability of log messages, including ***Structure***, ***Information***, and ***Wording***. For each aspect, we also derive several specific practices that can improve readability. Our survey participants acknowledge the importance of these aspects and the effectiveness of improvement practices. Among the three aspects, *Information* is considered as the most important aspect: 87.5% of the participants consider it is "Very important" and 12.5% consider it is "Important".

RQ2: How is the readability of log messages in large-scale open source software systems?

We use the data set of logging statements provided by a prior study [83] to manually investigate the readability of log messages based on the three aspects discussed in RQ1. We find that 61.9% of the log messages on average have adequate readability in all three aspects, meaning that a large portion

of the log messages (i.e., 38.1%) in these systems have inadequacy in terms of their readability.

RQ3: What is the potential of automatically classifying the readability of log messages? We explore the potential of automatically classifying whether a log message has readability issue or not using several deep learning and machine learning approaches (e.g., Bi-LSTM, Random Forest, Decision Tree). We find that Bi-LSTM achieves the best results in classifying each of the three aspects of readability, with an average balanced accuracy of 84.4%.

The contributions of this study are as follows:

- We are the first study that investigates the readability of log messages by conducting interviews with industrial practitioners. We derive three aspects that are related to the readability of log messages and several corresponding practices to improve the readability for each aspect.
- We find that a large portion of the log messages in large-scale open source systems actually have inadequate readability.
- We explore the potential of automatically classifying the log messages whose readability might need improvement and achieve encouraging results.

Chapter Organization. Section 5.2 summarizes the related work. Section 5.3 describes the research methodology of our study. Section 5.4 presents the results by answering three research questions. Section 5.5 discusses the implications of our study to practitioners and researchers. Section 5.6 discusses the threats to validity of our study. Section 5.7 concludes the chapter.

5.2 Related Work

Empirical Studies on Logging Practices. Yuan et al. [157] studied the logging practices in C/C++ applications and found that developers often improve log messages as after-thoughts. Chen et al. [17] further studied the logging practices in Java applications and pointed out the similarities and differences of logging practices compared to C/C++ applications. Zeng et al. [159] and Patel et al. [110] investigated the logging practices in Android applications and Linux kernel, respectively. Their findings show that logs are essential for debugging and maintenance purposes in different domains and platforms. Other prior studies focused on assisting developers in making logging decisions. For example, Fu et al. and Li et al. [43, 80] investigated where logging statements were placed to identify the common categories of logging locations. Zhu et al. [168] proposed an automated tool for suggesting logging locations. Several prior studies [74, 83, 86] proposed automated approach to help developers select the appropriate verbosity level. These studies focus on empirically studying logging practices or provide supports for deciding the logging locations or verbosity levels. In our

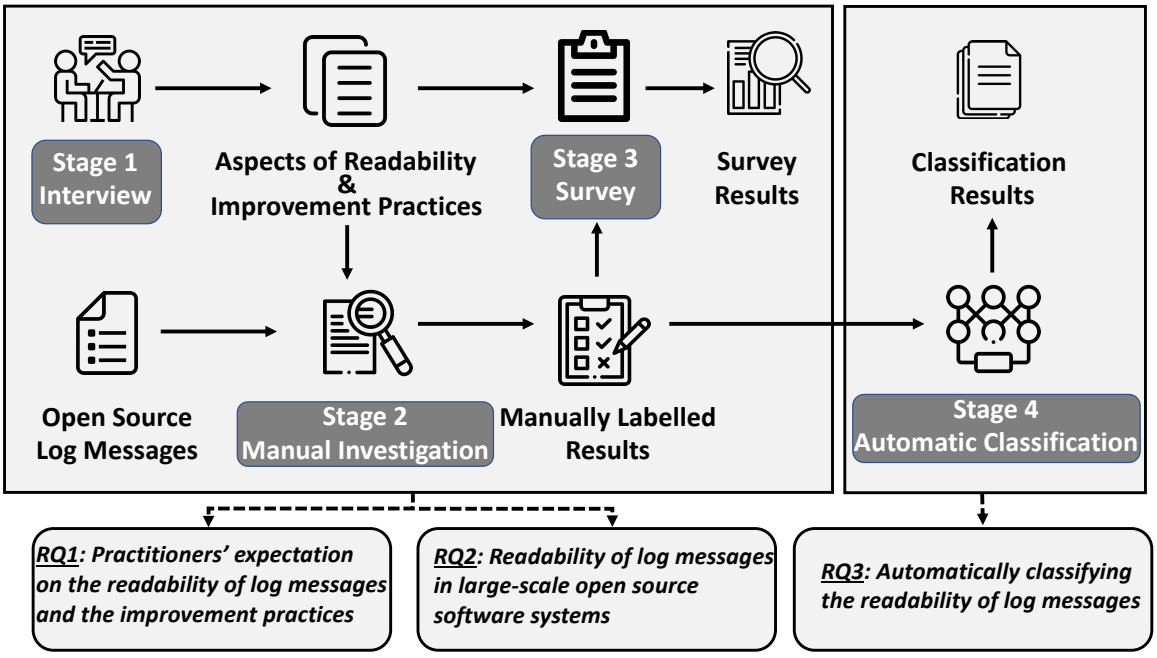


Figure 10: Overview of our research methodology and corresponding research questions.

study, we investigate practitioners' expectation on the readability of log messages, which complement prior studies on improving logging practices.

Studies on Log Messages in Logging Statements. He et al. [53] empirically studied the n-gram patterns of log messages and proposed an information retrieval based approach that generates log messages from similar code snippets. Ding et al. [37] formed the process of log message generation as neutral machine translations and achieved promising results in such generations. Mastropaolo et al. [94] proposed a deep learning based approach that can generate complete logging statements, including log messages for Java methods. Despite the extensive studies on log messages in logging statements, the readability of those log messages has not been investigated thoroughly. In this study, we systematically study the readability of log messages and derive three aspects related to the readability. For each aspect, we also derive several improvement practices based on our interviews with industrial practitioners.

5.3 Research Methodology

Our research methodology consists of four stages, as shown in Figure 10. *Stage 1:* Semi-structured interviews [92, 66, 139] with practitioners from industry on their experiences in reading log messages, their perspective on the readability of log messages and how to improve it. *Stage 2:* Manual investigation on how prevalent are log messages that may need improvement based on the aspects of

readability derived from the interview results. *Stage 3:* A questionnaire survey [67, 10, 109, 147] for confirming the aspects of log message's readability with the corresponding improvement practices that are summarized from the interview, and verifying the manual investigation results in the prior stage. *Stage 4:* Study of the potential of automatically classifying the readability of log messages.

5.3.1 Stage 1: Interviews

In our interviews with industrial practitioners, we investigate their perspective on the readability of log messages and their expectation on the specific practices that can improve the readability.

Interview Process. We first developed an interview guideline by brainstorming a set of open-ended questions. We then followed the guideline and conducts a series of individual interviews using online video-conferencing tools with 17 software practitioners. Before the start of each interview, we first sent the introduction part of the guideline to the interviewee to let them know the background information of our study, ensure that they are aware of the interview being recorded, and emphasize that we will protect participants' identities. Each interview takes 30-40 minutes, and is semi-structured with three parts of questions.

Part 1: We asked some questions about the interviewees' background information (e.g., years of experiences, role of responsibility, and programming languages used in daily job).

Part 2: We asked open-ended questions about their experiences in reading and analyzing log messages (e.g., "What kind of information provided by the log messages is important to you?", "Have you ever seen some log messages that are confusing or not helpful?").

Part 3: We asked the interviewees about their expectations on log messages with good readability, and what practices can practitioners do to improve the readability of log messages.

At the end of each interview, we thank the interviewee and verify there is no sensitive information mentioned in the recorded interview.

Interviewees. The interviewees are full-time employees working in software engineering related roles from 11 companies worldwide that are leading in their domains (the 17 interviewees are denoted as *I-1* to *I-17* when discussing their answers). The domain of those companies includes software development, telecommunications, electronics, investment management, and digital currency management. The role of interviewees includes developers, software architects, algorithm engineers, data analysts, and test engineers. We only invite interviewees who indicate that they have experience and knowledge in logging. Their years of experience in software development and maintenance is 7.5 on average, ranging from 4 to 18 years.

Data Analysis. After we completed all the interviews, we transcribed the interview record and performed open coding to generate an initial set of codes from the transcripts. We then verified

Table 10: An overview of the studied systems. LOC: Lines of code, NOL: Number of logging statements.

System	LOC	NOL	Sample
Cassandra	432K	1,316	298
ElasticSearch	1.50M	2,619	337
Flink	177K	2,455	333
HBase	1.26M	5,524	360
JMeter	143K	1,848	319
Kafka	267K	1,563	308
Karaf	133K	706	251
Wicket	216K	413	201
Zookeeper	97K	1,245	295
Total	4.2M	17,689	2,702

the codes and provided suggestions for improvement. We further removed the codes that are not related to the readability of log messages (e.g., some interviewees mention that the timestamp of logs should have a consistent time zone setting, which is not related to the composition of log messages). We then perform open card sorting [127] on the generated codes to analyze the thematic similarity. Specifically, we independently analyze the codes and sort the generated codes into potential themes that indicate the expected practices on the readability of log messages. We use Cohen's Kappa [97] to measure the agreement. Overall, we have a Cohen's Kappa value of 0.76, which indicates a substantial agreement. We then discuss the disagreements until a consensus is reached. Eventually, we derive three aspects that are related to the readability of the log message, including ***Structure***, ***Information***, and ***Wording***. Each aspect corresponds to several specific improvement practices that can be used to improve the readability. Some of the improvement practices are *corrective practices*, which are to improve the inadequacy of readability in log messages. Some are *enhancing practices*, developers may decide whether to apply them or not based on the situations and needs. We discuss each aspect and the corresponding practices in the results of RQ1 (Section 5.4).

5.3.2 Stage 2: Manual Investigation

In this stage, we manually investigate the readability of log messages in real-world open-source systems based on the aspects derived from the interviews.

Data Preparation. To investigate the readability of log messages in widely used large-scale software systems, we conduct our manual study on the data set of logging statements discussed in

Chapter 4. Table 10 presents the details of the data set. The data set includes a total of 17.7K logging statements collected from nine large-scale open source Java systems. The number of logging statements in each system ranges from 413 in Wicket to 5,524 in HBase. For each system, we randomly sample a set of logging statements to conduct the manual investigation based on 95% confidence level and 5% confidence interval [13]. We randomly sample 2,702 logging statements from the nine systems. The sample size of each system varies from 201 in Wicket to 360 in HBase.

Manual Investigation Process. We examine the sampled logging statements (i.e., 2,702 logging statements in total) with their surrounding code snippets. For each sampled logging statement, we independently label whether the readability of its log message is adequate for each of the three aspects (i.e., *Structure*, *Information*, and *Wording*). When the labeling is finished, we then compare their results and discuss each disagreement until reaching a consensus. We have a Cohen's Kappa [97] value of 0.83 in this process, which indicates a substantial agreement.

5.3.3 Stage 3: Survey

To quantify the findings derived from our interviews and verify the manually investigated results, we conduct an online questionnaire survey with a larger number of participants.

Survey Design. The survey has five parts: Part 1 to Part 4 include multiple-choice questions, and Part 5 includes an open-ended question.

Part 1: We ask some background information related to the role and experience of the participants.

Part 2: We ask the participants for their perspective on the three aspects of readability derived from our prior interviews. We illustrate each aspect by providing two real-world examples which are against and for the readability in the corresponding aspect. The participants then choose their consideration on the importance of the aspect to the readability of log messages from "Very important", "Important", "Neutral", "Unimportant", and "Very unimportant". At the end of this part, we further ask the participant for their overall perspective on the three aspects.

Part 3: We ask the participant for their perspective on the practices that can improve the readability of log messages from the corresponding aspect. We illustrate each practice by using a set of examples. We then provide a statement indicating the effectiveness of each practice and ask the participant to choose their agreement level on the statement following a 5-point Likert scale (i.e., "Strongly agree", "Agree", "Neutral", "Disagree", and "Strongly disagree").

Part 4: We randomly select seven logging statements from our manual investigation results (i.e., *Stage 2*, Section 5.3.2) and ask the participants to examine their readability. The participant can choose whether the log message of each logging statement is good or bad in terms of each of the three derived readability aspects. The main purpose of this part is to verify our manually labelled log messages in *Stage 2*.

Part 5: We ask if the participants have other comments or ideas regarding the readability of log messages.

In each multiple-choice question, we also provide an additional option "Not sure" if the participant cannot understand the question or does not have a clear answer. We also provide a comment field for each question where the participants are free to leave their comments related to the question.

In Part 4, each participant has different randomly selected logging statements to examine their readability. Due to the randomness of this part, we use online document platform (e.g., Google Doc) to design the surveys. Specifically, we prepare a series of survey documents in which Part 4 has unique logging statements and other parts have identical questions. Each participant has a unique link to the survey where the participants can directly write their answers.

We conduct a pilot survey with a small number of practitioners first to collect their feedback on the overall design of our survey. We make minor modifications to refine the description of questions based on their feedback and have a final version of the survey. We then distribute our final version of the survey to the participants. We exclude the responses collected from the pilot survey when we analyze and present the survey results.

Participants. We contact professionals in leading IT companies from our networks and ask their help to disseminate our survey to their colleagues. In total, we send out 80 surveys and receive 56 responses from the participants. Their years of experiences vary from 1 to 17 years, with an average of 5.4 years. The top two roles of the participants are developer (25 participants) and algorithm engineer (13 participants).

Data Analysis. We discard all the answers that select "Not sure". For the answers in Part 2 and Part 3, we report the percentage of each selected option. For the answers in Part 4, we analyze the results labelled by the participants and compare with our manual study results in *Stage 2* to examine the agreement level. We discuss the comments and feedback that we receive from the participants in Section 5.4.

5.3.4 Stage 4: Automatic Classification

As the first step to help improve the quality of log messages, in this stage, we seek to explore the potential of automatically classifying the readability of log messages. Specifically, for each aspect of readability (i.e., *Structure*, *Information*, and *Wording*), we classify whether a log message's readability is adequate or not in such aspect.

Data Preparation. We use the manually labelled log messages in *Stage 2* to train the models for automatic classification. For each log message, we tokenize it by space and attach its verbosity level (e.g., *info* or *error*) as the input feature, and use the labelled results as the target to predict.

There are two steps for verifying the manually labelled log messages: 1) We independently label the log messages and discuss any disagreement until a consensus is reached. The Cohen's Kappa value of this process is 0.83, which is a substantial agreement; 2) In our survey discussed in *Stage 3*, we also ask the participants to label seven randomly sampled log messages. We receive 392 labelled log messages from the 56 participants. We further exclude the results of log messages with answers that are "Not sure". We then have 366 available log messages labelled by the survey participants, which is larger than the statistically significant sample size of 337, computed from the 2,702 log messages based on a 95% confidence level and a 5% confidence interval [13]. In this process, we find that a large number of the log messages labelled by the participants (81%) are exactly consistent with ours (i.e., the labels of all the three aspects are the same), which indicates that the results of manual investigation have high agreement with the survey participants.

Classification Process. Deep learning and machine learning approaches are widely used in the tasks of Software Engineering [142, 93, 58, 35]. We use one deep learning and four machine learning approaches to explore the potential of automatic classification. For deep learning, we use Bi-LSTM [120]; for machine learning, we use Logistic Regression [98], Decision Tree [116], Random Forest [15], and SVM [114]. We use Keras [3] and Scikit-learn [6] to implement the deep learning approach and machine learning approaches, respectively. As a majority of the log messages may have adequate readability, we use a state-of-the-art oversampling technique on the training data, namely ADASYN [52], to mitigate the potential impact of imbalanced data. For the vectorization of input features, we use Skip-gram from Word2vec [2] to train the word embeddings and transform the input features into numeric vectors. We then use each approach to train the models and evaluate their performance.

5.4 Results

5.4.1 RQ1: What are Practitioners' Expectation on the Readability of Log Messages and How to Improve It?

In this RQ, we discuss the three aspects that are related to the readability of the log message derived from our interviews with practitioners, including *Structure*, *Information*, and *Wording*. For each aspect, we discuss: 1) Real-world example log messages that are against and for the corresponding aspect, respectively; 2) Discussion of the interview and survey results; 3) Practices that can improve the readability. Some of the practices are *"corrective practices"*, which are practices to improve the inadequacy of readability in log messages. Some of the practices are *"enhancing practices"*, where developers can decide whether to apply them or not based on the situations and

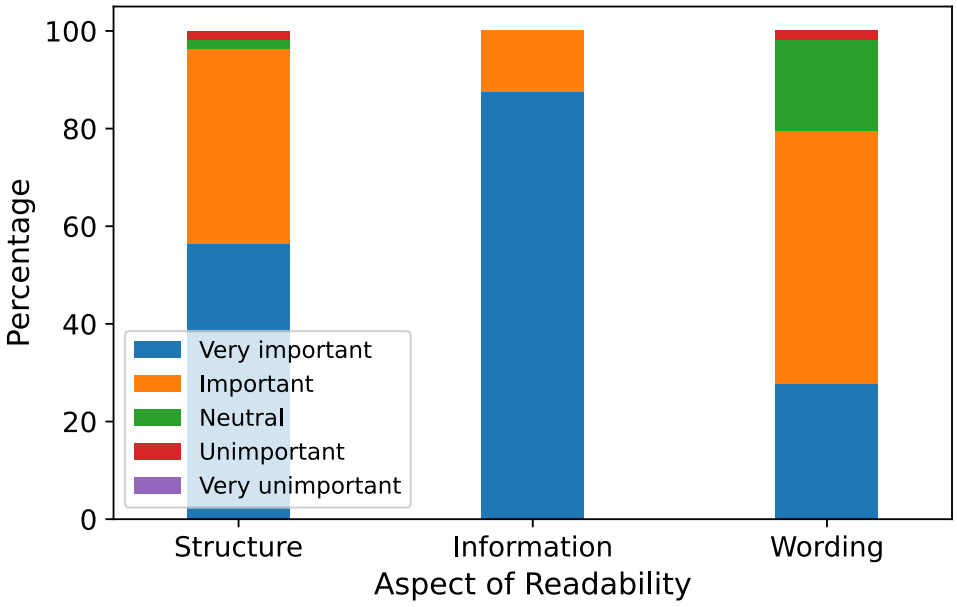


Figure 11: Survey participants' rating for the importance of the three aspects.

needs.

Aspect 1 - Structure

Description. Format and organization of words and variables that a log message presents its information.

Discussion. Below, we discuss the interview results and survey results related to the aspect of *Structure*, respectively.

Interview Results. Among the 17 participants, 9 participants mention that Structure is important to the readability of log messages. For example, interviewee *I-8* expects that the log message should be *"well structured so it is easy to read by human"*. Interviewee *I-3* also mentions that:

"Log message with good readability should have clear structure. For example, log messages that clearly separates variables could be easier to read. Don't present variables closely that are hard to judge boundaries."

Survey Results. In our survey, we ask the participants for their perspective on the importance of each aspect. Figure 11 presents the percentage of each rate of importance given by the survey participants. We exclude three answers which are "Not sure" in all the three aspects and compute the percentage based on the remaining answers. Overall, more than half of the participants (55.3%) consider that Structure is "Very important" to the readability of log messages, and 39.3% of the participants consider it is "Important". Some survey participants also comment their perspective on

this aspect. For example, one participant mentions that:

"The aspect of Structure affects how the message is formulated. Better formulated log messages are always easier to read than unformulated ones".

Improvement Practices. We derive three practices related to the aspect of *Structure*, including one corrective practice (i.e., practices to improve the inadequacy of readability in log messages) and two enhancing practices (i.e., practices that developers can apply them based on the situations and needs). Below, we discuss each practice with corresponding examples.

SP1 (Corrective Practices): Have clear boundaries and distinctions among items.

Different items in the log messages (e.g., variables) should have clear boundaries and descriptions to be easily distinguished. As shown in the example below, the four variables in Example 1 are presented one by one which might be difficult to understand the meaning of each variable. Example 2 shows the log message that adopts this corrective practice, where each variable is added with a description of its meaning. In our interviews, 5 out of the 17 participants mention that this practice can improve the readability of log message.

```
//Example 1 - WITHOUT this practice
LOG.debug("Reading from {} {} {} {}",
    tableDesc.getTableName(),
    region.getRegionNameAsString(),
    column.getNameAsString(),
    Bytes.toStringBinary(startKey));

//Example 2 - WITH this practice
LOG.debug("Reading from table: {}, region: {},
    column: {}, key: {}",
    tableDesc.getTableName(),
    region.getRegionNameAsString(),
    column.getNameAsString(),
    Bytes.toStringBinary(startKey));
```

SP2 (Enhancing Practices): Use an easy-to-parse structure if needed and possible.

Five interviewees mention that developers could consider formatting the log message that is easy to be automatically parsed by scripts for further analysis. For example, the code snippet shown below uses a comma (",") to separate each part. The ideal situation is to have log messages that are both human-readable and machine-readable.

```
//Example - WITH this practice
```

```
logger.info("Summary of the change, term: {},
    version: {}, reason: {}",
    newClusterState.term(),
    newClusterState.version(), task.source);
```

SP3 (Enhancing Practices): Use parameterized logging to present the variables.

Two interviewees mention that the log message in the logging statement with parameterized logging is easier to revisit and revise. Moreover, though not related to readability, parameterized logging has better performance compared to simply concatenating the strings (according to the documentation of Log4j2 [12]).

```
//Example 1 - WITHOUT this practice
LOG.error("Exception when formatting: '" + dateStr
    + "' from: '" + fromFormat + "' to: '" +
    toFormat + "'", e);

//Example 2 - WITH this practice
logger.info("Exception when formatting: '{}' from
    '{}' to '{}' ", dateStr, fromFormat, toFormat,
    e);
```

Aspect 2 - Information

Description. Semantic information conveyed by the log message to record system execution behaviors.

Discussion. Below, we discuss the interview results and survey results related to the aspect of *Information*, respectively.

Interview Results. All of our 17 interviewees consider that the actual information that a log message conveys is important to its readability. For example, interviewee *I-9* mentions that:

"The context of the log is important. When diagnosing the log, I would like to know how it happened. Like is it caused by an incorrect path or failed creation of files. It's also useful to know what is the consequence. Such as the consequence of the missing file. Will the system use the default configuration file or handle it with a different procedure."

Survey Results. As shown in Figure 11, most of the participants (87.5%) consider that Information is "Very important" to the readability of log messages, and the remaining participants consider it is "Important". The results show that the participants highly acknowledge the importance of *Information* to the readability of log messages. For example, a survey participant comments that:

"With more accurate information, the information aspect helps readers to better understand the message communicated by developers".

Improvement Practices. From our interviews, we derive three practices related to the aspect of *Information*, including two corrective practices and one enhancing practice. Below, we discuss each practice with corresponding examples.

IP1 (Corrective Practices): Provide proper context for the run-time behaviors.

As shown in the examples below, the system execution behavior is the interruption of a current thread. In Example 1, the log message is just "Interrupted", while it's unclear what is interrupted. In Example 2, some context information of the execution behavior (i.e., the current thread) is added to the log message. It would be even better to include the thread ID if available.

```
//Example 1 - WITHOUT this practice
Thread.currentThread().interrupt();
LOG.info("Interrupted");

//Example 2 - WITH this practice
Thread.currentThread().interrupt();
LOG.info("The current thread is interrupted");
    //(also add thread ID if available)
```

In our interviews, the participants suggest some context information that can be added into the log messages. We summarize the information into the following categories:

- Intention of this log message (clearly show whether it needs instant attention or not).
- Traceable information (e.g., thread and application ID).
- Clear "main character" of what happened *from* or what happened *to*.
- What is happening at the time.
- What is the consequence of this event.
- Possible reason of an unexpected event.

Note that it is not necessary to always include all of the context information every time, but our interviewees mention that the log messages should at least provide useful information and important events should include as sufficient context information as possible.

IP2 (Corrective Practices): Write a self-explanatory log message that is independent of other log messages.

Six interviewees mention that log message should be self-explanatory and not depend on other log messages. As shown in Example 1 below, the log message in *debug* level is "Full exception". However, these two info and debug logging statements may not always be generated closely together (i.e., other logs may be generated in between). If other logging statements appear before the debug one, it can be confusing to only see "Full exception" without the prior message. Hence, in Example 2, complete information is added to the *debug* level log to make it self-explanatory and avoid potential confusion.

```
//Example 1 - WITHOUT this practice
} catch (final AmazonClientException e) {
logger.info("Exception while retrieving instance
    list from AWS API: {}", e.getMessage());
logger.debug("Full exception:", e); //depending on
    the prior info log

//Example 2 - WITH this practice
} catch (final AmazonClientException e) {
logger.info("Exception while retrieving instance
    list from AWS API: {}", e.getMessage());
logger.debug("Exception while retrieving instance
    list from AWS API, full exception: ", e);
    //provides complete information that does not
    depend on other log messages
```

IP3 (Enhancing Practices): Minimize noise, emphasize the key information.

Four interviewees mention that they want to concisely see the key information without too much noise. As shown in Example 1, the log message gives the instruction first and only mentions the error code and error message at the end. In Example 2, we simplify the log message to emphasize the error code and error message. Developers could consider adding another log message or use another way to provide additional instructions if needed.

```
//Example 1 - WITHOUT this practice
```

```
LOG.warn("An HTTP error response in WebSocket
    communication would not be processed by the
    browser! If you need to send the error code
    and message to the client then configure
    custom WebSocketResponse via
    WebSocketSettings#newWebSocketResponse()
    factory method and override #sendError()
    method to write them in an appropriate format
    for your application. The ignored error code
    is '{}' and the message: '{}'.", sc, msg);

//Example2 - WITH this practice
LOG.warn("An HTTP error response in WebSocket
    communication would not be processed by the
    browser. Ignored error code: '{}', message:
    '{}'. ", sc, msg);
/*mention the key information first, can add
    another log, or use another way to write the
    additional instruction if it's necessary*/
```

Aspect 3 - Wording

Description. Lexical usage of words and punctuation in the log message.

Discussion. Below, we discuss the interview results and survey results related to the aspect of *Wording*, respectively.

Interview Results. Among the 17 participants, 7 participants mention that Wording is important to the readability of log messages. Some interviewees describe the scenarios where the wording affects the readability of log messages. For example, interviewees *I-1* and *I-13* mention that:

"Similar to writing source code, we should have consistent naming conventions for the words of log messages too. Otherwise it might be confusing to the users".

"I've read some logs that have weird names included, hard to understand their meaning. Like are they identifiers or the abbreviations of anything".

Survey Results. As shown in Figure 11, 50.0% of the participants consider that Wording is "Important" to the readability of log messages, and 26.8% of the participants consider it's "Very important". There are also 17.9% of the participants consider the importance of Wording is "Neutral". The survey results show that participants generally acknowledge the importance of *Wording*, but the priority is lower than *Information* and *Structure*. Some participants also provide comments to this aspect, for

example:

"Wording is important, but to a certain extend. Like tiny lexical mistakes can be acceptable."

"If the log message uses very emotional wording, I will obviously pay more attention to it and unhappy to see if it's just a trivial event".

Improvement Practices. We derive five practices related to the aspect of *Wording*, including three corrective practices and two enhancing practices. Below, we discuss each practice with corresponding examples.

WP1 (Corrective Practices): Use standard English words (e.g., avoid typos, incomplete words).

Four interviewees mention that we should avoid typos and incomplete words when writing the log messages. As shown in the example below, the word "preform" is a typo and should be "perform".

```
...
    //Example - WITHOUT this practice
    LOG.debug("Failed to preform reroute after cluster
        settings were updated."); //"preform" is a
        typo and should be "perform"
...
```

WP2 (Corrective Practices): Follow the convention of written language (e.g., capitalization, tense of verbs, not too colloquial).

Three interviewees mention that log messages are better to follow the convention of written language. As shown in the example below, we do not "exists" is incorrect and should be "exist".

```
//Example - WITHOUT this practice
LOG.debug("Pinging a master {} but we do not
    exists on it, act as if its master failure");
    //we do not "exists" should be "exist"
```

WP3 (Corrective Practices): Try to use impartial and neutral wording (e.g., avoid being too emotional or abusing capitalization).

Three interviewees mention that the emotion of log messages should try to be neutral and objective. The examples shown below are both log messages with improper emotional wording. In Example 1, the log message is oral and emotional, which does not help with understanding the log message. Example 2 abuses the capitalization for a non-critical system event (i.e., *info* level).

```
//Example 1 - WITHOUT this practice
LOG.error("!!!!!Uh-oh, didn't find any action
    handlers!!!!!");


//Example 2 - WITHOUT this practice
```

```
LOG.info("Added to offline, CURRENTLY NEVER
    CLEARED!!!");
```

WP4 (Enhancing Practices): Be careful on using abbreviations and acronyms.

Four interviewees mention that proper usage of abbreviations and acronyms is important. Developers should ensure that users can understand the meaning of abbreviations and acronyms before writing them into the log message. As shown in the example, the abbreviation "TGT" is not a well-known word. Probably only users with corresponding domain knowledge can understand the meaning.

```
//Example
LOG.warn("No TGT found: will try again at {}");
```

WP5 (Enhancing Practices): Consistent on the wording of domain-specific terms.

Six interviewees mention that the use of domain-specific terms should be consistent, otherwise it might be confusing for the users to understand their meaning. As shown in the example, "Incident ID" and "IncID" refer to the same thing. If possible, developers should consider keeping a consistent convention on the wording of domain terms to mitigate potential confusion.

```
//Example - WITHOUT this practice
LOG.info("Incident ID {}: a new incident is
    reported.", incID);
...
LOG.info("IncID {}: the incident is closed.",
    incID);
```

Overall Perspectives on the Aspects and Improvement Practices

In our survey, we also asked the participants for their overall perspectives on the three aspects above. Particularly, we asked if these three aspects can reflect the readability of log messages. Participants can choose from "Very positive", "Positive", "Neutral", "Negative", "Very negative", and "Not sure". Overall, 51.8% of the participants' responses are "Very positive", and the remaining responses are "Positive". The results show that our survey participants acknowledge that the three aspects can reflect the readability of log messages.

We also ask the survey participants for their agreement on the effectiveness for each improvement practice. For example in *Information Practice 1 (IP1)*, we provide a statement: *"This practice can improve the readability of log messages from the aspect of Information"*. Participants can choose their agreement level based on a 5-point Likert scale (i.e., "Strongly agree", "Agree", "Neutral", "Disagree", "Strongly disagree"), and an additional option of "Not sure". We exclude the answers that are "Not

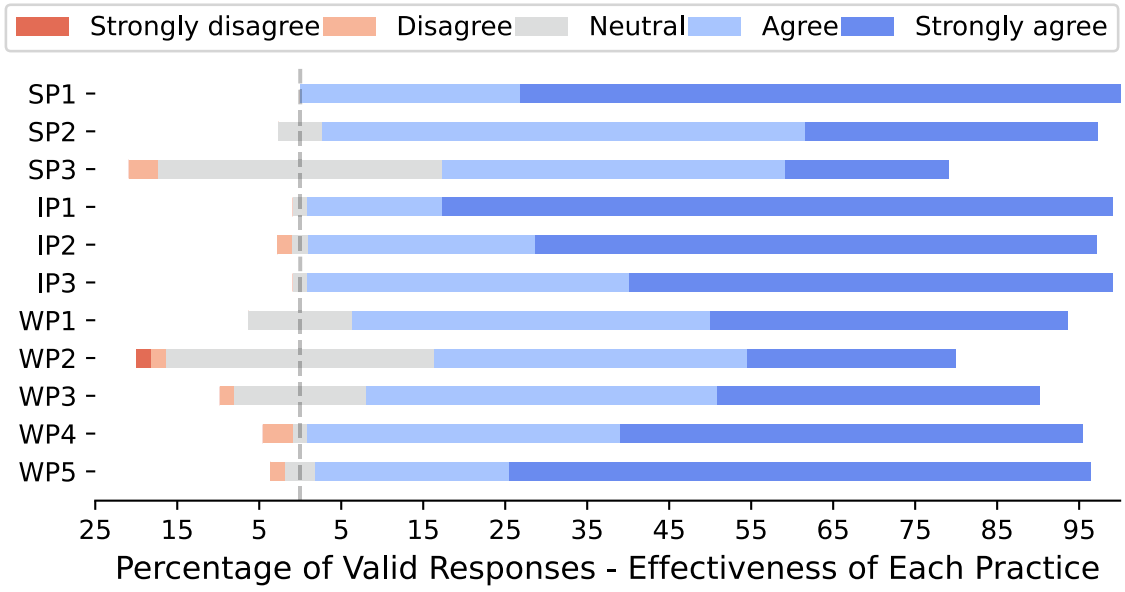


Figure 12: Survey participants' rating for each improvement practice.

sure" (1.3% of the total answers) and present the distribution of results in Figure 12. We find that for all the improvement practices, most of the responses have positive ratings (i.e., "Strongly agree" and "Agree"). Among the improvement practices for each aspect, *Information Practices* have the highest percentage of positive ratings, with an average of 97.7% for the effectiveness.

By analyzing the interview records, we derive three aspects that are related to the readability of log messages, including ***Structure***, ***Information***, and ***Wording***. For each aspect, we also derive several specific practices can be used to improve the readability in such aspect. Our survey participants acknowledge the importance of these aspects. Among the three aspects, *Information* is considered as the most important aspect: 87.5% of the participants consider it is "Very important" and 12.5% consider it is "Important".

> We derive three aspects that are related to the readability of log messages and several practices to improve each aspect. Among the three aspects, *Information* is considered as the most important aspect: All of our survey participants consider it is either "Very important" or "Important".

5.4.2 RQ2: How is the Readability of Log Messages in large-scale Open Source Software Systems?

In this RQ, we present the results of our manual investigation on the readability of 2,702 logging statements sampled from nine large-scale open source systems, following the process discussed in *Stage 2* of Section 5.3. We analyze the manual investigation results and present the results for: 1)

Table 11: Percentage (%) of log messages in each system that have adequate readability for all the three aspects, or inadequate in each of the aspect.

Data set	**Adequate**	**Inadequate**		
		Structure	Information	Wording
Cassandra	60.1	16.7	23.2	26.2
Elasticsearch	46.9	14.5	22.8	49.9
Flink	76.3	12.3	17.7	14.7
HBase	55.6	25.0	24.7	30.0
JMeter	52.0	27.0	30.7	36.4
Kafka	67.5	23.7	15.9	11.0
Karaf	75.7	12.0	16.7	21.1
Wicket	70.1	12.9	17.9	24.9
Zookeeper	60.0	15.6	23.1	34.6
Overall	61.9	18.2	21.7	28.1

Readability for log messages in different systems; 2) Readability for different lengths of log messages.

Readability for Log Messages in Different Systems. Table 11 presents the percentage of log messages in each data set that have adequate readability for all the three aspects (i.e., the column of *Adequate*), or inadequate in each of the aspect (i.e., *Structure*, *Information*, and *Wording* under the column of *Inadequate*). The row of *Overall* shows the overall percentage computed from all the data combined together. We find that the percentage of log messages with adequate readability varies in different systems, from 46.9% in Elasticsearch to 76.3% in Flink. We also find that the distribution of aspects is different for log messages with inadequate readability among the systems. For example, 49.9% of the log messages in Elasticsearch have inadequate readability in the aspect of Wording, while for Structure and Information the percentages are 14.5% and 22.8%, respectively.

Readability for Log Messages with Different Lengths. Figure 13 presents the percentage of log messages with adequate or inadequate readability for different lengths. We compute the length of a log message based on its number of words. *Adequate* refers to the percentage of log messages that have adequate readability in all the three aspects, *Inadeq-S*, *Inadeq-I*, and *Inadeq-W* refer to log messages that have inadequate readability in the aspect of *Structure*, *Information*, and *Wording*, respectively. We find that when the log messages are very short (i.e., length$\leqslant$2), only 7.4% of the log messages have adequate readability in all the three aspects, with a very high percentage of *Inadeq-I* (89.4%). In contrast, log messages whose length is within the range between 6 to 10 words have the highest percentage with adequate readability (80.6%). When the log messages have more than 10

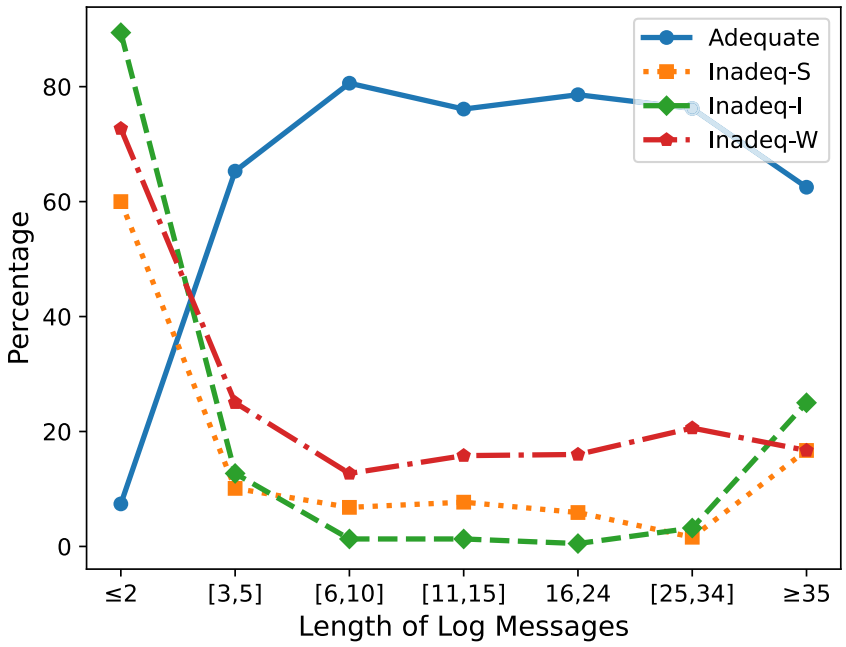


Figure 13: Percentage of log messages with adequate or inadequate readability for different lengths. Length refers to the number of words of a log message.

words, we then find that the readability has a downward trend as the length increases. For example, we find that the percentage of *Adequate* drops from 76.2% (log messages with number of words between 25 and 34) to 62.5% when the log messages have more than 35 words. Overall, the results show that the length of a log message might be an indicator of its readability, especially when the length is very short.

Moreover, we also ask the interviewees for their expectations on the length of log messages in the interviews. In total, 6 out of the 17 interviewees expect that the log message should be neither too short nor too long, 4 interviewees consider that the log message should not be too short and 2 interviewees consider it should not be too long. There are also 5 interviewees do not have a specific expectation on the length itself, but the log message should provide clear and useful information. We find that our results in this RQ confirm the expectations from the interviewees. Compared to extremely short or long, log messages with a proper length tend to be more readable and are preferred by the practitioners.

We find that only 61.9% of the log messages in our studied data set have adequate readability in all the three aspects, meaning that a large portion of the log messages (i.e., 38.1%) in these systems have inadequacy in terms of their readability.

5.4.3 RQ3: What is the Potential of Automatically Classifying the Readability of Log Messages?

We take a preliminary step to help developers improve log message by classifying whether a message has readability issue or not. In this RQ, we present the results of automatic classification for the readability of log messages. We use the 2,702 manually labelled log messages to train and test the models using each approach discussed in Section 5.3. We then perform a stratified 10-fold cross validation to estimate the performance of each approach and report the average results. Specifically, we randomly split the data set into ten subsets, with stratified random sampling [113] to ensure the same distribution of readability for each subset. The validation has ten rounds in total. For each round of validation, we use one subset for testing, and the remaining subsets for training.

We first examine the balanced accuracy of each approach on classifying the three aspects of readability. Balanced accuracy is widely used by prior studies to evaluate the performance of binary classification on imbalanced data [168, 80, 75]. As shown in Table 12, we find that Bi-LSTM achieves the best balanced accuracy in all the three aspects of readability, with an average of 84.4%. The machine learning approaches achieve a balanced accuracy from 65.0% by Logistic Regression to 80.2% by Random Forest. Overall, we find that deep learning and machine learning approaches can both achieve promising classification results. Among them, Bi-LSTM achieves the best balanced accuracy.

We further examine the performance of Bi-LSTM on classifying the adequacy and inadequacy of each aspect. Table 13 shows the Precision, Recall, and F1 score of classifying each aspect of readability using Bi-LSTM. When *Adequate* readability in each aspect is considered as the positive class, Bi-LSTM achieves an average precision, recall, and F1 score of 93.1%, and 89.8%, and 91.4%, respectively. When *Inadequate* readability in each aspect is considered as the positive class, the average precision, recall, and F1 score are 71.5%, and 79.0%, and 75.1%, respectively. Overall, we find that Bi-LSTM can effectively classify each aspect of the readability.

> Deep learning and machine learning approaches can both achieve promising results in the classification. Among the approaches, Bi-LSTM can effectively classify each aspect of the readability and achieve the best balanced accuracy.

5.5 Implications

We discuss the implications of our study for practitioners and researchers, respectively.

Implication for Practitioners. Due to the lack of well-defined guidelines on writing the log message, it is a challenging task to write log messages with good readability that can clearly and sufficiently record system run-time behaviors. Moreover, it is also difficult to decide what are log

Table 12: Balanced Accuracy (%) of different approaches on classifying the readability for each aspect.

	Structure	**Information**	**Wording**	*Average*
Bi-LSTM	86.6	90.4	76.3	84.4
Random Forest	80.2	88.1	72.6	80.3
Decision Tree	78.8	86.5	73.2	79.5
Logistic Regression	65.0	78.3	60.1	67.8
SVM	67.9	85.3	72.4	75.2

Table 13: Precision, Recall, and F1 score (%) of Classifying each aspect of readability using Bi-LSTM.

Metric		**Structure**	**Information**	**Wording**	*Average*
Adequate	**Precision**	95.4	96.2	87.8	93.1
	Recall	93.7	94.3	81.5	89.8
	F1	94.5	95.2	84.6	91.4
Inadequate	**Precision**	73.6	81.0	60.0	71.5
	Recall	79.6	86.4	71.1	79.0
	F1	76.5	83.6	65.1	75.1

messages with "good readability". In our study, we conduct a series of interviews with industrial practitioners and derive three aspects that are related to the readability of log messages (i.e., *Structure*, *Information*, and *Wording*). For each aspect, we also discuss several specific practices that may improve the readability in such aspect. Practitioners can consider to refer our findings to have a clearer comprehension of the readability when composing and revising the log messages.

We also explore the potential of automatically classifying the readability of log messages. We find that several widely used deep learning approaches and machine learning approaches (e.g., Bi-LSTM, Random Forest, and Decision Tree) are effective in such classifications. Practitioners can leverage the automated approach to examine the readability of log messages they compose and obtain a suggestion of whether any aspects of the readability can be improved.

Implication for Researchers. In RQ2, we find that 61.9% of the studied log messages in large-scale open source systems have adequate readability. Therefore, there is still a large portion of the log messages with in adequate readability. Some prior studies work on automatically generating log messages using existing source code and log messages [53, 37]. However, we observe that these studies directly use the log messages to train and evaluate the models without a verification on the quality of those log messages. As a consequence, log messages with poor readability may be generated and thus decrease the reliability of such approaches. Future studies may leverage the findings in our study to examine the readability of log messages and prompt automated generation using more well-verified log messages.

5.6 Threats to Validity

Internal Validity. We manually label the readability of log messages for each aspect. To mitigate the potential subjectivity, we label the log messages independently, and discuss each disagreement until a consensus is reached. The Cohen's Kappa value in this process is 0.83, which shows a substantial agreement. In our survey discussed in *Stage 3*, we also ask the participants to label 7 randomly sampled log messages. We receive 366 valid log messages labelled by the survey participants, which is larger than the statistically significant sample size of 337, computed from the 2,702 log messages based on a 95% confidence level and a 5% confidence interval [13]. We find that 81% of the log messages labelled by the participants are exactly consistent with ours.

External Validity. We derive three aspects of readability and the corresponding improvement practices from the interviews. The logging practices might vary in different companies and thus the interview results may be different. To mitigate such threat, we invite participants from a variety of large companies to participate in our study, and the domain of their companies range from software development to digital currency management. These participants represent a variety of roles and

level of software development and maintenance expertise.

5.7 Conclusion

Due to lack of guidelines on writing log messages, it is unclear on how to systematically write log messages with readability that succinctly and sufficiently record system execution behaviors. In this chapter, we investigate practitioners' expectation on the readability of log messages by conducting a series of interviews with industrial practitioners. We derive three aspects related to the readability of log messages along with several improvement practices for each aspect. Our findings receive encouraging feedback from subsequent online questionnaire surveys. We also find that a considerable proportion of the log messages in large-scale open source systems have inadequate readability. Therefore, we further explore the potential of automatically classifying the readability of log messages and find that both deep learning and machine learning approaches can effectively perform such classifications. The findings of our study provide a systematic understanding of the readability of log messages and shed light for future studies on providing comprehensive and automated supports for practitioners' logging practices.

Part III

Assist in Log Analysis

Chapter 6

Studying and Exploring Variable-Aware Log Abstraction

Due to the sheer size of software logs, developers rely on automated techniques for log analysis. One of the first and most important steps of automated log analysis is log abstraction, which parses the raw logs into a structured format. Prior log abstraction techniques aim to identify and abstract all the dynamic variables in logs and output a static log template for automated log analysis. However, these abstracted dynamic variables may also contain important information that is useful to different tasks in log analysis. In this chapter, we investigate the characteristics of dynamic variables and their importance in practice, and explore the potential of a variable-aware log abstraction technique. Through manual investigations and surveys with practitioners, we find that different categories of dynamic variables record various information that can be important depending on the given tasks, the distinction of dynamic variables in log abstraction can further assist in log analysis. We then propose a deep learning based log abstraction approach, named VALB, which can identify different categories of dynamic variables and preserve the value of specified categories of dynamic variables along with the log templates (i.e., variable-aware log abstraction). Through the evaluation on a widely used log abstraction benchmark, we find that VALB outperforms other state-of-the-art log abstraction techniques on general log abstraction (i.e., when abstracting all the dynamic variables) and also achieves a high variable-aware log abstraction accuracy that further identifies the category of the dynamic variables. Our study highlights the potential of leveraging the important information recorded in the dynamic variables to further improve the process of log analysis.

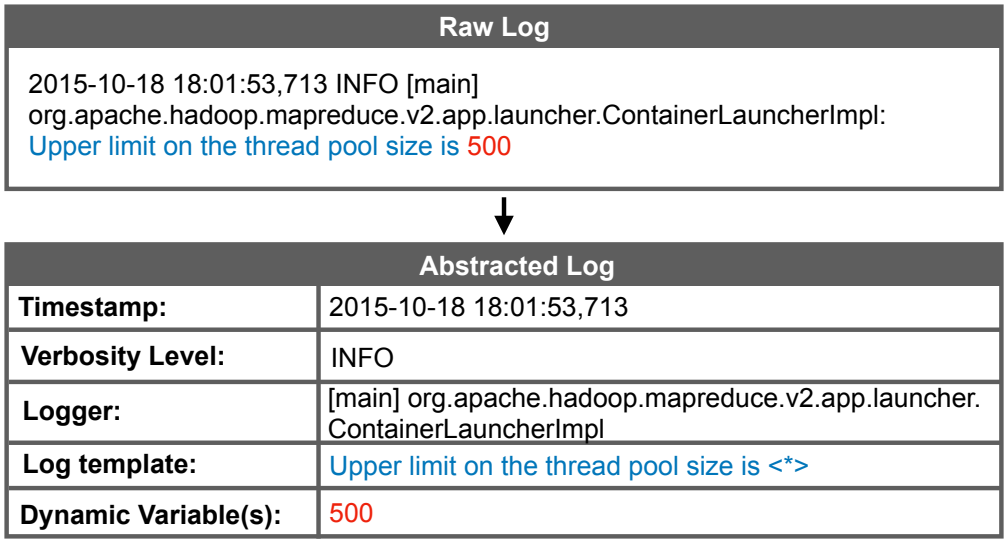


Figure 14: An example of the log abstraction process.

6.1 Introduction

Logs play an important role in software systems to record system execution behaviors. Practitioners leverage logs to assist in various tasks in the process of software development and maintenance, such as failure diagnosis [158, 167, 119, 129, 163], program comprehension [99, 105, 104, 68, 31, 44], and anomaly detection [125, 150, 162, 164]. Although logs contain rich system run-time information, there are challenges associated with log analysis [151]. For example, modern software systems generate a large number of logs on a daily basis, resulting in tens of gigabytes or even terabytes of data [169, 33, 88]. Therefore, developers usually rely on automated techniques for log analysis.

To effectively facilitate automated log analysis, log abstraction (also called log parsing) techniques [125, 169, 33, 54, 64] are used to process the raw logs into a more structured format. Figure 14 shows an example of the log abstraction process. The raw log in the example is composed of a message header (e.g., timestamp) that can be configured via the logging library, the static words that remain constant, and the dynamic variables that may vary depending on the run-time behaviors. The goal of log abstraction techniques is to identify the static log templates and abstract the dynamic variables from the raw logs and output the log templates. The sequence of log templates can then be leveraged for further log analysis (e.g., anomaly detection [55, 164]).

Prior studies propose various log abstraction techniques using different algorithms [169], such as frequent pattern mining (e.g., Logram [33] and LFA [103]), clustering algorithms (e.g., SHISO [101] and Lenma [124]), heuristics (e.g., Drain [54], AEL [64], and IPLoM [90]), and combined approaches (e.g., ULP [121]). Their common goal is to abstract all dynamic parts of logs and output the

remaining static words. However, the recorded dynamic values can provide valuable information to assist log understanding and analysis. In this study, we aim to understand the characteristics of dynamic variables and their importance in log analysis. We then seek to propose a log abstraction approach that can selectively abstract dynamic variables that belong to specific categories based on the needs.

We first empirically study the dynamic variables in logs. We manually study a widely used log abstraction benchmark data set [169] to uncover the characteristics of dynamic variables in logs and uncover 10 categories of dynamic variables. We then have a questionnaire survey in a world-leading software company (Company X) to investigate how do practitioners in the industry consider the usage and importance of dynamic variables in logs. Through our empirical study and survey, we find that different categories of dynamic variables record valuable information that can be important for different tasks. In our survey with industry practitioners, we also find that the practitioners acknowledge the importance of dynamic variables in logs, and a log abstraction technique that can preserve the categories of dynamic variables as specified may further help log analysis.

Motivated by our findings and practitioners' feedback, we then propose VALB, which is a **V**ariable-**A**ware **L**og a**B**straction approach that can identify the static and dynamic parts in logs (as the prior log abstraction techniques do), and also further identify the categories of dynamic variables. **Practitioners can specify the categories of dynamic variables based on their needs, and the values of such dynamic variables will be preserved along with the log templates**. Overall, VALB achieves an average accuracy of 96.1% for general log abstraction, which is better than other state-of-the-art log abstraction techniques (average accuracy ranges from 74.5% to 82.5%). VALB also achieves an average accuracy of 95.5% for variable-aware log abstraction that further considers the correctness of identifying different categories of dynamic variables. Moreover, we find that the models of VALB are still efficient on a new system by re-using the models trained from other systems and fine-tune the models with a small amount of logs in the new system.

The contributions of this study are as follows:

- We investigate the characteristics and importance of dynamic variables in logs, which are omitted by prior log abstraction techniques. We find that different categories of dynamic variables record valuable information that can be important for different tasks, point out the need of a variable-aware log abstraction technique.

- We propose a deep learning approach, VALB, which is the first log abstraction approach that can further identify the categories of dynamic variables in the process of log abstraction. VALB achieve promising results in variable-aware log abstraction and also outperforms prior state-of-the-art techniques in general log abstraction.

- We explore the potential of variable-aware log abstraction on assisting in log-based downstream tasks and find that variable-aware log abstraction can be leveraged to improve the performance of anomaly detection.

In short, our study uncovers the importance of dynamic variables and highlights future research opportunities on studying the potential of leveraging dynamic variables in logs to further assist in log analysis.

Chapter Organization. Section 6.2 summarizes the related works. Section 6.3 discusses the motivating examples of our study. Section 6.4 presents our empirical study on dynamic variables in logs. Section 6.5 describes the data preparation and the deep learning framework of VALB. Section 6.6 evaluates VALB by proposing and answering three research questions. Section 6.7 discusses the potential of variable-aware log abstractions on assisting in log-based downstream tasks. Section 6.8 discusses the threats to the validity of our study. Section 6.9 concludes the chapter.

6.2 Related Works

Research on Log Abstraction. There are many prior studies that propose log abstraction techniques to assist in log analysis. Some prior studies use frequent pattern mining (e.g., SLCT [135], LFA [103], LogCluster [136], Logram [33]) to identify the static words that occur frequently in logs. Some studies leverage clustering algorithms to cluster similar logs (e.g., LKE [42], LogSig [130], SHISO [101], LenMa [124], and LogMine [49]), since logs in the same cluster then tend to have the same log template. Some prior studies use heuristics or combined approaches to identify the static and dynamic parts of logs [54, 64, 90, 121]. For example, Drain [54] uses a fixed-depth tree to maintain log groups with the same log template. IPLoM [90] leverages an iterative partitioning strategy to partition logs into different groups. ULP [121] combines string matching and local frequency analysis to parse large log files. In addition to prior log abstraction techniques that aim to identify and abstract all the dynamic parts in logs, our approach can also distinguish different categories of dynamic variables. Developers can specify the categories variables to keep their values based on needs.

Deep Learning in Log-related Studies. Recent studies apply deep learning techniques to address log-related problems. Specifically, those studies are related to logging (i.e., writing logging statements) and log analysis (e.g., anomaly detection). For logging, Liu et al. [87] and Li et al. [83] proposed deep learning based approaches to suggest the variables and verbosity level for logging statements, respectively. For log analysis, Zhang et al. [162] proposed an attention-based Bi-LSTM framework to detect log sequences that have anomalies. Yang et al. [150] used probabilistic label

	Sequence of Raw Logs
1	2017-05-16_13:55:31 2931 INFO nova.compute.manager [instance: 96abccce-8d1f] Took 19.96 seconds to spawn the instance on the hypervisor.
2	2017-05-16_13:55:31 2931 INFO nova.compute.manager [instance: 96abccce-8d1f] VM Resumed (Lifecycle Event)
3	2017-05-16_13:55:31 2931 INFO nova.compute.manager [instance: 96abccce-8d1f] Took 20.71 seconds to build instance.

	Parsed Sequence of Log Templates
1	[instance: <*>] Took <*> seconds to spawn the instance on the hypervisor.
2	[instance: <*>] VM Resumed (Lifecycle Event)
3	[instance: <*>] Took <*> seconds to build instance.

Figure 15: An example of log parsing before analyzing the sequences of logs. The dynamic variables in the raw logs and abstracted variables in the parsed log templates are marked in red.

estimation and proposed a semi-supervised anomaly detection framework. Different from prior studies working on logging or log analysis, our study uses deep learning techniques in the process of log abstraction.

6.3 Motivating Examples

Log abstraction has shown to be a crucial first step towards further log analysis [169, 125]. Prior log abstraction techniques aim at abstracting all the dynamic variables and output a static log template [169]. However, during our collaboration with Company X, which is a world-leading software company, practitioners mention that such log abstraction process will result in the loss of important information recorded by dynamic variables. They consider that different categories of dynamic variables record different information, which can be important to specific tasks.

We take anomaly detection as an example. Anomaly detection tools [150, 162] analyze the sequences of log templates generated from log abstraction techniques to detect system anomalies. Since the dynamic variables have already been abstracted, the input log templates only contain the information of static words in logs. Figure 15 shows an example of the process of log abstraction on a sequence of logs. Due to Company X's policy, we take the example logs from OpenStack, an open-source anomaly detection data set provided by the *LogPAI* project [56]. The data sets provide sequences of logs that are generated from normal and abnormal (e.g., system crash) system execution behaviors. In the example shown in Figure 15, the raw logs record a series of run-time information of an instance (i.e., a node) "96abccce-8d1f". The dynamic variables (i.e., instance ID and seconds

taken by an action) are then abstracted as wildcards in the output sequences of log templates.

The anomaly detection approach then learns from the sequences of log templates in the training data, and predicts whether a given log sequence indicates a normal or abnormal run-time behavior. However, the dynamic variables may also play an important role in distinguishing normal and abnormal system behaviors. For example, in the log template *"[instance: *] Took * seconds to spawn the instance on the hypervisor"*, the average value of *"seconds"* in normal log sequences is 19.78, while the average value in abnormal sequences is 53.87. For the other log template *"[instance: *] Took * seconds to build instance."*, the average values of *"seconds"* are 20.63 and 39.21 in normal and abnormal log sequences, respectively. This indicates that this anomaly might be a performance issue due to network latency.

Therefore, apart from the static parts in the logs sequences, dynamic variables may also contain important information for log analysis, yet such information is usually abstracted by prior log abstraction techniques. An investigation of the characteristics of dynamic variables and their importance in practice may further help improve log abstraction techniques and to better assist in log analysis.

6.4 Studying the Dynamic Variables in Logs

Motivated by practitioners' awareness of the importance of dynamic variables, in this section, we study the dynamic variables in logs. We first conduct a manual study on a widely used log data set [56] to study the characteristics of the dynamic variables and categorize the information they record. We then conduct a questionnaire survey [67, 66] in a world-leading software company to investigate how do practitioners consider the role that dynamic variables play in log analysis.

6.4.1 Manually Studying and Characterizing the Dynamic Variables in Logs

Studied Data Sets. We use the log data sets and benchmarks provided by *LogPAI* [56] to study the characteristics of dynamic variables in logs. LogPAI includes 16 data sets of logs generated from both open-source and commercial systems across various domains, and is widely used in prior log analysis studies [162, 150]. Each data set also provides a subset of 2,000 logs with manually derived log templates as the ground truth for evaluating the accuracy of log abstraction techniques. The manually derived log templates are commonly used in prior log abstraction studies for evaluating log abstraction accuracy [169, 33, 54]. We call this subset of data sets *log abstraction benchmark data sets* in the rest of this chapter.

In our manual analysis, we study the above mentioned log abstraction benchmark data sets (i.e.,

16 data sets in total, 2000 logs per data set, each log is manually labeled with its corresponding log template for the evaluation of log abstraction). Table 14 presents the details of the log abstraction benchmark data sets, including the number of log templates (NOL), number of log templates that have abstracted dynamic variables with the percentage among the total number of log templates (TWV (%)), and the number of abstracted dynamic variables (NOV). The number of log templates ranges from 6 (Apache) to 341 (Mac), and the number of abstracted dynamic variables ranges from 8 (Apache) to 595 (Mac). We also find that, in each data set, more than half of the log templates have abstracted dynamic variables (i.e., not pure static messages).

Manual Investigation on Dynamic Variables. In order to investigate what kind of information do those dynamic variables record and their potential importance, we manually study the logs in the log abstraction benchmark data sets to uncover the characteristics of dynamic variables in logs. Our study involves the following steps:

Step 1 : We go through all the log templates that have abstracted dynamic variables in the data sets (992 log templates in total). For the abstracted dynamic variables in each log template, we check its corresponding original logs and note down what kind of information is recorded by each dynamic variable.

Step 2 : We independently re-visit the notes in Step 1 and derive a list of categories that can describe the characteristics of the dynamic variables [14]. We then discuss the derived results. The categories are refined and revised during the discussion in this step.

Step 3 : We use the categories derived in Step 2 to label every dynamic variable in the data sets (2,188 dynamic variables in total) independently. The results have a Cohen's Kappa of 0.79, which is a substantial level of agreement [126]. We discuss the disagreements until a consensus is reached. We then compute the number of dynamic variables that belongs to each category.

Categories of Dynamic Variables. In our manual investigation, we uncover 10 categories of dynamic variables. Table 15 presents each category with its abbreviation (Abbrev.), description, example, and the number of variables that belong to this category. The dynamic variables in the example are marked in bold and underline.

Overall, we find that Object ID (OID), Location Indicator (LOI), Object Amount (OBA), and Object Name (OBN) have a relatively higher presence in our studied dynamic variables (255 to 739, out of 2,188). Among them, Object ID (OID) records the identification information of an object, such as a session ID or user ID. Location Indicator (LOI) shows the location information of an object. It can be path information, a URI, or an IP address. Object Name (OBN) shows the name of an object (e.g., domain name, task name, job name), and Object Amount (OBA) records the amount of an object (e.g., number of errors, number of nodes).

Time or Duration of an Action (TDA) and Computing Resources (CRS) also have over 100

Table 14: An overview of the log abstraction benchmark data sets. NOL: Number of log templates, TWV (%): Percentage of log templates with variables, NOV: number of variables

Name	NOL	TWV (%)	NOV
Android	166	107 (64.5%)	320
Apache	6	6 (100.0%)	8
BGL	120	106 (88.3%)	269
Hadoop	114	90 (78.9%)	160
HDFS	14	14 (100.0%)	36
HealthAPP	75	44 (58.7%)	74
HPC	46	27 (58.7%)	52
Linux	118	67 (56.8%)	117
Mac	341	268 (78.6%)	595
OpenSSH	27	23 (85.2%)	56
OpenStack	43	42 (97.7%)	91
Proxifier	8	8 (100.0%)	23
Spark	36	24 (66.7%)	39
Thunderbird	149	97 (65.1%)	202
Windows	50	31 (62.0%)	66
Zookeeper	50	38 (76.0%)	80
Total	1363	992 (72.8%)	2188

Table 15: The manually-derived categories of dynamic variables with their corresponding abbreviations (Abbrev.). Dynamic variable in the example is marked in **bold and underline**

Name	Abbrev.	Description	Example
Object ID	OID	Identification information of an object.	Added attempt **1445144423722** to list of failed maps.
Location Indicator	LOI	Location information of an object.	Adding path spec: /**mapreduce**.
Object Name	OBN	Name of an object.	ServerFileSystem domain **root10-local** is full.
Type Indicator	TID	Type information of an object or an action.	Using configuration type **1**.
Switch Indicator	SID	Status of a switch variable.	Saw change in network reachability (isReachable= **2**).
Time/Duration of an Action	TDA	Time or duration of an action.	Scheduled snapshot period at **10** second(s).
Computing Resources	CRS	Information of computing resource.	Combo kernel: **126MB** LOWMEM available.
Object Amount	OBA	Amount of an object.	Total of **23** ddr error(s) detected and corrected.
Status Code	STC	Status code of an object or an action.	mod-jk child workerEnv in error state **7**.
Other Parameters	OTP	Other information other than the above categories.	calvisitor kernel: payload Data **0700** to list of failed maps.

Table 16: The survey results of "Opinions on the Categories of Dynamic Variables in Logs". UI: usually important, CBI: can be important in some situations, UNI: usually not important.

Name	UI	CBI	UNI
Object ID (OID)	**45.5%**	40.9%	13.6%
Location Indicator (LOI)	**50.0%**	40.9%	9.1%
Object Name (OBN)	**59.1%**	31.8%	9.1%
Type Indicator (TID)	**63.6%**	27.3%	9.1%
Switch Indicator (SID)	**54.5%**	27.3%	18.2%
Time/Duration of an Action (TDA)	31.8%	**63.6%**	4.6%
Computing Resources (CRS)	**45.5%**	36.4%	18.1%
Object Amount (OBA)	**40.9%**	**40.9%**	18.2%
Status Code (STC)	**68.2%**	31.8%	0.0%
Other Parameters (OTP)	9.1%	**63.6%**	27.3%
Average	46.8%	40.5%	12.7%

dynamic variables in our studied data sets. TDA shows the time that an action happens or the duration of an action, and CRS shows how many computing resources are in use or left (e.g., memory or disk space). For the remaining four categories (i.e., Type Indicator (TID), Switch Indicator (SID), Status Code (STC), and Other Parameters (OTP)), they have relatively fewer numbers among the studied dynamic variables. However, they can also be important in some situations. For example, Status Code (STC) can show the code for some crucial events (e.g., an error code), which is usually important for error diagnosis.

6.4.2 A Survey on Log Analysis and Dynamic Variables

In our manual study, we find that different categories of dynamic variables in logs record various system run-time behaviors and uncover 10 categories of dynamic variables. In order to investigate how do developers in the industry consider the usage of dynamic variables in logs when doing log analysis, we conduct a questionnaire survey in Company X. Specifically, we conduct the survey in several production teams in Company X with more than one hundred full-time engineers in total. We first have a pilot survey with three engineers to collect their feedback and adjust the design of the survey accordingly. The final version of the survey has 17 questions, divided into four parts. We then distribute the survey link to the group chat of those teams. We receive 22 responses in total, the participants are engineers from various production teams such as cloud computing and social network services. We discuss each part of survey questions in detail.

Experience of the Participants (Q1). We ask the participants how many years of experience do they have in software development and maintenance. On average, the participants have 6 years of experience.

Opinions on the Categories of Dynamic Variables in Logs (Q2–Q11). We provide the participants all the 10 categories of dynamic variables that we uncovered with a corresponding example and ask the participants to consider the importance of each category in practice. Table 16 present the results of the participants' opinions on the categories of dynamic variables. For each category, participants can select if the category is "Usually important (UI)", "Can be important in some situations (CBI)", or "Usually not important (UNI)". The highest number for each category is marked in **bold**.

Overall, most of the participants consider the dynamic variables are usually important, or can be important in some situations. For all the 10 categories, from 27.3% to 63.6% of the participants consider that they can be important in some situations. For 5 out of the 10 categories (LOI, OBN, TID, SID, and STC), more than half of the participants consider that they are usually important. The results show that developers acknowledge the importance of dynamic variables in logs, while some variables are usually important and some are important depending on the situations.

Follow-up Questions on Dynamic Variables and Log Analysis (Q12-Q16). In this part, we ask the participants five multiple-choice questions related to the dynamic variables in logs and log analysis. For each question, participants can choose one score from 1 to 5, where 1 represents for "Very low extent", and 5 represents for "Very high extent". Table 17 presents the survey questions and results. The column of *Avg.* and *Med.* shows the average and median score, respectively.

Overall, the average and median scores for Q12 - Q16 are all above 4.0. The results of Q12 (an average of 4.5 and a median of 5.0) show that the participants acknowledge the importance of dynamic variables in log analysis. The results of Q14 (an average of 4.2 and a median of 4.0) then show that developers believe further distinguishing the categories of dynamic variables may further help log analysis. Based on the results of Q13 (an average and a median of 4.0), the participants consider that our derived categories can represent the dynamic variables to a high extent. Combining the results of Q15 and the percentage of CBI in Table 16, participants consider that different categories of dynamic variables may play different roles in log analysis, depending on the specific tasks or requirements. In Q16, most of the participants consider that it will be more helpful if the log abstraction technique can further identify and keep some certain categories of dynamic variables during the log abstraction process.

Additional Comments from the Participants (Q17). We provide an open-ended question to ask the participants if they would like to share some experience or leave some comments related to log analysis.

Table 17: List of questions and results for "Follow-up Questions on Dynamic Variables and Log Analysis", the answers are in a scale from 1 (very low extent) to 5 (very high extent).

	Question	**Avg.**	**Med.**
Q12	To what extent do you think the dynamic variables are important for log analysis?	4.5	5.0
Q13	For the 10 categories of dynamic variables, to what extent do you think they can represent the dynamic variables in practice? (The larger the number, the higher the representativeness of the categories)	4.0	4.0
Q14	To what extent do you think that distinguishing the dynamic variables into categories can further help log analysis?	4.2	4.0
Q15	To what extent do you think that, for different specific tasks/requirements, different categories of dynamic variables may have different importance?	4.5	5.0
Q16	Existing log abstraction technique is used to abstract the dynamic variables of logs and assist in automatic log analysis. If there is an alternative log abstraction tool that can further keep certain categories of dynamic variables (as specified by the user) and abstract the rest, to what extent do you think this alternative tool can be helpful in log analysis?	4.5	5.0

Some participants provide comments indicating the importance of dynamic variables in logs. For example, two participants commented that:

"In practice, we would like to pay attention to the specific parameter in a log, while sometimes we can just use the log template to pinpoint the issue. So this may be related to the specific issue. It should be interesting to study the relationship between issues and parameters or templates."

"Determining whether a variable is important or not really depends on the task. For example, if we want to find something related to a detected failure, the error code or the identifiable information will be very important."

There are also some participants who provide examples related to how can dynamic variables help log analysis. More details will be discussed in Section 6.7. Overall, developers consider that dynamic variables are important, and their importance is related to what information they record and the specific tasks.

We find that different categories of dynamic variables record valuable information that can be important depending on the tasks. We also find that practitioners in our survey consider the distinction of dynamic variables in the process of log abstraction can further help log analysis.

6.5 An automated Approach for Variable-aware Log Abstraction

Motivated by our empirical findings and practitioners' feedback, in this section, we propose a deep learning based log abstraction approach, called VALB, which is a **V**ariable-**A**ware **L**og a**B**straction technique. Given a set of logs, VALB can identify the static words (i.e., log templates), dynamic variables, and the categories of dynamic variables. Hence, VALB can be used as a general log parser when developers decide to abstract all the categories of dynamic variables. Moreover, as we found in the result of our survey, practitioners consider that different categories of dynamic variables can have different importance depending on the tasks. Therefore, VALB also allows developers to decide which categories of variables they want to keep and preserve the values of such dynamic variables.

We formulate our variable-aware log abstraction process as a sequence tagging problem, which is widely studied in the natural language processing area [32, 62]. A typical usage of sequence tagging in NLP is named entity recognition (e.g., given a sentence, to recognize which word is a person, which word is an organization, etc.). In our study, for a given log message, VALB aims to identify which words are static words, which words are dynamic variables and what are their corresponding categories. Below, we discuss how we annotate the dynamic variables and static words in logs, and the deep learning framework and implementation of VALB.

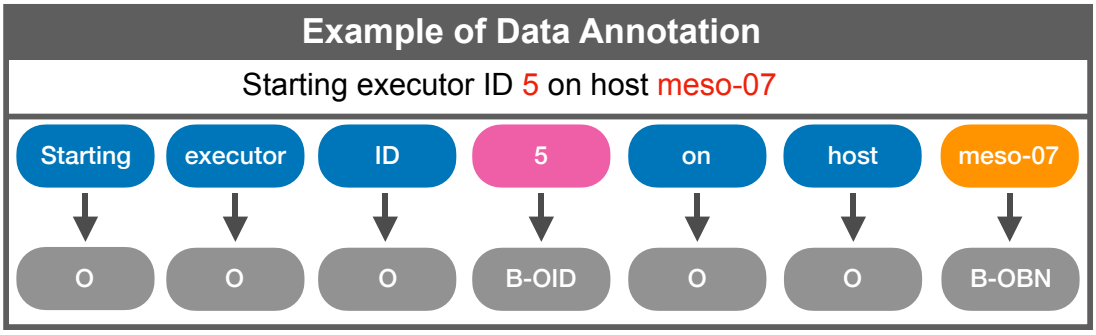


Figure 16: An example of our log annotation process. Static words are annotated with *O*, object ID is annotated with *B-OID*, and object name is annotated with *B-OBN*.

6.5.1 Data Annotation

VALB is based on supervised deep learning. In order to train the model that can identify the dynamic variables, their corresponding category, and static words in logs, we need to prepare the annotated training data. The training data consists of a specific amount of logs, and each word in the log is annotated with its category. For each word in the log, we use the IOB (inside-outside-beginning) annotation format [117] to annotate it with the categories that we found in Section 6.4. This format uses *"B-"* as the prefix of the beginning word of a named entity and uses *"I-"* as the prefix for the following word (if the following word exists). For the outside word of a named entity, it uses *"O"* as the annotation. In our study, for each dynamic variable, we use the aforementioned prefix as well as the abbreviation illustrated in Table 15 to annotate the category of each dynamic variable. For each static word, we annotate it as *"O"*.

Figure 16 shows the annotation process of a simplified log from the Spark data set. In the example, the log message is *"Starting executor ID 5 on host meso-07"*. The words *"Starting"*, *"executor"*, *"ID"*, *"on"*, and *"host"* are static words, so we annotate them as *"O"*. For the word *"5"*, it belongs to the category of *Object ID*, so we annotate it as *"B-OID"*. Similarly, we annotate the word *"meso-07"* as *"B-OBN"* (i.e., *Object Name*).

6.5.2 Deep Learning Framework and Implementation

Overall Architecture. Figure 17 shows the overall architecture of our deep learning framework. We first feed the log vectors into an embedding layer. Due to the variety of words in logs (e.g., numbers, normal words, and compound words), we use a combination of word embedding and character-level representation as our embedding layer. The embedding layer learns the relationship among the words and characters in logs and transfers the log vectors into probabilistic

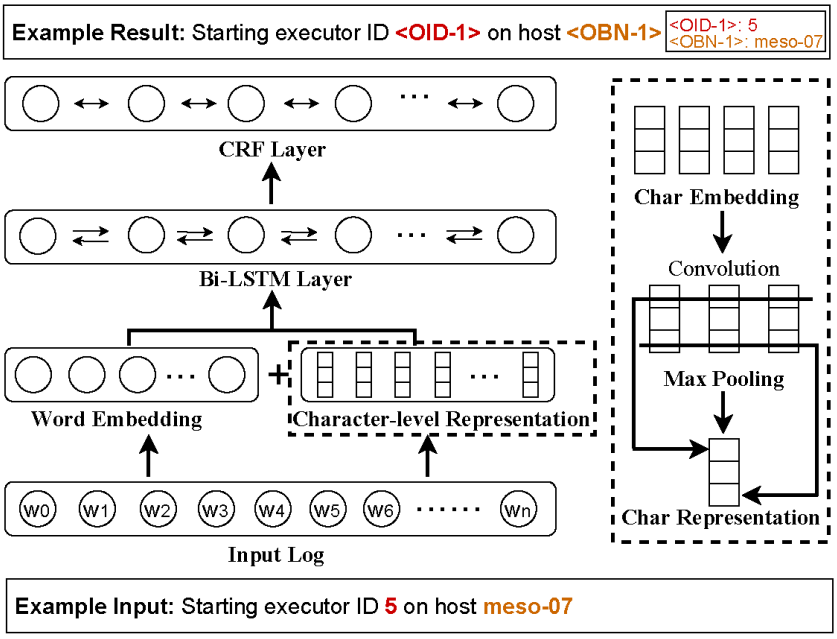


Figure 17: Overall diagram of our framework. Areas surrounded by dashed lines on the right illustrate the detailed structure for the character-level representation.

representations. We then use a Bi-LSTM (Bidirectional Long Short-Term Memory) layer to model the dependencies among the words in logs. Finally, we use a CRF (Conditional Random Fields) layer to model the relationships among the annotations of categories (e.g., which annotations are likely to appear together) and output the annotation of each word. We then analyze the annotation results and input logs to output the variable-aware log templates as the final results.

Embedding Layer. In the embedding layer, we use the concatenation of word embedding and character-level representation to transfers the log vector into probabilistic representations.

Word Embedding. For each log in the data set, the word embedding layer captures the relationship among the words in the log and transfers the log into probabilistic representations (i.e., probabilistic vector). Similar words tend to have a close distance in the vector space [133, 85, 140]. In our study, we use the GloVe embeddings [111] which are trained from six billion words collected from Wikipedia and the web.

Character-level Representation. Unlike static words that are always constant, the dynamic variables in logs can have various values based on different system behaviors. Moreover, many of the dynamic variables are numeric words (e.g., the category Object Amount in Section 6.4). The simple numbers from 0 to 9 can almost have unlimited potential of creating "new words" depending on various run-time information. This may result in very large size of the vocabulary and the OOV (out-of-vocabulary) problem while applying the models [87]. Hence, we also include character-level

representation together with the word embedding layer. The combined embedding layer can catch the relationships among both the words as well as the characters [38, 30]. We train the word embedding from the words in logs and then use CNN (Convolutional Neural Network) with max pooling to capture the relationship among the characters in words and build the character-level representation [23, 107]. We then concatenate the word embedding vector and character-level representation together and feed the combined embedding vector into the next layer.

Bi-LSTM Layer. Recurrent Neural Network is powerful at capturing the dependencies in sequential data [149, 95]. Log is a series of words that have sequential dependencies among the words. Similar to sentences in natural language, the words in a log may also have dependencies on past (i.e., words on the left) or future words (i.e., words on the right). Hence, we use Bi-LSTM (Bidirectional Long Short-Term Memory) [62, 106], a variant of RNN to capture the long term dependencies in the words from both directions. We then feed the output vectors in this layer that contains the dependency information in logs to the next layer.

CRF Layer. In sequence tagging tasks, the CRF (Conditional Random Fields) layer leverages the past and future tag in a sentence to predict the current tag [70]. It can learn the relationships and dependencies among the resulted annotations (e.g., which annotations are likely to appear together, and which annotations are not). In our deep learning framework, the CRF layer uses the vectors from the Bi-LSTM layer and leverages the category annotations of the past and the future words to predict the annotation (i.e., variable category) of the current word. For each line of log, the CRF layer outputs the category of each word in the log as the final result. We then analyze the results from CRF layer and output the static words (i.e., annotated as "O") and categories of dynamic variables as the result of variable-aware log templates (as shown in "Example Result" of Figure 17). The result also includes the value of each dynamic variable, developers can specify the categories and preserve their values.

Implementation and Hyper-parameters. We use Tensorflow [9] to implement our deep learning framework. To mitigate the impact of overfitting, we apply the dropout method for embedding layer and RNN layer, with a dropout rate of 0.2 [128, 58, 141]. For the embedding layer, we set the dimension as 300, filter size as 50, and kernel size as 3 for character-level embedding and CNN, and set the dimension as 100 for word embedding [160, 62, 87]. For the RNN layer, we set the hidden units as 128 [87]. For the training process, we set the number of epoch as 30 and the batch size as 8 [161, 58, 83].

6.6 Evaluation of VALB

In this section, we first discuss the experimental setup to evaluate VALB. We then propose three research questions and discuss the results.

6.6.1 Experimental Setup.

Data Preparation: We continue to use the log abstraction benchmark data sets provided by the *LogPAI* project [56] (as discussed in Section 6.4), which is widely used by prior log abstraction studies as the evaluation benchmarks [169, 33, 54], to train and evaluate the models. Specifically, we annotate all the 2,000 logs in each of the 16 data sets following the process discussed in Section 6.5. For each data set, we randomly split the 2,000 logs into training (20%), validation (20%), and testing data sets (60%). The intention of choosing a small size of training and validation data set is that, we want to investigate if training on a small data set can also achieve promising results. If so, the effort of preparing the training data sets can then be mitigated.

Baseline: Since there is no prior work on abstracting specific categories of dynamic variables in logs, we use the framework of VALB that excludes the char-level representations (i.e., only using regular word embedding) as the baseline approach. The purpose is to examine how character-level representation can help to model the diverse lexical usage of dynamic variables.

6.6.2 Research Questions

We discuss the results by answering three research questions. In RQ1, we use VALB as a general log parser that abstracts all the dynamic variables and compare the accuracy with other state-of-the-art log parsers. In RQ2, we examine the accuracy of VALB on variable-aware log abstraction that further identifies the category of dynamic variables. In RQ3, we investigate whether the trained models of VALB can be easily adopted to a new project.

RQ1: What is the Accuracy of VALB on General Log Abstraction?

Motivation. Prior log abstraction techniques aim at identifying the dynamic parts in the logs and completely abstract them [169]. Similar to prior works, VALB can also be used for general log abstraction if we only identify the dynamic variables and do not consider their categories. In this RQ, we investigate the accuracy of VALB when we use it for general log abstraction and compare it with other state-of-the-arts.

Approach. For each data set, we train and validate the model using the training and validation data sets and evaluate the accuracy on the testing data set. When we are training and evaluating the models, we first transfer the annotations of all the categories of dynamic variables to a single annotation that indicates the word is a variable, regardless of their categories. Given a log, VALB

can thus identify which words are static words and which words are dynamic variables, and output log templates without dynamic variables as what prior log abstraction works do.

For the accuracy of log abstraction, there are mainly two definitions: 1) a log is considered as correctly parsed if its event template corresponds to the same group of log messages in the ground truth [169, 54]; 2) a log is considered correctly parsed if and only if all of its static words and dynamic variables are correctly identified [33] (the category of dynamic variable is not considered). The first definition of accuracy does not examine if each word is correctly parsed. Therefore, we use the second definition of accuracy to examine the performance of VALB and other works on general log abstraction. The result of accuracy is computed as the ratio of correctly parsed logs against all the parsed logs. We refer to this accuracy as *general accuracy* in the rest of chapter. We use VALB as a general log parser (i.e., abstracting all the identified dynamic variables) and compare the accuracy with the top-3 state-of-the-art log parsers that have the highest accuracy reported in a prior study [33] (i.e., Logram [33], Drain [54], and AEL [64]) as well as our baseline approach.

Results and Discussions. Table 18 presents the general accuracy of our approach (VALB), the baseline (Base), and the other state-of-the-art log parsers. Each number indicates the ratio of the correctly parsed logs. The accuracy that is higher than 90.0% is marked in bold, and the highest accuracy among all the log parsers is marked with a star mark (*). Overall, VALB achieves the best accuracy in 15 out of the 16 data sets and the highest average accuracy across the data sets (96.1%). For the data set that VALB does not achieve the best accuracy, the accuracy of VALB is also close to the highest approach (e.g., VALB achieves an accuracy of 97.0% in HDFS, which is slightly lower than the highest accuracy of 99.9% achieved by AEL and Drain).

> VALB achieves a high accuracy in general log abstraction that abstracts all the identified dynamic variables (96.1% on average) , which outperforms other state-of-the-arts.

RQ2: What is the Accuracy of VALB on Variable-aware Log Abstraction?

Motivation. In our empirical study and survey, we find that practitioners acknowledge the importance of dynamic variables, and different categories of dynamic variables may have different usages depending on the tasks or scenarios. The findings point out the need of a variable-aware log abstraction technique that can preserve the value of specific categories of dynamic variables in the process of log abstraction. In this RQ, we evaluate the accuracy of VALB on variable-aware log abstraction, as well as the performance on identifying each category of dynamic variables. We study two sub-RQs:

RQ2-A: What is the accuracy of VALB on variable-aware log abstraction that can identify the static and dynamic parts in logs, and also further identify the categories of dynamic variables?

RQ2-B: What is the performance of VALB on identifying different categories of dynamic variables in logs?

Approach. Below, we discuss the approach of each sub-RQ.

RQ2-A: Apart from identifying static and dynamic parts in logs (i.e., general log abstraction), VALB can also identify the categories of dynamic variables (i.e., variable-aware log abstraction). To compute the accuracy of variable-aware log abstraction, we consider a log is correctly parsed when: 1) the static and dynamic parts are correctly identified **and** 2) all the categories of dynamic variables in a log are also correctly identified. We refer to this accuracy as *variable-aware accuracy* in the rest of this chapter. For each data set, we train and validate the model using the training and validation data sets and evaluate the variable-aware accuracy on the testing data set. Note that since prior log abstraction approaches cannot distinguish the categories of dynamic variables, we only compare the variable-aware accuracy of VALB with the baseline approach.

RQ2-B: In this sub-RQ, we further investigate the performance of VALB on identifying each category of dynamic variables. Specifically, we combine the results of all the 16 data sets in RQ2-A and compute an overall precision, recall, and F1 score for each of the 10 categories of the dynamic variables. These metrics are widely used by prior studies on sequence tagging [32, 62]. For each category, precision represents the ability of correctly identifying this category of dynamic variables (i.e., true positive divided by the sum of true positive and false positive); recall represents the ability of how many words in the log that belong to this category can be identified (i.e., true positive divided by the sum of true positive and false negative); and F1 score evaluates if the approach can both accurately and sufficiently identify the words that belong to this category. We also repeat the same process for the baseline and compare the baseline's performance with VALB.

Results and Discussions. We present and discuss the results of the two sub-RQs, respectively.

RQ2-A: Table 19 presents the variable-aware accuracy of VALB and the baseline approach. Overall, VALB achieves a high variable-aware accuracy ranging from 86.2% in Mac to 100.0% in Proxifier, which is also close to the general accuracy as discussed in RQ2-A. The average variable-aware accuracy of VALB is 95.5%, which is higher than the baseline (i.e., 86.2%). The results show that apart from general general log abstraction, VALB can also efficiently identify the categories of dynamic variables in the logs to perform a variable-aware log abstraction. Practitioners can specify the categories of dynamic variables based on their needs, and the values of such dynamic variables will be preserved along with the log templates for further log analysis.

RQ2-B: Table 20 shows the results of identifying different categories of dynamic variables using our approach (VALB) and the baseline (Base). We present the average results on identifying each category of dynamic variables from all the data sets to concisely show the overall performance. Each number represents for the average number computed from all the data sets. The *Average* line shows the arithmetic mean value of the corresponding column. Overall, VALB achieves over 90% in precision, recall, and F1 score for all the categories of dynamic variables and performs better than the baseline. VALB achieves an average precision of 96.2%, an average recall of 96.5%, and an

Table 18: Accuracy (%) of VALB on general log abstraction compared with other log parsers and the baseline (Base). **bold** numbers: higher than 90, star mark (*): highest accuracy in each row.

Dataset	AEL	Drain	Logram	Base	VALB
Android	86.7	**93.3**	84.8	79.8	**93.5***
Apache	69.3	69.3	69.9	**91.0**	**100.0***
BGL	81.8	82.2	74.0	83.0	**91.3***
Hadoop	53.9	54.5	**96.5**	**92.6**	**97.7***
HDFS	**99.9***	**99.9***	**98.1**	**91.1**	**97.0**
HealthAPP	61.5	60.9	**96.9**	75.8	**99.3***
HPC	**99.0**	**92.9**	**95.9**	**90.8**	**99.2***
Linux	24.1	25.0	46.0	**93.8**	**96.5***
Mac	57.9	51.5	66.6	67.2	86.6*
OpenSSH	24.7	50.7	54.5	**95.8**	**98.2***
OpenStack	71.8	53.8	84.7	**92.0**	**93.8***
Proxifier	**96.8**	**97.3**	**95.1**	**100.0***	**100.0***
Spark	**96.5**	**90.2**	**90.3**	**91.2**	**99.3***
Thunderbird	78.2	80.3	76.1	83.4	88.1*
Windows	**98.3**	**98.3**	**95.7**	**91.3**	**99.2***
Zookeeper	**92.2**	**96.2**	**95.5**	**92.1**	**98.3***
Average	74.5	74.8	82.5	88.2	**96.1***

Table 19: Variable-aware Accuracy (%) of VALB and the baseline (Base) discussed in RQ2, and Fine-tuning models with 50 logs from the target data set (F-50) discussed in RQ3.

	RQ2		RQ3
Dataset	**Base**	**VALB**	**F-50**
Android	76.0	**91.6**	82.3
Apache	**90.5**	**99.3**	**97.0**
BGL	82.0	89.6	86.7
Hadoop	**91.8**	**96.8**	**90.1**
HDFS	88.9	**96.5**	**95.0**
HealthAPP	75.1	**98.8**	**92.9**
HPC	86.6	**99.0**	**95.8**
Linux	**91.6**	**95.9**	**91.0**
Mac	63.8	86.2	78.0
OpenSSH	**90.1**	**97.6**	**91.5**
OpenStack	89.5	**93.2**	88.9
Proxifier	**100.0**	**100.0**	**100.0**
Spark	**90.7**	**99.1**	**92.3**
Thunderbird	80.6	87.8	82.3
Windows	**90.4**	**99.0**	**96.7**
Zookeeper	**91.7**	**98.1**	**95.4**
Average	86.2	**95.5**	**91.0**

Table 20: The results of identifying different categories of dynamic variables by our approach (VALB) and the baseline (Base).

	Precision (%)		Recall (%)		F1 (%)	
Category	VALB	Base	VALB	Base	VALB	Base
Object ID	96.5	89.2	95.9	93.1	96.2	91.1
Location Indicator	97.1	95.2	96.3	91.3	96.7	93.2
Object Name	99.8	95.5	98.3	95.8	99.0	95.6
Type Indicator	92.8	74.1	95.9	67.2	94.3	70.5
Switch Indicator	96.7	87.3	98.2	83.8	97.4	85.5
T. or D. of an Action	99.7	92.1	98.0	97.8	98.8	94.9
Computing Resources	98.7	91.2	97.3	91.7	98.0	91.4
Object Amount	92.5	77.5	96.9	87.8	94.6	82.3
Status Code	97.2	91.5	95.2	87.3	96.2	89.3
Other Parameters	91.1	72.9	93.0	82.2	92.0	77.2
Average	96.2	86.6	96.5	87.8	96.3	87.1

average F1 score of 96.3%; while the baseline achieves 86.6%, 87.8%, and 87.1%, respectively. VALB also has over 99% precision for Object Name (99.8%) and Time or Duration of an Action (99.7%).

VALB can effectively identify the categories of dynamic variables and achieves a high accuracy in variable-aware accuracy (95.5% on average), which outperforms the baseline approach without char-level representations (86.2% on average).

RQ3: Can the models of VALB be easily leveraged in a new project?

Motivation. Given that VALB is a supervised approach, the effectiveness of the models may rely on the training data. In this RQ, we would like to investigate how generalizable are the models of VALB. Specifically, we study how effective is VALB when the models are trained from other data and fine-tuned with a small size of data in the target data set.

Approach. We apply fine-tuning on existing models to investigate if VALB can be easily adopted to a new project with mitigated effort on data preparation. Specifically, for each target data set in the 16 data sets, we combine the training and validation data from the remaining 15 data sets and train a model. We then use a small size of logs (5, 10, 30, 50, and 100) from the training and validation data sets, respectively, from the target data set to fine-tune the model. We further use the fine-tuned model on the target testing data set to examine the variable-aware accuracy and compute an average number by combining the results of all the 16 target data sets together to show an overall trend for different size of fine-tuning logs.

Results and Discussions. Figure 18 shows the average variable-aware accuracy of models fine-tuned with different number of logs in the target data set (i.e., F-5, F-10, F-30, F-50, and F-50),

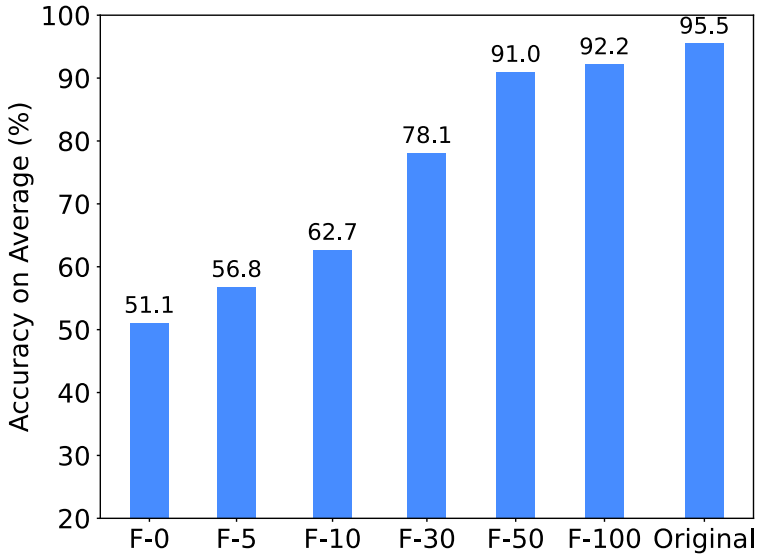


Figure 18: Average variable-aware accuracy of models fine-tuned with different number of logs in the target data set comparing with the original results in RQ2-A.

models without fine tuning (i.e., F-0), and the original results discussed in RQ2-A (i.e., Original). Overall, the average accuracy increases as the growth of the size of fine-tuning logs in the target data set, from 51.1% without fine-tuning logs to 92.2% with 100 fine-tuning logs. It is worth noting that the average variable-aware accuracy is fairly high when the models are fine-tuned with 50 logs in the target data set (i.e., 91.0% for F-50) and comparable to the original results discussed in RQ2-A.

In the last column of Table 19, we further present the detailed variable-aware accuracy of F-50 for each data set. We find that the fine-tuned models using 50 logs from the target data set (F-50) also achieve a high variable-aware accuracy with an average variable-aware accuracy of 91.0%, which is close to the average accuracy of the original results (i.e., 95.5%). Hence, after using the pre-trained models and a small data set of the target project, the models of VALB can be easily adopted to other projects.

VALB achieves a high variable-aware accuracy using the models fine-tuned with a small amount of data in the target system, and thus can be easily leveraged in a new project.

6.7 Discussion

Exploring the potential of variable-aware log abstraction on assisting in log-based downstream tasks. As discussed in Section 6.3, dynamic variables may also contain important information for log analysis tasks, and such information can be preserved using variable-aware log abstraction

of VALB. Hence, we further explore how can variable-aware log abstraction help the downstream tasks in log analysis. We conduct our exploration on the log-based anomaly detection benchmark provided by LogPAI [55, 56], which is widely used by other log-based anomaly detection studies [150, 162]. We use the HDFS data set provided by the benchmark to examine the performance of general log abstraction and variable-aware log abstraction on anomaly detection. HDFS data set contains over 11M log messages generated by running Hadoop-based MapReduce jobs on more than 2,000 Amazon's EC2 nodes for 38.7 hours. After grouping the logs with their block ID, there are 575,062 log sequences in total. Around 2.9% of the log sequences indicate anomalies, which are manually labeled by domain experts. We find that the logs in HDFS data set has four categories of dynamic variables: Object ID (OID), Location Indicator (LOI), Computing Resources (CRS), and Object Amount (OBA).

We use the anomaly detection techniques provided by the benchmark [55] with top-5 F1-scores (i.e., Decision Tree, SVM, LR, IM, and Clustering). For each technique, we use log sequences without dynamic variables as what prior log abstraction studies do (i.e., Original), with the value of each category of dynamic variable (i.e., OID, LOI, CRS, and OBA), and with all the values of dynamic variables (i.e., All) as the input data, respectively, to examine their performance on anomaly detection. Note that we further exclude the results of Decision Tree since the Precision, Recall, and F1-score are already nearly perfect (99.8%) for Original, and the results are very similar when using log sequences with each category of dynamic variables (from 99.7% to 99.9%).

Figure 19 presents the F1 scores achieved by each anomaly detection technique (excluding Decision Tree) using sequences of log templates without variables (i.e., Original), and using sequences of log templates with corresponding category of variables. As we find that the overall trends of Precision and Recall are similar to F1-score, we only present the results of F1-score to have a more concise view. Overall, we find that CRS (i.e., when using the log sequences with dynamic variables of which the category is Computing Resources) achieves the highest F1-score for each of the anomaly detection technique. For other category of variables, there is a fluctuation on the results compared with Original. In short, we find that log sequences with specific categories of dynamic variables (e.g., CRS in this experiment) can improve the performance of log-based anomaly detection.

Apart from anomaly detection, some participants in our survey (as discussed in Section 6.4) also mention some scenarios that dynamic variables in logs can assist in different tasks. For example, one participant comments that:

"Dynamic variables in the log are very important for parameter tuning works. Especially when the number of parameters is large, using dynamic variables in logs can help to track the performance of each parameter and easy to repeat the best performance."

The participant mentions that dynamic variables that record the hyper-parameters (e.g., the

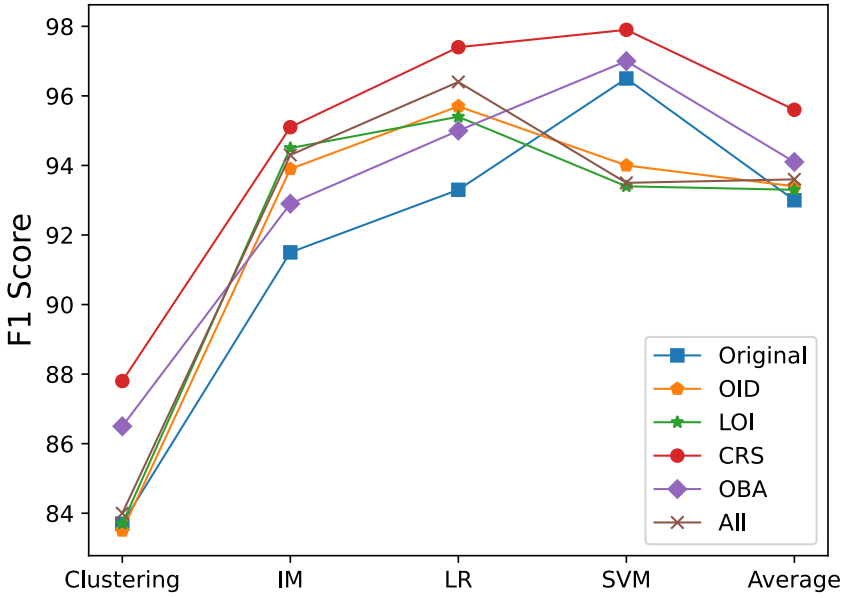


Figure 19: F1 score achieved by different anomaly detection techniques using sequences of log templates without variables (Original), and using sequences of log templates with corresponding category of variables.

number of epochs can be represented by the category of *Object Amount* in Section 6.4) can assist in parameter tuning works. Moreover, one participant also mentions that:

"Some types of variable can be very important for trouble shooting, like the status code. However, it's time-consuming to design regular expressions to grep such variables in each case. It will be helpful to identify such variables without ad-hoc efforts every time. "

Overall, practitioners acknowledge the importance of dynamic variables in practice, and such importance usually depends on the specific tasks. Our study explores the potential of variable-aware log abstraction on assisting in log analysis and sheds light on better leveraging the information in dynamic variables to improve log analysis for future studies.

6.8 Threats to Validity

Construct Validity. Our approach is based on supervised deep learning, the process of annotation on training and validation data may require extra effort in practice. However, as we discussed in Section 6.6, our approach can achieve promising results when training on small data sets and test on large data sets. Moreover, as we discussed in Section 6.7, our approach can also achieve encouraging results on the model trained from other projects and fine-tuned on a very small data set (e.g., 50 logs) of the target project. Hence, developers may not need significant time on manually labeling

the data.

External Validity. We conduct our study on open source log data sets provided by *LogPAI* [56] project. Conducting the study on different log data sets may have different results. For example, new categories of dynamic variables may be derived from other data sets. However, the data sets in *LogPAI* are across various domains and are widely studied by prior log-related studies [33, 162, 54, 150]. Moreover, the categories of dynamic variables are flexible to be updated by leveraging developers' data annotations.

6.9 Conclusion

Log abstraction is an important first step for automated log analysis. Prior log abstraction studies aim to completely abstract the dynamic variables in logs, without considering the great values that dynamic variables may have. Through an empirical study and a survey with industrial practitioners, we find that different categories of the dynamic variables in logs can be important for different tasks, and the distinction of dynamic variables in the process of log abstraction may further help log analysis. We then propose a deep learning based approach, VALB, which can further identify the category of dynamic variables in the process of log abstraction. VALB outperforms state-of-the-art log abstraction techniques on general log abstraction, and also achieves promising results on variable-aware log abstraction. Future studies may investigate the relationship between different categories of dynamic variables and their role in different tasks, in order to better leverage the information recorded in the dynamic variables and further help log analysis.

Part IV

Conclusion and Future Work

Chapter 7

Conclusion and Future Work

In this chapter, we summarize the contributions of this thesis as conclusions and discuss the potential directions of future work.

7.1 Thesis Contribution

Due to the lack of practical guidelines on logging and log analysis, the process of making logging decisions and log analysis remain challenging. In this thesis, we conduct studies to address the problem and help developers on logging practices in two aspects: 1) assist in making logging decisions, and 2) assist in log analysis.

For *assist in making logging decisions*, we have three research outcomes to improve the logging practice including suggesting logging locations, recommending log levels, and provide a systematic comprehension of the readability of log messages. We achieve promising results in the suggestions of logging locations and verbosity levels over the baseline approaches. The findings of our study on log messages provide a systematic understanding of the readability of log messages.

For *assist in log analysis*, we have one research outcome to study the dynamic variables in the process of log abstraction, which is usually omitted by prior log abstraction studies. We find that different dynamic variables may also be important based on the tasks and needs and explore the potential of a variable-aware log abstraction. . We propose a deep learning based approach that can identify the category of dynamic variables in the process of log abstraction. Our approach outperforms state-of-the-art log abstraction techniques on general log abstraction, and also achieves promising results on variable-aware log abstraction.

In conclusion, we summarize the contributions of this thesis as follows:

- We empirically study the characteristics of logging locations at the block level and propose

a deep learning based approach to suggest whether a code block needs to insert a logging statement (Chapter 3).

- We propose an automated deep-learning based approach that leverages the ordinal nature of log levels to make suggestions on choosing log levels. Our approach outperforms the existing state-of-the-art approaches in suggesting log levels (Chapter 4).

- We are the first study that investigates the readability of log messages by conducting interviews with industrial practitioners. We derive three aspects that are related to the readability of log messages and several corresponding practices to improve the readability for each aspect (Chapter 5).

- We investigate the characteristics and importance of dynamic variables in logs, which are omitted by prior log abstraction techniques. We find that different categories of dynamic variables record valuable information that can be important for different tasks, point out the need of a variable-aware log abstraction technique. We then propose a deep learning based log abstraction approach that can further identify the categories of dynamic variables in the process of log abstraction (i.e., variable-aware log abstraction). Our approach achieves promising results in variable-aware log abstraction and also outperforms prior state-of-the-art techniques in general log abstraction (Chapter 6).

7.2 Future work

This thesis makes an important step towards the goal of improving logging practices and log analysis. There are still many open challenges and research opportunities that may complement this thesis and further provide a logging guideline for developers to improve the quality of logging code. We discuss some potential directions for future work.

Providing Complete Logging Suggestions. Prior studies on improving logging practices mainly focus on one aspect of the logging challenges (i.e., *where-to-log*, *what-to-log*, and *how-to-log*) and provide partial supports for developers. In this thesis, we also explore the potential of providing automated supports for practitioners by conducting several studies on particularly improving one of the aspects above. In the future, we may investigate the potential of a complete logging suggestion technique, including the logging location, the messages, as well as the log level.

Providing Benchmarks for Techniques on Improving Logging Practices. Prior studies provide benchmarks for log abstraction and anomaly detection. For example, the LogPAI project shares 16 data sets of logs generated from both open-source and commercial systems across various domains, each data set also provides a subset of 2,000 logs with manually derived log templates

as ground truth for evaluating the accuracy of log abstraction techniques [169]. Moreover, LogPAI provides benchmarks for evaluating log-based anomaly detection techniques [55, 56], including two data sets and the implementations of several anomaly detection techniques. In future studies, we may provide benchmarks for the techniques on improving logging practices, such as suggesting logging locations, recommending log levels, and even complete logging statements.

Examining the Quality of Logging in Large Scale Software Systems. In this thesis, we take a preliminary step on investigating the readability of log messages in nine large-scale software systems and we find that a non-negligible portion of the log messages may not have adequate readability. Given that the effectiveness of models for automated suggestions and recommendations rely on the quality of training data, in the future, we may further investigate the general quality of existing logging code in large-scale software systems and study its impact on the effectiveness of the trained models. If the models are trained from data with mostly "good" data, then the models may very likely provide "good" suggestions as well. Therefore, we can further study how to filter and curate the training data to have more reliable and effective models.

Providing Automated Supports to Other Tasks of Log Analysis. In our thesis, we study dynamic variables in the process of log abstraction and find that variable-aware log abstractions can help the tasks of log-based anomaly detection. Logs are widely used in a various of tasks throughout the stages of software development and maintenance, including debugging, testing, requirement verification, performance tracking, and program comprehension. In future studies, we may seek to provide supports for other log-based tasks in practice such as failure diagnosis and log searching.

Bibliography

[1] Corpus of contemporary american english. https://www.english-corpora.org/coca/. Last checked Aug. 2020.

[2] gensim Word2vec embeddings. https://radimrehurek.com/gensim/models/word2vec.html. Last checked Feb. 2020.

[3] Keras: The python deep learning library. https://keras.io/. Last checked Aug. 2022.

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

[5] Oracle java documentation. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html. Last checked Mar. 2020.

[6] scikit-learn: Machine learning in python. https://scikit-learn.org. Last checked Aug. 2022.

[7] Simple logging facade for Java (SLF4J). http://www.slf4j.org. Last checked Feb. 2018.

[8] Simple logging facade for java (slf4j). http://www.slf4j.org/faq.html. Last checked Aug. 2020.

[9] Tensorflow: An end-to-end open source machine learning platform. https://www.tensorflow.org/. Last checked Aug. 2021.

[10] Reem Alsuhaibani, Christian Newman, Michael Decker, Michael Collard, and Jonathan Maletic. On the naming of methods: A survey of professional developers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 587–599, 2021.

[11] Harald Altinger, Steffen Herbold, Friederike Schneemann, Jens Grabowski, and Franz Wotawa. Performance tuning for automotive software fault prediction. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 526–530, 2017.

[12] Apache. log4j2. `https://logging.apache.org/log4j/2.x/manual/messages.html`, 2022. Last accessed August 2022.

[13] S. Boslaugh and P.A. Watters. *Statistics in a Nutshell: A Desktop Quick Reference*. In a Nutshell (O'Reilly). O'Reilly Media, 2008.

[14] Geoffrey Bowker and Susan Leigh Star. Sorting things out. *Classification and its consequences*, 4, 1999.

[15] Leo Breiman. Random forests. *Machine learning*, pages 5–32, 2001.

[16] Jeanderson Cândido, Jan Haesen, Maurício Aniche, and Arie van Deursen. An exploratory study of log placement recommendation in an enterprise system. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 143–154, 2021.

[17] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation. *Empirical Software Engineering*, pages 330–374, 2017.

[18] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 71–81, 2017.

[19] Boyuan Chen and Zhen Ming (Jack) Jiang. Extracting and studying the logging-code-issue-introducing changes in java-based large-scale open source software systems. *Empirical Software Engineering*, 24(4):2285–2322, Aug 2019.

[20] Boyuan Chen and Zhen Ming (Jack) Jiang. Studying the use of java logging utilities in the wild. In *Proceedings of the 42nd International Conference on Software Engineering,*, ICSE 2020, pages 1–12, 2020.

[21] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 305–316, 2018.

[22] Jinfu Chen, Weiyi Shang, Ahmed E. Hassan, Yong Wang, and Jiangbin Lin. An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 669–681, 2019.

[23] Junjie Chen, Shu Zhang, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Yu Kang, Feng Gao, Zhangwei Xu, Yingnong Dang, et al. How incidental are the incidents? characterizing and prioritizing incidents for large-scale online service systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 373–384, 2020.

[24] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 666–677, 2016.

[25] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 243–252, 2017.

[26] Tse-Hsun Chen, S. W. Thomas, Meiyappan Nagappan, and A.E. Hassan. Explaining software defects using topic models. In *Proceedings of the 9th Working Conference on Mining Software Repositories*, MSR '12, 2012.

[27] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919, 2016.

[28] Tse-Hsun Chen, Shang Weiyi, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 1001–1012, 2014.

[29] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *CoRR*, abs/1901.01808, 2019.

[30] Jason PC Chiu and Eric Nichols. Named entity recognition with bidirectional lstm-cnns. *Transactions of the Association for Computational Linguistics*, pages 357–370, 2016.

[31] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald 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*, pages 393–403, 2015.

[32] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, pages 2493–2537, 2011.

[33] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering*, 2020.

[34] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, pages 248–255, 2009.

[35] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: are we there yet? In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*, pages 612–621, 2018.

[36] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 USENIX Annual Technical Conference, USENIX ATC '15,*, pages 139–150, 2015.

[37] Zishuo Ding, Heng Li, and Weiyi Shang. Logentext: Automatically generating logging texts using neural machine translation. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 349–360, 2022.

[38] Cicero Dos Santos and Bianca Zadrozny. Learning character-level representations for part-of-speech tagging. In *International Conference on Machine Learning*, pages 1818–1826. PMLR, 2014.

[39] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. A quantitative analysis framework for recurrent neural network. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 1062–1065, 2019.

[40] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 158–167, 2007.

[41] Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[42] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158, 2009.

[43] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE-SEIP '14, pages 24–33, 2014.

[44] Daniele Gadler, Michael Mairegger, Andrea Janes, and Barbara Russo. Mining logs to model the use of a system. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 334–343, 2017.

[45] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 57–76, 2007.

[46] X. Gu, H. Zhang, and S. Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering, ICSE 2018*, pages 933–944, 2018.

[47] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 810–822, 2019.

[48] Huong Ha and Hongyu Zhang. Deepperf: performance prediction for configurable software with deep sparse neural network. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, pages 1095–1106, 2019.

[49] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1573–1582.

[50] David J Hand and Robert J Till. A simple generalisation of the area under the ROC curve for multiple class classification problems. *Machine learning*, 45(2):171–186, 2001.

[51] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, 2018.

[52] Haibo He, Yang Bai, Edwardo A Garcia, and Shutao Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence)*, pages 1322–1328. IEEE, 2008.

[53] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 33rd IEEE international conference on Automated software engineering*, pages 1–11, 2018.

[54] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*, pages 33–40, 2017.

[55] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[56] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Loghub: A large collection of system log datasets towards automated log analytics. *CoRR*, 2020.

[57] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo. Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel. *IEEE Transactions on Software Engineering*, 2019.

[58] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019*, pages 34–45, 2019.

[59] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[60] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018*, pages 200–210, 2018.

[61] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Yingfei Xiong, and Zibin Zheng. Learning code context information to predict comment locations. *IEEE Trans. Reliability*, 69(1):88–105, 2020.

[62] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF models for sequence tagging. *CoRR*, abs/1508.01991, 2015.

[63] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 135–146, 2017.

[64] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *J. Softw. Maintenance Res. Pract.*, 20(4):249–267, 2008.

[65] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D. Syer, and Ahmed E. Hassan. Examining the stability of logging statements. *Empir. Softw. Eng.*, 23(1):290–333, 2018.

[66] Eirini Kalliamvakou, Christian Bird, Thomas Zimmermann, Andrew Begel, Robert DeLine, and Daniel M German. What makes a great manager of software engineers? *IEEE Transactions on Software Engineering*, pages 87–106, 2017.

[67] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[68] Sandeep Kaur Kuttal, Anita Sarma, and Gregg Rothermel. History repeats itself more easily when you log it: Versioning for mashups. In *2011 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, pages 69–72, 2011.

[69] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 628–639, 2019.

[70] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 282–289, 2001.

[71] Bei Li, Hui Liu, Ziyang Wang, Yufan Jiang, Tong Xiao, Jingbo Zhu, Tongran Liu, and Changliang Li. Does multi-encoder help? A case study on context-aware neural machine translation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 3512–3518, 2020.

[72] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. Studying software logging using topic models. *Empirical Software Engineering*, Jan 2018.

[73] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering*, pages 1–17, 2020.

[74] Heng Li, Weiyi Shang, and Ahmed E. Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716, Aug 2017.

[75] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4):1831–1865, 2017.

[76] Xiaochen Li, He Jiang, Yasutaka Kamei, and Xin Chen. Bridging semantic gaps between natural languages and apis with word embedding. *IEEE Transactions on Software Engineering*, 2020.

[77] Zhenhao Li. Characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019*, pages 147–149, 2019.

[78] Zhenhao Li. Studying and suggesting logging locations in code blocks. In *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume*, pages 125–127, 2020.

[79] Zhenhao Li. Towards providing automated supports to developers on writing logging statements. In *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume*, pages 198–201, 2020.

[80] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. Where shall we log? studying and suggesting logging locations in code blocks. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, pages 361–372, 2020.

[81] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. DLFinder: characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, pages 152–163, 2019.

[82] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. Studying duplicate logging statements and their relationships with code clones. *IEEE Transactions on Software Engineering*, pages 1–19, 2021.

[83] Zhenhao Li, Heng Li, Tse-Hsun Peter Chen, and Weiyi Shang. Deeplv: Suggesting log levels using ordinal based neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1461–1472. IEEE, 2021.

[84] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 102–111, 2016.

[85] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485, 2020.

[86] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. Tell: log level suggestions via modeling multi-level code block information. In Sukyoung Ryu and Yannis Smaragdakis,

editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 27–38, 2022.

[87] 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. Early Access.

[88] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela F. Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 241–254, 2018.

[89] Shiqing Ma, Xiangyu Zhang, Dongyan Xu, et al. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, volume 2, page 4, 2016.

[90] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264, 2009.

[91] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 117–126, 2008.

[92] Zainab Masood, Rashina Hoda, and Kelly Blincoe. What drives and sustains self-assignment in agile teams. *IEEE Transactions on Software Engineering*, 2021.

[93] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering*, 2022.

[94] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2279–2290, 2022.

[95] Alejandro Mazuera-Rozo, Anamaria Mojica-Hanke, Mario Linares-Vásquez, and Gabriele Bavota. Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 276–287. IEEE, 2021.

[96] Peter McCullagh. Regression models for ordinal data. *Journal of the Royal Statistical Society: Series B (Methodological)*, 42(2):109–127, 1980.

[97] Mary L. McHugh. Interrater reliability: the kappa statistic. *Biochemia Medica*, 22(3):276–282, 2012.

[98] Scott Menard. *Applied logistic regression analysis*. Number 106. Sage, 2002.

[99] Salma Messaoudi, Donghwan Shin, Annibale Panichella, Domenico Bianculli, and Lionel Briand. Log-based slicing for system-level test cases. In *2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[100] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013*, 2013.

[101] Masayoshi Mizutani. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing*, pages 595–602, 2013.

[102] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1287–1293, 2016.

[103] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 114–117, 2010.

[104] Meiyappan Nagappan, Kesheng Wu, and Mladen A. Vouk. Efficiently extracting operational profiles from execution logs using suffix arrays. In *ISSRE'09: Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*, pages 41–50, 2009.

[105] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 353–366, 2012.

[106] Bui D. Q. Nghi, Yijun Yu, and Lingxiao Jiang. Bilateral dependency neural networks for cross-language algorithm classification. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019*, pages 422–433, 2019.

[107] Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. Automatic fault detection for deep learning programs using graph transformations. *TOSEM*, 2021.

[108] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, February 2012.

[109] Jevgenija Pantiuchina, Bin Lin, Fiorella Zampetti, Massimiliano Di Penta, Michele Lanza, and Gabriele Bavota. Why do developers reject refactorings in open-source projects? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 1–23, 2021.

[110] Keyur Patel, João Faccin, Abdelwahab Hamou-Lhadj, and Ingrid Nunes. The sense of logging in the linux kernel. *Empirical Software Engineering*, pages 1–47, 2022.

[111] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[112] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009*, pages 87–102, 2009.

[113] Heidar Pirzadeh, Sara Shanian, Abdelwahab Hamou-Lhadj, and Ali Mehrabian. The concept of stratified sampling of execution traces. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011*, pages 225–226, 2011.

[114] John Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, pages 61–74, 1999.

[115] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[116] J. Ross Quinlan. Induction of decision trees. *Machine learning*, pages 81–106, 1986.

[117] Lance A Ramshaw and Mitchell P Marcus. Text chunking using transformation-based learning. In *Natural language processing using very large corpora*, pages 157–176. 1999.

[118] Barbara Russo, Giancarlo Succi, and Witold Pedrycz. Mining system logs to learn error predictors: a case study of a telemetry system. *Empirical Software Engineering*, pages 879–927, 2015.

[119] Daan Schipper, Maurício Finavaro Aniche, and Arie van Deursen. Tracing back log data to its log statement: from research to practice. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019*, pages 545–549, 2019.

[120] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, pages 2673–2681, 1997.

[121] Issam Sedki, Abdelwahab Hamou-Lhadj, Otmane Ait-Mohamed, and Mohammed A Shehab. An effective approach for parsing large log files. 2022.

[122] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.

[123] Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan, and Zhen Ming Jiang. Understanding log lines using development knowledge. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 21–30, 2014.

[124] Keiichi Shima. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213*, 2016.

[125] Donghwan Shin, Zanis Ali Khan, Domenico Bianculli, and Lionel Briand. A theoretical framework for understanding the relationship between log parsing and anomaly detection. In *The 21st International Conference on Runtime Verification*.

[126] Julilus Sim and Chris C. Wright. The kappa statistic in reliability studies: Use, interpretation, and sample size requirements. *Physical Therapy*, 85(3):257–268, March 2005.

[127] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[128] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014.

[129] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Why my app crashes understanding and benchmarking framework-specific exceptions of android apps. *IEEE Transactions on Software Engineering*, 2020.

[130] Liang Tang, Tao Li, and Chang-Shing Perng. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 785–794.

[131] Jörg Tiedemann and Yves Scherrer. Neural machine translation with extended context. In *Proceedings of the Third Workshop on Discourse in Machine Translation, DiscoMT@EMNLP 2017, Copenhagen, Denmark, September 8, 2017*, pages 82–92, 2017.

[132] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, pages 25–36, 2019.

[133] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 542–553, 2018.

[134] Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *J. Artif. Intell. Res.*, 37:141–188, 2010.

[135] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*, pages 119–126, 2003.

[136] Risto Vaarandi and Mauno Pihelgas. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*, pages 1–7, 2015.

[137] Harold Valdivia Garcia and Emad Shihab. Characterizing and predicting blocking bugs in open source projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 72–81, 2014.

[138] Mario Linares Vásquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. On using machine learning to automatically classify software applications into domain categories. *Empir. Softw. Eng.*, 19(3):582–618, 2014.

[139] Zhiyuan Wan, Xin Xia, David Lo, and Gail C Murphy. How does machine learning change software development practices? *IEEE Transactions on Software Engineering*, pages 1857–1871, 2019.

[140] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Trans. Softw. Eng. Methodol.*, 30(4):56:1–56:30, 2021.

[141] Mohammad Wardat, Wei Le, and Hridesh Rajan. Deeplocalize: Fault localization for deep neural networks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 251–262. IEEE, 2021.

[142] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 1–58, 2022.

[143] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 262–273, 2016.

[144] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*, pages 326–337, 2019.

[145] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Xiaohu Yang. Elblocker: Predicting blocking bugs with ensemble imbalance learning. *Information & Software Technology*, 61:93–106, 2015.

[146] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016*, pages 29:1–29:10, 2016.

[147] Xin Xia, Zhiyuan Wan, Pavneet Singh Kochhar, and David Lo. How practitioners perceive coding proficiency. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 924–935, 2019.

[148] Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, and Xiaohong Li. Automatic loop summarization via path dependency analysis. *IEEE Trans. Software Eng.*, 45(6):537–557, 2019.

[149] Yan Xu, Lili Mou, Ge Li, Yunchuan Chen, Hao Peng, and Zhi Jin. Classifying relations via long short term memory networks along shortest dependency paths. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015*, pages 1785–1794, 2015.

[150] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. Plelog: Semi-supervised log-based anomaly detection via probabilistic label estimation. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*, pages 230–231, 2021.

[151] Nan Yang, Pieter J. L. Cuijpers, Ramon R. H. Schiffelers, Johan Lukkien, and Alexander Serebrenik. An interview study of how developers use execution logs in embedded software engineering. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 61–70, 2021.

[152] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: suggesting and updating logging locations for web-based systems performance monitoring. *Empirical Software Engineering*, 25(1), 2020.

[153] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 21–30, 2018.

[154] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3320–3328, 2014.

[155] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, 2014.

[156] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 143–154, 2010.

[157] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering*, pages 102–112, 2012.

[158] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *ASPLOS '11: Proceedings of the 16th international conference on Architectural support for programming languages and operating systems*, pages 3–14. ACM, 2011.

[159] Yi Zeng, Jinfu Chen, Weiyi iShang, and Tse-Hsun (Peter) Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, pages 1–41, 2019.

[160] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, pages 783–794, 2019.

[161] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael R. Lyu, and Miryung Kim. An empirical study of common challenges in developing deep learning applications. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019*, pages 104–115, 2019.

[162] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 807–817, 2019.

[163] Xu Zhang, Yong Xu, Si Qin, Shilin He, Bo Qiao, Ze Li, Hongyu Zhang, Xukun Li, Yingnong Dang, Qingwei Lin, Murali Chintalapati, Saravanakumar Rajmohan, and Dongmei Zhang. Onion: identifying incident-indicating logs for cloud systems. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1253–1263.

[164] Nengwen Zhao, Honglin Wang, Zeyan Li, Xiao Peng, Gang Wang, Zhu Pan, Yong Wu, Zhen Feng, Xidao Wen, Wenchi Zhang, Kaixin Sui, and Dan Pei. An empirical investigation of practical log anomaly detection for online service systems. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1404–1415, 2021.

[165] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 565–581, 2017.

[166] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. An exploratory study of logging configuration practice in java. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pages 459–469, 2019.

[167] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*, pages 683–694, 2019.

[168] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 415–425, 2015.

[169] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. Tools and benchmarks for automated log parsing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 121–130. IEEE / ACM, 2019.