Log-driven Automated Software Reliability Engineering

HUO, Yintong

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
July 2024

Thesis Assessment Committee

Professor MENG Wei (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor KING Kuo Chin Irwin (Committee Member)

Professor BRIAND Lionel (External Examiner)

Abstract of thesis entitled:

Log-driven Automated Software Reliability Engineering

Submitted by HUO, Yintong

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in July 2024

Modern software systems are integral to our daily lives, powering search engines, cloud services, and operating systems. However, these systems are susceptible to failures, such as node failure, network issues, and storage failure, leading to significant financial losses and damage to customer trust. As software systems grow increasingly complex, traditional rule-based methods for ensuring software reliability become impractical due to the massive volumes of related data and the intricate correlations between different types of failures.

To monitor, diagnose, and analyze these complex software failures, a variety of monitoring data is gathered to reflect the system's runtime status. Among all these data types, logs are particularly critical for several reasons. First, logs encapsulate the intentions of developers by documenting crucial runtime events and behaviors. Second, logs often serve as the only available data that accurately records the system's runtime status,

making them indispensable for diagnosis and monitoring. Third, logs are the bridges between development and operational teams by executing the logging statements, facilitating issue-handling processes in the operation period. Given the importance of logs, our research focuses on leveraging them to enhance software reliability engineering as follows.

First, we introduce SemParser, the first semantic-aware log parser that effectively extracts meaningful events and runtime parameters from logs. Our framework integrates an end-to-end semantics miner, which extracts explicit semantics from individual log messages, with a joint parser that uses domain knowledge to infer implicit semantics and combine them into comprehensive parameter semantics. This method significantly enhances downstream tasks like anomaly detection and failure diagnosis by incorporating semantic information often overlooked by traditional parsers.

Second, we develop EvLog, an anomalous log localization framework designed to remain effective despite software evolution. EvLog comprises a multi-level representation extractor and an anomaly discriminator augmented with attention mechanisms to pinpoint anomalous logs in an unsupervised manner. By addressing issues like evolving log events and unstable log sequences, EvLog reduces the need for frequent retraining, thereby maintaining high anomaly detection accuracy over time.

Third, to tackle the limitation of insufficient datasets, we propose AutoLog, a framework that synthesizes comprehensive

log sequences based on program analysis. AutoLog generates log sequences by analyzing program execution paths related to logging statements. It builds comprehensive call graphs, prunes them to identify scalable log-related execution paths, and propagates expert labels to generate flexible log sequences across methods based on their calling relationships. AutoLog improves anomaly detection capabilities by generating diverse and high-coverage log data, which are critical for training robust machine learning models.

Last but not least, we conduct an extensive evaluation of large language models (LLMs) for generating logging statements. The study involves five research questions from two perspectives: (1) effectiveness: how do LLMs perform in logging practice? and (2) generalizability: how well do LLMs generate logging statements for unseen code? Using our LogBench dataset, which includes both seen and unseen code after transformation, we draw eight findings, five implications, and benchmarks for future research on LLM-powered logging tools, highlighting their potential and current limitations.

In summary, this thesis aims to develop automated log analysis techniques to address challenges associated with high log variety, rapid software evolution, insufficient datasets, and inaccurate logging statements. Extensive experiments on public datasets demonstrate the effectiveness and efficiency of our proposed algorithms. Collectively, these techniques make significant contributions to the field of software reliability engineering.

論文題目：日誌驅動的自動化軟件工程

作者　　：霍茵桐

學校　　：香港中文大學

學系　　：計算機科學與工程學系

修讀學位：哲學博士

摘要　　：

現代軟件系統在我們的日常生活中扮演著不可或缺的角色，驅動著搜尋引擎、雲端服務和操作系統。然而，這些系統容易遭遇異常，例如節點故障、網路問題和存儲故障，導致重大的財務損失和客戶信任度受損。隨著軟件系統日益複雜，傳統的基於規則的方法，由於相關數據量巨大和不同類型故障之間的複雜關聯性，已經變得不切實際。為了監控、診斷和分析這些複雜的軟件故障，各種監控數據被收集起來以反映系統的運行狀態。在所有類型的數據中，日誌因以下幾個關鍵原因扮演著至關重要的角色。首先，日誌通過記錄關鍵的運行時事件和行為，封裝了開發者的意圖。其次，日誌通常是唯一準確記錄系統運行狀態的數據，對於診斷和監控來說不可或缺。日誌通過執行日誌語句，在開發和運營團隊之間架起溝通的橋樑，有助於更快地解決系統運行故障。鑒於日誌的重要性，我們的研究重點在於利用日誌來增強軟件可靠性工程，如下所述。

首先，我們介紹了 SemParser，這是第一個語義感知日誌解析器，能夠有效地從日誌中提取有意義的事件和運行時參數。我們的框架整合了端到端的語義挖掘器，從單個日誌訊息中提取顯式語義，並通過使用領域知識的聯合解析器推導隱式語義並將其整合為全面的參數語義。這種方法顯著改善了如異常檢測和故障診斷等下游任務，因為它利用了傳統解析器常常忽視

的語義信息。

其次，我們開發了 EvLog，一個設計用來在軟件演變過程中仍能保持有效的異常日誌定位框架。EvLog 包括一個多層次代表提取器和一個增加了注意力機制的異常識別器，能在無監督的情況下準確定位異常日誌。通過解決如日誌事件變化和日誌序列不穩定等問題，EvLog 減少了頻繁重訓的需求，從而在異常檢測準確性方面保持高效。

第三，為了解決數據集不足的限制，我們提出了 AutoLog，一個基於程序分析的綜合日誌序列生成框架。AutoLog 透過分析與日誌記錄語句相關的程序執行路徑來生成日誌序列。它構建了綜合的呼叫圖，並對其進行修剪以識別可擴展的日誌相關執行路徑，並傳播專家標籤以根據其調用關係生成靈活的日誌序列。AutoLog 通過生成多樣化和高覆蓋率的日誌數據，提高了異常檢測能力，這對於訓練穩健的機器學習模型至關重要。

最後,我們對大型語言模型（LLMs）在生成日誌語句方面進行了廣泛的評估。該研究從兩個角度探討了五個研究問題：（1）有效性：LLMs 在日誌實踐中的表現如何？（2）普適性：LLMs 在對未見代碼生成日誌語句方面的表現如何？通過使用我們的 LogBench 數據集，包括轉換後的已見和未見代碼，我們得出了八個結論、五個啟示和未來研究 LLM 驅動的日誌工具的基準，強調了其潛力和現有的局限性，指名了未來改進方向。

總之，本論文旨在開發自動化的日誌分析技術，以應對日誌多樣性、快速軟件演變、數據集不足和日誌語句不準確等挑戰。基於公開數據集進行的廣泛實驗證明了我們所提出算法的有效性和高效性。綜合來看，這些技術在軟件可靠性工程領域做出了重大貢獻。

Acknowledgement

Embarking on my Ph.D. journey at the Chinese University of Hong Kong has been the most transformative and fulfilling chapter of my life. These past four years have been marked by intense challenges, but also profound joy, exhilaration, and a deep sense of accomplishment as I explored the complexities of research.

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Michael R. Lyu, for the unwavering support of my Ph.D study and research. His patience, boundless knowledge, and visionary guidance have been the cornerstone of my Ph.D. experience. Beyond his academic mentorship, his generosity and compassion have profoundly influenced my personal growth and will remain with me throughout my life. I couldn't have asked for a more exemplary advisor.

Second, I am also profoundly grateful to my thesis assessment committee members (sorted by name), Prof. Irwin King, and Prof. Wei Meng. Their insightful comments and invaluable suggestions have significantly enhanced the quality of this thesis and all my term reports. Moreover, a special thanks to

Prof. Lionel C. Briand from the University of Ottawa, whose role as the external examiner brought insightful perspectives and constructive feedback that enriched my work.

Throughout this journey, I have never felt alone. I owe a heartfelt thank you to my mentors (sorted by name): Prof. Pinjia He, and Prof. Yuxin Su, whose expertise and guidance helped me grow fast into a mature researcher. I also extend my gratitude to Prof. David Lo and Prof. Karthik Pattabiraman for their support and career advice, which have been instrumental in shaping my professional path.

I could not forget all the help and encouragement I received from my collaborators (sorted by name): Dr. Jiazhen Gu, Mr. Zhihan Jiang, Ms. Baitong Li, Mr. Yichen Li, Mr. Jinyang Liu, Ms. Shiwen Shan, Mr. Zifan Xie, Dr. Hongming Zhang, and Mr. Renyi Zhong. Your contributions have been invaluable.

In the end, my deepest gratitude goes to my family for their unreserved love. This thesis would not have been possible without your constant support. Thank you for being my greatest source of strength and inspiration.

This thesis is dedicated to my beloved family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Modern software systems play a crucial role in our daily lives, including search engines, cloud services, and operating systems. However, these systems can experience failures, such as service disruptions, which can lead to financial losses and damage customer trust. For instance, in October 2021, Facebook and its subsidiaries were disrupted globally for six to seven hours, causing a nearly 5% drop in the company' s stock value and at least 60 million US Dollars of loss in revenue[1].

To address these issues, software reliability engineering (SRE) is essential. SRE focuses on developing and maintaining software systems to ensure they operate without failures for a specified period [1]. Traditional SRE techniques involve rule-based methods that require significant expert effort to analyze failures. As software systems become more complex, with highly corre-

[1]https://en.wikipedia.org/wiki/2021_Facebook_outage.

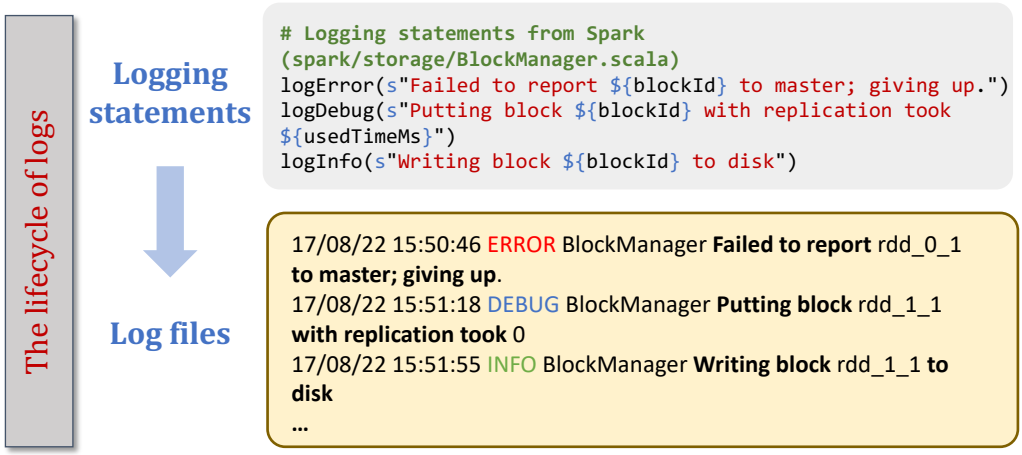


Figure 1.1: The lifecycle of logs.

lated failures and large volumes of monitoring data, automated approaches are needed for effective software development and operations.

Software logs are extensively utilized in various reliability assurance tasks for several key reasons. Firstly, they encapsulate the code intentions as written by the programmers. Secondly, they often provide the only available data that records software runtime information [2]. Thirdly, logs serve as a bridge between software development and operational engineers by appending log files from logging statements at runtime. Furthermore, logs are crucial in data-driven decision-making within the industry [3]. By analyzing these logs, system operators and administrators can monitor software status [4], detect anomalies [5, 6], and troubleshoot system issues [7].

Typically, logs are semi-structured texts generated by logging statements (e.g., `logger.info()`) in the source code. Fig-

ure 1.1 illustrates the lifecycle of logs containing two stage: logging statements and log files. In this figure, the first-line log messages are generated by the logging statement `logError(``Failed to report $blockId to master; giving up.'')`. The log event (e.g., "failed to report ... to master; giving up") is an unstructured statement crafted by developers to describe specific runtime behavior. Conversely, the remaining parameters (e.g., rdd_0_1) are system variables automatically generated during runtime.

Handling logs in modern software systems is challenging, primarily due to the massive amount of log files generated. In traditional standalone systems, engineers could manually inspect logs or create rules to detect failures using their expertise, often relying on keyword searches (e.g., "error") or regular expression matching [8]. Nevertheless, this method becomes impractical with the current scale of log production, where systems can generate about 50 gigabytes of logs per hour, amounting to roughly 120-200 million lines [9]. Manually extracting essential information from such a vast volume of logs is nearly impossible, eliciting the need for automated solutions. In addition to volume, log-based software reliability engineering faces several other challenges.

- **High variety.** Although the number of log events in a system is limited, these events can generate millions of different log messages based on their auto-generated param-

eters. For instance, a dataset [10] with 30 log events can include more than 11 million different log messages. This variety complicates the design of rules to filter out noisy logs, creating a significant barrier to identifying key log messages.

- **Fast evolution.** After the initial release, software undergoes continuous development to meet customer demands, fix bugs, and add new functionalities. During this evolution, logging statements can be revised and changed, resulting in evolving log events. Such evolution introduces challenges for log analysis by introducing unseen log events, modifying the order of log sequences, and altering event distribution.

- **Insufficient datasets.** Modern data-driven models require ample data for training and testing against, of course, log analysis models are no exception. However, existing datasets are often collected from specific workloads, covering only limited system behavior patterns, which creates a gap with the complex activities of real-world systems. Additionally, logs often contain sensitive information, making service providers reluctant to publicize them.

- **Inaccurate logging statements.** Effective log analysis relies on high-quality logging statements, but existing automated logging statement generation frameworks are

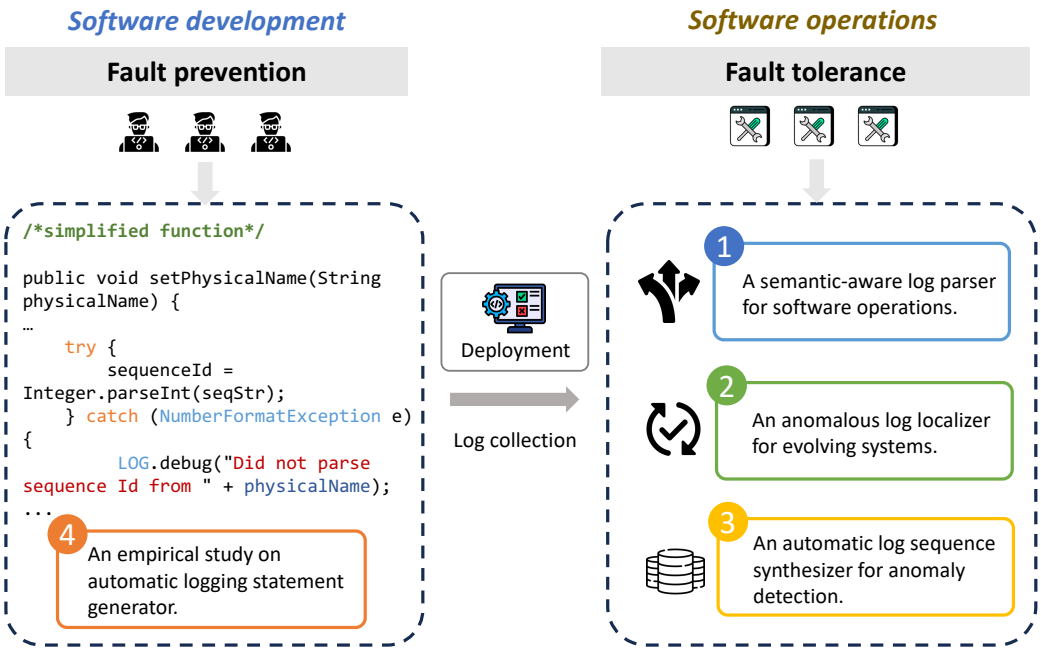


Figure 1.2: Log-driven automated software reliability engineering studies.

far from satisfactory. Only when log messages accurately record anomaly patterns and failure details can engineers discern discrepancies between normal and abnormal logs, thereby mitigating issues.

In response to these challenges, our research focuses on log-driven intelligent software reliability engineering. The challenges we address include high variety, rapid evolution, insufficient datasets, and inaccurate logging statements, which span the two stages of logs: logging statements and log files. Tackling these challenges ensures software reliability from different angles, each aligning with a traditional Software Reliability Engineering (SRE) solution. Firstly, improving the quality of logging statements during the software development phase aligns with

fault prevention in traditional SRE. Secondly, enabling effective log analysis during the software operation phase corresponds to fault tolerance in traditional SRE.

Figure 1.2 outlines the structure of log-driven automated software engineering studies in this thesis, encompassing both proactive logging statement generation during software development and reactive log analysis during software operations. The figure lays out our solutions, which are divided into four parts. First, we introduce the first semantic-aware log parser for software operations. This parser not only extracts log events but also emphasizes the significance of runtime parameters in log messages, examining their role in software failure analysis. Second, we develop an anomalous log localizer that automatically identifies anomaly logs during software runtime. This algorithm remains effective throughout software evolution, reducing the training cost for frequently updated software. Third, we introduce AutoLog, a log sequence synthesizer designed to generate log sequences that supplement existing log-based anomaly detection datasets. This approach addresses the challenge of insufficient training log data by actively generating log sequences that cover a wide range of system behaviors, based on program analysis. Finally, we conduct an empirical study on LLM-based logging statement generation models. We focus on two critical evaluation aspects: the effectiveness of log generation and the generalizability of the models to unseen code. This study identifies key factors influencing model performance, providing

insights for future researchers to develop large language models (LLMs) as intelligent logging assistants.

1.2 Thesis Contributions

In this thesis, we make the following contributions toward log-centered software reliability engineering.

1. **A semantic-aware log parser for software operations.**

   Before obtaining more insights about the run-time status of the software, a fundamental step of log analysis, called log parsing, is employed to extract structured templates and parameters from the semi-structured raw log messages. However, existing log parsers treat each message as a mere character string, overlooking the semantic information embedded in the parameters and templates. To address this, we introduce SemParser [11], which consists of two main components: a semantic miner and a joint parser. To evaluate the effectiveness of our semantic parser, we first demonstrate its ability to extract high-quality semantics from log messages collected from seven widely used systems. Next, we perform two representative downstream tasks, showing that current techniques enhance their performance in anomaly detection and failure diagnosis when they utilize appropriately extracted semantics.

2. **An anomalous log localizer over software evolution.** Given that software can generate a vast amount of logs daily, automatically distinguishing anomalous logs from normal ones is a critical challenge for engineers. While existing methods reduce the burden on software maintainers, they rely on a flawed yet crucial assumption: that logging statements remain unchanged. However, as software evolves, our empirical study identifies three resulting challenges from software evolution: log parsing errors, evolving log events, and unstable log sequences. In this chapter, we introduce a novel unsupervised approach called Evolving Log Analyzer (EvLog) [12] to address these issues. EvLog has demonstrated its effectiveness on two real-world system evolution log datasets, achieving average F1 scores of 0.955 and 0.847 in intra-version and inter-version settings, respectively. This performance surpasses that of other state-of-the-art methods by a wide margin.

3. **An automatic log sequence synthesizer.** Although existing log datasets are available for anomaly detection, they have limitations regarding the comprehensiveness of log events, scalability across diverse systems, and flexibility in log utility. To overcome these limitations, we introduce AutoLog [13], the first automated log synthesis methodology for anomaly detection, leveraging program analysis. AutoLog begins by identifying compre-

hensive logging statements associated with an application's call graphs. It then constructs execution graphs for each method, pruning the call graphs to identify log-related execution paths in a scalable manner. Finally, AutoLog assigns anomaly labels to each identified execution path based on expert knowledge. Experiments conducted on 50 popular Java projects reveal that AutoLog captures significantly more (*9x-58x*) log events than existing log datasets from the same systems and generates log messages much faster (*15x*) using a single machine compared to existing passive data collection methods. Furthermore, we demonstrate AutoLog's practicality by showing that it enables log-based anomaly detectors to achieve better performance (*1.93%*) compared to using existing log datasets.

4. **An empirical study on LLM-based logging statement generation.**

   Automated logging statement generation supports developers in documenting critical software runtime behavior. With the significant advancements in natural language generation and programming language comprehension, LLMs hold the potential for generating logging statements, yet this area remains unexplored. To fill this gap, this chapter presents the first study exploring LLMs for logging statement generation. We build a dataset, *LogBench*, consisting of two parts: (1) LogBench-O, containing 3,870 methods

with 6,849 logging statements collected from GitHub repositories, and (2) LogBench-T, featuring transformed unseen code from LogBench-O. Using LogBench, we evaluate the effectiveness and generalization capabilities of eleven top-performing LLMs, ranging from 60M to 175B parameters. Our study yields eight findings and five implications, offering practical advice for future logging research. Our empirical analysis highlights the limitations of current logging approaches, demonstrates the potential of LLM-based logging tools, and provides actionable guidance for developing more effective models.

1.3 Thesis Organization

The remainder of this thesis is organized as follows.

- **Chapter 2: Background review**
  In this chapter, we review background knowledge and related work on the log lifecycle in system reliability engineering, encompassing automated log analysis and logging statement generation. First, we provide a brief introduction to logs as system runtime data, highlighting their role in performance monitoring and software operations. We then delve into the fundamental step of log parsing for log analysis and examine related work in Section 2.2. Following this, we discuss the widely studied task of log-based

anomaly detection in Section 2.3. Finally, we explore approaches for generating logging statements automatically based on code context in Section 2.4.

- **Chapter 3: Semantic-aware log parser**
  Log parsing is essential for automated log analysis as it converts unstructured log messages into structured log events. Chapter 3 addresses the limitations of existing log parsers, such as their lack of semantic awareness and the potential negative impact on subsequent log analysis. To overcome these limitations, we introduce SemParser, the first semantic-aware log parsing framework. Specifically, Section 3.1 covers the basics of log parsing and our motivation for incorporating semantic parsing. Sections 3.2 and 3.3 present the problem definition and our methodology, respectively. The evaluation results are detailed in Section 3.5, followed by a discussion of threats to validity in Section 3.6. Finally, we summarize the chapter in Section 3.7.

- **Chapter 4: Anomalous log localizer for software evolution**
  This chapter introduces EvLog, an anomalous log localization framework designed for evolving software. The core technique of EvLog is an event alignment mechanism that encodes paraphrased log messages similarly to the origi-

nal ones. By representing each log in this manner, they can be compared with normal patterns to detect deviations (anomalies). Specifically, Sections 4.1 and 4.2 discuss the motivation for developing a log localizer for evolving software systems. Section 4.3 defines the problem, while Section 4.4 outlines the design of EvLog. We evaluate its performance in Section 4.6 and provide a case study in Section 4.7. Finally, we conclude the chapter in Section 4.8.

- **Chapter 5: Log sequence synthesis for anomaly detection**

  This chapter presents a log sequence synthesis framework for log-based anomaly detection. The comprehensiveness of the log dataset has been a gap between the lab-collected dataset and the data from real-world complicated software. To this end, we develop the first log sequence synthesis framework, namely AutoLog, that could actively analyze the program and generate simulated log sequences without executing the program. In particular, Section 5.1 and Section 5.2 illustrate the motivation of synthesizing log sequences for anomaly detection. Section 5.3 includes the framework design of AutoLog. We present the evaluation of AutoLog in Section 5.5 and discussion in Section 5.6. Lastly, we summarize this chapter in Section 5.8.

- **Chapter 6: Evaluation study on LLM-powered logging statement generation**

This chapter presents an empirical study on the performance of LLMs in logging statement generation tasks, including their ability to predict log levels, log variables, and log texts. We collect the benchmark named LogBench, including one dataset LogBench-O for evaluating LLMs' effectiveness and another dataset LogBench-T for evaluating their generalizability on unseen code. In particular, Section 6.1 illustrates our motivation for this empirical study. Section 6.2 reviews the existing approaches for generating logging statements. Section 6.3 describes how we develop the benchmark LogBench. Upon the experiments, we draw eight findings in Section 6.4 and five implications in Section 6.5 for future researchers and developers. Section 6.7 concludes this chapter in the end.

- **Chapter 7: Conclusion and Future Work**
  The final chapter summarizes this thesis in Section 7.1 and outlines three future research directions in Section 7.2.

To ensure each chapter is self-contained, we may briefly reiterate key aspects such as basic concepts, problem motivations, and model definitions in various chapters.

□ **End of chapter.**

Chapter 2

Background Review

This chapter offers a concise overview of foundational concepts related to log utility in both runtime operations and software development, which underpin our research. Initially, we discuss system runtime data with a particular emphasis on system logs. The subsequent part is further divided into three main subsections, each addressing a specific problem examined in this thesis. These are: (1) Log parsing (Section 2.2), (2) Log-based anomaly detection (Section 2.3), (3) logging statement generation (Section 2.4). Each subsection begins with a description of the relevant research field, followed by a problem description and a review of existing literature. Figure 2.1 depicts the taxonomy of prior research works presented in this paper.

2.1 System Runtime Data

Runtime data is generated by systems or programs while in operation. In today's environment, characterized by com-

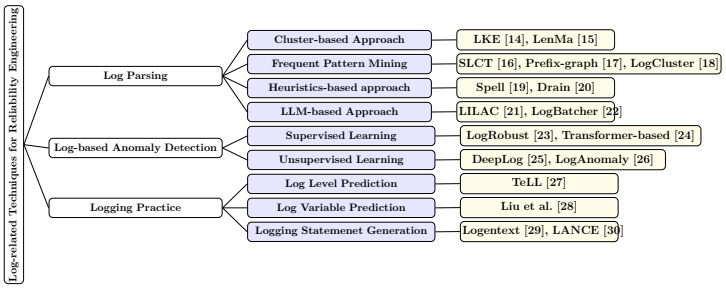


Figure 2.1: The taxonomy of log-related techniques in this thesis.

plex, large-scale systems like Amazon and Alibaba Cloud, these systems are operational around the clock to cater to millions of global customers via the internet. The reliance on software hosted by such online services means that even minimal downtime can lead to significant revenue losses for both service providers and users. Furthermore, the complexity and scale of modern systems make them prone to operational failures, such as node failures, highlighting the need for effective software reliability management.

To manage software reliability, engineers need to monitor the system's runtime status continuously, detect anomalies, and rectify operational issues swiftly. System runtime data, often the only source available for such analysis, is critical for these jobs.

Runtime data can be categorized into metrics, topologies, and logs. Metrics are standardized measurements of system performance, including parameters like service response times and CPU usage. Topologies represent graphical representations within cloud systems, detailing the interactions and dependen-

cies among various system components. Logs, on the other hand, are semi-structured textual records that document critical events and operations during system runtime. Of these categories, logs are typically the most prevalent and crucial because of two factors: first, logs provide contextual information that is essential for diagnosing failures; and second, they record the chronological behavior of the system, which is vital for tracking the sequence of events leading up to certain actions.

Figure 2.2 (Upper part) presents examples of raw log messages from Zookeeper's running logs. Each log message comprises several elements: the log time, log level (e.g., INFO), system components (e.g., NIOServerCxn.Factory), and the log content itself. The log content can be further divided into two segments: a static part authored by developers and a dynamic part generated by the system. The static part offers a description of system behavior, while the dynamic part captures runtime variables within the program. For instance, in the first example log, the static part is "Client attempting to establish new session at," and the dynamic part is "/10.10.34.12:58913." In this thesis, we refer to the static part of a log message as the "log event" and the dynamic part as either "parameters" or "variables." Additionally, unless specified otherwise, we use "log message" or "logs" interchangeably to denote the log content.

With the rise of distributed systems and cloud computing, log management has become increasingly challenging due to the stringent requirements for reliability assurance and the sheer

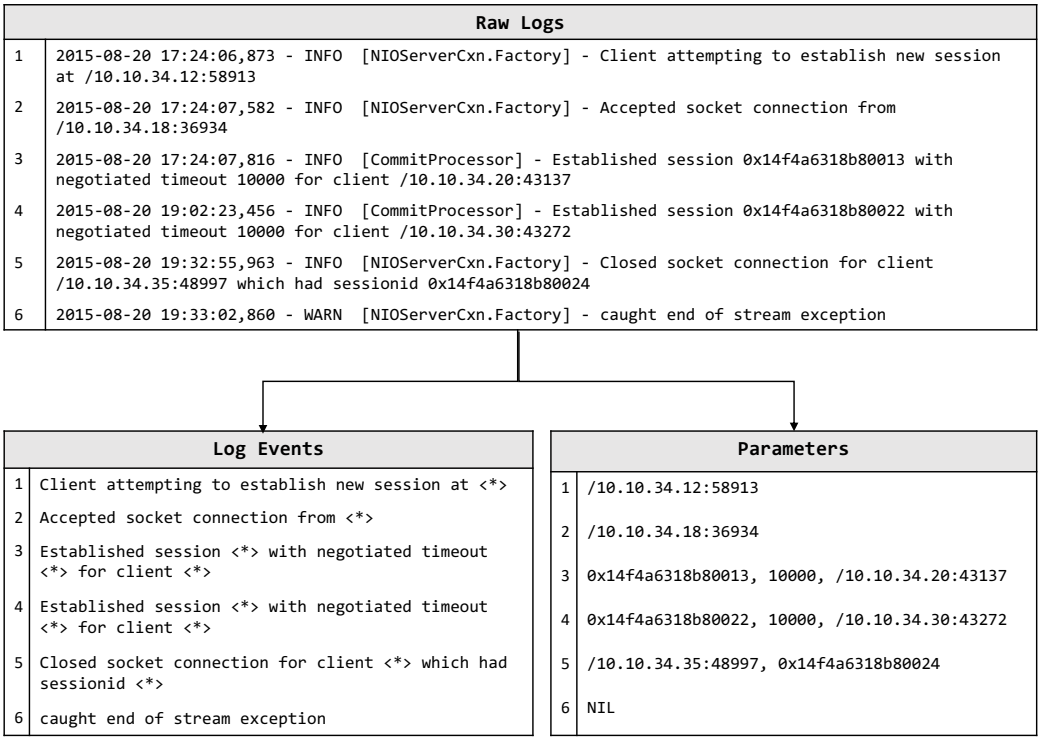


Figure 2.2: Examples of log parsing.

volume of log data. Numerous studies have addressed these challenges by providing tools and methodologies for handling extensive log data. For instance, Wei et al. [31] developed a log compression and query tool named Loggrep that efficiently structures and organizes log data into fine-grained units by leveraging both static and runtime patterns. Similarly, Yu et al. [32] introduced a non-intrusive log reduction framework based on eBPF (Extended Berkeley Packet Filter), which significantly aids in managing log hotspots and diagnosing cascading failures. Another notable solution, FLAP [33], employs data mining techniques in an end-to-end approach to assist log analysts in investigating system status. In both industry and academia,

the utilization of logs for system monitoring has gained great attention for decades.

2.2 Log Parsing

In this section, we delve into the background knowledge of log parsing. Log parsing is the process of converting unstructured log messages into structured log events. A log message typically consists of free-form text that describes system activities. This text includes natural language written by software developers and auto-generated variables during software execution. Figure 2.3 illustrates the typical workflow for log analysis. Most log mining algorithms, such as Support Vector Machines (SVM), require structured input in the form of matrices. Therefore, a fundamental step in automated log analysis is log parsing. By transforming free-form log messages into structured log events, also known as log templates, the data becomes more adaptable for subsequent analytical modules.

2.2.1 Problem Description

Log parsing can be viewed as an information extraction problem with the primary objective of differentiating log events from variables. The input for a log parser is a sequence of log messages $L = (l_i : i = 1, 2, ...)$ during system runtime, where each log message l_i can be represented as a sequence of tokens $l_i = (t_j : j = 1, 2, ...)$. As illustrated in Figure 2.2, each log

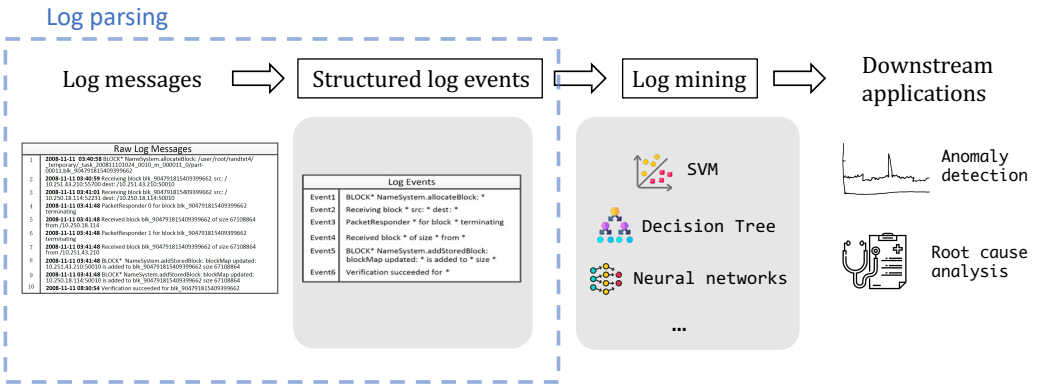


Figure 2.3: The overflow for log analysis.

message comprises a timestamp, log level, components, and log content. While fields like timestamps typically follow standardized formats, the log content is usually in a free-text format, defined by developers through logging statements. The output of a log parser is a sequence of log events $E = (E_i : i = 1, 2, ...)$ and a sequence of parameters $P = (P_i : i = 1, 2, ...)$ corresponds to each log message.

For instance, in Figure 2.2, the log parser processes 6 lines of raw logs and outputs corresponding sequences of log events and parameters for each log line. After parsing, the third line produces a log event "Accepted socket connection from $< * >$ and parameters "/10.10.34.18:36934". Here, $< * >$ acts as a placeholder for the variable part. Different log messages can share the same log events when they describe identical system activities.

It is important to note that all log parsing methods proposed in this thesis rely solely on the log messages as input and

do not require any source code information. While log events can naturally be extracted from source code using program analysis and then mapped to log messages, this approach is often impractical in large-scale online services due to the use of third-party libraries and security concerns that might render source code inaccessible.

2.2.2 Literature Review

A series of data mining approaches are proposed for log parsing, which can be further divided into three categories [2]: frequent pattern mining, heuristics, and clustering.

Clustering-based log parsing

Some parsers approach log parsing as a clustering problem: Given a sequence of log messages $L = (l_i : i = 1, 2, ...)$, a clustering-based log parsing method groups them into a set of K clusters $C = C_1, C_2, ..., C_k$. Log messages within the same cluster exhibit high similarities, whereas those in different clusters have low similarities. After clustering, the log parsing method extracts a single log template from each cluster, representing the core event for all messages within that cluster. Various clustering techniques can be employed, including hierarchical clustering, density-based clustering (e.g., DBSCAN [34]), and centroid-based clustering (e.g., KNN [35]). Below are two specific examples:

- LKE [14]: This method hierarchically clusters log messages based on a weighted edit distance threshold determined by a K-Means algorithm. Messages with distances below the threshold are grouped together. Each resulting group is then subdivided based on the positions of the least frequent tokens, forming the final clusters.

- LenMa [15]: This algorithm encodes each log message into a vector based on the length of each token. Log messages with high cosine similarity in their word length vectors are clustered together. LenMa also supports incremental updates for new incoming logs, appending a message to an existing cluster with the highest similarity or classifying it into a new cluster if no sufficient similarity is found.

Frequent pattern mining

Some log parsers implement Frequent Pattern Mining (FPM) [36], a traditional data mining technique used to uncover patterns that appear above a certain support threshold [37]. These methods generally concentrate on two types of frequent patterns derived from the fundamental characteristics of system logs: (1) *Token frequency*: Most tokens occur infrequently, so frequently occurring tokens are typically constants. (2) *N-gram frequency*: There are often strong correlations between consecutive tokens that appear frequently together. We will illustrate this approach with two specific examples:

- SLCT [16]: SLCT was an pioneering method in automated log parsing. It determines whether a token is a variable or a constant based on its frequency, assuming that frequent tokens are constants.
- Prefix-Graph [17]: This method uses a probabilistic graph to model sequences of consecutive tokens. Starting with a directed acyclic graph, it iteratively merges branches with similar frequent tokens. A template extraction algorithm is then used to derive log events from the graph.

Heuristics-based approach

Some log parsers utilize heuristic algorithms or specific data structures to encode logs and extract templates. The most common approaches include identifying the longest common subsequences (LCS) between log messages, developing parsing trees, and extracting heuristic features for machine learning. Below, we detail two such methods:

- Spell [19]: Spell actively identifies LCS between logs and maintains an LCS map for already parsed logs. During parsing, Spell searches this map to find the match with the maximum LCS length for an incoming log message.
- Drain [20]: Drain employs a fixed-depth parse tree, where each internal leaf node encodes specifically designed heuristic filtering rules, such as word length and preceding tokens, to streamline the parsing process.

2.3 Log-based Anomaly Detection

Anomalies can occur in running software systems, such as online services, manifesting as node failures, connection errors, disk full conditions, and unexpected outages. Anomaly detection aims to identify these abnormal system behaviors promptly, playing a crucial role in managing failures in large-scale systems. Timely detection allows system developers and operators to quickly pinpoint and resolve issues, thereby minimizing system downtime.

Most unexpected anomalies are recorded in log files, but error messages could be overwhelmed by normal log entries. To alleviate the burden of manually inspecting log files, numerous studies have focused on developing automatic log-based anomaly detection techniques. By narrowing down log files to a small subset that indicates anomalies, engineers can address these issues more quickly. In the following sections, we provide an overview of the background and related work on log-based anomaly detection.

2.3.1 Problem Description

Log-based anomaly detection can be viewed as a binary classification problem, where the goal is to categorize a small sequence of log messages as either normal or anomalous. The input to these anomaly detectors is a sequence of log messages $L = (l_i : i = 1, 2, ...)$, and the output is the label for each subse-

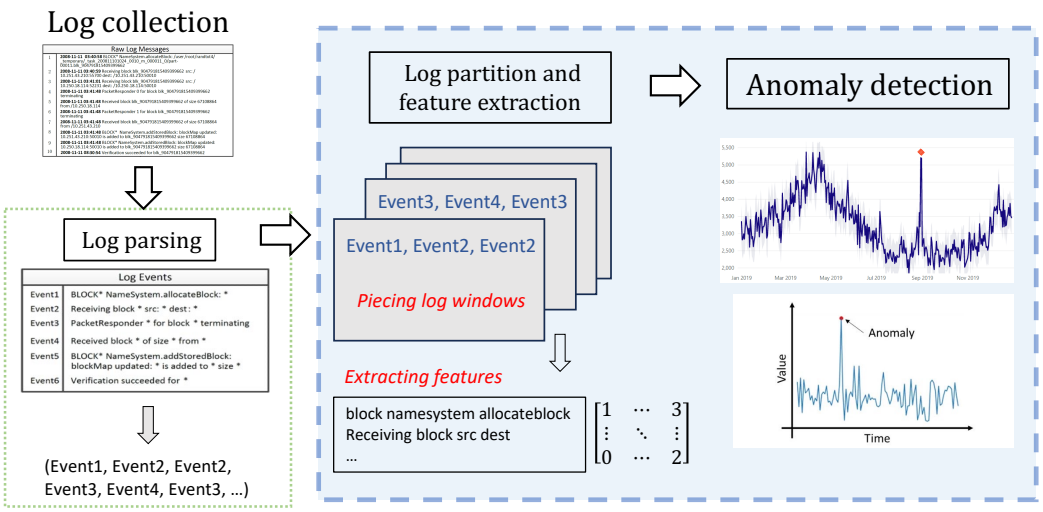


Figure 2.4: Framework of anomaly detection.

quence within L. Figure 2.4 outlines the typical framework for anomaly detection, which includes four main components: log collection, log parsing, log partitioning and feature extraction, and anomaly detection. The blue segment highlights the core focus of the anomaly detection process.

Specifically, once each log message is represented by its log template (through log parsing), the log partitioning step slices the log sequence into log windows. In this step, timestamps and identifiers (e.g., task ID, job ID) are commonly used to segregate the original long log sequence (millions) into segments (hundreds to thousands). The result is a series of log windows, each containing a subsequence of log messages. Next, feature extraction algorithms are applied to construct numerical features from each log window. These features are then fed into machine learning algorithms to determine whether a log win-

dow contains anomalies. In practice, only the log windows that indicate anomalies are forwarded to system maintainers for further diagnosis, thereby reducing their workload and improving efficiency.

2.3.2 Literature Review

A significant number of studies focus on log-based anomaly detection, with many adopting learning-based approaches over the past few decades. These methods build models from historical data and can be broadly categorized into supervised and unsupervised anomaly detection.

Supervised Log-Based Anomaly Detection

Supervised log-based anomaly detection requires that anomaly labels in the training data be available in advance. This allows the model to automatically learn features that help distinguish abnormal samples from normal ones. Specifically, given an input window $W = (e_1, e_2, ...)$ containing log events e_i and a corresponding label y_W, the supervised model is trained to perform binary classification by maximizing the conditional probability distribution $P(y = y_W|W)$. The main efforts in developing supervised log-based anomaly detection models focus on two aspects: (1) feature extraction, and (2) designing appropriate machine learning algorithms to work with the numerical features. Here, we illustrate two methods:

- LogRobust [23]. To accommodate the evolving log events and sequences during the software development process, LogRobust encodes log templates using TF-IDF scores, and then applies an attention-based bi-LSTM network for anomaly detection. The attention mechanism enables the model to learn the relative importance of different log events.
- Transformer-based [24]. The approach utilizes the Transformer encoder [38] to encode log windows with a multi-head self-attention mechanism to detect anomalies in logs. This allows the model to capture contextual information effectively.

Unsupervised log-based anomaly detection

In some cases, acquiring labels for anomaly detection is labor-intensive, necessitating the use of unsupervised log-based anomaly detection methods. Instead of performing classification like supervised models, unsupervised models focus on forecasting subsequent log events based on previous log sequences. The fundamental assumption behind unsupervised methods is that logs generated during normal system operations exhibit stable patterns. When an anomaly occurs, these normal log patterns are disrupted, such as by the appearance of error logs, changes in the order of log events, or premature termination of stable patterns. By learning these normal patterns, any deviation from them can be identified as an anomaly.

Specifically, given an input window $W_i = (e_{t-m}, e_{t-m+1}, ..., e_{t-1})$ and the next log event following the input window e_t, the model aims to learn a conditional probability distribution $P = (e_t = e_i|W)$ for all e_i in the set of log events $E = e_1, e_2, ..., e_n$ [39]. Here, we demonstrate two methods:

- DeepLog [40]. It uses a forecasting-based approach that predicts the next log event based on preceding event ID sequences via LSTM. Anomalies are detected based on prediction discrepancies, making this approach a pioneering unsupervised one in the community.
- LogAnomaly [26]. This approach takes into account the semantic information of logs by introducing a template2Vec distributed representation to encode words in log templates. Template2Vec distinguishes between synonyms and antonyms, ensuring that words like "success" and "fail" have distinct meanings. It is also a forecasting method.

2.4 Logging Practices

Log analysis relies heavily on high-quality logging statements embedded in the source code. Engineers can only use this information to investigate root causes if the printed logging statements accurately describe system behavior and failure details. In addition to supporting software operations, logging is also widely used during various phases of the software lifecycle,

including testing [41] and debugging [42].

While developers theoretically have the option to log every event (e.g., every exception), doing so can degrade the efficiency and effectiveness of a program. Conversely, missing logging information can obscure runtime failures, complicating the diagnosis process. To maintain a balance between the quantity and quality of logging statements, developers strategically insert log statements, specify appropriate log levels (e.g., error, debug, info), and craft concise yet informative text messages. Despite having expertise, determining where and what to log remains a challenging task for developers.

2.4.1 Problem Description

Most studies on logging practices focus on the automatic suggestions on logging statements, treating it as a non-functional code line completion problem. Logging practices can be divided into two key tasks: determining where to log and what to log. (1) Where-to-log: Given a piece of code (typically a function), the model predicts which line should have a logging statement inserted. (2) What-to-log: Given code snippets (typically a function) and a specific line, the model is asked to complete the full logging statement, including log level, variables, and log events. Figure 2.5 illustrates an example for each of these tasks.

The structure of logging statements naturally makes logging generation a task that combines code comprehension and

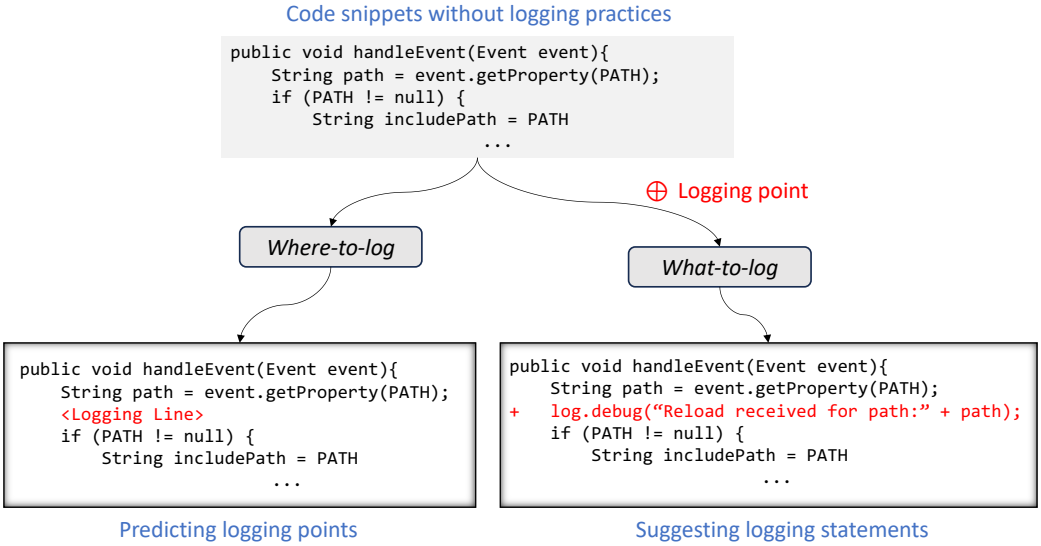


Figure 2.5: Example of where-to-log and what-to-log.

text generation. However, unlike typical code completion tasks, generating logging statements presents two unique challenges: (1) inferring critical runtime statuses of the software and (2) creating complex text that seamlessly integrates both natural language and code elements.

2.4.2 Literature Review

A number of studies in logging practices aim to provide automatic logging suggestions for developers during programming. These studies can be broadly categorized into two types: deciding the contents of logging statements (what-to-log) and recommending appropriate locations for logging statements (where-to-log).

Where-to-Log

Where-to-log studies focus on suggesting optimal logging points in the source code [43, 44]. Excessive logging can increase unnecessary effort in development and maintenance, while insufficient logging can miss key information needed for system diagnostics [45, 3]. Existing research on where-to-log concentrates on understanding code structures to automate log placement. Previous studies address log placement in specific code constructs, such as catch blocks [46], if statements [46], and exceptions [47]. Li et al. [43] propose a deep learning-based framework to suggest logging locations by fusing syntactic, semantic and block features extracted from source code. The most recent model in T5 architecture, LANCE [30], provides a one-stop logging statements solution for deciding logging points and logging contents for code snippets.

What-to-Log

What-to-log studies are interested in producing concrete logging statements, which include deciding the appropriate log level (e.g., warn, error) [48, 27, 49], choosing suitable variables [28, 50, 51], and generating proper logging text [30, 29]. For example, ordinal-based neural networks [48] and graph neural networks [27] have been applied to learn syntactic code features and semantic text features for log-level suggestions. LogEnhancer [51] aims to ease the burden of failure diagnosis by in-

serting causally-related variables into logging statements using program analysis techniques, whereas Liu et al. [28] predict logging variables for developers using a self-attention neural network to learn tokens in code snippets.

□ **End of chapter.**

Chapter 3

Semantic-aware Log Parser

3.1 Introduction

A log message is a type of semi-structured language comprising a natural language written by software developers and some auto-generated variables during software execution. Since most log analysis tools require structured input, the fundamental step for automated log analysis is log parsing. Given a raw log message, a log parser identifies a set of fields (e.g., verbosity levels, date, time) and the message content, which is then represented as structured event templates (i.e., constants) with corresponding parameters (i.e., variables). For example, in Figure 3.1 (up), the template "Listing instance in cell <*>" describes the system event, and "949e1227" serves as the parameter corresponding to the placeholder "<*>" in the template.

Despite the challenges associated with automatic log parsing, researchers have made significant progress. For instance, SLCT [16] and LFA [52] create log templates by counting his-

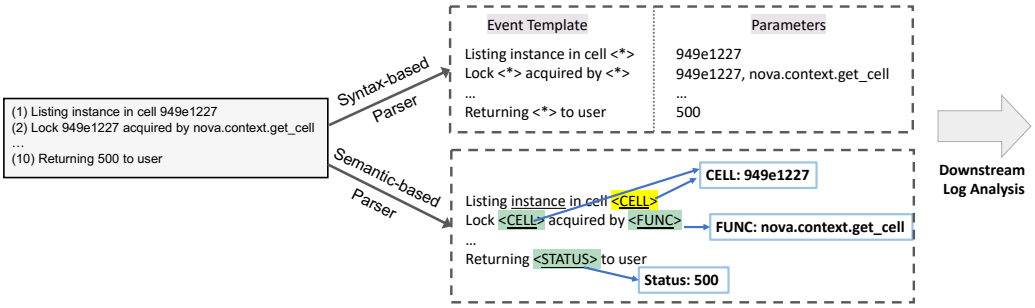


Figure 3.1: Difference between syntax-based parsers and semantic-based SemParser. Logs are sampled from OpenStack.

torically frequent words. Logram [53] focuses on frequent n-gram patterns. LogSig [54] and SHISO [55] encode logs based on word pairs and word length, respectively, and then apply clustering algorithms for partitioning. Prefix-Graph [17] uses a probabilistic graph approach for parsing. The most widely-used parser in industry, Drain [20], constructs log templates by traversing leaf nodes in a tree structure. However, we argue that all current parsers are *syntax-based*, relying on superficial features such as word length, log length, and frequency. These parsers have limited capability for high-level semantic understanding. In this chapter, we categorize these limitations into a three-level hierarchy to better analyze and address the shortcomings of existing log parsing methods.

The first limitation is the inadequate attention to individual *informative tokens*. For instance, in the first log shown in Figure 3.1, the parameter "949e1227" and technical terms like "cell" are far more noteworthy compared to prepositions such as "in." Syntax-based log parsers only differentiate between param-

eters and templates, treating each log message as a sequence of characters without considering the significance of technical concepts. Previous research [56] has shown that technical terms and topics in logs are informative when examining six large software systems. Therefore, both parameters and domain-specific terms should be localized to enhance log comprehension.

Secondly, the *semantics within a message* should be highlighted. While humans rarely use digits or character strings (like "949e1227") in everyday communication, these parameters in log messages are crucial and carry specific meanings. Unfortunately, syntax-based parsers regard each parameter as a meaningless character string. Intuitively, a parameter in a log is used to specify another technical concept within the same log. For example, in the first log of Figure 3.1, the token "949e1227" is understood to refer to "cell," making "949e1227" a cell ID. Exploiting such intra-message semantics can significantly improve the understanding of parameters.

Thirdly, the *semantics between messages* are often overlooked. All previous parsers process each log message independently, ignoring the relational context between them. However, historical logs can provide domain knowledge about a parameter, helping to resolve the implicit meanings of the same parameter in subsequent logs. In Figure 3.1, although the second log does not explicitly disclose the semantics of the parameter "949e1227," we infer it refers to a cell based on the information provided in the first log. Since parameters rarely appear in

daily language, understanding their semantics in log messages is inherently different from comprehending common language, highlighting the importance of mining these semantics from logs.

Some studies have recognized the aforementioned limitations and have attempted to address them. For instance, LogRobust [23] assigns weights to each token based on its TF-IDF value when encoding logs, aiming to highlight informative tokens. While this approach tends to assign higher weights to rare words, common technical terms can also be illuminating and should not be overlooked. For semantic mining, several approaches like Drain [20], LKE [14], MoLFI [57], and SHISO [55] use regular expressions to recognize block IDs, IP addresses, and numbers when parsing HDFS datasets. However, creating handcrafted rules is a tedious process and can become impractical during system migrations. It is nearly impossible to account for all potential scenarios, so these rules can only cover a limited portion of the logs. Furthermore, these regular expressions struggle to distinguish between polysemous parameters. For example, the variable ”200” could refer to an HTTP status code if the system makes REST API calls, but it might also represent a thread identifier (TID) in Spark. Moreover, text mining approaches [58, 59] that are effective in human language understanding fail to grasp the specific meanings of variables in log messages. As illustrated in the last log in Figure 3.1, serious information omissions and misinterpretations of erroneous status codes, such as ”500,” can accumulate, complicating further

anomaly detection tasks. This accumulation ultimately hinders the aim of preventing incidents and ensuring system reliability. This highlights the need for more sophisticated techniques that can accurately parse and understand the semantic context of logs, thereby improving the overall efficiency and effectiveness of log-based anomaly detection.

To address the aforementioned complex yet critical limitations, we introduce a novel *semantic-based* log parser, **SemParser**. This is the first work aiming to parse logs by focusing on their semantic meanings. We begin by defining two levels of semantic granularity in logs: *message-level* and *instance-level* semantics. Message-level semantics involves identifying technical concepts (e.g., "cell") within log messages (highlighted in Figure 3.1), while instance-level semantics concerns understanding what the instance (e.g., parameters) describes. To achieve this, we design an end-to-end semantics miner and a joint parser. This system can not only recognize the templates of given logs but also extract explicit semantics within a log and uncover implicit inter-log semantics. Specifically: (1) The end-to-end semantics miner is designed to identify the semantics of log messages (e.g., concepts like "instance" and "cell") and the explicit semantics of instances (e.g., "949e1227" as a "cell ID"). This step allows us to capture noteworthy tokens and explicit parameter semantics, addressing the first and second limitations, respectively. (2) The joint parser then infers the implicit semantics of parameters using domain knowledge acquired from prior

logs, thereby overcoming the third limitation of missing inter-log relations. Figure 3.1 illustrates the key distinctions between syntax-based parsers and the proposed SemParser. Explicit semantics are highlighted in yellow, while implicit semantics are highlighted in green. Not only does SemParser serve as an accurate log-template extractor, like existing syntax-based parsers, but it also provides additional structured semantics. This enriches the information available for downstream analysis, enhancing overall system diagnostics and anomaly detection.

We conduct an extensive study to evaluate the performance of SemParser on six system log datasets from two perspectives: (1) the effectiveness of semantic mining, and (2) its impact on two typical downstream log analysis tasks. The experimental results demonstrate that our approach can capture semantics with high accuracy, achieving an average F1 score of 0.985 in semantic mining. Additionally, SemParser outperforms state-of-the-art log parsers, with an average improvement of 1.2% and 11.7% on two anomaly detection datasets and an 8.65% enhancement on a failure identification dataset. These compelling results highlight the superiority of SemParser and underscore the critical importance of semantics in log analytics, particularly given the increasing complexity of modern software systems.

In summary, the contribution of this chapter is threefold:

- To the best of our knowledge, SemParser is the first semantic-based parser capable of actively capturing both message-

level and instance-level semantics from logs, while also collecting and leveraging domain knowledge for parsing.

- We evaluate SemParser in terms of its semantic mining accuracy on six system logs, demonstrating that our framework can effectively extract semantics from logs.
- We apply SemParser to the tasks of failure identification and anomaly detection. The promising results reveal the critical importance of semantics in the field of log analytics.

3.2 Problem Statement

This chapter focuses on parsing logs with respect to semantics, which could further be decoupled into message-level semantics and instance-level semantics. Message-level semantics are defined as a set of *concepts* (i.e., technical terms) appearing in log messages, such as "cell". We use the term *instance* [1] to denote variables in log messages, then the instance-level semantics are represented by a set of *Concept-Instance pairs (CI pairs)*, which describe the concept that the instance refers to, such as (cell, 949e1227). A *Domain Knowledge database* maintains a list of detected CI pairs from historical logs. After obtaining *instances*, *concepts* and *CI pairs* from a log message, we replace the instances with their corresponding concepts and name the new message as *conceptualized template*.

[1] The term "instance" is rather closed to the "parameters" or "variables" in the syntax-based parser. One concept can be instantiated by multiple instances.

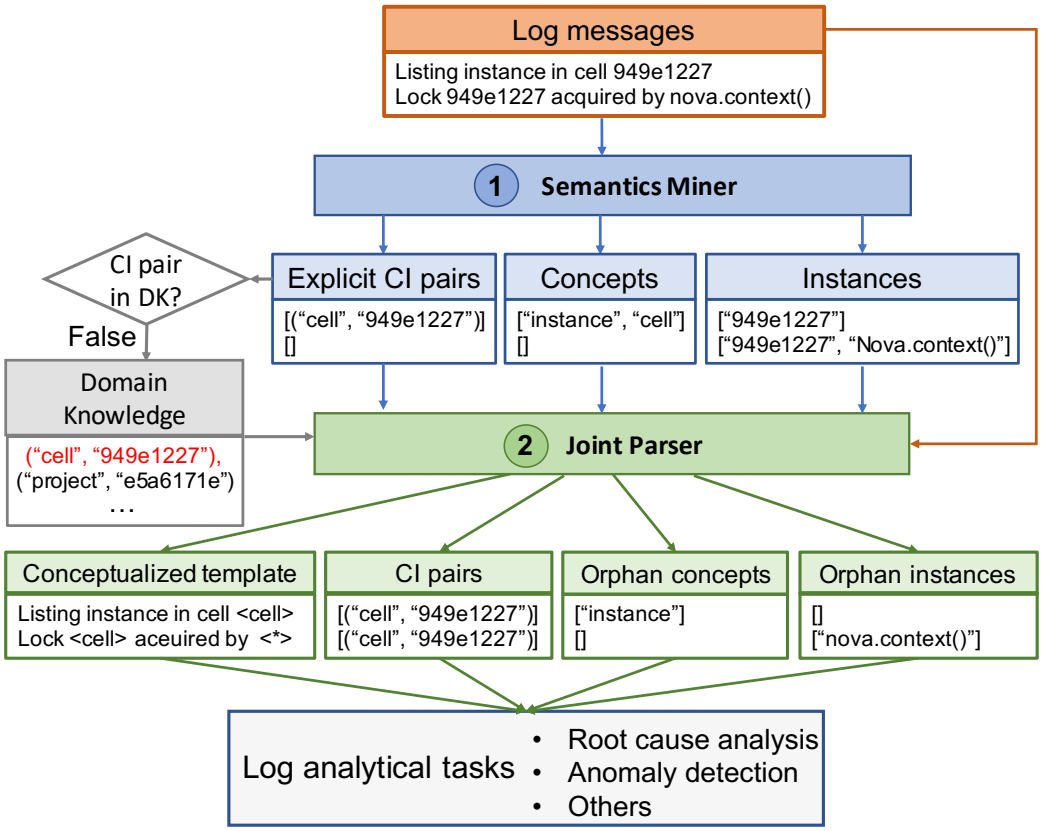


Figure 3.2: The pipeline of SemParser.

The semantic parser task can be regarded as follows. Given a log message[2], the structural output is composed of a conceptualized template T, a set of CI pairs $CI = \{(c_0, i_0), ..., (c_n, i_n)\}$, as well as other orphan concepts $OC = \{oc_0, ..., oc_j\}$ and orphan instances $OI = \{oi_0, ...oi_k\}$ which cannot be paired with each other.

[2]The log message refers to log content without fields in this chapter by default.

3.3 Methodology

3.3.1 Overview of SemParser

Our framework consists of two main components: an end-to-end semantics miner and a joint parser. Figure 3.2 provides an example to illustrate how our framework processes log messages. To begin with, log messages are sent to the semantics miner to extract template-level semantics (i.e., *concepts*) and explicit instance-level semantics (i.e., *explicit CI pairs*) from each log message independently. This step directly addresses the first two challenges mentioned earlier. Any new explicit CI pairs identified are added to the *Domain Knowledge database* to keep the knowledge base updated. Additionally, instances in log messages are retained to uncover potential implicit semantics using domain knowledge, thus addressing the challenge of missing inter-log relations. Subsequently, the joint parser processes the outputs from the semantics miner, focusing on inferring implicit semantics with the assistance of domain knowledge. The newfound implicit instance semantics, combined with the explicit semantics, form the comprehensive instance-level semantics, referred to as *CI pairs*. Remaining concepts and instances that cannot be paired are stored separately as *orphan concepts* and *orphan instances*, respectively. Additionally, *conceptualized templates* are generated by replacing instances with their corresponding concepts (if available) or with placeholders such as "<*>" otherwise.

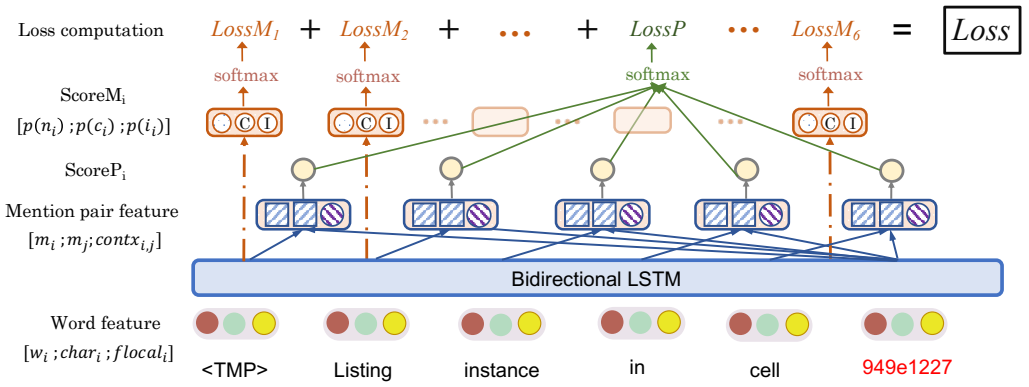


Figure 3.3: The architecture of semantics miner.

The final structured output of SemParser consists of *conceptualized templates*, *CI pairs*, *orphan concepts*, and *orphan instances*. As the foundational step for log analysis, SemParser facilitates a wide range of downstream log analysis tasks. The subsequent two subsections will provide detailed explanations of the semantics miner and the joint parser.

3.3.2 End-to-end semantics miner

Semantics miner aims to mine semantics on both the instance-level and the message-level. To acquire a set of explicit *concepts*, *instances* , and *CI pairs* within a log message, we model the task into two sub-problems: finding CI pairs and classifying each token into a type in $\{concept, instance, none\}$. As shown in Figure 3.3, an end-to-end model with three modules is proposed to solve the two sub-tasks simultaneously. First, a log message is fed into a *Contextual Encoder* for acquiring context-based word representation. Then, the contextualized words are separately

used in *Pair Matcher* and a *Word Scorer* for extracting CI pairs and determining the type of each word, respectively. As the total loss is the sum of the Pair Matcher loss and Word Scorer loss, the model is forced to learn from both sub-tasks jointly. We elaborate on the details of the three modules as below.

Contextual encoder

Intuitively, log messages can be regarded as a special type of natural language due to its semi-structured essence of mixing unstructured natural language and structured variables.

Motivated by the success of long short-term memory networks (LSTM) across natural language processing tasks [60] (e.g., machine translation, language modeling), we design an bi-directional LSTM-based network (bi-LSTM) [61] to capture interactions and dependencies between words in log messages.

However, it is not practical to directly feed the word embeddings into the LSTM network because of the severe out-of-vocabulary (OOV) problem, which is due to the large portion of customized words in log messages (e.g., function names, cell ID, request ID), resulting drastic performance degradation. To solve the problem, we devise two additional features associated with word representations. Firstly, inspired by previous findings that character-level representation helps exploit sub-word-level information [62], we adapt a Convolution Neural Network (CNN) to extract character-level features of each word. Secondly, following several studies [63, 64] that leveraged local fea-

tures for sequential representations, we also deliberate a set of local features for each word concerning its shape, length, and other morphological features.

The word representations $word_i$, character representations $char_i$, as well as the local features f_i^{local}, are concatenated as word features and fed into the bi-LSTM indicated in Equation 3.1. Afterward, the hidden state of bi-LSTM is used as the contextual embedding for each word.

$$m_i = LSTM([word_i; char_i; f_i^{local}]) \tag{3.1}$$

Pair matcher

This module is designed for acquiring explicit instance-level semantics. Numerous studies focus on identifying key elements in texts and classifying them into several categories by assigning each word into one of the pre-defined categories. For example, the combination of bi-LSTM and Conditional Random Field (CRF) is deployed to identify 100 log entities (e.g., IP address, identifier) in log messages [65], or uncover 20 software entities (e.g., class name, website) in software forum discussions [66]. However, such token classification-based framework relies on a *closed-world assumption* that all categories are known in advance. The assumption makes sense when dealing with a specific and small system with limited concepts. Unfortunately, it will break down if we want to migrate the approach across software systems, or the system we are facing is huge and sophisticated.

To get over the closed-world assumption limitation, the pair matcher is required to discern the (*concept*, *instance*) pairs between words in a log message. We abstract this problem as a multi-classifier problem: for each word w_i in a sentence $S = w_1, w_2, ..., w_n$, the matcher determines what previous word $w_j (0 \leq j < i)$ does the word w_i refer to[3].

To achieve the goal, we rank the confidence score of each word pair candidate $(w_i, w_j), \forall 0 \leq j < i$, which is determined by a feed-forward neural network $FFNN_a$ as in Equation 3.2. Intuitively, if a word "is" exists between w_i and w_j, the pair has a higher probability formed by the two words, so we consider the interval context between the candidate pair (w_i, w_j) as the average word embedding value between the pair, denoted as $contx_{i,j}$. In summary, we construct the pair-level features $f_{i,j}^{pair}$ for scoring by concatenating contextual representation (i.e., m_i, m_j) obtained from last step, as well as the abovementioned interval context $contx_{i,j}$, as shown in Equation 3.3.

$$ScoreP_i(i,j) = FFNN_a(f_{i,j}^{pair}) \quad (3.2)$$

$$f_{i,j}^{pair} = [m_i; m_j; contx_{i,j}] \quad (3.3)$$

Figure 3.3 shows a simple case for matching pair for the red word w_5. After acquiring contextual word representations from the contextual encoder, we form the pair-level feature for each word pair in $\{(w_5, w_4), (w_5, w_3)...(w_5, w_0)\}$. These pair features

[3]We add a dummy word <TMP> (w_0) to indicate the word does not refer to any of the previous word in the message (e.g., in).

will be scoring by a softmax function on top of a feed forward neural network for loss computation.

Word scorer

Apart from the pair matcher, we also design a word scorer to determine whether each token is a *concept*, *instance* or *neither of both*. The token's category is crucial for two reasons. First, the message-level semantics can be perceived via extracted concepts. Second, we notice that some instance-level semantics cannot be resolved via the pair matcher if the instance's corresponding concept does not occur in a single message (e.g., the second log in Figure 3.1), which we call *implicit instance-level semantics*. In this case, we need to store the *instances* for further processing. To this end, we devise the word scorer with a feed-forward neural network $FFNN_b$ to learn the possibility of three types for each token. The score is computed as follows:

$$ScoreM_i = FFNN_b(m_i) \tag{3.4}$$

Afterwards, the possibility of three categories will pass through a softmax layer for normalization before computing loss.

Loss function

Multi-task learning (MTL) is a training paradigm that trains a collection of neural network models for multiple tasks simultaneously, leveraging the shared data representation for learning

common knowledge[67, 65]. The fruitful achievements of MTL motivate us to train pair matcher and word scorer simultaneously by aggregating their losses. Therefore, the total cost of semantics miner is defined as:

$$cost = \sum_i CELoss(P_i') + \sum_i CELoss(M_i') \tag{3.5}$$

where P_i' and M_i' denotes the outputs of $ScoreP_i$ and $ScoreM_i$ after passing a softmax layer, respectively. Here, we adopt Cross Entropy Loss (i.e., CELoss) as the loss function due to its numerical stability. By minimizing the cost, the model naturally learns the pairs and the word types for each token with shared contextual representations generated from bi-LSTM network.

In the inference, for each word, we regard the highest probability of its pairs and its type score as the final results.

3.3.3 Joint parser

The joint parser leverages concepts, instances, and CI pairs obtained from the end-to-end semantics miner, as well as log messages to deal with : (1) uncovering implicit instance-level semantics using domain knowledge; and (2) semantic parsing log messages. The next sections go into the specifics.

Implicit instance-level semantics discovery

We apply a novel domain knowledge-assisted approach to resolve the implicit instance-level challenge of concepts and in-

stances not coexisting in one log message. Naturally, suppose we have recognized a CI pair in historical logs, then we are able to identify the semantics of such instance in the following logs, even though the following logs do not explicitly contain such pair information.

The knowledge-assisted approach maintains a high-quality domain knowledge database when processing logs by incorporating newly discovered CI pairs acquired from the semantics miner. To guarantee the quality of the domain knowledge, we only add the superior CI pairs, which are defined by *if and only if there is a concept and an instance in the predicted pair*. The joint parser examines whether the orphan instances have their corresponding concepts in the high-quality knowledge base, to uncover implicit CI pairs. As a result, fresh CI pairs of the log messages are stored if found. In such a way, we merge the explicit CI pairs and new implicit CI pairs into the final CI pairs. Details are in Algorithm 1.

Semantic parsing

As a semantic parser, SemParser is able to extract the template for a given log message obeying two rules:

- For the instance in CI pairs, replacing the instance with the token <concept> of its corresponding concept.
- For the orphan instances, replacing the instance with a dummy token <*> as syntax-based parsers do.

Algorithm 1 Implicit instance-level semantics discovery

Input: Log message $M = m_0, ..., m_n$, instance indices $I = [i_0, ...i_j]$, concept indices $C = [c_0, ...c_k]$, explicit CI pair indices $P = [(s_0, t_0), ..., (s_u, t_u)]$
Output: Instances I', Concepts C', CI pairs P'

$P' = []$
$C' = []$
for all p such that $p \in P$ **do**
    if p contains 1 instance cur_I and 1 concept cur_C **then**
        DomainKnowledge.add($M[cur_C]$,$M[cur_I]$)
        I.REMOVE(cur_I)
        C.REMOVE(cur_C)
    end if
end for;
for all i such that $i \in I$ **do**
    if FINDCONCEPTFROMDOMAINKNOWLEDGE($M[i]$) **then**
        P'.APPEND([newfound concept, $M[i]$])
        C'.APPEND(newfound concept)
        I.REMOVE(i)
    end if
end for
I' = INDEXTOWORD(I)
C' += INDEXTOWORD(C);
P' += INDEXTOWORD(P)

The rules are straightforward but reasonable. Compared to other technical terms or common words, instances (e.g., ID, number, status) are more likely to be variables in logging statements automatically generated by software systems. As the retrieved template takes in concepts, we name it "conceptualized template" instead of the vanilla template with only <*> representing parameters.

Finally, the conceptualized template, CI pairs, orphan concepts, as well as orphan instances are the structured outputs of our SemParser. The results are extensible for a collection of

downstream tasks, and we will elaborate them later.

3.4 Tool Implementation

3.4.1 Dataset annotation

We implement the SemParser framework on a public dataset [68] containing log messages collected from OpenStack for training. Considering that it is labor-intensive to annotate a large dataset in a real-world scenario, we randomly sample 200 logs from the dataset for human annotation, with the sample rate of 0.05%. A practical model should be able to learn from a small amount of data. The trained model from such data is named the "base model" for further evaluation.

All annotation is carried out as follows. For each log, we invite two post-graduate students experienced in OpenStack to independently manually label: (1) whether a word is a concept, instance, or neither of both; and (2) the explicit CI pairs within a sentence. If the two students provide the same answer for one log, the answer will be regarded as the ground-truth for training; otherwise another student will join them to discuss until a consensus is reached. The inter-annotator agreement [69] before adjudication is 0.881. Finally, we remove the sentences without any CI pair annotation to mitigate the sparse data problem, yielding 177 labeled messages for training the semantics miner.

3.4.2 Pre-trained word embeddings

Although existing pre-trained word embeddings show the large success in representing semantics of words, it is not appropriate for understanding logs. Log message is a very domain-specific language, where the words have quite distinct semantics from daily life. Hence, we train domain-specific word embeddings on a representative cloud management system, OpenStack corpus. The corpus is made up of 203,838 sentences crawled from its official website. We train the pervasive skip-gram model [70] on Gensim [71] for ten epochs and set the word embedding dimension to be 100.

3.4.3 Implementation details

When implementing the model, we set the character-level embedding dimension to be 30. We select the two-layer deep bi-LSTM with a hidden size of 128. The model is trained for 30 epochs[4] with an initial learning rate of 0.01. The learning rate decays at the rate of 0.005 after each epoch. It takes one hour for training, and the trained model occupies only 25 MB. SemParser runs 25 messages per second in a single batch and single thread during inference.

[4]The model converges within 30 epochs.

3.5 Evaluation

We evaluate SemParser from two perspectives, the ability of semantic mining and the usefulness in downstream tasks, with three research questions:

- RQ1: How effective is the SemParser in mining semantics from logs?
- RQ2: How effective is the SemParser in anomaly detection?
- RQ3: How effective is the SemParser in failure identification?

3.5.1 Experiment details

RQ1–Semantic mining

Dataset. LogHub [72] is a repository of system log files for research purposes, which has been used by plenty of log-related studies [73, 39, 74]. We manually label six representative log files for semantic mining evaluation ranging from distributed, operating, and mobile systems. The dataset has a total of six different system log files with 12,000 log messages and 20,636 annotated CI pairs. Details are shown in Table 3.1, where # Logs, # Pairs, # Temp., and Unseen denotes the number of log messages, CI pairs, log templates, and the percentage of *unseen templates* in the test set, respectively.

Settings. As SemParser is a semantic-based parser, we consider its semantic mining ability for evaluating how effective

Table 3.1: Statistics of dataset for semantic mining.

System type	System	#Logs	#Pairs	#Temp.	Unseen
Mobile system	Android	2,000	6,478	166	82.8%
Operating system	Linux	2,000	2,905	118	86.8%
Distributed system	Hadoop	2,000	2,592	14	84.6%
	HDFS	2,000	3,105	30	47.0%
	OpenStack	2,000	4,367	43	52.3%
	Zookeeper	2,000	1,189	50	75.9%

it is when mining instance-level semantics from log messages. Specifically, given a log message, we report the correct proportion of the model's extracted CI pairs (Precision), the proportion of actually correct positives extracted by the model (Recall), and their harmonic mean (F1 score). As we hope the model could learn semantics from small samples, we fine-tune the base model (i.e., train from Section 3.4) on a small dataset of 50 randomly sampled logs for each system and evaluate the performance on the remaining 1,950 logs.

RQ2–Anomaly detection

Dataset. We evaluate the anomaly detection performance on two datasets. (1) We first follow the previous studies to evaluate the HDFS [10] dataset, which includes log messages by running map-reduce tasks on more than 200 nodes. (2) The second F-Dataset [68] is initially created for investigating software failures by injecting 396 failure tests in major subsystems of the widely used cloud computing platform OpenStack, covering 70% of bug reports in the issue tracker. For each failure

Table 3.2: Statistics of anomaly detection datasets.

Dataset	#Message	Anomaly rate
HDFS dataset	11,175,629	3%
F-Dataset	1,318,860	0.22%

injection test, the authors all *log data* in major subsystems, the *labeled anomaly log messages*, as well as the exception raised by a service API call named as *API Error*, such as "server create error". Statistics of both datasets are shown in Table 3.2.

Settings. In the anomaly detection task, the detector predicts whether anomalies exist within a short period of log messages (i.e., session). Motivated by previous studies [8, 39], we decouple the anomaly detection framework into two components, a *log parser* to generate templates, and a *detection model* to analyze template sequences in a session. A dependable parser should perform well as a foundational processor for log analysis, regardless of the downstream detection model used. Our experiments compare the performance of different baseline parsers under various anomaly detection techniques.

Specifically, we compare SemParser to the following log parsers as baselines: (1) **LenMa [15].** This online parser encodes each log message into a vector, where each entry refers to the length of the token. Then, it parses logs by comparing the encoded vectors; (2) **AEL [75].** This chapter devises a set of heuristic rules to abstract values, such as "value" in "word=value"; (3) **IPLoM [76].** IPLoM partitions event logs

into event groups in three steps: partition by the length of the log; partition by token position; and partition by searching for bijection between the set of unique tokens; (4) **Drain [20].** It leverages a fixed depth parse tree with heuristic rules to maintain log groups. Its ability to parse logs in a streaming and timely manner makes it popular in both academia and industry.

We also reproduce four widely-applied anomaly detection models as follows: (1) **DeepLog** [25] employed a deep neural network, LSTM, to conduct anomaly detection and fault localization on logs, taking the context information into account; (2) To handle the ever-changing log events and sequences during the software evolution, **LogRobust** [23] detected anomaly detection by an attention-based bi-LSTM network. The attention mechanism allows the model to learn the different importance of log events; (3) **CNN** [77] is also utilized to detect anomalies in big data system logs inspired by its benefits in general NLP analysis; and (4)**Transformer.** [24] detected anomalies in logs via the Transformer encoder [38] with a multi-head self-attention mechanism, allowing the model to learn context information.

When conducting experiments, we feed parsing results from log messages into different models. Different from previous work [25, 23, 77, 24] that only employs templates to form the input sequence $x_0, x_1, ..., x_m$ where x_i refers to the i^{th} message in the sequence, we equip the sequence with extracted semantics. Specifically, for each log message in the sequence, we concate-

nate template, concepts, and instances as follows:

$$\tilde{x} = [template; < SEP >; sem_0; sem_1; ...; sem_n] \quad (3.6)$$

$$sem_i = [concept_i; instance_i]. \quad (3.7)$$

To specify the corresponding relationship within a CI pair, we concatenate the concept and instance in sem_i. Otherwise, an <NIL> token replaces another half pair, indicating the orphan situation. A special <SEP> token is used to separate the template and semantics. Afterwards, the sequence $\tilde{x_0}, \tilde{x_1}, ..., \tilde{x_m}$ containing m messages will be fed into the model for prediction. Following previous anomaly detection work [25, 23, 77, 24], we use Precision, Recall, and F1 as the evaluation metrics.

RQ3–Failure identification

Dataset. While anomaly detection identifies present faults from logs, failure identification looks deeper into the problems and identifies what type of failure occurs. To make the F-Dataset appropriate for failure identification, we utilize the labeled anomaly log messages and their corresponding API error in each injection test as the input and ground-truth. Entirely, we collect 405 failures with 16 different types of API errors. With the splitting training ratio of 0.5, we obtain 194 and 211 failures for the train and test set, respectively. Typical API errors include “server add volume error”, “network delete error” and so on.

Table 3.3: Sample log messages and ground-truth templates.

Log	After Scheduling: PendingReds:1 CompletedReds:0 ...
GT-Template	After Scheduling: PendingReds:<*> CompletedReds:0 ...
Log	TaskAttempt: [attempt_14451444] using containerId ...
GT-Template	TaskAttempt: [attempt_<*>] using containerId ...

Settings. In this chapter, we formulate the failure identification task as follows: given the *anomaly log messages* from one injection test in F-Dataset, the model is required to determine what *API error* emerges. Similar to the anomaly detection task, we also compare the performance of different baseline parsers associated with several log analysis models (i.e., DeepLog, LogRobust, CNN, and Transformer). The only difference is that we change the node number of the last prediction layer of the above-mentioned techniques from 2 to 16 to make it a 16-class classification task for 16 error types in the dataset.

Recall@k is widely used in recommendation systems to assess whether the predicted results are relevant to the user(s) [78, 79]. Similarly, we are also interested in whether top-k recommended results contain the correct API error. Hence, we report the Recall@k rate as the evaluation metric.

Discussion–log parsing comparison

In this section, we discuss why we do not compare SemParser to other syntax-based parsers in the log parsing task where only the templates and parameters are extracted. Firstly, the ground-truth for log parsing is not suitable for the semantic

parser. For the logs and their ground-truth templates shown in Table 3.3 with highlighted improper parts, we observe that "0" is not a parameter but a token in the template, because the value for "CompleteReds" is always "0" in 2000 logs in this template. In contrast, "0" will be regarded as an instance in our model, since "0" is used to describe "CompleteReds" semantically. Besides, we show how different tokenizer affects the results in the second example, where we consider "attempt_14451444" as an instance for the concept "TaskAttempt", but the syntax-based log parsers only regard the number "14451444" as parameters, excluding the same prefix "attempt". This kind of widely-present distinction occurs 817 times among 2000 logs in the Hadoop log collection. As a result, it is unfair to compare SemParser with syntax-based parsers in the log parsing task. Instead, we investigate the semantic mining ability in the first research question.

Secondly, log parsing is more of a pre-processing technique for downstream applications rather than an application by itself, and therefore, it will be more meaningful to concern about how the log parsers promote performance in downstream tasks. For example, if a developer wants to detect anomalies in overwhelming logs, the extracted templates and their parameters are not what he/she needs, but the result from an automated anomaly detection model is. From this perspective, we compare SemParser with four baseline parsers in two log analysis tasks to demonstrate our semantic parser's effectiveness. On the other

hand, our approach could provide accurate log templates with extra underlying semantics, so it would naturally promote generalized downstream tasks.

To conclude, SemParser is developed as a semantic-based parser instead of a syntax-based parser, so the evaluation should be related to its semantic acquisition ability and how the acquired semantics benefit log analysis for downstream tasks in an end-to-end fashion.

3.5.2 RQ1: How effective is the SemParser in mining semantics from logs?

In this experiment, we focus on evaluating the *explicit CI pair extraction* in the semantics miner as it serves as a vital step. A high-quality domain knowledge database and further joint parser process could be conducted if and only if the semantics miner extracts high-quality explicit CI pairs from log messages.

Basically, mining the instance-level semantics from log messages is difficult to do with handcrafted rules. Taking logs in Hadoop as examples, there are several ways to describe an instance associated with one concept TaskAttempt:

- TaskAttempt: [attempt_14451444] using containerId ...
- attempt_14451444 TaskAttempt Transitioned from ...
- Progress of TaskAttempt attempt_14451444 is ...

The evaluation result across six representative system logs is

Table 3.4: Experimental results of mining semantics from logs.

(a) Experiments for Android, Hadoop, and HDFS logs.

	System								
	Android			Hadoop			HDFS		
Framework	P	R	F1	P	R	F1	P	R	F1
SemParser	0.951	0.935	**0.943**	0.993	0.978	**0.985**	1.000	1.000	**1.000**
- w/o F_{char}	0.981	0.909	**0.943**	0.988	0.953	0.970	1.000	0.998	0.999
- w/o F_{local}	0.979	0.858	0.915	0.993	0.880	0.933	1.000	0.999	0.999
- w/o $LSTM$	0.979	0.858	0.915	0.993	0.879	0.932	1.000	0.999	0.999
- w/o F_{contx}	0.977	0.060	0.113	0.984	0.253	0.403	0.999	0.289	0.449

(b) Experiments for Linux, OpenStack, and Zookeeper logs.

	System								
	Linux			OpenStack			Zookeeper		
Framework	P	R	F1	P	R	F1	P	R	F1
SemParser	0.998	0.977	**0.987**	0.999	0.998	**0.999**	1.000	0.989	**0.995**
- w/o F_{char}	0.995	0.957	0.976	0.995	0.989	0.992	0.993	0.987	0.990
- w/o F_{local}	0.992	0.947	0.969	0.994	0.989	0.992	0.997	0.940	0.968
- w/o $LSTM$	0.995	0.909	0.951	1.000	0.963	0.981	0.966	0.953	0.959
- w/o F_{contx}	0.999	0.242	0.389	1.000	0.256	0.407	0.842	0.197	0.319

presented in Table 3.4. Since our work is the first to extract semantics from logs, we do not set baselines for comparison. Other general text mining techniques in the NLP field can only extract keywords (e.g., LDA [58]), but they are not be capable of extracting semantic pairs or parsing log messages to structured templates. Instead, we conduct ablation studies to explore the effectiveness of each element in the semantics miner, where w/o F_{char}, w/o F_{local}, w/o $LSTM$ and w/o F_{contx} refers to removing the character-level feature, local word feature, LSTM network, and interval context, respectively. The best F1 score for each system is in bold fonts.

In conclusion, our model could extract not only high quality

but also comprehensive instance-level semantics from log messages. We achieve an average F1 score of 0.985 for six systems logs even though we only fine-tune the base model on 50 annotated samples and a large portion of templates are unseen in the test set (the last column in Table 3.1). The promising result indicates our framework has a powerful ability for capturing semantics from log messages.

We attribute the outstanding concept-instance pairs mining ability of SemParser to its comprehensive architectures. The ablation experiments indicate that removing components degrade the performance in varying degrees. Firstly, to minimize the impact of a large portion of unknown words (e.g., attempt_14451444) to the model, we devise a character-level feature extraction convolutional network and a local feature extraction method since similar words are always composed of similar character structures. For example, although attempt_14451444 is different from attempt_14415371, they share the same structures that the word "attempt" following by an underscore and a sequence of numbers. Secondly, a recurrent network is designed to capture the contextual representation for each word in a sentence, since the same word may have various meanings under different contexts. By removing the bi-LSTM network, words in the sentence are equally regarded as a bag of words. Thirdly, SemParser naturally learns the patterns between concepts and instances by incorporating the interval context. For instance, if a colon separates two words, the latter word is prob-

ably an instance of the prior one, even if the latter one is an unseen word. We find such interval context is quite important, as a dramatic degradation is observed when we remove it. To conclude, the experiment shows the superiority of the our model by achieving an average F1 score of 0.985 across various system logs.

3.5.3 RQ2: How effective is the SemParser in anomaly detection?

To illustrate how SemParser benefits the anomaly detection task, we compare SemParser with four baseline parsers on four different anomaly detection models, and the results are shown in Table 3.5. Each row represents the performance of four anomaly detection models associated with the selected parser for upstream processing. The last row (Δ) displays how much our semantic parser outperforms the best baseline parser of F1 score, and the negative score indicates how the percentage of ours performs lower than the best baseline.

In the base HDFS dataset with only 31 templates, although all parsers provide a good performance, we still observe that SemParser also outperforms syntax-based parsers by an average F1 score of 1.22% over four techniques. In the more challenging F-Dataset, we observe that SemParser performs at rates approximately above ten percent overall baselines in the F1 score, indicating its effectiveness and robustness across various mod-

Log message	… ""GET /v2.1/5250c/flavors"" status: 200 …
C-Template	… ""GET /<*>/<*>/flavors"" status: <*status*> …
CI pairs	[(status, 200), (project, 5250c)]

Log message	Returning 500 to user …
C-Template	Returning <*status*> to user …
CI pairs	[(status, 500)]

Figure 3.4: A case for anomaly detection.

els. It outperforms baselines regarding DeepLog, LogRobust, CNN, and Transformer by 11.80%, 10.17%, 8.27%, and 16.58% respectively, with an average F1 score of 0.926. The results on Precision, Recall, and F1 reveal the effectiveness of acquired semantics from logs.

We attribute SemParser's distinct superiority on its precision to the awareness of semantics we extract, particularly instance-level semantics. Previous studies only use log template sequences to detect anomalies automatically, suffering from missing important semantics. Taking a case in Figure 3.4 as an example, where C-Template refers to the conceptualized templates. The CI-pairs are either extracted explicitly or implicitly via a domain knowledge database. The green tick indicates a normal log message, while the red cross stands for an anomaly log. A service maintainer must understand that "status: 500" returned by a REST API request reflects the internal server error, while the "status: 200" means the request is successful

Table 3.5: Experiment results for anomaly detection.

(a) HDFS Dataset.

	Technique											
	DeepLog			LogRobust			CNN			Transformer		
Baseline	P	R	F1	P	R	F1	P	R	F1	P	R	F1
LenMa	.897	.994	.943	.914	.995	.953	.924	.995	.958	.872	.908	.890
AEL	.896	.994	.943	.935	**.996**	.964	.922	.995	.958	**.893**	.904	.898
Drain	.908	.994	.949	.934	.994	.963	.925	.995	.959	.886	.871	.878
IPLoM	.898	.994	.944	.940	.994	.966	.926	**.996**	.960	.889	.904	.896
SemParser	**.940**	**.995**	**.967**	**.954**	.995	**.974**	**.931**	.995	**.962**	.881	**.954**	**.916**
Δ%	+1.86%			+0.82%			+0.21%			+2.00%		

(b) F-Dataset.

	Technique											
	DeepLog			LogRobust			CNN			Transformer		
Baseline	P	R	F1	P	R	F1	P	R	F1	P	R	F1
LenMa	.717	**.938**	.813	.714	**.924**	.806	.793	.815	.804	.685	.896	.776
AEL	.738	.934	.824	.791	.877	.832	.747	.924	.826	.503	**.962**	.660
Drain	.824	.867	.845	.810	.886	.846	.737	**.943**	.827	.693	.919	.790
IPLoM	.863	.833	.848	.808	.877	.841	.834	.834	.834	.929	.683	.787
SemParser	**.971**	.927	**.948**	**.952**	.913	**.932**	**.907**	.899	**.903**	**.938**	.904	**.921**
Δ%	+11.80%			+10.17%			+8.27%			+16.58%		

based on ad-hot knowledge. In this way, the maintainer can easily recognize that an API request fails if the return status equals to 500. Similarly, feeding semantics like (“status”, “500”) and (“status”, “200”) into the anomaly detection model forces the model to learn the relation between “500” and “anomaly” (or the relation between “200” and “normal”). As a result, the model will not mistake a log containing a normal status (e.g., 200) for an anomaly. The instance-level semantics also resolve problems for unseen logs. Even if the model has never encountered the template before, it is able to correctly predict it as a normal one according to a success status code, and vice versa.

Table 3.6: Experimental results in failure identification task.

(a) Part 1

	Model					
	LSTM			Atten-biLSTM		
Baseline	Rec@1	Rec@2	Rec@3	Rec@1	Rec@2	Rec@3
LenMa	0.839	0.924	0.953	0.858	0.943	0.957
AEL	0.844	0.919	0.953	0.853	0.915	0.962
Drain	0.844	0.919	**0.972**	0.863	0.938	0.953
IPLoM	0.848	0.943	0.957	0.863	0.948	0.962
SemParser	**0.954**	**0.968**	0.968	**0.954**	**0.968**	**0.972**
Δ%	+12.50%	+2.65%	-0.41%	+10.54%	+2.11%	+1.04%

(b) Part 2

	Model					
	CNN			Transformer		
Baseline	Rec@1	Rec@2	Rec@3	Rec@1	Rec@2	Rec@3
LenMa	0.877	0.962	0.967	0.919	0.934	0.948
AEL	0.810	0.905	0.929	0.858	0.929	0.953
Drain	0.867	0.948	0.967	0.853	0.919	0.943
IPLoM	0.867	**0.967**	**0.986**	0.839	0.910	0.948
SemParser	**0.945**	0.963	0.972	**0.954**	**0.958**	**0.968**
Δ%	+7.75%	-0.42%	-1.44%	+3.81%	+2.46%	+2.11%

Note that without the deliberately established CI Pairs, previous syntax-based parsers cannot distinguish the above normal v.s. anomaly status.

3.5.4 RQ3: How effective is the SemParser in failure identification?

This section demonstrates how effectively our semantic parser enhances failure identification. The experimental results are shown in Table 3.6, where each row represents the performance with the selected parser and several model architectures. The

last row reveals how much SemParser increases the F1 score when compared to the best baseline results. Given that there are 16 types of API errors in F-Dataset, we report Recall@1, Recall@2, Recall@3 score, as we want the top-k suggested errors to cover the real API error.

It is noteworthy that our semantic parser outperforms four baselines by a wide margin, regardless of the analytical techniques. We can observe that our parser surpasses others by 12.5%, 10%, 7.75%, and 3.81% for LSTM, Atten-biLSTM, CNN, and Transformer in Recall@1, respectively. In general, SemParser shows the promising Recall@1 score of 0.95, indicating the effectiveness of semantics for failure identification.

The impressive performance can be attributed to several reasons. Firstly, our parser can extract precise conceptualized templates, serving as a basis for downstream task learning. We extract conceptualized templates by replacing the instances with their corresponding concepts while reserving all concepts in the template, based on the observation that instances (e.g., time, len, ID) are more likely to be generated in running time. The template number dramatically decreases after conceptualization, giving the sequence of abstract log messages for primitive learning.

Secondly, the instance-level semantics benefits failure identification. In the case shown in Table 3.7a, "853cfe1b" will be regarded as a meaningless character string by the traditional syntax-based parser; however, SemParser recognizes it as

Table 3.7: Cases for failure identification.

(a) A case for instance-level semantics.

API error	server add volume
Log message	... Cannot 'attach_volume' instance 853cfe1b ...
C-Template	... Cannot 'attach_volume' instance <*server*> ...
CI Pairs	[(server, 853cfe1b)]

(b) A case for message-level semantics.

API error	network create
Log message	... POST /v2.0/networks ...
C-Template	... POST /<*>/networks ...
Concepts	[POST, networks]

a "server" from previous log messages. Therefore, the preserved semantics allows the downstream technique to understand that the original log message is talking about the concept *server*, as well as the concept *attach_volume*, then it will not be hard to infer the API error behind the failure is "server add volume".

Thirdly, our parser provides strong messages-level semantics, clues model in resolving failures. For example, Table 3.7b shows how the semantic parser extracts the concept "network" with the actual API error being "network create". With the help of the concept "network", the model focuses on network errors and filters other server errors or volume errors. To sum up, SemParser benefits the failure identification task by providing message-level semantics and instance-level semantics altogether.

3.6 Discussions

3.6.1 Threats to Validity

Threats to CI pair granularity. Our approach can only discover semantic pairs in a single word. For example, for one Zookeeper log "Connection request from old client /10.10.31.13:4 0061", the extracted CI pair is "(client, /10.10.31.13:40061)" instead of "(old client, 10.10.31.13:40061)". Using "old client" is more precise than "client" to describe this instance. Fortunately, based on our observation, since such multi-word concepts infrequently occur in log messages, using the single-word concept will not alter the semantics too much.

Threats to transferability. Our model mines semantics relying on manually labeled data. The sampled data for annotation and annotation quality both affect its performance. Fine-tuning with new annotation is required to transfer the model across different systems. In this case, we consider that our model can easily adapt to a new system after fine-tuning with a small amount of data (e.g., Our RQ1 shows that 50 annotated logs are sufficient to transfer a model from OpenStack to Hadoop, with 84.6% templates in the test set are unseen).

Threats to efficiency. Despite the fact that the neural network used in our approach can effectively mine semantics, it is not as computationally efficient as other statistical parsers. Nevertheless, the issue can be mitigated by batch operation or GPU acceleration. Moreover, missing identification of

an anomaly can also be very costly. As RQ2 and RQ3 demonstrate SemParser's effectiveness over other parsers in anomaly detection and failure identification, it is worthy of mining such semantics by sacrificing controllable computational efficiency.

3.7 Conclusion

In this chapter, we first identify three key limitations of current log parsers: inadequate attention to informative tokens, missing semantics within log messages, and overlooked relations between log entries. To address these issues, we introduce SemParser, a semantic parser that operates in two phases: a semantics miner to extract explicit semantics from logs and a joint parser that uses domain knowledge to infer implicit semantics. We conduct extensive experiments to evaluate SemParser using six representative system logs for its semantic mining capabilities, achieving an impressive average F1 score of 0.985. Additionally, we assess our approach in two downstream log analysis tasks, namely anomaly detection and failure identification. The experimental results demonstrate that our method significantly outperforms syntax-based log parsers, underscoring the importance of understanding semantics in log analysis.

□ **End of chapter.**

Chapter 4

Anomalous Log Localizer over Software Evolution

4.1 Introduction

Nowadays, intelligent log analytics is designed to manage overwhelming logs [80] for failure troubleshooting, and anomaly detection [81]. Existing automated log analytics can be categorized into two types based on granularity: coarse-grained tasks and fine-grained tasks. Coarse-grained models, such as the anomaly detectors [40] and failure predictors [82], detect (or predict) anomalies given the logs from a period of time. Taking anomaly detection as an example, the model accepts a session of logs to determine whether an anomaly exists in this session. Although the coarse-grained models show promising results in open datasets, they provide limited evidence of failure diagnosis for software maintainers. On the other hand, fine-grained tasks aim to further identify the individual/single anomalous

logs within a session showing possible interpretations of the failure [83, 84, 85]. Even if coarse-grained models free maintainers from inspecting massive log lines, it is still time-consuming to analyze hundreds of log lines within a session to find the anomalous log for troubleshooting [86]. To ease the burden of software maintainers, we focus on this more challenging yet significant task, individual anomalous log identification, in this chapter.

An *anomalous log* signals an anomaly in the system, such as network error [86]. The following example shows a log message that may indicate a connection problem caused by a network fault within the system:

Container launch failed for container_32h: Connection refused.

Anomalous logs are crucial for diagnosing failures, but they are often accompanied by numerous normal logs, which can be overwhelming for maintainers. To distinguish them from normal run-time logs, existing studies [84, 85, 87, 88] constructed a *reference model* from training log sequences and then identified which log violated the reference model. Specifically, they abstracted log event sequences into a directed graph via either a finite state machine (FSM) [84, 85, 87] or causal dependencies [88] as the reference model. Subsequently, any deviations from this model would be regarded as an indication for anomaly and marked for troubleshooting.

However, both FSM-based and causal graph-based approaches following the *closed-world assumption* suffer limitations for pro-

cessing the unseen data. However, after the initial version is released, software experiences continual development to fulfill customers' demand, to fix bugs, and to extend to new functionalities, which is well-known as *software evolution* [89, 90]. Previous studies pointed out that logging statements change over software evolution is so pervasive that around 33% of the log are revised as after-thoughts [91, 92]. The changed logging statements during the evolution activities raise challenges for existing approaches:

(1) Parsing errors. Log parsers extract static events (e.g., *Container launch failed for <*>: Connection refused.*) and dynamic parameters (e.g., *container_32h*) from log messages. However, as discussed in Section 4.2.2, parsers may misalign revised log events in evolving software versions, causing log parsing errors. These parsing errors further downgrade the subsequent log analytics performance. ***(2) Evolving events.*** Even if state-of-the-art parsers work as expected, software evolution brings new logging statements or paraphrases old logging statements, which we refer as *evolving events* in this chapter. ***(3) Unstable sequences.*** Apart from log events, the log sequences from running identical jobs can vary, named *unstable sequences*. Such variation can be caused by interleaving logs produced from multiple threads [84]. Moreover, software evolution may alter the function invocation sequences, leading to new sequential patterns.

While solutions to the first two challenges still remain un-

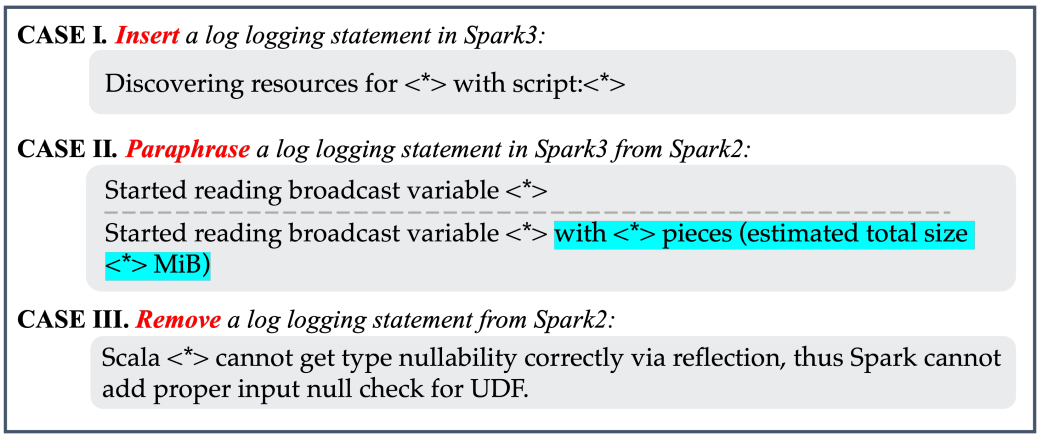


Figure 4.1: Evolving logging statement cases for Spark2 and Spark3.

explored, there have been several attempts to handle the third challenge. For example, previous study [93] tried to resolve the interleaving logs by considering multiple predecessors and successors of a log event, instead of just the direct ones. Another study [94] mitigated unstable sequences challenge by learning causal relationships between event pairs from historical data. Nevertheless, none of the existing approaches considered the software evolution scenario, which can negatively impact the performance of identifying anomalous logs if left unaddressed.

To address the above challenges, we propose an unsupervised anomalous log identification solution over software evolution (**EvLog**). The design of our approach is based on two insights: *1) the majority of logs are normal in a healthy system; and 2) the anomalous logs are unknown a priori because we cannot iteratively inject all kinds of failures.* In particular, we design EvLog with two steps. The first step aims to tackle the *parsing errors* and the *evolving events* issues. We derive multi-

level representations directly from logs to prevent introducing *parsing errors*. The representations at different levels undertake different functions: 1) the semantic-rich representation aims to fully retain semantics from log messages, which is extracted by pre-trained language models; and 2) the abstract representation to align similar logs across software evolution, which is derived from the hierarchical clustering approach. Such multi-level representations maintain the pertinent semantics while leaving out unnecessary trifles to address the *evolving events* issue.

In the second step, we address the *unstable sequence* issue by constructing an anomaly discriminator with an attention mechanism. The core idea is to learn a transformation function (e.g., neural networks) that embeds normal log features (source domain) to stay close (enclosed in a hyper-sphere) to a target domain, then the logs that are largely distant from this hyper-sphere are considered as anomalous ones. Specifically, EvLog constructs log features for each single log and its surrounding log contexts based on multi-level representations. It then applies neural networks to discriminate the anomalous logs instead of rigorously comparing new sequences with existing ones. Once trained, EvLog can be directly applied to a future software version without any fine-tuning.

Our new approach is evaluated using two realistic datasets (i.e., LogEvol) and a synthetic dataset (i.e., SynEvol) to simulate logging evolution. The experiment results illustrate that EvLog reaches a promising average F1 score of 0.955 and

0.847 in intra-version identification and inter-version anomalous log identification on two representative system logs, respectively.

To conclude, the contribution of this chapter is threefold:

- We empirically identify three challenges (i.e., parsing errors, evolving events, unstable sequence) brought by software evolution for anomalous log identification, which has never been properly addressed before.
- To overcome the above challenges, we develop EvLog, an unsupervised anomalous log identification approach with a multi-level representation extractor and an anomaly discriminator. To our best knowledge, EvLog is the first solution to tackle the problem of identifying anomalous logs over software evolution.
- By evaluating EvLog on real system log datasets and a synthetic dataset, we show our approach can effectively identify anomalous logs across different software versions without fine-tuning or manual labeling.

4.2 Motivating study

4.2.1 How do logging statements evolve?

Developers may modify logging statements when updating the software, producing unseen log messages in system run-time for maintenance. To examine how logging statements evolve

Table 4.1: Logging evolution ratio between Spark2 and Spark3.

Percentage	Unchanged	Inserted	Paraphrased	Removed
Log message	91.16%	0.07%	8.75%	0.02%
Logging statement	76.12%	12.69%	1.49%	9.70%

during software updates, we analyze Spark, an open-source cluster computing system for the parallel processing of large-scale data. In particular, we run benchmark workloads in Spark 2.4.0 (denoted as Spark2) and Spark 3.0.3 (denoted as Spark3) with details shown in Section 4.5.1 and compare the collected log messages.

We categorize the change of logging statements into three types: *insert*, *paraphrase* and *remove*. We show three cases in Fig. 4.1, where "<*>" refers to the dynamic parameters generated in running time. In Case I, a new logging statement is added in Spark3 to indicate the attached resources. In Case II, the logging statement is paraphrased by adding information on the number of pieces and the estimated size of the variable to gain a deeper understanding of the system performance. In Case III, a logging statement is removed from Spark2 due to the deprecation of "*UserDefinedFunction*" in Spark3.

Table 4.1 displays the statistics of the three types of changes on collected log messages and logging statements. It is observed that nearly 24% logging statements are changed from Spark2 to Spark3, resulting in almost 10% changed log messages. Although 12.69% and 9.70% logging statements are inserted or

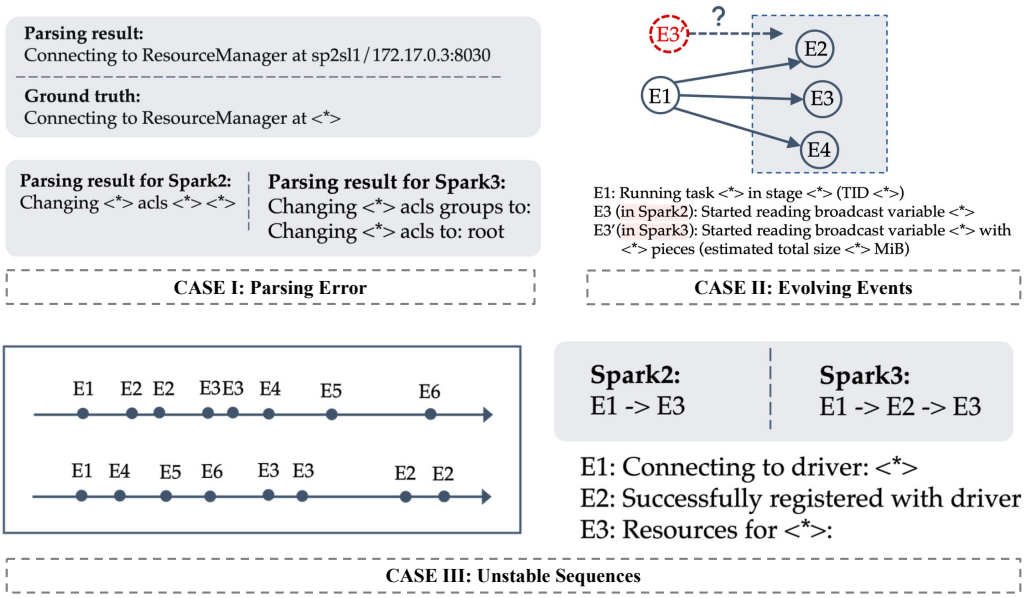


Figure 4.2: Three challenges brought by software evolution. E1, E2, etc., represent different log events.

removed, respectively, they only make up less than 0.1% collected logs, meaning they appear in a low frequency. However, the high proportion of paraphrased logs implies developers are likely to modify the commonly used logging statements. To conclude, logging statements change over software evolution. The non-negligible amount of changes motivates us to reckon with the software evolution issue.

4.2.2 How does evolution raise challenges for anomalous log identification approaches?

Parsing errors

Log parsers extract constant strings (i.e., events) and runtime parameters from log messages. However, existing log iden-

tification models *only* use the extracted events and do not consider the original log messages. This can be problematic because log parsers can introduce errors, and the evolution of logs over time can make parsing even more challenging [95]. CASE I in Fig. 4.2 displays two parsing mistakes from a widely-used parser, Drain [20], where <*> denotes parameters. The top one is caused by confusing parameters with constant strings, and the bottom one shows inconsistent parsing results in Spark2 and Spark3. Since current log parsers are parameter-sensitive and not versatile enough [17], and the hyper-parameters that work well for one software version may not be suitable for others. Since systematic log analytics should operate on raw log messages, it is essential to find ways to avoid parsing mistakes.

Evolving events

Log identification models also face a challenge in dealing with evolving events. Typically, these models detect anomalous logs by examining whether the actual next log is in line with the predicted next logs based on contextual information. The idea works well when all the events are known; however, if the actual next log is an unseen event, it can never be matched with any predicted next logs. For instance, in CASE II of Fig. 4.2, event E3' cannot be matched with the predicted logs since it is a paraphrase of event E3. According to its decision logic, such inconsistency leads to a significant issue where all unseen events are treated as false positives. This issue becomes severe

for all existing log-based approaches considering that 8.75% of the collected logs have been paraphrased.

Unstable sequences

Ideally, we expect that the log message sequences perfectly match the execution sequences of a program. However, there are situations where log messages from different threads can interleave, resulting in what we refer to as "unstable sequences". Additionally, introducing new logging statements in a software update can create new log events during run-time, leading to sequential pattern changes. As shown in CASE III of Figure 4.2, unstable sequences can be caused by interleaving logs [84] and new log events from software evolution. To resolve the issue, identifying relevant and informative log messages in a sequence is of great essence.

In summary, our empirical findings suggest that logging evolution can affect existing models in three ways: the potential parsing errors, the evolving events, as well as the unstable sequences. These influential factors have never been explored, yet their impact can hardly be ignored.

4.3 Problem illustration

In this chapter, we consider the anomalous log identification problem as in the literature [84, 85, 96], which enables pinpointing a collection of fault-indicating anomalous logs [86]. Given

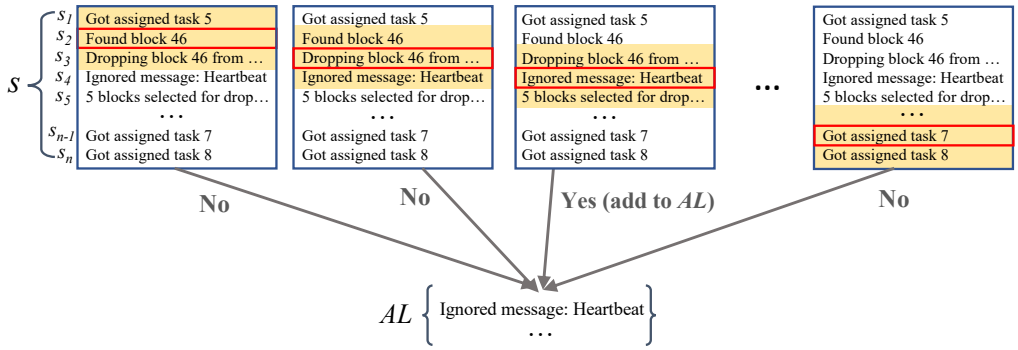


Figure 4.3: Anomalous logs localization problem illustration.

a sequence of log messages $s = s_1, s_2, ..., s_n$, the task asks the model to find a set of anomalous logs $AL = \{s_i | 1 \leq i \leq n\}$ within the message sequence. Compared with the anomaly detection task that determines whether a problem exists in a session (session-based), anomalous log identification is a more fine-grained and challenging task that needs to localize individual fault-indicating logs (message-based). We use *context* to represent the surrounding logs of a specific log (named as the *center log*) and analyze whether the log is anomalous based on its context.

To resolve the subset AL, we check every individual log, that is, for all $s_i \in s$, the model determines whether the center log s_i is an anomalous log in the given context of s_i. We will add the center log into AL if it is considered anomalous. Fig. 4.3 shows the identification process, where the center log and its corresponding context are highlighted with a red rectangular and yellow background, respectively.

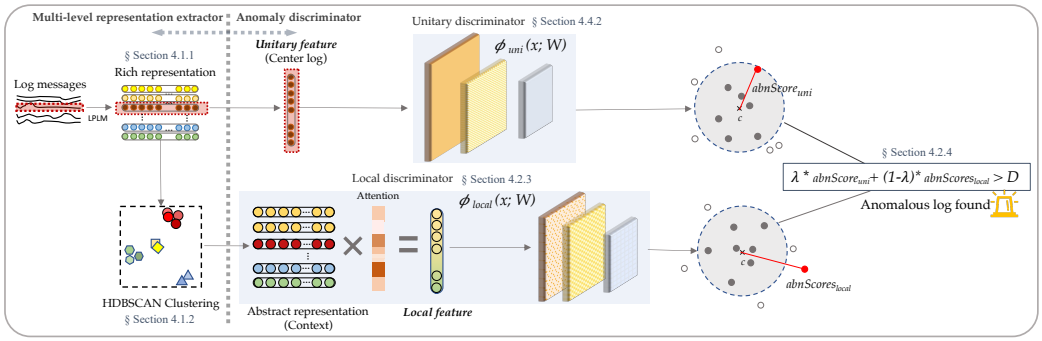


Figure 4.4: EvLog with a multi-level representation extractor and an anomaly discriminator.

4.4 Approach

This section introduces our novel approach, called EvLog, shown in Fig. 4.4, in tackling the anomalous log identification challenges over software evolution. EvLog has two components, i.e., a *multi-level representation extractor* to derive multi-level robust log representations, followed by an *anomaly discriminator* with the attention mechanism to pinpoint the anomalous logs. In particular, the multi-level representation extractor targets extracting *rich representations* as informative as possible and *abstract representations* to capture high-level commonalities among similar logs. Then these representations are fed into the anomaly discriminator to automatically localize the anomalous logs in an unsupervised manner.

4.4.1 Multi-level representation extractor

We exploit multi-level representations with various information from log messages to semantically understand them.

This section illustrates how to extract multi-level semantic representations, that is, a *rich representation* and an *abstract representation*. The low-level rich representation provides a concrete understanding of a certain log. In contrast, the high-level abstract representation captures the commonality of logs with similar semantics, regardless of their slight differences (e.g., parameters difference, revised log events).

Rich representation

Semantics in both log events and their corresponding parameters has advantageous for log analysis [11, 97]. To obtain informative representations from logs with respect to their semantics, we fine-tune a pre-trained language model [98] (PLM) on our collected log datasets.

The PLMs have shown the powerful semantic encoding ability for many software engineering tasks, such as log-based anomaly detection [99] and code comprehension [100]. In our work, to overcome potential parsing errors and to make the best usage of information inside log messages, EvLog acquires domain-specific semantic representations via PLMs. On one side, system logs share some fundamental knowledge with natural languages since humans write logging statements. After being trained on a large corpus, the PLMs learn more information about word senses, not limited to system logs. On the other side, we notice that these PLMs are not sufficient for domain-specific tasks due to the knowledge gap. Hence, we fine-tune

the massive language models to further capture domain-specific semantics. In specific, we employ the widely-used masked language modeling strategy [30, 101, 102] to fine-tune the PLMs, by randomly masking 10% tokens in each log and asking the model to predict the masked tokens.

Specifically, given a log message x, the rich representation x_{rich} is designed to capture its detailed semantics. This parser-free representation extractor accepts log messages instead of events, allowing it to get away from potential parsing mistakes.

Algorithm 2 Abstract representation acquisition.

Input: Rich representation to be clustered $E = [e_1, e_2, ..., e_n]$
Output: Abstract representation $C = [c_1, c_2, ..., c_n]$

```
C = []
Centroid={}
E' = PrincipalComponentAnalysis(E)
ClusterIds = HDBSCAN(E')
"""Compute centroid for each cluster"""
for all ClusterID from 1 → Set(ClusterIds) do
    Centroid[ClusterID] = Mean(E'[ClusterIds==ClusterID])
end for
"""Compute abstract representation"""
for all i from 1 → n do
    C.append(Centroid[ClusterIds[i]])
end for
```

Abstract representation

Apart from the rich representation, we also extract a high-level semantic representation, x_{abs}, that remains stable on similar log events over logging evolution. To this end, we develop a cluster-based approach on top of the rich representa-

tions. Previous studies [103, 104] have demonstrated the effectiveness of clustering approaches in grouping similar texts together based on their intrinsic characteristics. Motivated by theirs, we also employ a cluster-based approach to group log messages. Specifically, we adopt the idea from the previous log clustering study [105] using Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [106], whose efficiency and effectiveness has been presented in many domains [107, 108]. Compared to other clustering approaches, HDBSCAN inherits two special advantages for our scenario: (1) It can automatically extract the "dense" cluster without predefining the number of clusters (e.g., Kmeans [109]), which is important in the case that we may never know the number of clusters of logs. (2) HDBSCAN has a few parameter numbers, and its robustness to parameter choice [105, 106] makes it versatile for diverse log data.

Eventually, the abstract representation x_{abs} for each log message x is the centroid of its corresponding cluster by averaging all points (logs) belonging to the cluster. Algorithm 2 shows the abstract representation computation process.

4.4.2 Anomaly discriminator

This section illustrates how we pinpoint anomalous logs from the acquired rich and abstract representations. In particular, for each input log, the unitary discriminator processes

the log, and the local discriminator processes the log's context. Two processed results are integrated as the final output.

Basic idea

Existing unsupervised log-based reference sequences models [40, 84, 26] build a reference model from training data and check whether the testing log violates the prediction from the model. Unfortunately, these models are ineffective at handling evolving events over software evolution. Moreover, the anomalous logs are unknown a priori because we cannot iteratively inject all types of faults. Thus, we propose a different approach to handle the problem by learning the "normality" of the normal log features instead of predicting the subsequent events.

Motivated by the Support Vector Machine [110] (SVM) that learns a hyperplane to separate data, our idea is to develop a neural network that learns a hyper-sphere to separate normal logs and anomalous logs. The neural network maps log features (in the source domain) to a target domain where normal features stay as close as possible (enclosed in a hyper-sphere). We measure the distance between a mapped log feature and the center of the hyper-sphere as *normality* (e.g., grey circle in Figure 4.4), with logs far from the center being considered anomalous due to deviating from the normality. In this way, logs with evolving events can be transformed into the target domain where they are close to the previous semantically similar logs, minimizing adverse effects on the anomaly discriminator

results. For example, if a normal log is paraphrased during software evolution, the evolved log with similar semantics will be mapped within the normality. This new approach in localizing anomalous logs is superior via two advantages: 1) It delivers better performances than other traditional methods due to the neural network's proven learning ability. 2) Our approach frees humans from labor-intensive labeling since it can learn the normality naturally in an unsupervised manner from large-scale normal logs that can be easily collected from stable software.

Specifically, the goal is to train a neural network model (mapped features from the source domain to the target domain) while minimizing the hyper-sphere volume that encloses the normal data features in the target domain. In this way, the model is forced to learn implicit semantics since it must map the normal log features closely to the hyper-sphere's center. Thus the unseen log events with similar semantics can also be embedded close in the target domain. To achieve the above goal, the objective function is:

$$J = \min_{W} \quad \frac{1}{n}\sum_{i=1}^{n} \|\phi(x_i; W) - c\|^2 + \frac{\alpha}{2}\|W\|^2, \tag{4.1}$$

where $\phi(x_i; W)$ refers to using the model ϕ with its parameters W to map each input sample $x_i \in x$ to a hyperspace $\mathbb{R}^n$; $c \in \mathbb{R}^n$ refers to the hyper-sphere center; the last term serves as a regularization term with weight α to avoid over-fitting. The objective function forces the normal data features to stay close

to the center c. Theoretically, the mapping model ϕ can be replaced by any neural network architecture, demonstrating the extensibility of our approach. The following two sections show how we develop an appropriate neural network for mapping multiple features. We then describe how to integrate the mapped features for identifying anomalous logs in Section 4.4.2.

Unitary discriminator

We first look into single logs, as the single log that contains negative words (e.g., "failure" and "error") usually indicates an anomaly. The unitary discriminator works on rich representation of individual logs, aiming to map normal logs to a hypersphere that describes the normality. The motivation behind the unitary discriminator is that, the negative terms in anomalous logs exhibit significantly different semantics than words in normal logs (e.g., "running", "success"). These anomalous logs' features will be mapped far away from the center of the hypersphere; thus they are considered as normality-deviating ones. To this end, we adopt the strong learning ability from neural networks and build the unitary discriminator (ϕ_{uni}) with a two-layer feed-forward neural network denoted as $FFNN$. We describe the architecture as follows:

$$\phi_{uni}(x_{rich}; W_{uni}) = FFNN_b((FFNN_a(x_{rich})), \tag{4.2}$$

where x_{rich} refers to the rich representation containing full log semantics (i.e., *unitary feature*) of the center log x.

Local discriminator

Looking into one individual log is not sufficient to comprehensively understand the running status, so it is noteworthy to exploit its contextual information. On the one hand, it is pointed out that different logs possess different importance [23]. For example, some miscellaneous logs regularly appear regardless of what job the system is running, whereas other logs provide richer guidance for analysis. On the other hand, log data transmission, collection, and software evolution affect synchronization temporally, leading to unstable sequences. To focus on beneficial logs and leave the uninformative logs out, we leverage the attention mechanism [111] to focus on beneficial logs. In the local discriminator, we use the center log and its contexts to acquire a *local feature* against unstable sequences and then learn the normality of such a local feature.

Given a center log x, we construct its context representation x_{ctx} by forming its abstract representation of context as a matrix. Then we compute the weights across the context by the attention mechanism [38], allowing the model to learn the importance of surrounding logs, thus addressing the unstable sequence issue. Specifically, given a center log x_{rich} as query and its context x_{ctx} as value, we compute the weighted context representation as the

local feature (denoted as x_{local}) as follows:

$$\begin{aligned} x_{local} &= softmax(\frac{x_{query}x_{ctx}^{T}}{\sqrt{d_k}})x_{ctx}, \\ x_{query} &= FFNN_c(x_{rich}), \end{aligned} \tag{4.3}$$

where d_k refers to the dimension of x_{ctx}, and $FFNN_c$ transforms x_{rich} to the target domain that shares the same dimension with x_{ctx}.

After that, another two-layer neural network with an activation function is applied to the local feature x_{local} for learning normality from contexts. To sum up, we describe the network for the local discriminator (ϕ_{local}) as in Equation 4.4:

$$\phi_{local}(x_{rich}, x_{ctx}; W_{local}) = FFNN_e((FFNN_d(x_{local}))). \tag{4.4}$$

Integration

The unitary discriminator learns normality for individual logs, whereas the local discriminator learns the context normality in running status. To fully exploit these two different information sources, we propose the total objective function with a weighted sum in Equation 4.5 to simultaneously optimize two sub-discriminators:

$$J_{total} = \lambda * J_{uni} + (1 - \lambda) * J_{local}, \tag{4.5}$$

where J_{uni} and J_{local} are the functions defined in Equation 4.1 for unitary discriminator ϕ_{uni} and local discriminator ϕ_{local}, re-

spectively. The objective functions allow two discriminators to learn the normality by minimizing their hyper-sphere volume.

The distance between a log message (after mapping by discriminators) to the hyper-sphere center measures the degree of normality of the log. We apply an *abnormal score* ($abnScore$) to describe how the log deviates from the normality, which is the weighted sum of the abnormal sub-scores from two independent discriminators. The abnormal sub-score $abnScore_i$ is defined by the Euclidean distance from the feature embedding to its corresponding hyper-sphere center, denoted by Equation 4.6:

$$\begin{aligned} abnScore &= \lambda * abnScore_{uni} + (1-\lambda) * abnScore_{local}, \\ abnScore_i &= \|\phi_i(x; W_i) - c_i\|^2, i \in \{uni, local\}. \end{aligned} \tag{4.6}$$

The center log is eventually predicted as an anomaly if and only if its abnormal score is larger than the threshold D (Equation 4.7). We put all identified logs into the anomalous log set AL, which provides detailed clues to troubleshoot the system conveniently.

$$\text{center log} = \begin{cases} NormalLog, & abnScore \leq D \\ AnomalousLog, & abnScore > D. \end{cases} \tag{4.7}$$

4.5 Implementation Setup

4.5.1 Data collection

Infrastructure

Despite many log datasets being collected for research [40, 112, 113, 114], there is no open-source dataset documenting the evolution process. To fill this blank, we collect a new dataset LogEvol containing log data from the most widely-applied data processing system Spark [115] (LogEvol-Spark) and Hadoop [116] (LogEvol-Hadoop), across different versions.

To this end, we employ HiBench [117], a big data benchmark suite, to generate logs by running a set of workflows in Spark and Hadoop, respectively, from basic to sophisticated scenarios. In total, we run 22 workloads (shown in Table 4.2) on the systems to cover more practical scenarios, while other existing datasets [114, 112] are collected from simply running two straightforward tasks (i.e., page rank and word count).

Then, we repeat the procedure of running workloads using different versions of the software systems mentioned above, covering a wide time range and various data size scales. We select two typical versions of Spark (i.e., Spark2.4.0 and Spark3.0.3) and Hadoop (i.e., Hadoop2.10.2 and Hadoop3.3.3), as they have undergone systematic changes with significant differences.

Table 4.2: Workloads for collecting LogEvol.

Categories	Workloads
Micro task	Sort, Wordcount, etc.
Machine learning	Bayes Classification, Gradient Boosted Trees, etc.
SQL	Aggregation, Join, Scan etc.
Websearch	Pagerank
Graph	NWeight, Graph Pagerank
Streaming	Repartition

Fault Injection

We inject 18 typical types of faults into the system to simulate real-world production failures: (1) *Process suspension*: Suspend processes in multiple types of nodes, one at a time; (2) *Process killing*: Kill processes in seven types of nodes, one at a time; (3) *Resource occupation*: Inject other computation programs to occupy CPU and memory; and (4) *Network faults*: Establish network faults such as losing packages, network delay, and connection lost.

In total, we collect 6,703,460 log messages (# Logs)with recognized 69,513 anomalous logs (# Anomalous logs), whose statistics are shown in Table 4.3. To guarantee dataset quality, anomalous logs are discussed and annotated by two engineers who have two years of development experience with the Spark system. Since annotators have read a lot of logs in their development experience, they can provide reliable annotations.

Table 4.3: Statistics of LogEvol.

	Spark2	Spark3	Hadoop2	Hadoop3
# Logs	931,960	1,600,273	2,120,739	2,050,488
# Anomalous logs	1,702	2,430	35,072	30,309

4.5.2 Implementation details

In the multi-level representation extractor, we use BERT [98] as the pre-trained language model and fine-tune it with Hugging Face [118]. For the anomaly discriminator, we specifically choose leaky ReLU [119] as the activation function between two layers in the perceptron so as to resolve the "all-zero-solution" issue [120]. We set the dynamic threshold D to be 0.4 times of the maximum normality (hyper-sphere radius) in the training data in intra-version and 0.6 times for the inter-version. We set λ=0.5 in the experiments as the unitary and local features both serve as an important role in fault localization. We randomly split the collected logs into training, development, and testing sets for each software version with a standard 8:1:1 splitting. In contrast, the training set only contains logs collected in the fault-free periods as we assume the majority of logs are normal in a healthy system. All experiments are conducted in 64-bit CentOS 7 with Intel(R) Xeon(R) CPU and 1 GeForce RTX 2080 GPU for acceleration. It takes approximately 15 seconds for the anomaly discriminator to train in an epoch.

4.6 Experiments

To evaluate the effectiveness of EvLog, we investigate three research questions:

RQ1: How effective is EvLog in identifying anomalous logs?

RQ2: How effective is EvLog in resolving evolving events and evolving sequences?

RQ3: How effective are different components in EvLog?

4.6.1 Experimental settings

Baselines

We select four unsupervised log-based analytics as baselines, including two anomalous log identification models and two anomaly detection (AD) models. LogAnomaly and LogSed are the state-of-the-art AD and log localization models, respectively. The reason why we choose AD baselines is, they both work for anomaly analysis with different granularities (i.e., coarse-grained and fine-grained); For AD models, we use the historical sequences to train a reference model and predict the next event as in the original paper. The actual next event that outside the predicted list of candidate events will be considered as anomalous due to its deviation from the reference model. In our implementation, we use the state-of-the-art log parser [17] to extract events for all baselines as they all require the parsing phase. In specific, we briefly characterize four baselines as follows.

- LOGAN [96] built the diagnosis system by constructing a directed graph from normal log event sequences. Then any of the test time logs that deviate from the directed graph will be considered anomalous.
- LogSed [84] addressed the interleaving logs problem by developing a two-stage approach to mine the important sequential relationship from log sequences. The incoming log message that violates that sequence will be regarded as anomalous.
- DeepLog [40] utilizes an LSTM network to capture sequential information of log data. It accepts the sequence of log event IDs to predict the next log, the actual log ID outside prediction will be regarded as an anomaly.
- LogAnomaly [26] is proposed for unsupervised anomaly detection with semantic representation for log events via an attention-based LSTM network.

Dataset

EvLog is evaluated on two datasets: a software evolution dataset collected from two representative systems (LogEvol) and a synthetic dataset (SynEvol).

LogEvol. Although existing study [90] analyzed the evolution process of Hadoop, and mentioned the importance of new-emerging log messages [23], there lacks a public dataset showing

how logs change during software evolution. Hence, we evaluate our approach and compare it with baselines on the data collected in Section 4.5.1. To our best knowledge, LogEvol is the first publicly accessible log dataset recording software evolution activities.

SynEvol. To evaluate how EvLog resolves the challenges of unseen events and unstable sequences separately (Note that EvLog is parser-free), we build a synthetic dataset based on the collected Spark2 logs in LogEvol (denoted as LogEvol-Spark2). Following previous work [23], we inject unseen events and unstable sequences into LogEvol-Spark2 to simulate the real-world software evolution as follows:

1. Unseen events are introduced by logging statement alteration in software updates. Developers may paraphrase or insert logging statements for customized functionalities. Since EvLog does not use a parser, we simulate the change by creating a set of synthetic log messages via (1) inserting, (2) deleting, or (3) replacing a common word from an original log message. Such modification is more likely to reflect the changes in log events.

2. Unstable sequences occur both in log generation and log evolution. Logs from multiple transaction flows may be interleaving, making the direct predecessor or successor of a certain log different. Moreover, log evolution is likely to cause variations via function ensemble or the changes of function invocation sequences. To construct synthetic sequences, we randomly remove a few unimportant log messages (far away from anomalous logs),

repeat some log messages several times, or shuffle the log messages in a short time.

We inject the evolving events and unstable sequences into the original dataset, denoted as SynEvol-Events and SynEvol-Seqs correspondingly. The injection follows specific ratios. We inject the 5%, 10%, 15%, 25%, and 30% synthetic log messages and log sequences to LogEvol-Spark2, to observe how EvLog reacts to unseen and unstable sequences, respectively.

Evaluation metrics

To evaluate the effectiveness of EvLog in anomalous log identification, we apply Precision, Recall, and F1-score as evaluation metrics. In particular, Precision (P) is the percentage of logs that are correctly identified anomalous overall identified logs ($\frac{TP}{TP+FP}$). Recall (R) is the percentage of logs that are correctly identified anomalous over logs belonging to anomaly logs. ($\frac{TP}{TP+FN}$). F1 score (F1) is the harmonic mean of Precision and Recall ($2 * \frac{P*R}{P+R}$), where TP refers to the amount of anomalous logs that is correctly identified, FP refers to the number of normal logs that are wrongly predicted as anomalous, and FN means the number of anomalous logs that are identified as the normal logs.

4.6.2 RQ1: How effective is EvLog in identifying anomalous logs?

To evaluate how effective EvLog can pinpoint the anomalous logs with software evolution activities, we conduct experiments on our dataset LogEvol. The experiments engage two different settings: 1) Intra-version: identify the anomalous logs on the same system it is trained (e.g., Spark2 → Spark2); and 2) Inter-version: identify the anomalous logs in a different system version after training (e.g., Spark2 → Spark3).

We can draw two observations from the experimental results shown in Table 4.4. First, EvLog delivers an overall satisfactory performance under the intra-version setting with the average F1 score of 0.967 in Hadoop and 0.944 in Spark, which is comparable with other baselines. The experimental results indicate that EvLog can learn the normality and effectively identify anomalous logs from log sequences. Besides, we find that deep learning-based approaches perform better than FSM-based approaches, demonstrating that neural networks are capable of capturing intrinsic sequential patterns and log semantics.

Second, in the inter-version scenario, EvLog significantly outperforms all baselines by a wide margin, demonstrating its effectiveness and robustness in software evolution. We observe that all baseline performances drastically drop (approximately an F1 score of 0.55) while EvLog achieves an average F1 score of 0.87 for Hadoop, which contains 3% new logs. In the case of

Spark, where logging statement paraphrasing and insertion via software updating account for 10% logs, baseline performances are further significantly downgraded.

(a) Experimental results in identifying anomalous logs for LogEvol-Hadoop.

Intra-version						
Baseline	Precision	Recall	F1	Precision	Recall	F1
LOGAN	0.894	0.995	0.942	0.899	0.988	0.942
LogSed	0.910	0.995	0.951	0.925	0.986	0.955
DeepLog	0.913	0.985	0.947	0.926	1.000	0.961
LogAnomaly	0.926	0.994	0.958	0.939	0.988	0.963
Name	0.945	0.982	**0.963**	0.952	0.988	**0.970**
Inter-version						
Baseline	Precision	Recall	F1	Precision	Recall	F1
LOGAN	0.360	0.988	0.528	0.376	0.995	0.546
LogSed	0.371	0.988	0.540	0.390	0.993	0.560
DeepLog	0.386	0.999	0.556	0.410	0.971	0.576
LogAnomaly	0.389	0.998	0.560	0.407	0.995	0.578
Name	0.770	0.941	**0.847**	0.857	0.913	**0.884**

(b) Experimental results in identifying anomalous logs for LogEvol-Spark.

Intra-version						
Baseline	Precision	Recall	F1	Precision	Recall	F1
LOGAN	0.798	0.943	0.865	0.967	0.870	0.916
LogSed	0.842	0.914	0.877	0.907	0.923	0.915
DeepLog	0.862	0.952	0.905	0.858	0.976	0.914
LogAnomaly	0.931	0.939	0.935	0.898	0.947	**0.922**
Name	0.970	0.974	**0.972**	0.944	0.888	0.915
Inter-version						
Baseline	Precision	Recall	F1	Precision	Recall	F1
LOGAN	0.016	0.943	0.032	0.012	0.943	0.024
LogSed	0.013	0.917	0.026	0.010	0.914	0.020
DeepLog	0.017	0.947	0.032	0.014	0.909	0.026
LogAnomaly	0.020	0.923	0.038	0.017	0.948	0.034
Name	0.922	0.700	**0.795**	0.920	0.812	**0.863**

Table 4.4: Experimental results in identifying anomalous logs (train set→test set).

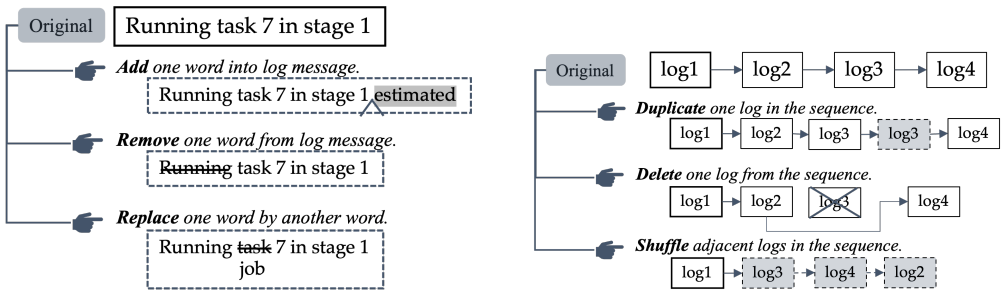
We analyze the reasons below. First, log parsers will generate unseen events when they encounter these new logs. Then, directed graph approaches (i.e., LOGAN, LogSed) and AD mod-

els (i.e., DeepLog, LogAnomaly) fail in matching these unseen events to any current events or predicted subsequent-event candidates. Consequently, current baselines label all unseen events as anomalous, leading to high false-positive rates (i.e., low precision). On the contrary, EvLog uses hierarchical clustering to learn abstract representations of log messages and aligns unseen events to similar past ones. In this way, the modified log message shares the consistent representation with its old one, so as to reduce false positives and improve anomalous log identification performance. Note that false positive rates in anomalous log identification, although not as severe as false negative cases, can still be problematic as they can lead to excessive work for maintainers.

Answers to RQ1: EvLog can effectively identify anomalous logs under both intra- and inter-version settings, all the while demonstrating its robustness and stability across software evolution activities.

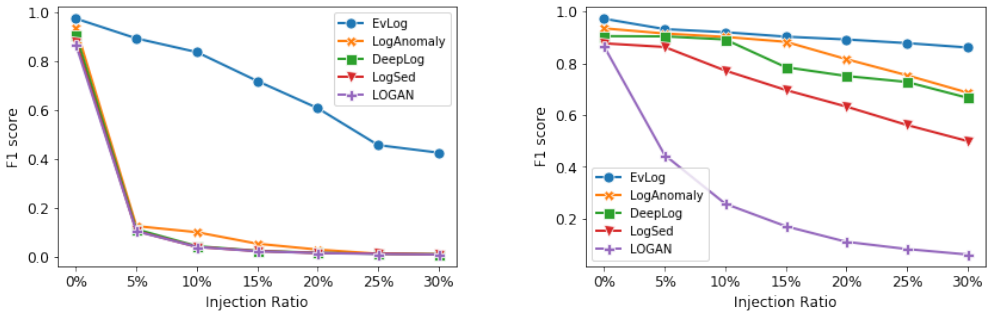
4.6.3 RQ2: How effective is EvLog in resolving evolving events and evolving sequences?

We overcome the parsing errors challenge naturally since our model is parser-free. Thus, we are interested in how well our model addresses the other two challenges, i.e., evolving events and evolving sequences. To do so, we measure EvLog on the synthetic dataset, including SynEvol-Events and SynEvol-



(a) SynEvol-Events example (b) SynEvol-Seqs example

Figure 4.5: Examples of synthetic dataset SynEvol.



(a) Experiments in SynEvol-Events (b) Experiments in SynEvol-Seqs

Figure 4.6: Experiment results on the synthetic dataset SynEvol.

Seqs. Fig. 4.5 shows the examples in the dataset.

Fig. 4.6 shows the F1 scores of baselines, and ours under the injection ratio varies from 0% to 30% (the injection ratio of 30% means 30% of the original dataset was replaced by the synthetic one). The results demonstrate our approach's effectiveness in both evolving events and sequences compared with baselines. In particular, EvLog achieves the F1 scores of 0.42 and 0.86 in SynEvol-Events and SynEvol-Seqs, even though the synthetic dataset replaces 30% of the messages and sequences in LogEvol-Spark2, respectively. We attribute the advantage to the extracted multi-level semantics, as well as the stability of

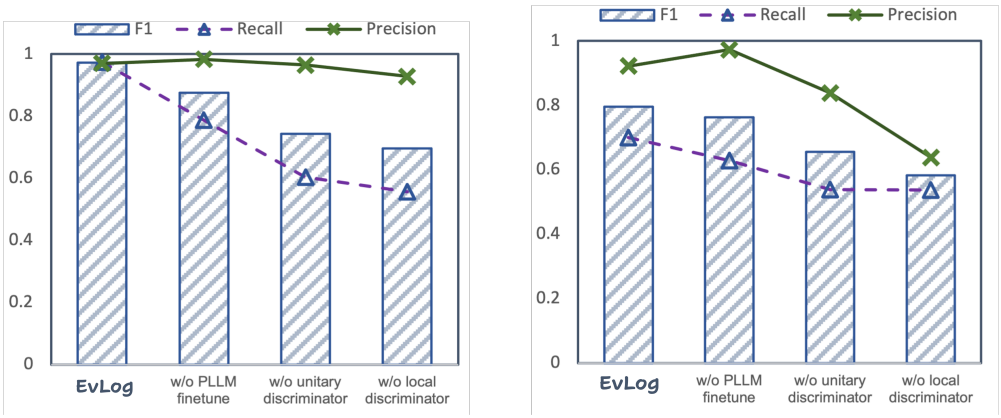
the normality learned by the anomaly discriminator.

Another observation is that log changes are more likely to damage the model's performance than sequence changes. This is because log changes bring unseen events to the trained model, posing greater difficulties for the model to deal with. On the one hand, our approach can still perform stably with evolving events due to EvLog's unique clustering mechanism that aligns old events with new ones. This result is in line with our experiments in RQ1 that all baselines perform unsatisfactorily during version transferring, as many events are changed from Spark2 to Spark3. On the other hand, in terms of the unstable sequences, we conclude that neural networks (used by LogAnomaly, DeepLog, and ours), particularly those with the attention mechanism (used by LogAnomaly and ours), force the model to pay attention to the informative log messages while getting rid of unstable sequences.

Answers to RQ2: EvLog reveals the robustness across different types of changes happening in software evolution, owing to its multi-level semantics extractor and attention mechanism.

4.6.4 RQ3: How effective are different components in EvLog?

This research question investigates an ablation study on how much each design contributes to EvLog. Specifically, we remove each focused component one at a time and conduct ex-



(a) Spark2 → Spark2 (b) Spark2 → Spark3

Figure 4.7: Effectiveness of finetuning, unitary discriminator and local discriminator, respectively (train set → test set).

periments on LogEvol-Spark. In particular, we remove (1) the fine-tuning phase in PLM, (2) the unitary discriminator, and (3) the local discriminator, separately.

Our experiments in Fig. 4.7 show that all three components of EvLog contribute to its effectiveness. The reasons based on the experiments are elaborated as follows. First, fine-tuning on the log dataset helps EvLog capture precise semantics by bridging the knowledge gap between Spark domain knowledge and common sense knowledge. Second, the unitary discriminator, which operates on individual logs, learns the commonality of single normal logs. Third, removing the local discriminator largely degrades the overall performance since it provides a more comprehensive view of the contextual running status.

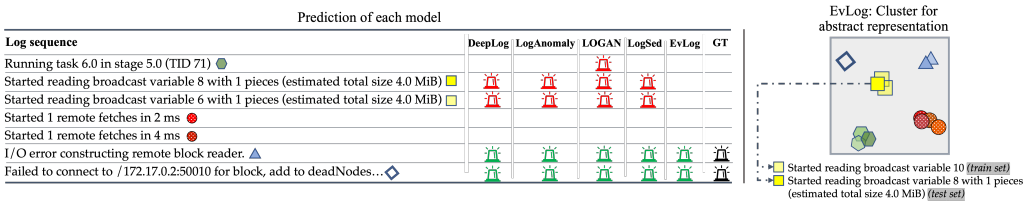


Figure 4.8: An example of how EvLog identifies anomalous logs over software evolution.

Answers to RQ3: The three components, i.e., PLM fine-tuning, unitary discriminator, and local discriminator, all show their effectiveness in the intended design of EvLog.

4.7 Discussion

4.7.1 Case Study

This section conducts a case study (Fig. 4.8) to show how EvLog successfully deals with unseen events and avoids false positives. Having been trained on Spark2, baselines and EvLog are tested in the case from Spark3, where their *AL* predictions are marked with lights. Green, red lights refer to true positive and false positive, respectively. "GT" refers to the ground-truth *AL* set. All baselines wrongly predict the line2 and line3 logs as anomalous. We attribute the false-positive results on the two logs to their evolving events. In fact, this event is paraphrased as follows:

Spark2: Started reading broadcast variable <*>
Spark3: Started reading broadcast variable <*> with <*> pieces (estimated total size <*> MiB)

where <*> refers to the run-time generated numeric values.

Facing such evolving logs, EvLog can mitigate the associated issue by the abstract representation shown on the right-hand side of Figure 4.8. Though line2 and line3 are unseen logs, they can be assigned to a cluster that contains historical semantically similar log messages, according to their rich semantic representations. The yellow squares represent the rich representations of logs in the hyperspace, where the logs before and after paraphrasing stay closely in one cluster. Therefore, the high-level abstract representation remains stable in the change from the original logs to the paraphrased logs, and these new logs will not be mapped far away from the hyper-sphere's center. Eventually, the model can identify the new paraphrased log as a normal one because it does not deviate from the normality.

4.7.2 Threat to Validity

Internal threats. (1) Dynamic threshold. EvLog requires a dynamic threshold D to identify anomalous logs. Our study found that the satisfied threshold for intra-version and inter-version identification is 0.4 and 0.6 times the maximum normality in the training data, respectively. The threshold strikes a balance between recall rate and precision rate. In practice, maintainers can customize the threshold based on different scenarios. (2) Domain knowledge gap. Technical terms in logs may have specific meanings not captured by PLMs. For example, we

use "volume" to describe a detachable block storage device in a computing system, but it usually refers to the degree of loudness or the amount of space in daily life. We fine-tune the PLM with the collected log messages to mitigate the threat.

External threats. (1) Software drastic evolution. Software systems possibly experience a drastic change, such as complete code restructuring or infrastructure renewal. In such scenarios, logging statements are likely to be altered significantly, and our approach has limitations to handle it without incremental learning. Nevertheless, our comparison between Spark2 and Spark3 over two years shows limited extreme changes. (2) Limited dataset. EvLog has been evaluated with only two real datasets and a synthetic dataset, and more real datasets with diverse job types are necessary to validate EvLog's effectiveness. However, as this is a brand-new task, datasets are sparse and challenging to collect. To address this issue, our created dataset is collected with representative 22 benchmark workloads from two widely-used systems. Although this dataset does not cover all possible workloads, it includes many commonly used ones and provides a practical simulation of the task.

4.8 Summary

Existing advanced log localization models are proposed to discover anomalous logs that may indicate faults in a system automatically, but they ignore software evolution activities. This

chapter first empirically identifies three challenges (i.e., parser errors, evolving events, and unstable sequences) carried with software evolution and discusses how these challenges can affect localization models. Second, we propose EvLog to address the above challenges. To deal with the first two challenges, we develop a parser-free extractor to mine multi-level semantic representation from logs. Then, an anomaly discriminator with an attention mechanism is built to overcome the unstable sequence issue. At last, the effectiveness of EvLog in identifying anomalous logs over software evolution is confirmed by evaluating it on large-scale system logs. This is a newly identified research task in anomalous log localization due to software evolution, and the associated code of EvLog as well as the newly collected datasets, are released for research purposes. We hope our study can motivate more future work on software evolution in the log analytics community.

□ **End of chapter.**

Chapter 5

Log Sequence Synthesis for Anomaly Detection

5.1 Introduction

The log analysis community has proposed numerous techniques to assist maintainers in automatically inspecting the large log files produced by modern complex systems [11, 121, 122, 123]. Despite the continued development of automated solutions for software maintenance through log analysis, there has been a lack of emphasis on the need for effective datasets. Consequently, only a few of these techniques have been successfully deployed in real-world settings. This highlights the gap between the limited log data used in academic research and the complexity of industry deployment [6].

Acquiring sufficient, high-quality, and representative logs for practical analysis is challenging. On the one hand, industrial logs from real-world large service providers [6, 93, 83, 124]

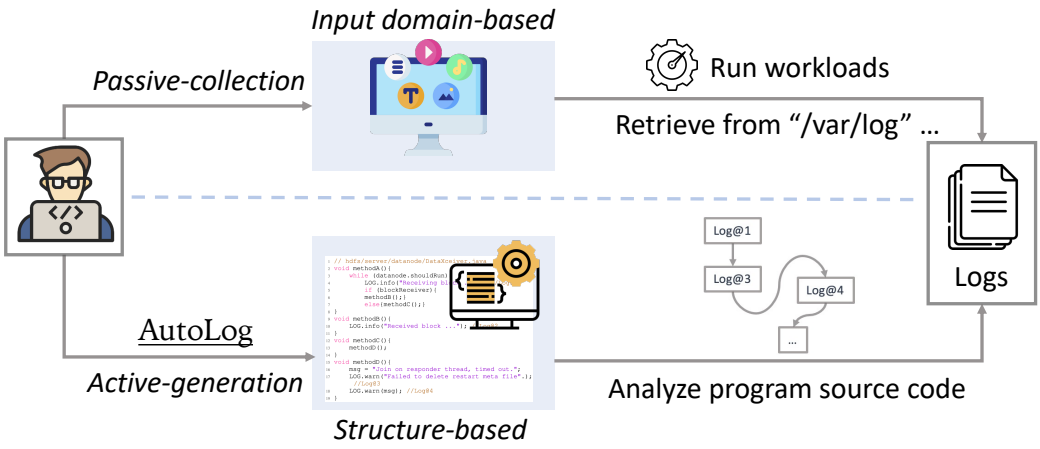


Figure 5.1: Difference of existing passive-collection approach and our active-generation methodology AutoLog.

(e.g., IBM, Microsoft) contain rich events but have privacy concerns, making it difficult to release them publicly for researchers. On the other hand, logs collected in laboratory environments using simulation [74, 10, 72] are publicly available but contain simple events produced from limited standard workloads. As a result, they do not accurately reflects the complex run-time workflow of large-scale software. In either case, existing datasets are based on a *passive-collection methodology* that retrieves log files after running applications in a computing system, as described in Fig. 5.1 (Up). The output logs from the passive-collection methodology are solely dependent on the value of inputs, which impedes exploring and evaluating models on three aspects, namely:

(1) *Comprehensiveness* of log events. Log events are formulated as logging statements in source code for execution. When

collecting logs passively, the diversity of log events is mainly determined by the variety of input workloads. For example, the widely-used Hadoop dataset [74] was generated using only two test applications. Even though the system is deployed in scale, it is still impractical to traverse all *what-if* scenarios [125] during execution to trigger complete logging statements by deliberately designed workloads.

(2) *Scalability* over diverse systems. Collecting logs from new systems requires engineers to re-deploy the system and re-implement workloads, which is time-consuming and labor-intensive [72, 113]. Hence, existing datasets are inadequate in terms of system logs, making it challenging to evaluate the generalization ability of proposed log analysis techniques across diverse systems. For example, the most popular dataset LogHub [72], integrates only five labeled system logs for evaluating anomaly detectors.

(3) *Flexibility* of log utility. While existing datasets allow for minor modifications (e.g., anomaly rate) for evaluation purposes [39, 126], these changes are inadequate for imitating diversified scenarios. For example, when maintainers need to analyze the functionality of a storage component, existing datasets with anomaly-or-not annotation cannot distinguish these component-specific logs. Hence, the complicated conditions in the actual production environment necessitate a more flexible and controllable log collection methodology.

The limitations of the current passive-collection methodol-

ogy have led to the need for effective log collection for building practical solutions. Since system logs are generated during the execution of logging statements in the source code, (e.g., `LOG.error(``failed to start web server.'')`), we consider the log collection problem as the task of constructing log sequences based on the execution order of logging statements. To create log sequences, we adopt the idea from program analysis to develop AutoLog, the first Automated Log generation methodology that *actively generates* effective datasets for anomaly detection (Fig. 5.1 (Down)). The novel static-guided approach can uncover execution paths and produce rational log sequences [41, 51, 44].

Specifically, AutoLog generates effective log datasets with three phases: (1) *logging statement probing phase* explores all methods containing the logging statements to achieve *comprehensive* log events coverage. (2) *log-related execution path finding phase*, which prunes the call graphs and acquires the execution paths related to the logging statements to ensure *scalability*. (3) *log graph walking phase*, which forms log sequences from execution paths and labels the anomaly sequences based on expert annotations. The controllable parameters of AutoLog enable researchers to customize the log dataset with different data sizes, anomaly ratios, and component indicators, providing *flexibility* in log utility.

We conduct an extensive evaluation of AutoLog to 50 popular Java projects and compare the resulting dataset to existing

passively-collected datasets, showing its superiority from three perspectives: (1) AutoLog generates `9x-58x` more log events than existing datasets from the same system, and covers 87.77% of all log events of the 50 projects on average. (2) AutoLog successfully produces logs at least `15x` faster than the collection time for existing log datasets, with tens of thousands of messages per minute on average. (3) AutoLog is built with options to change the data size, anomaly ratio, and certain components of logs, supporting a wide range of development environment simulations. Further, we show that existing anomaly detection models effectively gain improvements (1.93%) by learning from the comprehensive log dataset generated by AutoLog. Thus, we believe that *the data generation methodology AutoLog can serve as a critical resource for developing advanced log analysis algorithms, as well as for providing testing and benchmarking data for such algorithms to ensure software reliability.*

To summarize, the contributions of this chapter are:

- We present AutoLog, a novel, widely applicable automated log generation methodology that addresses three limitations of existing log datasets: lack of comprehensiveness, scalability, and flexibility.
- AutoLog has three phases: logging statement probing, log-related execution pathfinding, and log path walking.
- Extensive experiments show that AutoLog achieves a comprehensive (87.77%) coverage of log events and efficiently

Table 5.1: Statistics and descriptions of existing datasets.

Dataset	#Event	#Workload	#Failure	#Message	CollectionTime	Annotation
$\mathcal{D}$-HDFS	30	NA	11	11,175,629	38.7 Hours	✓
$\mathcal{D}$-Hadoop	242	2	3	394,308	NA	✓
$\mathcal{D}$-BGL	619	NA	NA	4,747,963	214.7 days	✓
$\mathcal{D}$-Zookeeper	77	NA	NA	207,820	26.7 days	✗

produces log datasets (over 10,000 messages/min) on Java projects, and offers the flexibility for adapting to multiple scenarios. We further demonstrate that AutoLog improves anomaly detectors by 1.93%.

5.2 Motivating study

5.2.1 Study Subject

We select four widely-used datasets (Table 5.1) as subjects released by different project teams, including distributed systems and supercomputers. In this table, #Event, #Workload, #Failure, and #Message show the number of log events, executed workloads, failure types, and log messages in each dataset, respectively. In particular, $\mathcal{D}$-HDFS [10], $\mathcal{D}$-Hadoop [74], and $\mathcal{D}$-BGL [113] datasets are collected for anomaly detection after running normal workloads and injecting several types of failures in each system, respectively. $\mathcal{D}$-Zookeeper [72] is collected by running several benchmarks without labeling.

```
1  // hdfs/server/datanode/DataXceiver.java
2  void methodA(){
3      while (datanode.shouldRun){
4          LOG.info("Receiving block " + block); //Log@1
5          if (blockReceiver){
6              methodB();}
7          else{
8              methodC();}
9      }
10 }
11 void methodB(){
12     LOG.info("Received block " + block); //Log@2
13 }
14 void methodC(){
15     methodD();
16 }
17 void methodD(){
18     msg = "Join on responder thread, timed out.";
19     LOG.warn("Failed to delete restart meta file."); //Log@3
20     LOG.warn(msg); //Log@4
21 }
```

Listing 5.1: A simplified example from HDFS.

5.2.2 Are Existing Datasets Comprehensive?

We first investigate whether the existing log datasets provide comprehensive coverage of logging statements. Considering the case in Listing 5.1 from Hadoop, `Log@2` will occur if `blockReceiver` is enabled; otherwise, `Log@3` and `Log@4` will be logged. Nevertheless, we find that the latter were absent in the $\mathcal{D}$-HDFS dataset, likely due to `blockReceiver` being enabled at all times. As a result, the anomaly detection techniques trained with an inadequate dataset may fail to tackle the unseen log,

and even their experiment conclusions may not be representative. For instance, neglecting the responder connection time anomaly, which is associated with `Log@3` and `Log@4`, can happen if the dataset used for training lacks these logs. Furthermore, Table 5.1 reveals that the existing datasets collected through passive-collection approaches are limited in the number of failure types and workloads they cover. The literature [127, 128] implies that although a range of workloads has been implemented, it is still unrealistic to inject all types of anomalies to trigger all log events. Thus, these datasets fall short of comprehensive coverage, creating a significant gap with real-world scenarios.

5.2.3 Are Existing Datasets Scalable?

Validating the generalization ability over diverse systems is critical for practical log analysis algorithms. We use the term scalability to measure the effort (e.g., time, workforce) we should put in to acquire logs from multiple system sources.

To determine the scalability of existing datasets, we summarize their log message amounts and collection time in Table 5.1. The table reveals that it takes a long time to collect logs, even after the system has been deployed and configured. For example, collecting 207,820 log messages from the Zookeeper takes more than 26 days. Additionally, since existing datasets are obtained from running system applications, it requires additional expert efforts to redeploy and rerun the applications

when extending the workloads to a new system. Unfortunately, software maintainers cannot afford to wait such a long time before developing or validating new algorithms. In short, existing datasets are not scalable enough, necessitating the exploration of a new, more efficient approach to log collection.

5.2.4 Are Existing Datasets Flexible?

Previous research has examined the impact of certain dataset characteristics, such as data distribution and data selection, on log-based anomaly detectors [39, 126]. However, to further investigate the performance of log analytics, it is necessary to customize log datasets to simulate diverse scenarios, which requires flexibility.

Ideally, all data can be collected, but in reality, data cannot be completely collected (or collected in a large enough quantity), and they suffer from restrictive flexibility. For example, to increase the anomaly ratio from 0.1% to 15%, researchers may need to remove a large number of normal logs (`150x`) from the existing dataset, sacrificing data quantity [126]. Moreover, logs collected from system files without component specifications raise difficulties for component-wise analysis. More ingredients of the flexibility are elaborated in Section 5.5.4. In this regard, existing datasets cannot flexibly demonstrate model effectiveness under various scenarios, motivating a controllable log sequence acquisition approach.

5.3 Methodology

5.3.1 Overview

Log files are created every time when a system executes logging statements. Therefore, we view the log sequence generation problem as a task of finding the execution paths related to logging statements in the program. Generating log sequences for anomaly detection involves two primary questions: how to generate program execution paths that include logging statements (Phase2), and how to identify whether the execution path produces an anomaly or not (Phase3).

To tackle the problem, AutoLog takes the program as input and outputs a dataset for anomaly detection by analyzing its execution paths. It comprises three phases, as illustrated in Figure. 5.2: The first phase builds call graphs for the project and marks all methods that contain logging statements (LogMethod), enabling *comprehensive* coverage of log events. The second phase prunes out the call graph nodes and records the log-related execution paths (LogEPs) within each remaining method to overcome the *scalability* issue. The third phase propagates logging statement labels from domain experts to all LogEPs. AutoLog eventually generates *flexible* log sequences with chaining LogEPs across methods based on their calling relationship.

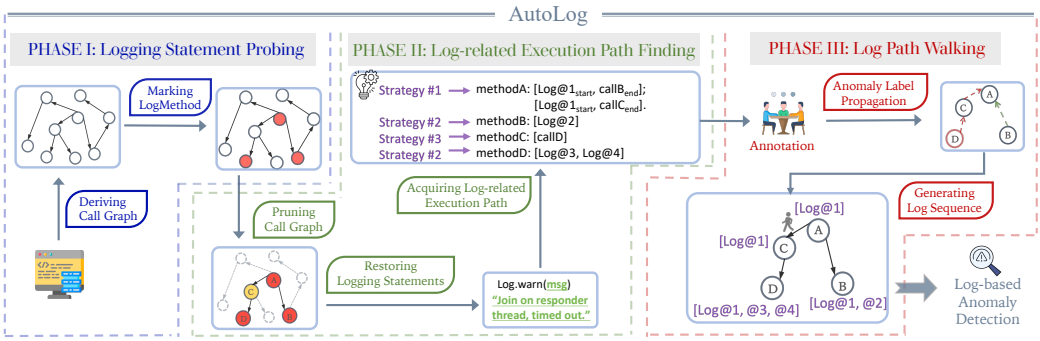


Figure 5.2: The overall framework of AutoLog with three phases: logging statement probing, log-related execution path finding, and log path walking. The details of "Acquiring Log-related Execution Path" are illustrated in Figure. 5.3.

5.3.2 PHASE1: Logging Statement Probing

To cover comprehensive log events, AutoLog starts with exploring the methods containing logging statements and the calling relationships of these methods.

Deriving Call Graphs

Initially, we perform a standard static analysis to construct a call graph that chains methods based on their execution-time relationships in a program [129]. A call graph is a directed graph where each node identifies a method and each edge represents a pair of (caller, callee) method-level relationships. AutoLog derives call graphs with the context-insensitive pointer analysis [130] to enhance the precision of the call graph in handling virtual method calls, calls through interfaces, and polymorphism. Afterwards, we treat each call graph as a directed acyclic graph after marking the cycles induced by recursion in-

vocation.

Marking LogMethod

Since we concentrate on logs, AutoLog identifies methods containing logging statements (LogMethods) by checking whether any logging API is used in a method. To detect a variety of logging APIs used in different projects, we summarize commonly-used logging frameworks (e.g., slf4j [131]) and capture logging APIs by analyzing the invocation of these popular frameworks. Compared to the regular expressions applied in LogCoCo [41], this API-based analysis is capable of recognizing more comprehensive customized logging APIs by examining the inheritance relationships. Figure. 5.2 illustrates how AutoLog manages Listing 5.1, where `methodA`, `methodB`, and `methodD` are recognized as LogMethod (highlighted in red).

5.3.3 PHASE2: Log-related Execution Path Finding

An intuitive idea to solve the execution path finding problem is to construct the entire execution graph of a project and traverse it. However, in most cases, large-scale software contains an infinite number of paths [132] so that exhaustive enumeration undoubtedly causes path explosion problem [133]. To overcome scalability challenges, AutoLog takes two steps: (1) pruning out the call graph nodes that will not induce LogMethods; (2) traversing the intra-method execution graph and recording

the execution paths that are related only to logging activities.

Pruning Call Graph

The goal of pruning is to eliminate redundant nodes from the call graph, particularly those that neither represent LogMethod nodes nor nodes that lead to any LogMethod nodes. A node may induce a LogMethod node in two ways: (1) by calling it directly, or (2) by calling it indirectly through other intermediate nodes. Identifying the LogMethod-inducing nodes can be viewed as a graph sorting problem that finds all ancestor nodes of specific nodes (i.e., LogMethod nodes) in the graph. To do so, AutoLog performs topological sorting [134] over the call graph. Using the topological sorting, a node is considered a non-LogMethod-inducing method if it is neither a LogMethod nor comes before any LogMethod, indicating that it is not an ancestor of any LogMethod. In Figure. 5.2, red, yellow, and dashed contour nodes are used to represent LogMethod, LogMethod-inducing methods, and non-LogMethod-inducing methods, respectively.

All non-LogMethod-inducing nodes and the edges associated with them (represented by dashed lines) will be pruned out. The resulting pruned graph (denoted as CG') is much smaller than the original call graph since many methods do not contain or induce other logging statements. For example, only *14.79%* of nodes are reserved after pruning call graphs for the HDFS system. Because only the remaining methods are used for fur-

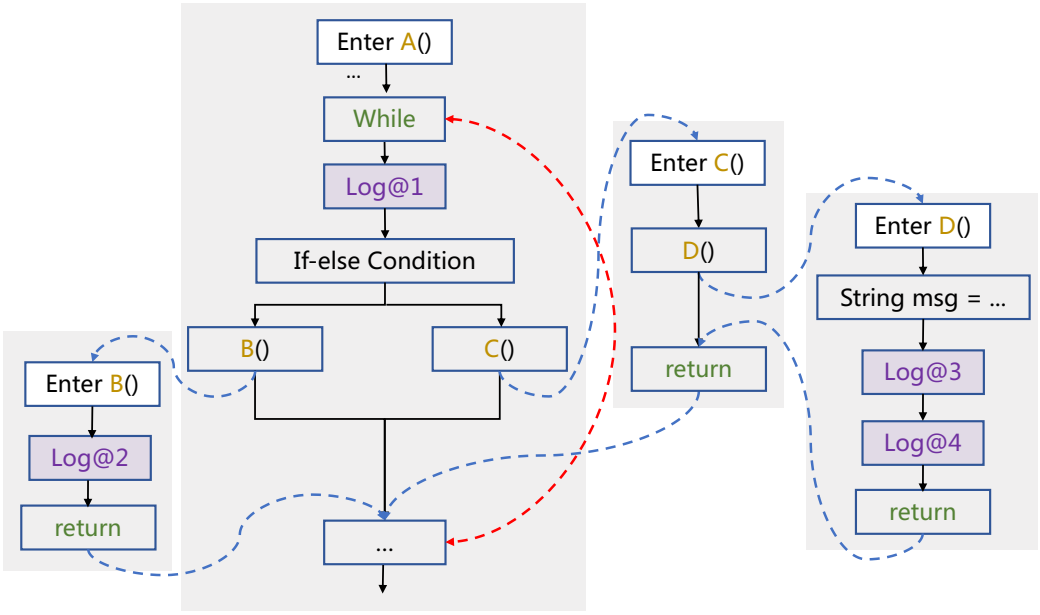


Figure 5.3: The simplified execution graphs for methods in Listing 5.1.

ther analysis, the pruning step significantly promotes AutoLog's efficiency and facilitates subsequent analysis.

Restoring Logging Statements

This step involves restoring logging statements by resolving run-time parameters to provide more detailed information beyond the specific logging statements. A logging statement typically includes constant strings written by developers (e.g., "`Receiving block`") and run-time parameters (e.g., `block`). Specifically, AutoLog resolves the parameters that are constant string variables inside the methods. For example, the parameter `msg` (Line20) is resolved from its assignment (Line18) in Listing. 5.1. For other types of parameters (e.g., `block`), AutoLog replaces them with a dummy token (i.e., <*>) since they

are typically removed during preprocessing for anomaly detection models [23, 77].

Acquiring Log-related Execution Paths

Log sequences generate along with software run-time, reflecting the control-flow paths and execution states [51, 135, 87, 136]. However, obtaining the order of realistic log events by enumerating execution paths from the entire application can be challenging due to their huge size [137]. To solve this issue, AutoLog constructs and traverses the small-scale execution graph for each method in the CG' to enable scalability.

Particularly, AutoLog builds an execution graph with control flow information for each method and links the invocation with the corresponding method in the entry and exit points. Each node in the execution graph represents an executed activity, and the edge represents the relationship between two activities in the temporal order.

Afterwards, AutoLog traverses each execution graph and only records log-related execution paths (*LogEPs*) for each method, which includes the *invocations* and *logging activities* (i.e., logging statements). The invocation is noteworthy because a log sequence in a method can be interrupted by another log sequence introduced by the invocation. AutoLog derives LogEPs for each method by using three strategies. Figure. 5.3 depicts the execution graphs for four methods of Listing 5.1 to illustrate the strategies.

- Strategy #1: For nodes that are *both non-leaf nodes and LogMethod* in the CG', AutoLog considers the execution order of invocations and logging activities. In our case, we obtain two LogEPs for `methodA`: `[Log@1, callB]`, and `[Log@1, callC]`.
- Strategy #2: For *leaf nodes* in CG' that are definitely *LogMethods*, AutoLog enumerates all possible execution sequence of logging activities. This strategy is applied to `methodB` and `methodD`, leading to the LogEP of `[Log@2]` and `[Log@3, Log@4]`, respectively.
- Strategy #3: For the *non-leaf and non-LogMethod* nodes in the CG', AutoLog records all possible invocation sequences in execution. In this case, one LogEP is derived for `methodC`: `[callD]` applying this strategy.

In AutoLog, loops (e.g., for, while) are viewed as paths that are repeatedly traversed in a tail-recursive way and are cycle-free. This crucial feature allows our approach to enumerate all possible execution paths. Nodes within the loop can occur multiple times in a row to mimic the actual execution sequences. When acquiring LogEPs, we mark the nodes (i.e., first and last node) in a loop with a special sign. For instance, LogEP for methodA will be marked as: `[Log@1`$_{start}$`, callB`$_{end}$`]`, and `[Log@1`$_{start}$`, callC`$_{end}$`]`.

Although LogEPs provide all possible executable paths, there exist some paths that are not executable under any in-

put values. To avoid such infeasible paths, we conduct intra-procedural constraint analysis following previous studies [51]. In specific, we gather all constraints for each LogEP and filter out any LogEPs that contain unsatisfiable constraints. This process makes the generated LogEPs more realistic. Using the scalable traversal strategy, AutoLog obtains all LogEPs in less than 1.5 hours in an HDFS system that includes 3,749 methods (pruned) and 2,535 logging statements.

5.3.4 PHASE3: Log Path Walking

This phase is devised with the goal of generating labeled log sequences to simulate the actual application execution by chaining LogEPs from each method. To achieve accurate labels while saving annotation effort, AutoLog uses a *seed-propagation strategy* that involves experts identifying a set of anomaly LogEPs as seeds, followed by automatically propagating the labels of these anomaly LogEPs to all acquired LogEPs. Labeled log sequences are eventually generated by a succession of random LogEP selection step which walks over the invoked methods. The choice of LogEPs can be controlled by setting specified data size, execution entrance and anomaly rate, allowing a flexible dataset that simulates execution logs in multiple scenarios.

Seed Anomaly LogEP Annotation

Since identifying anomalous logs is challenging and mostly relies on domain expert knowledge, we adopt the human-annotation process similar to existing log-based anomaly detection datasets [113, 74, 10]. However, AutoLog differs from existing datasets in that, it can automatically propagate labels from a set of anomaly unit to the sequence-level, significantly improving the annotation efficiency.

We use LogEP as an anomaly unit for expert labeling as it is a small execution path reflecting the system behavior in a period. The output of this step is a set of seed anomaly LogEPs that *must* produce anomalies. To obtain anomaly LogEPs, we design the annotation process with *alerting statement annotation* and *anomaly LogEP identification*. The alerting statement annotation is designed to filter out a large number of normal logging statements (e.g., `Log@1`). To do so, we present all logging statements, their corresponding code snippets as the execution context, as well as the nearby comments inserted by developers, for annotators to decide whether they are alerting logging statements for an anomaly. Taking the HDFS system as an example, we present the example alerting statements and their corresponding potential anomalies in Table 5.2. However, identifying alerting statements is not enough for profiling system activities. Hence, anomaly LogEP identification aims to further determine whether LogEP will certainly lead to anomalies. To recognize

Table 5.2: Alerting logging statements examples and their potential anomaly types in HDFS system.

Anomaly Type	Alerting Logging Statements Examples
Version mismatch	Layout version on remote node does not match this node's layout version.
Disk/Storage error	Unable to get json from Item. Unexpected health check result for volume $< * >$.
Dependency error	Unresolved dependency mapping for host . Continuing with an empty dependency list

anomaly LogEPs, we ask annotators to manually check each LogEP that contains alerting statements. The identified anomaly LogEP is considered anomaly seeds for propagation in the next step.

Anomaly Label Propagation

To generate an effective anomaly detection dataset, it is important to have labels at the sequence-level even though the annotations are done at the LogEP-level to save human effort. To this end, AutoLog uses a strategy called seed propagation to propagate the labels of seed anomaly LogEPs to other LogEPs, with the goal of figuring out whether a LogEP is infected by the anomaly label. The main idea is that: If a LogEP in one method contains an anomaly (e.g., `methodD`), then all other LogEP invokes this method (e.g., `[callD]`) *may* also induce an anomaly.

The propagation starts from the seed anomaly LogEPs marked *infected*. The propagation is done recursively by checking whether

a LogEP contains other infected LogEPs brought from invocations. After the propagation, infected LogEPs are *likely to*, but will not necessarily cause an anomalyz, whereas others *must* be anomaly-free.

Generating Log Sequences

Given the LogEPs, AutoLog eventually generates each actual log sequence by selecting and chaining the LogEPs in a top-down approach. The top-down random walking process works as follows: It starts at an entrance method and walks along invocations, with one LogEP being randomly chosen at each walking step. If an invocation exists in the current chosen LogEP, AutoLog walks to the callee method and then chooses a LogEP in the callee method. The logging statements in all chosen LogEPs are chained according to the invocation relationships to form the log sequence.

As one log sequence reflects the execution path of a single thread, AutoLog generates a labeled sequence at each time with two walking strategies and combines the sequences to form the datasets for anomaly detection:

- To generate an *anomaly log sequence*, we always select the infected LogEP in every step until we have selected an anomaly LogEP that may contain alerting logging statements.
- To generate a *normal log sequence*, we randomly select a

LogEP of each method but take a step back when selecting an anomaly LogEP, and re-choose another LogEP.

During the walking, invocations or logging statements within the loop may occur successively more than once. The log sequence generation process with random path selection enables the flexibility of datasets. It can be decorated with a set of hyper-parameters to generate more controllable log sequences. For example, the component indicator (CI) controls the starting point to simulate the execution path in a specific component (e.g., storage component), and the anomaly rate AR controls the anomaly ratio ($\frac{\#Anomaly\_Sequence}{\#All\_Sequence}$) in the generated dataset.

5.4 Implementation

5.4.1 Experiment Environment

AutoLog has been implemented by 5,182 lines of Java code with Soot [138], a Java bytecode optimization and analysis framework. We run all experiments on Ubuntu 18.04. The experiments are carried out on a machine with an Intel(R) Xeon(R) Platinum 8255C CPU (@2.50GHz) with 128GB RAM. We set the AR to be 3% and CI to be all possible paths to ensure the coverage, unless otherwise specified.

5.4.2 Annotation

To ensure the correctness of annotation, we invite three Ph.D. students who have at least two-year experience in distributed system research and development, two of whom annotate individually, and the other one works as an adjudicator to discuss the disagreements with annotators. They are all allowed to access the Internet for searching answers. The agreement score between annotators measured by Cohen's kappa [139] before adjudication is 0.841 and 0.834 in alerting statement annotation and anomaly LogEP identification, respectively. All annotators reach a consensus on the labels after discussing them with the adjudicator.

5.5 Experiments

We evaluate AutoLog using four research questions:

RQ1: How comprehensive are the datasets generated by AutoLog?

RQ2: Is AutoLog scalable for real-world applications?

RQ3: How flexible are the datasets compared with passively-collected datasets?

RQ4: Can AutoLog benefit anomaly detection problems?

5.5.1 Experimental Settings

Existing Datasets

To verify the effectiveness of AutoLog, we choose three widely-used publicly available datasets collected from Java applications. We use $\mathcal{D}$-*sys* and AutoLog-*sys* to denote the baseline datasets collected from system *sys* and collected by AutoLog, respectively. Details of the datasets are illustrated as follows:

- $\mathcal{D}$-Hadoop [74]. Hadoop is an open-source framework designed to store and process large-scale data efficiently. The dataset is collected via running two standard applications and injecting three types of failures for anomaly detection.
- $\mathcal{D}$-HDFS [10]. HDFS is a distributed file system for large-data storage, enabling high-throughput access to data. $\mathcal{D}$-HDFS is collected from a private cloud environment executing benchmark workloads with labeled anomalies.
- $\mathcal{D}$-Zookeeper [72]. Zookeeper provides a centralized service to manage a large set of hosts (e.g., synchronization, configuration information management). $\mathcal{D}$-Zookeeper is collected in a lab environment for log analysis *without labelling*.

Since this chapter presents the first methodology that actively generates log datasets without deploying and running the system, we compare AutoLog with all existing Java-based log datasets in our research questions.

Evaluation Subjects

Apart from three projects associated with existing log datasets, we extensively evaluate the effectiveness of AutoLog on the most popular 50 projects from the Maven repository [140], with more than 10,000 usage times for each. The selected projects include, but are not limited to, distributed streaming platforms (e.g., Kafka), core Java packages (e.g., Apache HttpClient), and unit testing frameworks (e.g., JUnit). Among them, we present the detailed result of three widely-studied distributed system projects below and report statistical results for other projects.

- Apache Storm [141]. It is a distributed real-time computation system, allowing large-scale data processing and high-velocity data streams.
- Flink [142]. Flink is a stream processing engine that can handle scale system events with low latency.
- Kafka [143]. Kafka is a distributed event storage and stream-processing platform applied in thousands of companies.

Metrics

- *Coverage.* Motivated by software testing studies [144, 41], we evaluate the comprehensiveness of *logging coverage*, which measures the percentage of the discovered log events to the total log events designed in the application ($\frac{\#Log\_Event}{\#Total\_Log\_Event}$). We also use $\mathcal{D}$-Coverage to evaluate the ratio of log events

Table 5.3: The comparison of datasets for comprehensiveness. $\mathcal{D}$-Coverage is reported for the systems with publicly log datasets.

Dataset	# Log Event	Logging Coverage	$\mathcal{D}$-Coverage	Increment (↑)
$\mathcal{D}$-Hadoop	242	242/3426 (7.1%)	219/242 (90.5%)	12x
AutoLog-Hadoop	2879	2879/3426 (84.0%)		
$\mathcal{D}$-HDFS	30	30/1700 (1.8%)	27/30 (90.0%)	58x
AutoLog-HDFS	1367	1367/1700 (80.4%)		
$\mathcal{D}$-Zookeeper	77	77/758 (10.2%)	77/77 (100%)	9x
AutoLog-Zookeeper	740	740/758 (97.6%)		
AutoLog-Apache Storm	1754	1754/1887 (93.0%)	-	-
AutoLog-Flink	1574	1574/1711 (92.0%)	-	-
AutoLog-Kafka	847	847/1002 (84.5%)	-	-

in existing datasets that are covered by AutoLog: $(\frac{\#Log\_Event\_Covered\_by\_AutoLog}{\#Log\_Event\_in\_Existing\_Dataset})$.

- *Execution Time.* To validate the scalability of AutoLog, we report the program execution time for each system, which includes the time for code analysis and data generation in a single machine.

5.5.2 RQ1: How Comprehensive are the Datasets Generated by AutoLog?

We evaluate the comprehensiveness of the dataset generated by AutoLog regarding the number of log events, logging coverage, and $\mathcal{D}$-Coverage, illustrated in Table 5.3.

Compared to the existing log datasets, AutoLog effectively enhances the comprehensiveness of log events. Firstly, AutoLog covers an average of 87.38% logging statements over six systems, demonstrating its ability to capture logging statements designed in code. The number of log events generated by AutoLog is

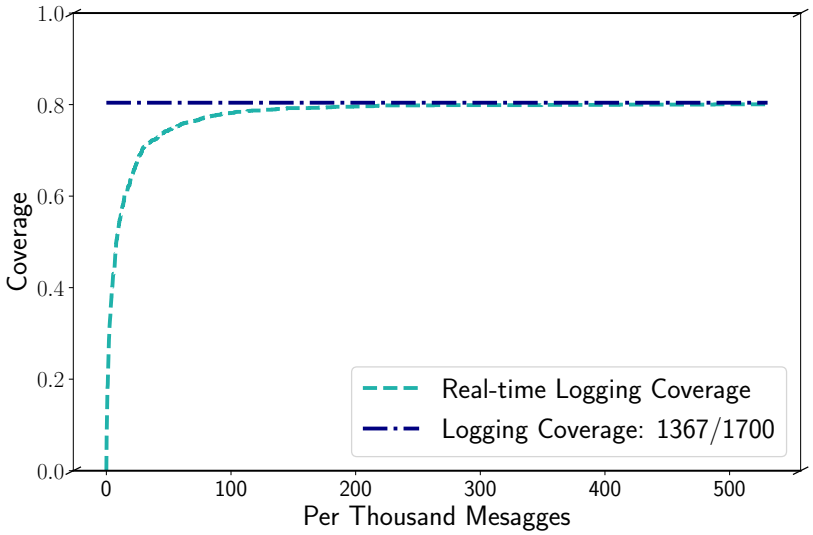
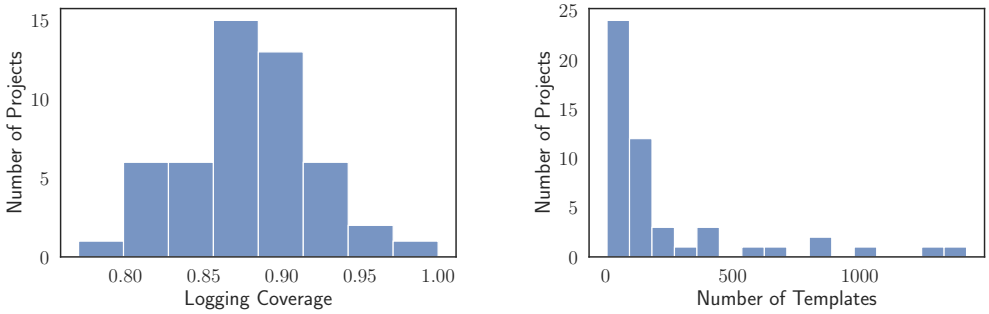


Figure 5.4: The number of generated log messages and their corresponding real-time logging coverage.

`12x, 58x,` and `9x` is more than existing datasets collected from Hadoop, HDFS, and Zookeeper, respectively. Regarding the missing parts, we analyze them as follows. They mainly come from the restrictive call graph construction phase [51], the limitation of logging statement restoring [145] across different methods, and unreachable execution paths that we have eliminated. Taking the restoring process as an example, if the constant string `msg` is defined outside methodD in Listing 5.1, AutoLog has difficulty restoring the `Log@4`. Secondly, the experiment results illustrate that AutoLog covers most of the log events in the existing dataset, achieving 219/242, 27/30, and 77/77 for Hadoop, HDFS, and Zookeeper systems, respectively. The missing log events mainly come from optional components of complicated systems and different deployment settings on varied platforms.



(a) Histogram of Logging coverage.

(b) Histogram of # Log Event.

Figure 5.5: The histogram of logging coverage and the number of log events over 50 popular projects.

Logging coverage implies the upper limit of the discovered log events in AutoLog; ideally, generating log sequences for enough time will eventually reach that coverage. Figure 5.4 shows the relationship between the number of real-time generated log messages and its corresponding logging coverage in the HDFS system (denoted as real-time logging coverage). The results indicate that AutoLog achieves its logging coverage after generating approximately 350,000 messages.

Besides, we display the logging coverage (left) and the number of generated log events (right) histogram over 50 popular Java projects in Figure 5.5. The results demonstrate that AutoLog achieves an average logging coverage of 87.77%, which is superior to existing log datasets. Additionally, the number of log events in different projects ranges from hundreds to thousands, indicating that existing log datasets with limited log events are inadequate.

Table 5.4: The comparison of existing datasets for scalability.

Dataset	# Message	Execution Time	# Messages/min (speed)	Acceleration (↑)
$\mathcal{D}$-Hadoop	394,308	NA	NA	-
AutoLog-Hadoop	392,427	3.41 hours	1,918	
$\mathcal{D}$-HDFS	11,175,629	38.7 hours†	4,813	15x
AutoLog-HDFS	11,376,233	2.62 hours	72,367	
$\mathcal{D}$-Zookeeper	207,820	26.7 days†	6	2072x
AutoLog-Zookeeper	211,425	17 mins	12,436	
AutoLog-Apache Storm	1,001,245	1.28 hours	13,037	-
AutoLog-Flink	1,003,416	1.21 hours	13,821	-
AutoLog-Kafka	1,002,629	39 mins	25,708	-

† Collection time from its original paper. NA means the authors do not report the collection time. "-" means the acceleration cannot be compared.

Answer to RQ1: AutoLog shows a significant improvement (9x-58x) on the number of log events and can effectively cover 87.77% logging statements on average over studied projects.

5.5.3 RQ2: Is AutoLog Scalable for Real-world Applications?

We evaluate the scalability of our log generation methodology, which measures the time required for generating data. To compare the scalability, we generated the same amount of data in a single machine as existing public datasets and compared the data collection time.

The result in Table 5.4 indicates that AutoLog is efficient in analyzing and generating log sequences from real-world applications. It can produce messages at high speed, with a range of 1,918 to 72,367 messages per minute. Moreover, compared with the HDFS dataset that took 38.7 hours to collect, AutoLog can

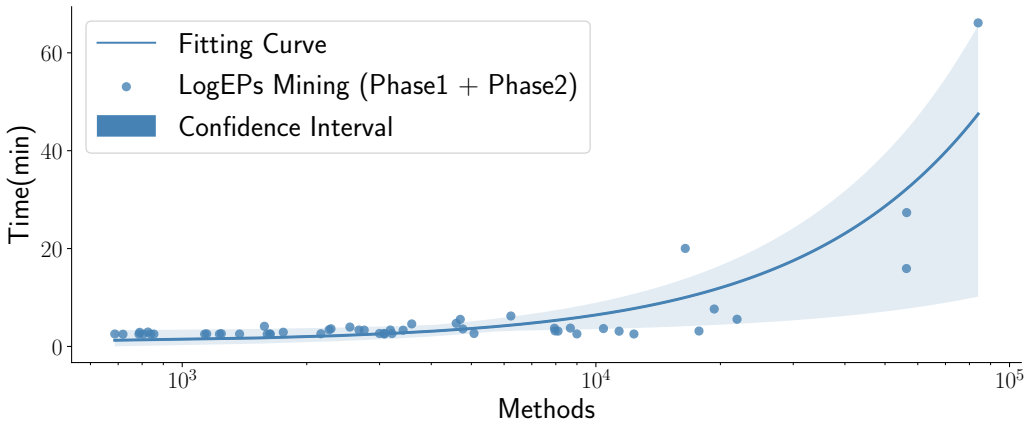


Figure 5.6: Time cost for acquiring LogEPs and the number of analyzed methods over 50 popular projects.

generate the same amount of data within 2.62 hours (15 times faster). This scalability is attributed to the call graph pruning and intra-method LogEP derivation steps, which decrease the number of methods for analysis and solve the path explosion problem. Moreover, because AutoLog is built on source code, deploying it into a new system is effortless, unlike the long time required for configuring and rerunning applications in existing passively-collection methods. Additionally, we investigate the time spent for execution path finding (Phase1 and Phase2) as LogEPs can be reusable to generate log sequences multiple times. In particular, we apply AutoLog on 50 Java projects and measure the execution time, ranging from 687 to 83,969 methods in a project (as shown in Figure 5.6). Our results demonstrate that AutoLog can analyze most projects within one hour, indicating its practicality and efficiency.

Although we did not include the annotation time in Ta-

Table 5.5: The comparison of log datasets for flexibility.

Aspects	$\mathcal{D}$-HDFS	$\mathcal{D}$-Hadoop	AutoLog
Data Size	Partial	Partial	**Complete**
Component Indicator	Partial	Partial	**Complete**
Anomaly Rate	Deterministic	Deterministic	**Non-deterministic**

ble 5.4, AutoLog has an efficient seed-propagation labeling strategy that requires less time than labeling each log sequence in passively-collected datasets. For instance, independent annotators spent an average of 2 hours on alerting statement annotation and 4 hours on anomaly LogEP identification for 1,367 log events in HDFS.

Answer to RQ2: AutoLog considerably shortens the dataset generation time (`15x`) compared with existing datasets. The promising path finding time further manifests its scalability for real-world applications.

5.5.4 RQ3: How Flexible are the Datasets Compared with Passively-collected Datasets?

We assess the flexibility of AutoLog and its benefits for log-based anomaly detection. In Table 5.5, we compare AutoLog with existing datasets regarding the flexibility over *data size*, *component-indicator*, and *anomaly rate*.

Different systems require varying amounts of data for algorithm development. For example, a large amount of data can

be exploited to build algorithms for distributed cloud systems whereas naive systems only preserve a small amount of data to study. AutoLog is capable of generating datasets of *any* size, whereas $\mathcal{D}$-HDFS and $\mathcal{D}$-Hadoop have partial size restrictions once they are released.

In addition, engineers may need to focus on specific components' functionality during software development and maintenance, which requires component-specific logs for anomaly detection. While the existing datasets provide limited component information, AutoLog is able to produce sequences of component-indicator logs by starting to walk from constrained entry nodes during generation.

Moreover, different systems have different fault-tolerance abilities, which affect the potential anomaly rates in collected logs. Existing datasets (e.g., $\mathcal{D}$-HDFS) have fixed numbers of anomaly sequences, requiring researchers to filter out some anomaly sequences or significantly lower the number of normal sequences to increase the anomaly rate in a deterministic way [126]. However, AutoLog can produce a huge amount of various sequences iteratively, introducing the unseen patterns and increasing the flexibility for anomaly detection.

Answer to RQ3: AutoLog effectively generates datasets with more flexibility of utilization (i.e., data size, component, anomaly rate) than existing datasets, allowing imitating a wider range of sophisticated application scenarios.

5.5.5 RQ4: Can AutoLog Benefit Anomaly Detection Problems?

This RQ explores how AutoLog helps to resolve log-based anomaly detection. In particular, we train state-of-the-art (SOTA) detectors on AutoLog and evaluate their performance. This section takes HDFS[1] to evaluate models on real-machine-generated logs $\mathcal{D}$-HDFS (denoted as $\mathcal{D}$) and AutoLog-HDFS (denoted as AutoLog) from the same software version, followed by a performance discussion.

Settings

We select the following representative log anomaly detection models for evaluation: (1) LogRobust [23], which leverages a bi-directional LSTM network (bi-LSTM) with an attention mechanism to learn the importance of each log for tackling unstable log patterns. (2) CNN [77], which transforms the log sequence to a trainable matrix, then applies a Convolutional Neural Network (CNN) for log-based anomaly detection. (3)

[1]Other widely-used anomaly detection datasets (e.g., BGL) are not generated from open-sourced software.

Table 5.6: Comparison of the anomaly detection models over two datasets $\mathcal{D}$ and AutoLog.

	Train set	$\mathcal{D}$			AutoLog		
Test set	Approach	P	R	F1	P	R	F1
$\mathcal{D}$	Transformer	0.889	0.904	0.896	0.892	0.996	**0.941**
	CNN	0.936	0.995	0.965	0.959	0.997	**0.978**
	LogRobust	0.942	0.994	**0.967**	0.947	0.988	**0.967**
AutoLog	Transformer	-†			0.723	0.755	0.739
	CNN	-†			0.697	0.790	0.741
	LogRobust	-†			0.673	0.875	0.761

†The large amount of unseen events will lead to considerably poor performance.

Transformer [24], which applies a Transformer encoder to learn context information to distinguish the anomaly logs from normal logs. When using AutoLog, we set the anomaly rate (3%), and data split ratio (train: test = 8: 2) to be the same as the original $\mathcal{D}$.

Following previous research work [23, 99], we adopt Precision (P), Recall (R), and F1 to evaluate the performance. *Precision* is calculated by the percentage of log sequences correctly identified with anomalies over all sequences that are recognized with anomalies. *Recall* is calculated by the percentage of log sequences correctly recognized with anomalies over the actual anomaly sequences. *F1 Score* (F1) is the harmonic mean between Precision and Recall.

Results

Table 5.6 presents the performance of different anomaly detection models on two datasets. First, we evaluate models on the real-world dataset $\mathcal{D}$ (test set). We observe that models trained with AutoLog perform *consistently better* (1.93% of F1 on average) than trained with $\mathcal{D}$ (train set). CNN can achieve an F1 score of 0.978 after training with AutoLog, even surpassing the SOTA performance by 1.1%. We analyze the reason for performance improvement after training on AutoLog as follows. AutoLog generates more comprehensive log events by imitating a more varied system behavior. Once an anomaly detector has seen such diverse log events in its training phase, it can effectively detect the anomalies in production environments. The result demonstrates that AutoLog is effective in generating realistic log sequences, which can help models learn more varied system behavior and improve anomaly detection accuracy.

Second, we observe that state-of-the-art approaches perform well in $\mathcal{D}$ dataset but can only reach an F1 score of 0.761 in AutoLog. We attribute the performance gap to the more comprehensive log events (e.g., 1367 rather than 30) and diverse sequential log patterns generated by AutoLog. This highlights the need for further research on log semantic encoding and system behavior profiling to address the challenges posed by complex log data in real-world deployments of anomaly detection models.

Answer to RQ4: AutoLog benefits anomaly detection detectors by providing the training resource that allows existing models to improve (1.93% on average) their performance consistently. It is effective in generating more sophisticated and practical log data than any existing Java log datasets. It can serve as a benchmark data generator and training resource for developing and validating promising anomaly detection models.

5.6 Case Study

Figure. 5.7 exhibits a case from an HDFS user who tries to delete blocks and encounters data loss issue[2]. The user reports the log sequences from Datanode (left) and Namenode (right). After matching this realistic log sequence in $\mathcal{D}$-HDFS and AutoLog-HDFS, we find that (1) both $\mathcal{D}$-HDFS and AutoLog-HDFS contain the sequence from Datanode, (2) only AutoLog-HDFS can cover the sequence from Namenode. In AutoLog, the acquired execution paths via program analysis simulate how logging statements are executed (recorded) in software to provide realistic log sequences in block deletion. In addition, the case illustrates the comprehensiveness of logs produced by AutoLog. Although it is impractical to configure and enumerate all system activities in software for log collection, AutoLog addresses the issue by actively analyzing all possible

[2]https://issues.apache.org/jira/projects/HDFS/issues/HDFS-16829

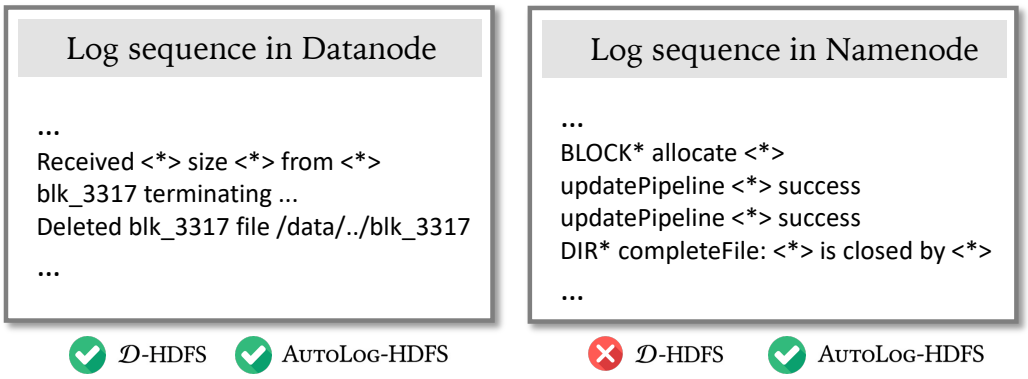


Figure 5.7: User-reported log event sequences from HDFS.

execution paths.

5.7 Threats to Validity

We have identified the following four major threats to validity. (1) *Unpredictable exception handlers*: A few run-time behaviors, such as exception catching, cannot be thoroughly analyzed without actually executing the program. A logging statement can be severed from the execution path by the exception handler. While exceptions can reflect anomalies, these anomalies are relatively "easy" to be detected (e.g., keyword search). Existing log-based anomaly detection mainly focuses on the remaining "difficult" anomalies. To mitigate this threat, we can simulate run-time exception-catching behaviors by randomly selecting try-catch statements and setting interruptions in the try body.

(2) *Potential multi-thread intervention*: AutoLog analyzes execution paths and generates log sequences in a single thread.

Thus, the intervention and communication between multiple threads may be neglected. However, we notice that modern anomaly detection models [23, 77, 24] are developed for single-thread analysis, which starts with separating different threads based on their thread IDs. In this regard, the lack of multi-thread log sequences will not hamper the evaluation of existing anomaly detection models. In the future, we plan to extend AutoLog to support multi-thread settings.

(3) *Unresolved parameters*: AutoLog uses a dummy token to replace unresolved parameters (e.g., addresses, digits, ID character strings) in the logging statements, which is supposed to carry system behavior information. However, previous studies [88, 39] reveal that log event (without run-time parameters) sequences are sufficient to capture system activities, and parameters may hinder deep learning-based detectors [23]. Even though the unresolved parameters are specified with certain values, they will be wiped out before feeding into anomaly detection models. In any case, we regard parameters resolving as an important future direction for their utilization in other log analytics (e.g., log parser).

(4) *Imprecise call graph*: Generating precise call graphs has been known as an open-challenging question in static analysis community for a long time [146, 147, 145]. Soot [138] uses Class Hierarchy Analysis (CHA) [148] to handle dynamic dispatch (e.g., finding the callee), which has a relatively low accuracy due to the polymorphism of Java. To mitigate the impact

of the imprecise graph, we refined the calling relationship with context-insensitive pointer analysis that significantly improves the precision of the call graph.

5.8 Summary

Although log-based anomaly detection has been widely studied, only a few approaches have been successfully deployed in the real world because existing datasets suffer from comprehensiveness, scalability, and flexibility limitations. To overcome these limitations, this chapter presents AutoLog, the first automated log generation methodology for anomaly detection, with three phases: logging statement probing, log-related execution path finding, and log path walking. Extensive experiments demonstrate that AutoLog produces comprehensive coverage of log events in application with scalability. AutoLog is also equipped with hyper-parameters to generate log datasets in flexible data size, component indicator, and anomaly rate. With our replication package released, we believe AutoLog could be a starting point for active dataset generation in the log analysis field, which provides benchmarking data for building practical anomaly detection algorithms.

□ **End of chapter.**

Chapter 6

Empirical Study on LLM-powered Logging Statement Generation

6.1 Introduction

As shown in the example below, a logging statement typically consists of three *ingredients*: a logging level, logging variables, and logging texts [2]. Specifically, as illustrated in the example below, logging level (e.g., *warn*) indicates the severity of a log event; logging variables (e.g., *url*) contain essential runtime information from system states; and logging texts (e.g., *Failed to connect to host: <>*) provides a description of the system's activities.

```
log.warn("Failed to connect to host: {}", url)
```

To help software developers decide the contents of logging statements (i.e., *what-to-log*), logging statement generation tools

are built to automatically suggest logging statements given code snippets. Conventional logging suggestion studies [149, 91] reveal that similar code tends to have similar logging statements, and thus, a retrieval-based approach is used to suggest similar logging statements from a historical code base [150]. However, such retrieval-based approaches are limited to the logging statements encountered in that code base. To overcome such limitation, recent studies employ neural-based methods to decide about *single ingredients* of logging statements (i.e., logging levels, logging variables, logging text). For example, prior work [48, 27] predicts the appropriate logging level by feeding surrounding code features to a neural network. While these tools have also shown improvements in suggesting important variables [28] or proper log levels [27, 49], they lack the ability to produce complete logging statements containing multiple ingredients simultaneously. Some tools [48] require the availability of certain ingredients to suggest others, which can be impractical for programmers who need to generate complete logging statements. However, the complete statement generation has been considered challenging as the model should analyze the code structure, comprehend the developer's intention, and produce meaningful logging text [30]. Moreover, existing neural-based tools are further restricted by training data with limited logging statements and may not generalize to unseen code.

Recent large pre-trained language models (LLMs) [151, 152] have achieved impressive performance in the field of natural lan-

guage processing (NLP). Inspired by this, the latest logging-specific model, LANCE [30], treats logging statements generation as a text-to-text generation problem and trains a language model for it. LLMs have proven their efficacy in many code intelligence tasks, such as generating functional code [153, 154] or resolving bugs [155], and have even been integrated as plugins for developers [156] (e.g., Copilot [157], CodeWhisperer [158]). However, their capacity for generating complete logging statements has not been comprehensively examined. To fill this gap, we pose the following question: *To what extent can LLMs produce correct and complete logging statements for developers?* We expect LLMs, given their strong text generation abilities, can improve the quality of logging statements. Further, LLMs have exhibited a powerful aptitude for code comprehension [159], which paves the way for uncovering the semantics of logging variables.

Our work. To answer our research question, this empirical study thoroughly investigates how modern LLMs perform logging statement generation from two perspectives: *effectiveness* and *generalization capabilities*. We extensively evaluate and understand the effectiveness of LLMs by studying (1) their ability to generate logging ingredients, (2) the impact of input instructions and demonstrations, and (3) the influence of external program information. To assess the generalizability of LLMs, since LLMs are trained on a significant portion of publicly available code, there is a potential data leakage issue in

which logging statements used for evaluation purposes may be included in the original training data [155, 160, 161]. It remains unclear whether LLMs are really inferring logging statements or merely memorizing the training data. Thus, we further evaluate the generalization capabilities of LLMs using unseen code.

In particular, we evaluate the performance of eleven top-performing LLMs encompassing a variety of types—including natural language and code-oriented models, covering both academic works and commercial coding tools on *LogBench-O*, a new dataset we collected, consisting of *2,430* Java files, *3,870* methods, and *6,849* logging statements. Additionally, we employ a lightweight code transformation technique to generate a semantics-equivalent modified dataset *LogBench-T*, which contains previously untrained data and thus can be used to evaluate the generalization capabilities of LLMs. Based on our large-scale empirical study on LogBench-O and LogBench-T, we summarize eight key findings and five implications with actionable advice in Table 6.1.

Contributions. The contribution of this chapter is threefold:

- We build a logging statement generation dataset, LogBench, containing the collection of *6,849* logging statements in *3,870* methods (LogBench-O), along with their functionally equivalent unseen code after transformation (LogBench-T).
- We analyze the logging effectiveness of eleven top-performing

LLMs by investigating their performance over various logging ingredients, analyzing prompt information that influences their performance, and examining the generalization capabilities of these LLMs with unseen data.

- We summarize our results into eight findings and draw five implications to provide valuable insights for future research on automated log statement generation.

Table 6.1: Summarization of key findings and implications in this chapter.

(a) Key findings

Key findings
👉 The performance of existing LLMs in generating complete logging statements *needs to be improved* for practical logging usage.
👉 Comparing the LLMs' logging capabilities presents a challenge, as models perform inconsistently on different ingredients.
👉 Directly applying LLMs yields *better performance* than conventional logging baselines.
👉 Instructions significantly impact LLMs, but there is consistency in the relative ranking of LLMs when used with the same instructions.
👉 Demonstrations help, but more demonstrations does not always lead to a higher logging performance.
👉 Since comments provide code intentions from developers, ignoring them leads to decreased effectiveness for LLMs.
👉 Compared to comments, LLMs gain greater advantages from considering *additional methods* in the same file.
👉 *Unseen code* significantly degrades all LLMs' performance, particularly in variable prediction and logging text generation.

(b) Key implications & actionable advice

Key implications & Actionable advice
➔ How to *generate proper logging text* warrants more exploration.
➔ Intriguing alternative, possibly *unified metrics* to assess the quality of logging statements.
➔ LLM-powered logging is promising. Refining prompts with instructions and demonstration selection strategies for effective few-shot learning should be investigated.
➔ Providing proper *programming contexts* over the projects that reveal execution information can boost LLMs' logging performance.
➔ To advance the generalization capabilities of LLMs, developing *prompt-based learning techniques* to capture code logic offers great potential for LLMs in automated logging.

6.2 Background

6.2.1 Problem Definition

This study focuses on the *logging statement generation* task (i.e., *what-to-log*), which can be viewed as a statement completion problem: given lines of code (typically a method) and a specific logging point between two statements, the generator is then required to predict the logging statement at such point. The prediction is expected to be similar to the one removed from the original file. Figure 6.1 (in dashed line) illustrates an example of this task, where an effective logging statement generator should suggest `log.debug("Reload received for path:" + path)` that is highlighted with green for the specified logging point[1]. Following a previous study [30], for the code lines with n logging statements, we create *n-1* inputs by removing each of them one at a time.

6.2.2 Challenges in Logging Statement Generation

The composition of logging statements naturally makes the logging generation problem a joint task of code comprehension and text generation. Compared to code completion tasks, the generation of logging statements presents two distinct challenges: (1) inference of critical software runtime status and (2) the creation of complicated text that seamlessly integrates both

[1]In this chapter, the logging statement that the generator should predict is always highlighted by green.

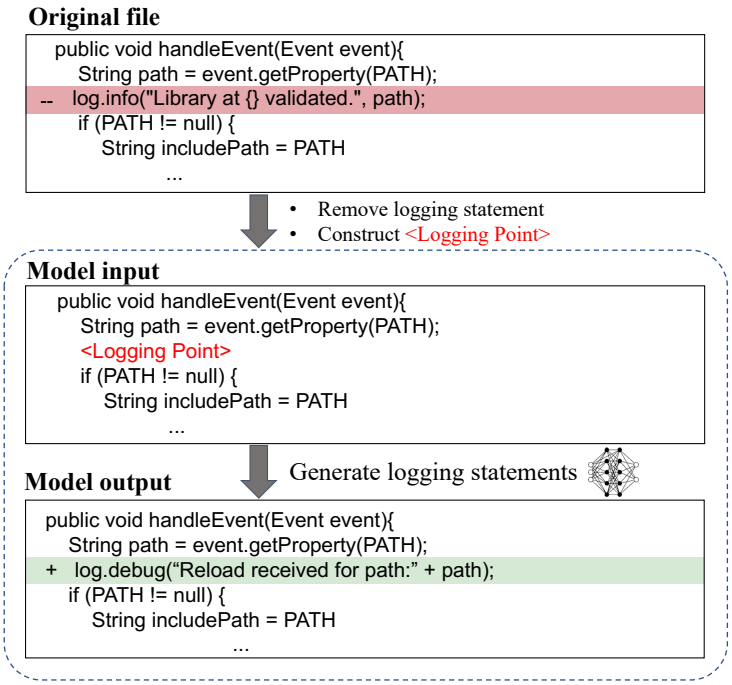


Figure 6.1: Task formulation for logging statement generation.

natural language and code elements.

First, while code generation produces short methods with a high degree of functional similarity, logging statements are *non-functional* statements not discussed in code generation datasets (e.g., HumanEval [162], APPS [163]). Nevertheless, logging statements are indispensable in large-scale software repositories for documenting run-time system status. To log proper system status, a logging statement generator shall comprehend program structure (e.g., exception handling) and recognize critical code activities worthy of logging. Second, integrating natural language text and code variables poses a unique challenge. Logging statement generators must be mastered in two distinct languages and harmoniously aligned. Developers describe code functional-

ities in natural language and then incorporate relevant logging variables. Likewise, a logging statement generator should be capable of translating runtime code activities into natural language and explaining and recording specific variables.

6.2.3 Study Subject

Motivated by the code-related text generation nature of the logging statement generation, we opt to investigate top-performing LLMs from three fields as our study subjects: LLMs designed for general natural text generation, LLMs tailored for logging activities, and LLMs for code intelligence. We also evaluate state-of-the-art logging suggestion models, which usually work on a single ingredient, to discuss whether advanced LLMs outperform conventional ones.

We summarize the details of eleven LLMs in and three conventional approaches in the following sections. Since we already included official models [164, 165, 166] from the GPT series, other models that have been tuned on GPT [167, 168] are not included in our study (e.g., GPT-Neo [167] and GPT-J [168]).

General-purpose LLMs

The GPT-series models are designed to produce natural language text closely resembling human language. The recent GPT models have demonstrated exceptional performance, dominating numerous natural language generation tasks, such as

question-answering [169] and text summarization [170]. Recently, Meta researchers built an open model, LLaMa, as a family member of LLMs [171], which showed more efficient and competitive results with GPT-series models. In our chapter, we select the two most capable GPT-series models based on previous work [172], i.e., Davinci, ChatGPT for evaluation. We also select one competitive open-sourced model, Llama2, as the representative of general-purpose LLMs.

- Davinci (175B) is derived from InstructGPT [173] is an "instruct" model meant to generate texts with clear instructions. We access the Text-davinci-003 model by calling the official API from OpenAI. We access the model by calling the official API from OpenAI.
- ChatGPT (175B) is an enhanced version of GPT-3 models [166], with improved conversational abilities achieved through reinforcement learning from human feedback [174]. It forms the core of the ChatGPT system [165]. We access the GPT3.5-turbo model by calling the official API from OpenAI.
- Llama2 (70B) [171] is an open-sourced LLM trained on publicly available data and outperforms other open-source conversational models on most benchmarks. We deploy the Llama2-70B model provided by the authors.

Logging-specific LLMs

To the best of our knowledge, LANCE [30] is the only work on training LLMs for automatically generating logging statements, which has been published in top-tier software venues (i.e., FSE, ICSE, ASE, ISSTA, TSE, and TOSEM). Consequently, we choose it as logging-specific LLMs.

- LANCE (60M) [30] accepts a method that needs one logging statement and outputs a proper logging statement in the right position in the code. It is built on the T5 model, which has been trained to inject proper logging statements. We re-implement it based on the replication package [175] provided by the authors.

Code-based LLMs

Inspired by the considerable success of LLMs in the natural language domain, researchers also derive Code-based LLMs that can support code understanding and generation tasks, so as to assist developers in completing codes. These LLMs are either commercial models powered by companies, or open-access models in academia. For the open-access models with publicly available weights, we follow the selection of code models on recent comprehensive evaluation studies [176, 177, 178], and reserve the LLMs with larger sizes than 6B. The process leads to four LLMs as our subjects, i.e., InCoder [153], CodeGeex [179], StarCoder [177], and CodeLlama [176]. In terms of the com-

mercial models, we select three popular developer tools as the study subjects, i.e., TabNine [180], Copilot [156], and CodeWhisperer [158] from Amazon.

- InCoder (6.7B) [153] is a unified generative model trained on vast code benchmarks where code regions have been randomly masked. It thus can infill arbitrary code with bidirectional code context for challenging code-related tasks. We deploy the InCoder-6.7B model provided by the authors.
- CodeGeeX (13B) [179] is an open-source code generation model, which has been trained on 23 programming languages and fine-tuned for code translation. We access the model via its plugin in VS Code.
- StarCoder (15.5B) [177] has been trained on 1 trillion tokens from 80+ programming languages, and fine-tuned on another 35B Python tokens. It outperforms every open LLM for code at the time of release. We deploy the StarCoder-15.5B model provided by the authors.
- CodeLlama (34B) [176] is a family of LLMs for code generation and infilling derived from Llama2. After they have been pretrained on 500B code tokens, they are all fine-tuned to handle long contexts. We deploy the CodeLlama-34B model provided by the authors.

- TabNine [180] is an AI code assistant that can suggest the following lines of code. It can automatically complete code lines, generate entire functions, and produce code snippets from natural languages. We access the model via its plugin in VS Code.
- Copilot [156] is a widely-studied AI-powered code generation tool relying on the CodeX [164]. It can extend existing code by generating subsequent code trunks based on natural language descriptions. We access the model via its plugin in VS Code.
- CodeWhisperer [158], developed by Amazon, serves as a coding companion for software developers. It can generate code snippets or full functions in real-time based on comments written by developers. We access the model via its plugin in VS Code.

Conventional Logging Approaches

Apart from LLMs that can offer complete logging statements, we also select conventional logging approaches that work on *single* logging ingredients for comparison. Specifically, for each ingredient, we choose the corresponding state-of-the-art logging approaches from the top-tier software venues: DeepLV [48] for log level prediction, Liu et al.'s [28] (denoted as WhichVar) for logging variable prediction, and LoGenText-Plus [181] for logging text generation. These approaches learn the relation-

ships between specific logging ingredients and the corresponding code features based on deep learning techniques[2].

- DeepLV [48] leverages syntactic context and message features of the logging statements extracted from the source code to make suggestions on choosing log levels by feeding all the information into a deep learning model. We reimplement the model based on the replication package provided by the authors.
- WhichVar [28] applies an RNN-based neural network with a self-attention mechanism to learn the representation of program tokens, then predicts whether each token should be logged through a binary classifier. We reimplement the model based on its chapter due to missing code artifacts.
- LoGenText-Plus [181] generates the logging texts by neural machine translation models (NMT). It first extracts a syntactic template of the target logging text by code analysis, then feeds such templates and source code into Transformer-based NMT models. We reproduce the model based on the replication package provided by the authors.

[2]All the baselines we have reimplemented have been organized in our artifacts.

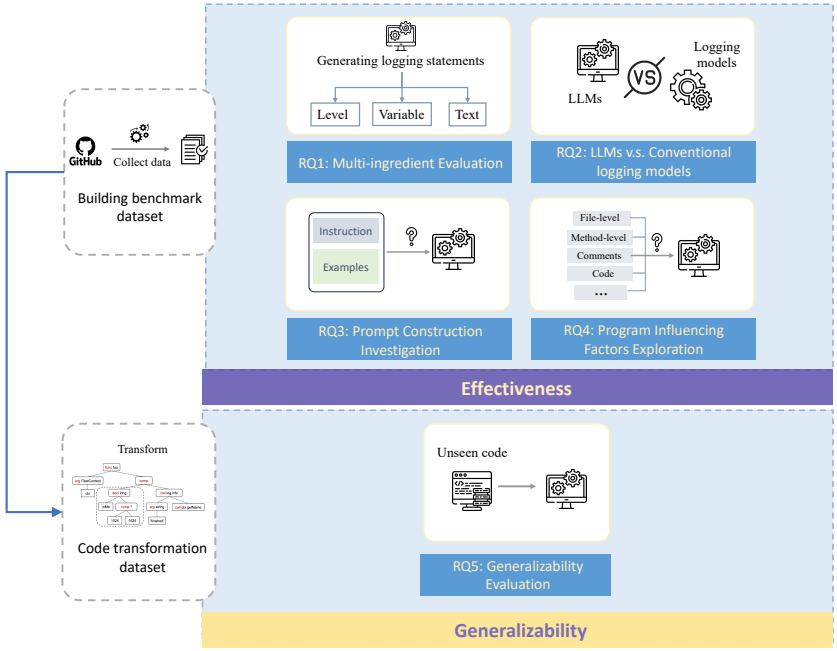


Figure 6.2: The overall framework of this study involving five research questions.

6.3 Study Methodology

6.3.1 Overview

Figure 6.2 depicts the overview framework of this study involving five research questions from two perspectives: (1) *effectiveness*: how do LLMs perform in logging practice? and (2) *generalizability*: how well do LLMs generate logging statements for unseen code?

To start, we develop a benchmark dataset LogBench-O comprising *6,849* logging statements in *3,870* methods by crawling high-quality GitHub repositories. Inspired by the success of LLMs in NLP and code intelligence tasks, our focus is on assessing their efficacy in helping developers with logging tasks. This

study first evaluates the effectiveness of state-of-the-art LLMs in terms of multiple logging ingredients (RQ1). We then conduct a comparative analysis between state-of-the-art conventional logging tools and LLMs, elucidating differences and providing insights into potential future model directions (RQ2). Next, we investigate the impact of instructions and demonstrations as inputs for LLMs, offering guidance for effectively prompting LLMs for logging (RQ3). Furthermore, we investigate how external influencing factors can enhance LLM performance, identifying effective program information that should be input into LLMs to improve logging outcomes (RQ4). Last but not least, we explore the generalizability of LLMs to assess their behavior in developing new and unseen software. To this end, we evaluate models on an unseen code dataset, LogBench-T, which contains code derived from LogBench-O that was transformed to preserve readability and semantics (RQ5).

6.3.2 Benchmark Datasets

Due to the lack of an existing dataset that can meet the benchmark requirements, we developed the benchmark dataset LogBench-O and LogBench-T for logging statement generation in this section. Although we chose Java as the target language of our study, due to its wide presence in industry and research [182], the experiments and findings can be extended to other programming languages.

Creation of LogBench-O

We build a benchmark dataset, consisting of high-quality and well-maintained Java files with logging statements, by mining open-source repositories from GitHub. As the largest host of source code in the world, GitHub contains a great number of repositories that reflect typical software development processes. In particular, we begin by downloading high-quality Java repositories that meet the following requirements[3]:

- Gaining more than *20* stars, which indicates a higher level of attention and interest in the project.
- Receiving more than *100* commits, which suggests the project is actively maintained and not likely to be disposable.
- Engaging with at least *5* contributors, which demonstrates the quality of its logging statements by simulating the collaborative software development environment.

We then extract the files that contain logging statements in two steps. We first select the projects whose POM file includes popular logging utility dependencies (e.g., Log4j, SLF4J), resulting in 3,089 repositories. We then extract the Java files containing at least one logging statement by matching them with regular expressions [41], because logging statements are always written in specified syntax (e.g., `log.info()`). Afterward, we randomly sample the collected files across various repositories, re-

[3]All repositories were archived on July 2023

Table 6.2: Our code transformation tools with eight code transformers, descriptions, and associated examples.

Transformer	Descriptions & Example
Condition-Dup	Add logically neutral elements (e.g., *&& True* or *|| False*) if (exp0) → if (exp0 || false)
Condition-Swap	Swap the symmetrical elements of condition statements if (var0 != null) → if (null != var0)
Local variable	Extract constant values and assign them to local variables var0 = const0; → int var1 = const0; var0 = var1;
Assignment	Separate variable declaration and assignment int var0 = var1; → int var0; var0 = var1;
Constant	Replace constant values with equivalent expressions int var0 = const0 → int var0 = const0 + 0
For-While	Convert *for-loops* to equivalent *while-loops* for (var0 = 0; var0 < var1; var0++) {} ↔
While-For	Convert *while-loops* to equivalent *for-loops* var0 = 0; while (var0++ < var1) {}
Parenthesis	Add redundant parentheses to expression var0 = arithExpr0 → var0 = (arithExpr0)

sulting in a dataset of 2,420 files containing 3,870 methods and 6,849 logging statements, which we refer to as LogBench-O.

Creation of LogBench-T Dataset to Avoid Data Leakage

LLMs deliver great performance in multiple tasks; however, evaluating their performance solely on publicly available data can be problematic. Since LLMs are trained on datasets that are obtained through large-scale web scraping [183], these models may have already seen the benchmark data during their training, raising concerns about assessing their generalization abili-

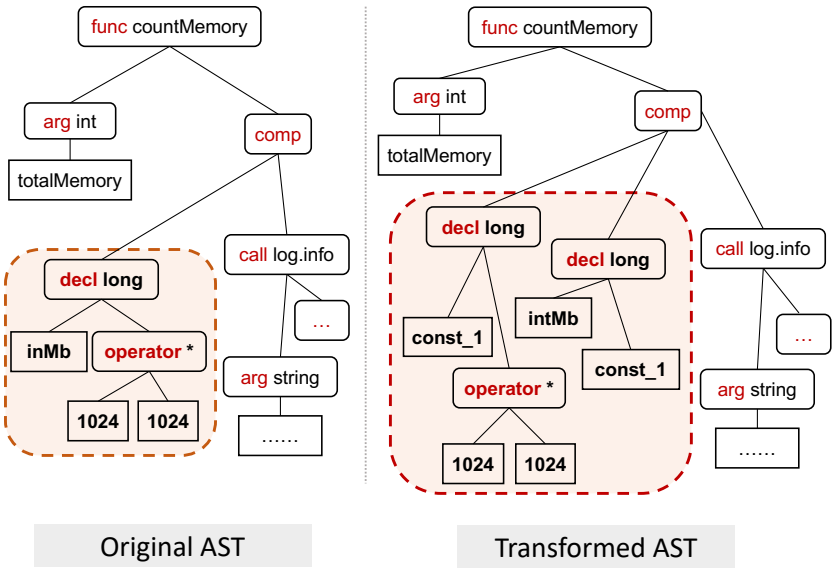


Figure 6.3: An example of how the code (constant) transformer works.

ties [155, 160, 161]. This issue, commonly known as *data leakage*, requires particular attention since most code models [153] have been trained on public code.

To fairly evaluate the generalization ability of LLMs, we further develop an unseen code dataset LogBench-T that consists of the code transformed from LogBench-O. Prior works have developed *semantics-preserving* code transformation techniques that do not change the functionality of the original code, for the purpose of evaluating the robustness of code models [184, 185, 186, 187]. However, these approaches randomly replace informative identifiers with meaningless ones, degrading the *readability* of the code. For example, after transforming an informative variable name (e.g., `totalMemory`) to a non-informative name (e.g., `var0`), even a programmer can hardly understand the variables

and log properly. Such transformations make the transformed code less likely to appear in daily programming and not suitable for logging practice studies. To avoid this issue, we devise a code transformation tool that generates semantics-preserving and readability-preserving variations of the original code.

In particular, our code transformation tool employs eight carefully engineered, lightweight code transformers motivated by previous studies [184, 186, 188, 189], whose descriptions, together with their examples, are illustrated in Table 6.2. These code transformation rules work at the Abstract Syntax Tree (AST) level, ensuring that the transformed code remains semantically equivalent to the original code. Besides, readability-degrading transformations, such as injecting dead code [190] and modifying the identifier names, are eliminated. Additionally, to affirm the soundness of our transformations, we have limited our selection to widely-used transformation rules that have been proven effective in various code-related tasks [187, 184, 191] over time. Transformation rules are further verified by executing unit tests on sample projects, which confirm that our code transformations will not hurt functionality.

The process of transformation begins with converting the source code into an AST representation using JavaParser [192]. To detect potential transformation points (i.e., specific nodes and subtrees) for each transformer, a series of predefined checkers traverse the AST in a top-down manner. Once the transformation points are identified, each checker will independently

call its corresponding transformer to perform a one-time transformation. We denote one-time transformation as $T : x \rightarrow x'$, where x and x' represent the source AST and the transformed AST, respectively. Each transformer functions independently, allowing multiple transformations to be applied to the same code snippet without conflicts. These single transformations are chained together to form the overall transformation: $\mathbb{T} = T_1 \circ T_2 \circ ... \circ T_n$. Once all the identified points have been transformed or the number of transformations reaches a predetermined threshold, the AST is converted back into the source code to complete the transformation process. Figure 6.3 exhibits a case concerning how a `Local variable` transformer works. The constant checker firstly detects transformation points, then the Local variable transformer replaces the constant expression `{inMb=1024*1024}` by `{const_1=1024*1024; inMb=const_1}` involving a new variable *const_1*. The AST changes via transformation are highlighted in red area.

6.3.3 Implementations

Evaluation

Based on the access ways offered by different LLMs, we evaluated them as follows.

(1) Released models (Llama2, LANCE, InCoder, StarCoder, CodeLlama): we ran them on a 32-Core workstation with an Intel Xeon Platinum 8280 CPU, 256 GB RAM, and 4x NVIDIA

GeForce RTX 4090 GPUs in Ubuntu 20.04.4 LTS, using the default bit precision settings for each model.

(2) APIs (ChatGPT, Davinci): we called their official APIs to generate the logging statement by providing the following instruction: *Please complete the incomplete logging statement at the logging point: [Code with corresponding logging point].* As we discussed in Section 6.4.4, we choose the median value of all metrics across the top five instructions, as determined by voting, to approximate the instructions most commonly utilized by developers. We set its temperature to 0 so that ChatGPT would generate the same output for the same query to ensure reproducibility. For ChatGPT and Davinci, we use the public APIs provided by OpenAI with gpt-3.5-turbo-0301 and text-davinci-003, respectively.

(3) Plugins (Copilot, CodeGeeX, TabNine, CodeWhisperer): we purchased accounts for each author to obtain the logging statement manually at the logging point that starts with the original logging API (e.g., `log.`). This starting point forces these plugins to generate logging statements instead of other functional codes.

For conventional logging approaches, we reproduced them based on the replication packages released by the authors, or the paper descriptions if the replication package is missing. For all experiments that may introduce randomness, to avoid potential random bias, we repeat them three times and report the median results following previous works [193, 194, 195].

Code Transformation

Our code transformation technique (Section 6.3.2) was implemented using 4,074 lines of Java code, coupled with the JavaParser library [192], a widely-used parser for analyzing, transforming, and generating Java code. All transformations were performed on the same workstation as in the evaluation.

6.4 Result analysis

6.4.1 Metrics

In line with prior work [2], we evaluate the logging statement generation performance concerning three ingredients: *logging levels*, *logging variables*, and *logging texts*. Although different ingredients emphasize various aspects of runtime information, they are indispensable and complementary resources for engineers to reason about system behavior.

(1) Logging levels. Following previous studies [48, 27], we use the level accuracy *(L-ACC)* and Average Ordinal Distance Score *(AOD)* for evaluating logging level predictions. L-ACC measures the percentage of correctly predicted log levels out of all suggested results. AOD [48] considers the distance between logging levels. Consequently, given the five logging levels in their severity order, i.e., `error, warn, info, debug, trace`, the distance of $Dis(error, warn) = 1$ is shorter than the distance of $Dis(error, info) = 2$. AOD takes the average distance

between the actual logging level a_i and the suggested logging level (denoted as $Dis(a_i, s_i)$). AOD is therefore formulated as $AOD = \frac{\sum_{i=1}^{N}(1-Dis(a_i,s_i)/MaxDis(a_i))}{N}$, where N is the number of logging statements and $MaxDis(a_i)$ refers to the maximum possible distance of the actual log level.

(2) Logging variables. Evaluating predictions from LLMs is different from neural-based classification networks, as the predicted probabilities of each variable are not known. We thus employ *Precision*, *Recall*, and *F1* to evaluate predicted logging variables. For each predicted logging statement, we use S_{pd} to denote variables in LLM predictions and S_{gt} to denote the variables in the actual logging statement. We report the proportion of correctly predicted variables (precision=$\frac{S_{pd} \cap S_{gt}}{S_{pd}}$), the proportion of actual variables predicted by the model (recall=$\frac{S_{pd} \cap S_{gt}}{S_{gt}}$), and their harmonic mean (F1=$2 * \frac{Precision * Recall}{Precision + Recall}$).

(3) Logging texts. To align with previous research [30, 29], we assess the quality of the produced logging texts using two well-established machine translation evaluation metrics: *BLEU* [196] and *ROUGE* [197]. These n-gram metrics compute the similarity between generated log messages and the actual logging text crafted by developers, yielding a percentage score ranging from *0* to *1*. A higher score indicates greater similarity between the generated log messages and the actual logging text. In particular, we use BLEU-K ($K = \{1, 2, 4\}$) and ROUGE-K ($K = \{1, 2, L\}$) to compare the overlap concerning K-grams between the generated and the actual logs. In

Table 6.3: The effectiveness of LLMs in predicting logging levels and logging variables.

Model	Logging Levels		Logging Variables		
	L-ACC	**AOD**	**Precision**	**Recall**	**F1**
General-purpose LLMs					
Davinci	0.631	0.834	0.634	0.581	0.606
ChatGPT	0.651	0.835	0.693	0.536	0.604
Llama2	0.595	0.799	0.556	0.608	0.581
Logging-specific LLMs					
LANCE†	0.612	0.822	0.667	0.420	0.515
Code-based LLMs					
InCoder	0.608	0.800	0.712	0.655	0.682
CodeGeex	0.673	0.855	0.704	0.616	0.657
TabNine	0.734	0.880	0.729	0.670	0.698
Copilot	**0.743**	**0.882**	0.722	**0.703**	0.712
CodeWhisperer	0.741	0.881	**0.787**	0.668	**0.723**
CodeLlama	0.614	0.814	0.583	0.603	0.593
StarCoder	0.661	0.829	0.656	0.649	0.653

addition to the token-based match in a sparse space, we also incorporate *semantic similarity* in our evaluation. Following prior works [198, 199, 194], we also leverage widely-used code embedding models, UniXcoder [154] and OpenAI embedding [200], to embed the logging texts to calculate the semantics similarity between generated and original logging texts, offering another evaluation metric from a semantic perspective.

6.4.2 RQ1: How do different LLMs perform in deciding ingredients of logging statements generation?

To answer RQ1, we evaluate eleven top-performing LLMs on the benchmark dataset LogBench-O. The evaluation results are shown in Table 6.3 (levels, variables) and Table 6.4 (logging

texts), where we **underline** the best performance score for each metric. † Note that since LANCE decides logging point and logging statements simultaneously, we only consider its generated logging statements with correct locations.

Intra-ingredient. Regarding the logging levels, we observe that Copilot achieves the best L-ACC performance, i.e., 0.743, indicating that it can accurately predict 74.3% of the logging levels. While other baselines do not perform as well as Copilot, they also accurately suggest logging levels for at least *60%* logging statements. Compared with logging levels, there are greater differences among models when recommending logging variables. While 70% of the variables are recommended by Copilot, LANCE can only correctly infer 42% of them. The recall rate for variable prediction is consistently lower than the precision rate across models, indicating the difficulty of identifying many of variables. Predicting variables is more challenging than logging levels, as variables are diverse, customized, and have different meanings across systems. To address this challenge, logging variables should be inferred based on a deeper comprehension of code structure, such as control flow information.

Concerning logging text generation shown in Table 6.4, both Copilot and CodeWhisperer demonstrate comparable performance across syntax-based metrics (BLEU, ROUGE) and semantic-based metrics, outperforming other baselines by a wide margin. The comparison between syntax-based metrics and

Table 6.4: The effectiveness of LLMs in producing logging texts.

Model	Logging Texts						
	BLEU-1	**BLEU-2**	**BLEU-4**	**ROUGE-1**	**ROUGE-2**	**ROUGE-L**	**Semantics Similarity**
General-purpose LLMs							
Davinci	0.288	0.211	0.138	0.295	0.127	0.286	0.617
ChatGPT	0.291	0.217	0.149	0.306	0.142	0.298	0.633
Llama2	0.235	0.168	0.102	0.264	0.116	0.261	0.569
Logging-specific LLMs							
LANCE†	0.306	0.236	0.167	0.162	0.078	0.162	0.347
Code-based LLMs							
InCoder	0.369	0.288	0.203	0.390	0.204	0.383	0.640
CodeGeex	0.330	0.248	0.160	0.339	0.149	0.333	0.598
TabNine	0.406	0.329	0.242	0.421	0.241	0.415	0.669
Copilot	**0.417**	**0.338**	0.244	**0.435**	0.247	**0.428**	**0.703**
CodeWhisperer	0.415	0.338	**0.249**	0.430	**0.248**	0.425	0.672
CodeLlama	0.216	0.146	0.089	0.258	0.103	0.251	0.546
StarCoder	0.353	0.278	0.195	0.378	0.195	0.369	0.593

semantic-encoding metrics reveals a consistent trend across various LLMs: models exhibiting strong syntax similarity also exhibit high semantic similarity. On average, the studied models produce logging statements with a similarity of 0.194 and 0.341 for BLEU-4 and ROUGE-L scores, respectively. The result indicates that recommending appropriate logging statements remains a great challenge.

Finding 1. *While existing models correctly predict levels for 74.3% of logging statements, there is significant room for improvement in producing logging variables and logging texts.*

Inter-ingredient. From the inter-ingredient perspective, we observe that LLM performance trends are *not consistently the same* across various ingredients, e.g., models that perform well in logging level prediction do not necessarily excel in generating

logging texts. For instance, Incoder fares worst in predicting logging levels but performs better in generating logging texts (the fourth best performer). Upon manual investigation, we observe that Incoder predicts *41%* of the cases with a `debug` level, most of which are actually intended for the `info` level statements. Nevertheless, either Copilot or CodeWhisperer outperforms other baselines in all reported metrics. This is likely because suggesting the three ingredients requires similar code comprehension capabilities, such as understanding data flows, specific code structures, and inferring code functionalities.

Finding 2. *LLMs may perform inconsistently on deciding different ingredients, making model comparisons more difficult based on multiple ingredient-wise metrics.*

6.4.3 RQ2: How do LLMs compare to conventional logging models in logging ability?

We compare the results of directly using LLMs for logging against conventional logging models on LogBench-O. As conventional logging models can only predict one ingredient, we opt for state-of-the-art models for each one (i.e., DeepLV, WhichVar, and LoGenText-Plus) and present their performance against LLMs in Figure 6.4. The boxplot illustrates the performance range of LLM-powered models, while the points depict conventional logging models.

Despite being carefully designed for the logging task, the

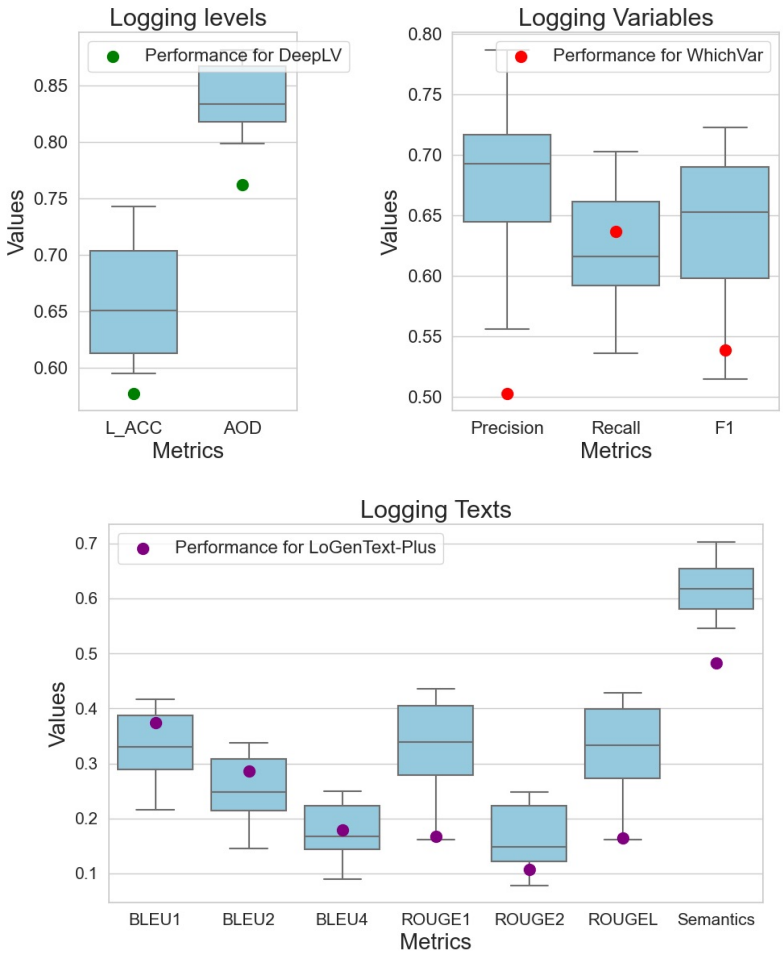


Figure 6.4: Comparison between traditional logging models and LLM-powered models.

conventional logging models do not surpass LLMs. As shown in Figure 6.4, conventional models exhibit inferior performance compared to any LLMs on five metrics (i.e., below the lower whiskers) and fall below the median on the other three metrics (i.e., below the line in the box). In terms of logging level prediction, DeepLV performs worse than any of our studied LLMs, correctly predicting only 57.7% of statements. Regarding generating logging variables and texts, WhichVar and LoGenText-Plus show comparable performance to LANCE, but lag behind other studied LLMs. While the most effective model (Copilot) achieves a 0.703 semantic-based similarity in logging texts, the state-of-the-art logging model, LoGenText-Plus, only produces a 0.485 similarity (yielding a 21.8% drop). These surprising results show that, *without any specific change or fine-tuning, directly applying LLMs for logging statement generation yields better performance compared to conventional logging baselines.*

Figure 6.5 displays the Venn diagram illustrating the logging levels correctly predicted by DeepLV in comparison to three chosen LLMs on the LogBench-O dataset. Notably, 97% of the cases handled by DeepLV can also be predicted by LLMs. In contrast, DeepLV can only handle 70%, 62%, and 60% of the cases successfully predicted by Copilot, ChatGPT, and StarCoder, respectively.

To demonstrate the ability of LLMs, we present Figure 6.6 to illustrate some statements produced by ChatGPT, InCoder, Copilot, and TabNine, respectively. Through pre-training, these

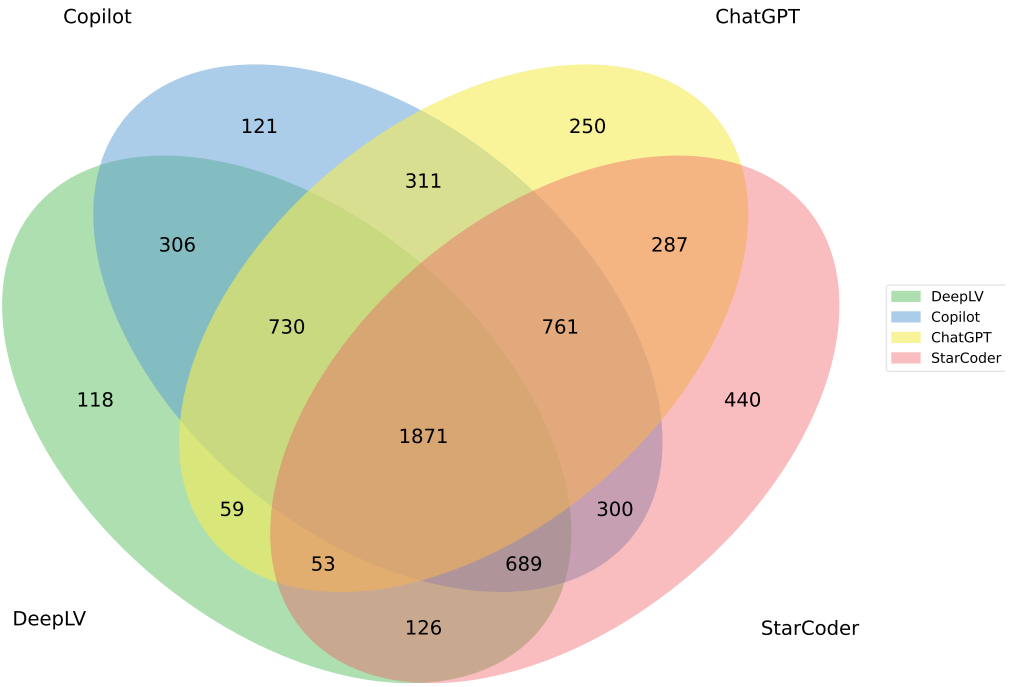


Figure 6.5: Venn diagram for logging levels prediction.

```
public Object addingBundle(Bundle bundle, BundleEvent event)
{       ...
        List<Driver> drivers = loadDrivers(bundle, providerURL);
        register(drivers);
+       LOG.debug("Registered {} drivers in bundle {}", drivers, bundle);
          ...
```

Models	Generated Logging Statements
ChatGPT	LOG.info(“Bundle {0} has been added in event.", bundle.getName())
InCoder	LOG.info("Registered JDBC drivers: {}", drivers);
TabNine	LOG.info("Registered " + drivers.size() + " drivers")
Copilot	LOG.debug("Found " + drivers.size() + " registered drivers in bundle” + bundle);

Figure 6.6: An example of the generation results from eight models.

LLMs gain a basic understanding of method activity in adding bundles with drivers, leading to the generation of relevant logging variables. Notably, code-based LLMs produce more accurate logging statements compared to models pre-trained for general purposes. In Figure 6.6, general-purpose LLMs (i.e., ChatGPT) mispredict the logging statement by focusing on the `event` variable in the method declaration, overlooking the driver registration process preceding the logging point. Conversely, most code models (e.g., InCoder) capture such processes, recognizing that `drivers` are critical variables describing a device status. We attribute the performance difference to the gap between natural and programming languages. Training on a code base enables these models to acquire programming knowledge, bridging the gap and enhancing logging performance.

Finding 3. *When directly applying LLMs to logging statement generation, without fine-tuning, they still yield better performance than conventional logging baselines.*

6.4.4 RQ3: How do the prompts for LLMs affect logging performance?

Previous literature has identified the variance of input prompts can significantly affect the performance of LLMs' [198]. For the LLMs that can take prompts (e.g., ChatGPT, LLaMa2), we investigate the influences of instructions and demonstrate examples for their logging purpose.

Impact of different instructions. LLMs have been shown to be sensitive to the instructions used to query the LLM sometimes. To compare the impact of different instructions, we conducted a two-round survey involving 54 developers from a world-leading technical company, each possessing a minimum of two years of development experience. To begin with, we ask the developers to individually propose 10 instructions that they would consider when utilizing LLMs for generating logging statements. Subsequently, we distributed a second questionnaire, asking developers to choose the top 5 instructions from the initial round that they likely to employ. Eventually, instructions receiving the top 5 votes will be considered for evaluation, shown as follows.

1. Your task is to generate the logging statement for the corresponding position.
2. You are an expert in software DevOps; please help me write the informative logging statement.
3. Complete the logging statement while taking the surrounding code into consideration.
4. Your task is to write the corresponding logging statement. Note that you should keep consistent with current logging styles.
5. Please help me write an appropriate logging statement below.

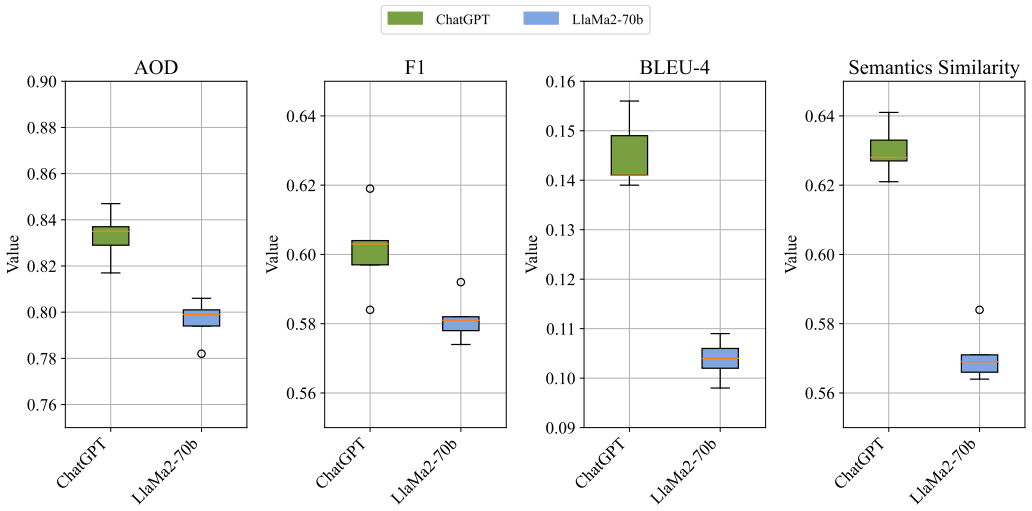


Figure 6.7: The selected metrics of LLMs' logging performance with different instructions.

We then feed these representative instructions into two studied LLMs, that is, ChatGPT, and LLaMa2, respectively. The box plot in Figure 6.7 exhibits logging performance associated with different instructions. The selected instructions result in approximately 3% performance variance for each metric, revealing the importance of designing prompts. Among all metrics, the difference in logging variable prediction for ChatGPT is slightly larger, but still in the range of 4% variation. Despite there being small variations due to different instructions, these variances do not alter the consistent superiority of ChatGPT over LLaMa2. In summary, as long as the logging ability of LLMs is evaluated using the same instructions, such evaluation and comparison are meaningful.

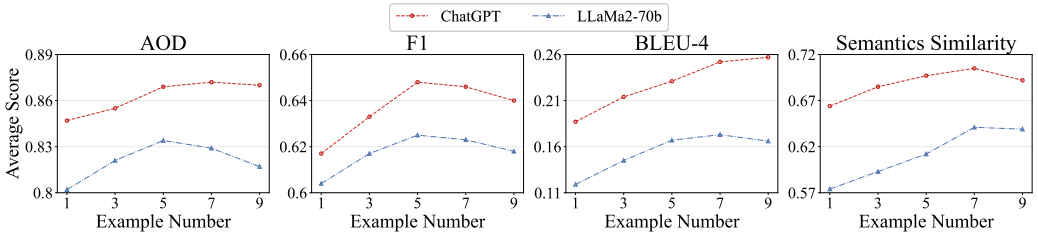


Figure 6.8: The selected metrics of LLMs' logging performance with different numbers of examples.

Finding 4. *Although instructions influence LLMs to varying extents, there is cohesiveness in the relative ranking of LLMs with the same instructions.*

Impact of different numbers of logging examples. In-context learning (ICL) is a prevalent prompt strategy, enabling LLMs to glean insights from few-shot examples in the context. Many studies have shown that LLMs can boost complicated code intelligence tasks through ICL implementation [198]. Despite being promising, there are intriguing properties that require further exploration, for example, the effects of parameter settings in ICL.

Figure 6.8 presents the logging performance (i.e., logging level, variable, texts) in terms of different numbers of demonstration examples provided. In this experiment, we vary the number of demonstrations for ChatGPT and LLaMa2 from 1 to 9. We select and order demonstration examples measured by using BM25 retrieval methods, as previous works have demonstrated its effectiveness in code tasks [198].

The figure illustrates the impact of the number of demonstration examples on LLMs' logging performance, resulting in a increment of 2%-8%. Initially, the performance of ICL improves across all metrics as the number of demonstration examples increases. However, when the number of examples surpasses 5, divergent trends emerge for different tasks. For instance, in determining logging levels (AOD) and logging variables (F1), the LLaMa performance peaks at 5 demonstration examples but experiences a decline with further increments to 7. Conversely, in logging text generation (BLEU-4, Semantics Similarity), LLaMa performance continues to rise and stabilizes beyond 7 examples. We attribute these diverse trends to the *model distraction problem* [201]. Tasks involving predicting logging levels and variables demand an intricate analysis of individual program structures and variable flows, and the introduction of additional examples with longer input lengths can potentially distract the model, leading to performance degradation. In contrast, logging text generation involves a high-level program understanding and summarization. More examples allow LLMs to learn proper logging styles from other demonstrations.

Finding 5. *More demonstration examples in the prompt do not always improve performance. It is recommended to use 5-7 examples in the demonstration to achieve optimal results.*

Table 6.5: The results of logging statement generation without comments.

Model	Logging Levels	Logging Variables	Logging Texts		
	AOD	F1	BLEU-4	ROUGE-L	Semantics Similarity
Davinci	0.834 (0.0%-)	0.587 (3.1%↓)	0.133 (3.6%↓)	0.283 (1.0%↓)	0.608 (1.5%↓)
ChatGPT	0.833 (0.2%↓)	0.592 (2.0%↓)	0.149 (0.0%-)	0.294 (1.3%↓)	0.614 (3.0%↓)
Llama2	0.789 (1.3%↓)	0.574 (1.2%↓)	0.099 (2.9%↓)	0.255 (2.3%↓)	0.544 (4.4%↓)
InCoder	0.789 (1.4%↓)	0.674 (1.2%↓)	0.201 (1.0%↓)	0.377 (9.2%↓)	0.622 (2.8%↓)
CodeGeex	0.848 (0.8%↓)	0.617 (6.1%↓)	0.149 (6.9%↓)	0.306 (8.1%↓)	0.578 (3.3%↓)
TabNine	0.876 (0.5%↓)	0.690 (1.1%↑)	0.239 (1.2%↓)	0.412 (0.7%↓)	0.655 (2.1%↓)
Copilot	**0.878** (0.5%↓)	0.696 (2.2%↓)	0.241 (1.2%↓)	**0.419** (2.1%↓)	**0.689** (2.0%↓)
CodeWhisperer	0.877 (0.7%↓)	**0.718** (0.7%↓)	**0.244** (2.0%↓)	0.418 (1.6%↓)	0.661 (1.6%↓)
CodeLlama	0.804 (1.2%↓)	0.581 (2.0%↓)	0.087 (2.2%↓)	0.247 (1.6%↓)	0.544 (0.3%↓)
StarCoder	0.823 (0.7%↓)	0.647 (0.9%↓)	0.193 (1.0%↓)	0.369 (2.4%↓)	0.591 (**0.3%**↓)
Avg.Δ	0.835 (0.8%↓)	0.638 (2.1%↓)	0.173 (2.2%↓)	0.338 (3.0%↓)	2.1%↓

6.4.5 RQ4: How do external factors influence the effectiveness in generating logging statements?

While RQ3 discusses the prompt construction for LLMs, some external program information is likely to affect their effectiveness in logging generation. In particular, we focus on how *comments* and *the scope of programming contexts* will impact the model performance.

With comment v.s. without comment. Inspired by the importance of human-written comments for intelligent code analysis [154, 202, 203], we also explore the utility of comments for logging. To this end, we feed the original code (with comment) and comment-free code into LLMs separately, compare their results, and analyze the corresponding performance drop rate (Δ) in Table 6.5 in terms of AOD, F1, BLEU, and ROUGE score. The results show that LLMs consistently encounter performance drops without comments, with an average drop rate on *0.8%*, *2.1%*, *2.2%*, and *3.0%* for AOD, F1, BLEU-4, and

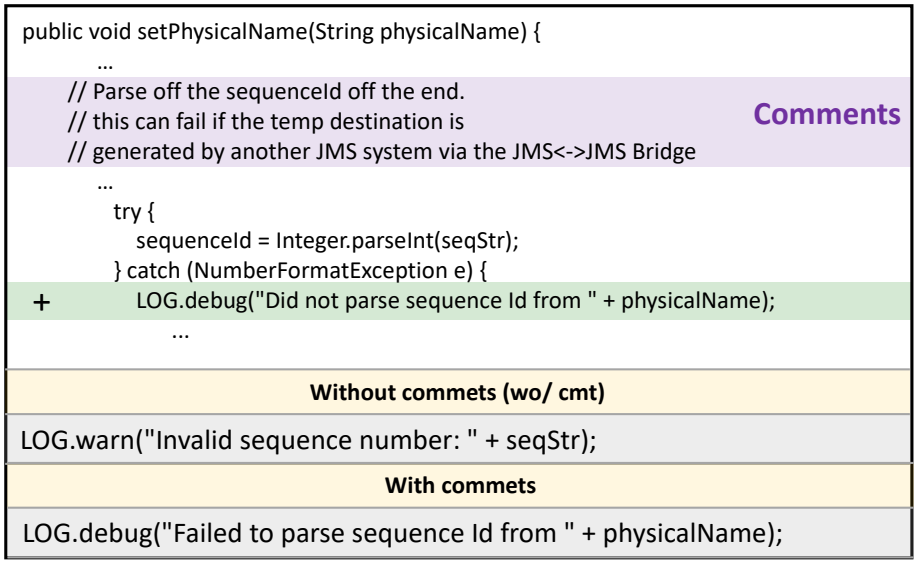


Figure 6.9: A logging statement generation case using code comments.

ROUGE-L, respectively. The reason is that, comments are used to describe the functionalities of the corresponding code, thus sharing similarities to logging practices that record system activities.

Figure 6.9 presents an example with CodeWhisperer that can be facilitated by reading the comment of `parse sequence Id`. Without the comment, CodeWhisperer only concentrates on the invalid sequence number but fails to involve parsing descriptions, which may further mislead maintainers on parsing failure diagnosis. Moreover, the comments highlight that the exception is a foreseeable and potentially common issue, which helps the LLMs in correctly selecting the log level, changing the logging level from `warn` to `debug`.

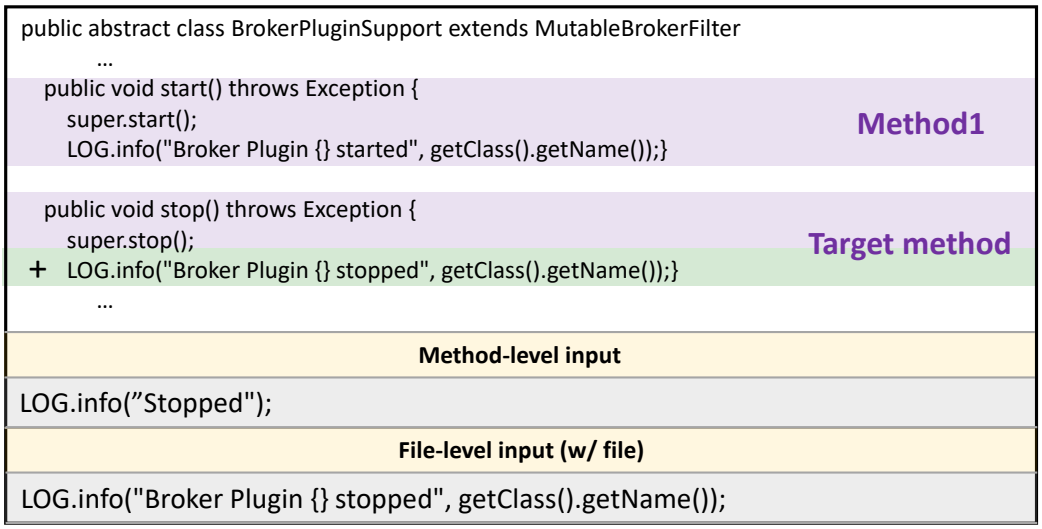


Figure 6.10: A logging statement generation case using different programming contexts.

Finding 6. *Ignoring code comments impedes LLMs in generating logging statements, resulting in an average 2.43% decrease when recommending logging texts.*

Programming contexts: method v.s. file. Current logging practice tools restrict their work on code snippets or methods [30, 29, 28], and ignore the information from other related methods [204]. However, methods that implement similar functionalities can contain similar logging statements [150], which can be used as references to resolve logging statements. In past works, this constraint was mainly due to the limits in input size in previous neural-based models. But since LLMs can now process thousands of input tokens without suffering from such limitations, we aim to assess the benefits of larger programming contexts, i.e., file-level input.

Table 6.6: The results of logging statement generation with file-evel contexts.

Model	Logging Levels	Logging Variables	Logging Texts		
	AOD	F1	BLEU-4	ROUGE-L	Semantics Similarity
Davinci	0.854 (2.6%↑)	0.638 (5.3%↑)	0.156 (13.0%↑)	0.318 (11.2%↑)	0.635 (2.9%↑)
ChatGPT	0.858 (2.8%↑)	0.650 (7.6%↑)	0.253 (51.5%↑)	0.389 (30.5%↑)	0.704 (11.2%↑)
Llama2	0.832 (4.1%↑)	0.617 (6.2%↑)	0.149 (46.1%↑)	0.392 (50.2%↑)	0.669 (17.6%↑)
InCoder	0.815 (1.9%↑)	0.745 (9.2%↑)	0.307 (51.2%↑)	0.521 (35.3%↑)	0.734 (11.7%↑)
CodeGeex	0.869 (1.6%↑)	0.696 (5.9%↑)	0.241 (50.6%↓)	0.395 (18.6%↑)	0.644 (7.7%↑)
TabNine	0.912 (3.6%↑)	0.767 (9.9%↑)	0.375 (55.0%↑)	0.530 (27.7%↑)	0.783 (17.0%↑)
Copilot	**0.916** (3.9%↑)	0.742 (4.2%↑)	0.346 (41.8%↑)	0.522 (22.0%↑)	**0.816** (16.1%↑)
CodeWhisperer	0.913 (3.6%↑)	**0.792** (9.6%↑)	**0.401** (61.0%↑)	**0.559** (31.5%↑)	0.811 (20.7%↑)
CodeLlama	0.817 (0.4%↑)	0.607 (2.4%↑)	0.144 (61.8%↑)	0.378 (50.6%↑)	0.642 (17.6%↑)
StarCoder	0.847 (2.2%↑)	0.714 (9.3%↑)	0.314 (61.0%↑)	0.517 (40.1%↑)	0.679 (14.5%↑)
Avg.Δ	2.7%↑	6.9%↑	49.3%↑	31.8%↑	13.7%↑

In this regard, we feed *an entire Java file* for generating logging statements rather than *the target method*. The result in Table 6.6 presents the effectiveness of file-level input (w/ File) and the corresponding increment ratio (Δ). The result suggests that file-level programming contexts consistently enhance performance in terms of all metrics where, for example, TabNine increases *3.6%*, *9.9%*, and *55.0%* for AOD, F1, and BLEU score, respectively. On average, all models generate logging statements that are *49.3%* more similar to actual ones (reflected by BLEU-4) than using a single method as input. We take Figure 6.10 as an example from CodeWhisperer to illustrate how LLMs can learn from an additional method, where the green line represents the required logging statements. The model learned logging patterns from Method1, which includes the broker plugin name and its status (i.e., `start`). Regarding `stop()`, CodeWhisperer may refer to Method1 and write similar logging statements by changing the status from `started` to `stopped`. Additionally, by analyzing the file-level context, LLMs can identify pertinent variables, learn

Table 6.7: The generalization ability of LLMs in producing logging statements for unseen code.

Model	Levels		Variables		Texts						Average
	AOD	Δ	F1	Δ	BLEU-4	Δ	ROUGE-L	Δ	Semantics	Δ	Avg. Δ
General-purpose LLMs											
Davinci	0.820	1.7%↓	0.523	13.7%↓	0.116	15.9%↓	0.234	20.7%↓	0.533	13.6%↓	13.1%↓
ChatGPT	0.830	0.6%↓	0.532	11.9%↓	0.118	20.8%↓	0.240	19.5%↓	0.541	14.5%↓	13.5%↓
Llama2	0.788	1.4%↓	0.568	2.2%↓	0.094	7.8%↓	0.213	18.4%↓	0.513	9.8%↓	7.9%↓
Logging-specific LLMs											
LANCE	0.817	0.6%↓	0.475	7.5%↓	0.153	8.4%↓	0.144	**11.1%**↓	0.301	13.3%↓	**8.2%**↓
Code-based LLMs											
InCoder	0.778	2.8%↓	0.587	13.9%↓	0.175	13.8%↓	0.316	17.5%↓	0.584	8.8%↓	11.4%↓
CodeGeex	0.850	0.6%↓	0.534	18.7%↓	0.115	28.1%↓	0.253	25.4%↓	0.549	8.2%↓	16.2%↓
TabNine	0.869	1.3%↓	0.596	14.6%↓	0.202	16.5%↓	0.342	18.8%↓	0.608	9.1%↓	12.1%↓
Copilot	**0.881**	**0.1%**↓	0.610	14.3%↓	**0.234**	**4.1%**↓	**0.377**	13.3%↓	**0.641**	8.8%↓	8.2%↓
CodeWhisperer	0.871	1.1%↓	**0.629**	13.0%↓	0.219	12.0%↓	0.362	14.6%↓	0.612	8.9%↓	9.9%↓
CodeLlama	0.801	1.6%↓	0.574	**3.2%**↓	0.078	12.6%↓	0.211	15.9%↓	0.482	11.7%↓	9.0%↓
StarCoder	0.811	2.2%↓	0.619	5.2%↓	0.175	10.3%↓	0.309	16.3%↓	0.546	**7.9%**↓	8.4%↓
Avg. Δ	-	1.4%↓	-	11.6%↓	-	15.0%↓	-	19.2%↓		10.4%↓	11.5%↓

relationships between multiple methods, and recognize consistent logging styles within the file. Last but not least, the comparison of Table 6.5 and Table 6.6 implies that expanding the range of programming texts has a stronger impact than incorporating comments, even though certain models (e.g., Copilot) are trained to generate code from natural language.

Finding 7. *Compared to comments, incorporating file-level programming contexts leads to a greater improvement in logging practice by providing access to additional functionality-similar methods, variable definitions and intra-project logging styles.*

6.4.6 RQ5: How do LLMs perform in logging unseen code?

In this RQ, we assess the generalization capabilities of language models by evaluating them on the LogBench-T (Table 6.2).

Original code

```
public void countMemory(ITestContext ctx, int totalMemory) {
  long inMb = 1024 * 1024;
+ log.info("Total memory : " + totalMemory / inMb + " MB");
  MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();
  MemoryUsage heapMemoryUsage = memoryMXBean.getHeapMemoryUsage();
  ...
```

Code after transformation

```
public void countMemory(ITestContext ctx, int totalMemory) {
  long const_1 = 1024 * 1024;
  long inMb = const_1;
+ log.info("Total memory : " + totalMemory / inMb+ " MB");
  MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();
  MemoryUsage heapMemoryUsage = memoryMXBean.getHeapMemoryUsage();
  ...
```

Original code	
CodeWhisperer	log.info("Memory usage: " + totalMemory / inMb + " MB");
ChatGPT	log.info("Current heap memory usage: " + heapMemoryUsage.getUsed() / inMb + " MB");
Incoder	log.debug("Memory used: " + totalMemory / inMb + "MB");
Tansformed code	
CodeWhisperer	log.info("Total Memory: " + totalMemory + " MB");
ChatGPT	log.info("Starting memory count...");
Incoder	log.warn("Memory usage: " + heapMemoryUsage);

Figure 6.11: A case of code transformation and its corresponding predicted logging statement from multiple models.

As stated in Section 6.3.2, predicting accurate logging statements does not necessarily imply that a model can be generalized to unseen cases well. As the modern software codebase is continuously evolving, we must explore LLMs' ability to handle these unseen cases in daily development.

We present the result in Table 6.7, where we **underline** the best performance for each metric and the lowest performance drop rate (Δ) compared to corresponding results in LogBench-O. Our experiments show that all models experience different degrees of performance degradation when generating logging statements on unseen code. LANCE has the smallest average decrease of 6.9% across metrics, while CodeGeex is most im-

pacted with a 16.2% drop. Copilot exhibits the greatest generalization capabilities by outperforming other baselines for three out of four metrics on unseen code. Additionally, we observe that predicting logging levels the smallest degradation in performance (1.4%), whereas predicting logging variables and logging text (BLEU-4) experience significant performance drops, 11.6% and 15%, respectively. Such experiments indicate that resolving logging variables and logging texts is more challenging than predicting logging levels, thus warranting more attention in future research.

Figure 6.11 illustrates a transformation case where we highlight code differences in red and demonstrate how LLMs (CodeWhisperer, ChatGPT, Incoder) log accordingly. Regarding the original code, all models correctly predict that `inMB` should be used to record memory. However, after transforming the constant expression `1024*1024` to a new variable `const_1` and then assigning `const_1` to `inMB`, all models fail to understand and identify `inMB` (or const_1) as a logging variable. CodeWhisperer and Incoder mistakenly predict `totalMemory` and `heapMemoryUsage` as the memory size indicator without dividing it by 1024*1024 to be converted into MB units, while ChatGPT does not suggest any variables. Even though the transformation retains code semantics, existing models exhibit a significant performance drop, indicating their limited generalization abilities.

Finding 8. *LLMs' performance on variable prediction and logging text generation drops significantly for unseen code by 11.6% and 15.0% on average across models, respectively, highlighting the need to improve the generalization capabilities of these models.*

6.5 Implications and Advice

Pay more attention to logging texts. According to Section 6.4.2, while existing models offer satisfactory predictions for logging levels, recommending proper logging variables and logging texts is difficult, particularly the latter. Since LLMs have shown stronger text generation ability than previous neural networks, future research should focus on using LLMs for the challenging problem of logging text generation instead of simply predicting logging levels.

Implication 1. *Future logging studies are encouraged to take advantage of prompting LLMs and focus on the challenging problem of logging text generation.*

Devise alternative evaluation metrics. Section 6.4.2 extensively evaluates the performance of LLMs in generating logging statements using twelve metrics over three ingredients. We observe that a model may excel in one ingredient while performing poorly in others, and such inconsistency makes any comparison and selection of LLMs difficult. Existing metrics like

BLEU and ROUGE, while suitable and being widely-used [30, 29], may not be optimal for logging statements evaluation because they do not consider semantics when assessing similarity between texts: they aggressively penalize lexical differences, even if the predicted logging statements are synonymous to the actual ones [205].

An alternative perspective to assessing the quality of logging statements involves examining the information entropy for operation engineers. Past research has highlighted that a small number of logging statements often dominate an entire log file [32], posing challenges for engineers in figuring out failure-indicating logs. These limitations underscore the need for a succinct and precise logging strategy in practical applications.

Implication 2. *It is recommended to investigate better, possibly unified metrics addressing all ingredients, to evaluate logging statement generation quality.*

Refine prompts with domain knowledge. In Section 6.4.4, we highlight that effective example demonstrations play a crucial role in enhancing the logging performance of LLM by imparting domain knowledge for few-shot learning. Nevertheless, our experiments reveal that augmenting the number of examples does not consistently result in improved performance. These insights elicit the development of an advanced selection strategy for choosing demonstrations, aiming to include the most informative ones in the prompt. The selection strategy can draw

inspiration from program structure similarity (e.g., try-catch), syntax text similarity (e.g., TF-IDF), or code functional similarity [206].

Implication 3. *Designing a demonstration selection framework for effective few-shot learning can yield better results.*

Provide broader programming contexts for LLMs. In Section 6.4.5, we investigate how expanding programming contexts can significantly enhance the logging performance of LLMs. Such a finding implies that extending the context to the file level, rather than the method level, is beneficial for acquiring extra information as well as learning logging styles. However, including the entire repository as input for LLMs may be impractical for large programs due to input token limitations. Additionally, LLM performance tends to decline with longer inputs, even when within the specified context length [207, 208]. To capture effective programming contexts for specific methods, a promising solution involves identifying methods with associated calling relationships and variable definitions. Providing methods spanning multiple classes can also contribute to generating logging statements consistent with existing ones, thereby learning intra-project logging styles.

Implication 4. *When using LLMs for logging, future research could broaden the programming context by incorporating information from function invocations and variable definitions.*

Enhance generalization capabilities of LLMs. In Section 6.4.5, we observe that current LLMs show significantly worse performance on unseen code, reflecting their limited generalization capabilities. The result can be attributed to the capacity of parameters in LLMs to memorize large datasets [160]. This issue will become more severe when tackling code in a rapidly evolving software environment, resulting in more unseen code. One effective idea is to apply a prompt-based method with few chain-of-thought demonstrations [209, 210] to foster the generalization capabilities of ever-growing LLMs. The chain-of-thought strategy allows models to decompose complicated multi-step problems into several intermediate reasoning steps. For example, we can ask models to focus on special code structures (e.g., if-else), then advise them to elicit key variables and system activities to log. While the chain-of-thought strategy has shown success in natural language reasoning tasks [211], future work should explore such prompt-based approaches to enhance generalization capabilities.

Implication 5. *We should investigate prompt-based strategies with zero-shot or few-shot learning to improve the generalization ability of LLMs.*

6.6 Threats to Validity

Internal Threats. (1) A concern of this study is the potential bias introduced by the limited size of the LogBench-O dataset, which consists of *3,840* methods. This limitation arises due to the fact that those plugin-based code completion tools impose usage restrictions to prevent bots; therefore, human efforts are needed. To address the threat, we acquired and sampled LogBench-O and LogBench-T datasets from well-maintained open projects, which we believe are representative. Note that existing Copilot testing studies also have used datasets of comparable sizes [202, 212].

(2) Another concern involves the context length limitations of certain language models [153, 165, 166] (e.g., *4,097* tokens for Davinci), which may affect the file-level experiment. To address this concern, we analyze the collected data and reveal that *98.6%* of the Java files fall within the *4096*-token limit, and *94.3%* of them are within the *2048*-token range. Such analysis implies that the majority of files in our dataset remain unaffected by the context length restrictions.

(3) The other threat is the potential effect of various prompts on Davinci and ChatGPT. To address this, we invited four authors to independently provide three prompts according to their usage habits. These prompts were evaluated using a dataset of 100 samples, and the one that demonstrated the best performance was selected. This approach ensures that the chosen

prompt is representative for daily development.

External Threats. One potential external threat stems from the fact that the LogBench-O dataset was mainly based on the Java language, which may affect the generalizability of our findings to other languages. However, according to previous works [48, 27, 30], Java is among the most prevalent programming languages for logging research purposes, and both SLF4J and Log4j are highly popular and widely adopted logging APIs within the Java ecosystem. We believe the representativeness of our study is highlighted by the dominance of Java languages and these APIs in the logging domain. The core idea of the study can still be generalized to other logging frameworks or languages.

6.7 Summary

In this chapter, we present the first extensive evaluation of LLMs for generating logging statements. To achieve this, we introduce a logging statement generation benchmark dataset, LogBench, and assess the effectiveness and generalization capabilities of eleven top-performing LLMs. While LLMs are promising in generating complete logging statements, they can still be promoted in multiple ways.

First, our evaluation indicates that existing LLMs are not yet adept at generating complete logging statements, particularly in producing effective logging texts. Nonetheless, their di-

rect application surpasses the performance of conventional logging models, indicating a promising future for leveraging LLMs in logging practices.

In addition, we delve into the construction of prompts that influence LLMs' logging performance, considering factors such as instructions and the number of example demonstrations. While our experiments demonstrate the advantages of incorporating demonstrations, we observe that an increased number of demonstrations does not consistently result in improved logging performance. Thus, we recommend the development of a demonstration selection framework in future research. Furthermore, we identify external factors, such as comments and programming contexts, that enhance model performance. We encourage the incorporation of such factors to enhance LLM-based logging tools.

Last but not least, we evaluate LLMs' generalization ability using a dataset that includes transformed code. Our findings indicate that directly applying LLMs to unseen code results in a significant decline in performance, highlighting the necessity to enhance their inference abilities. We suggest employing the chain-of-thought technologies to break down the logging task into smaller logical steps as a future step, unlocking LLMs' full potential. We hope this chapter can stimulate more work in the promising direction of using LLMs for automatic logging.

Chapter 7

Conclusion and Future Work

In this chapter, we summarize the main contributions of this thesis and present several promising directions for future work.

7.1 Conclusion

As modern software has evolved into large-scale, complex, and interconnected services, reliability engineering techniques have similarly progressed, shifting from human inspection to automated approaches. Logs play a critical role in the software lifecycle for two main reasons: (1) they are often the only accessible resource reflecting the system's runtime status, and (2) they bridge the gap between developers and operators through the execution of logging statements. This thesis focuses on enhancing software reliability engineering by utilizing logs for performance monitoring, anomaly detection, and system behavior profiling.

Chapter 3 addresses the challenges associated with analyzing high-variety logs. We revisit existing log parsers and identify their limitations in extracting semantics. Moreover, existing parsers predominantly focus on individual log messages and overlook correlations between different logs. To overcome these shortcomings, we propose the first semantic-aware log parser, AutoLog. This parser automatically identifies semantics from both intra-message and inter-message levels. Extensive experiments were conducted on six real-world log parsing datasets, along with evaluations in anomaly detection and failure identification.

Next, in Chapter 4, we developed an automatic anomalous log localization framework named EvLog for evolving software systems. The key insight behind EvLog is that although logging statements may be revised during software evolution, their semantics often remain consistent. Therefore, the core design of EvLog utilizes a cluster-based approach to align historical logs with new, revised logs, ensuring that the new representations remain similar. EvLog outperforms existing log mining frameworks on two real-world datasets, demonstrating its strong capability in handling frequently evolving systems.

After that, in Chapter 5, we address one longstanding challenge in the log analysis community: the insufficiency of comprehensive datasets. Existing datasets are passively collected by manually executing a limited number of workloads, which restricts the diversity of log events captured. To counter this lim-

itation, we devised AutoLog to actively generate log sequences based on program analysis. Experiments show that AutoLog can efficiently synthesize log sequences with high coverage of system behavior. Additionally, AutoLog proves effective in enhancing the performance of anomaly detection models when they are trained with synthesized data.

Last but not least, in Chapter 6, we conduct the first extensive evaluation study of large language models (LLMs) for their ability to generate logging statements. This study involves eleven top-performing LLMs and three state-of-the-art conventional logging approaches. Specifically, we introduce a logging statement generation benchmark dataset, LogBench, which includes a collection of *6,849* logging statements across *3,870* methods (LogBench-O), along with functionally equivalent unseen code (LogBench-T) to evaluate generalization ability. Based on the evaluation results, we summarize eight key findings and draw five implications, providing valuable insights for future research on automated log statement generation.

In summary, this thesis presents novel techniques for the automated processing and analysis of system runtime logs. These techniques include: (1) A semantic-aware log parser that captures both intra-message and inter-message semantics, (2) An anomalous log localizer tailored for evolving software systems, (3) A log sequence synthesizer that generates comprehensive log data via program analysis, (4) An extensive evaluation study on LLM-powered logging statement generators, providing crucial

insights for future research. These contributions collectively enhance our ability to effectively monitor, diagnose, and analyze complex software systems, thereby improving overall software reliability and operational efficiency.

7.2 Future Work

We envision a future where AI technologies are universally accessible, delivering substantial benefits to everyday life, and where everyone can depend on reliable software. My past research has demonstrated the effectiveness of data-driven methods in advancing software reliability engineering (SRE) through log analysis.

Although this thesis proposes several novel approaches that contribute significantly to reliability engineering, the era of large language models (LLMs) offers numerous further opportunities. Future work will explore these possibilities. The following sections introduce three promising directions: multi-modal intelligent software operations, operation-guided software development, and LLM-powered software reliability engineering ecosystems.

7.2.1 Multi-modal Intelligent Software Operations

Modern software generates diverse data during runtime, each reflecting system behavior from different perspectives. Besides logs, these data also encompass topologies, KPIs, and

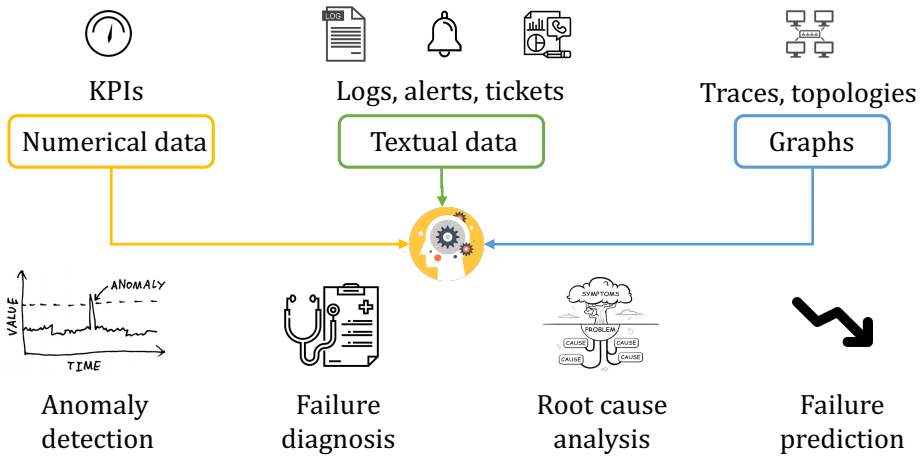


Figure 7.1: Multimodal operations example.

alerts: (1) KPIs (Key Performance Indicators): Metrics sampled uniformly from the running system, such as response time and memory usage. (2) Topologies: Graph data in cloud systems that indicate dependencies and interactions between various nodes. (3) Alerts: Textual data, also known as alarms, raised in real-time to signal potential issues. Alerts are based on predefined policies and conditions within the program. When a failure occurs in an online system, it may be indicated by warning logs, anomalous KPIs, and multiple alerts. Although these data types are highly correlated, past research has often tackled them in isolation.

The success of multi-modal large language models has inspired us to create a similar approach for operations. As shown in Figure 7.1, the desired model can process different types of data—textual (such as logs), numeric (such as KPIs), and graphical (such as topologies). Each data type provides complemen-

tary information. With this unified operation model, engineers can conduct more comprehensive analyses, such as performing root cause analysis by examining a combination of service dependency graphs and log entries. Such a multi-modal approach aims to provide a holistic understanding of system behavior, enabling more effective monitoring, diagnosis, and troubleshooting. This can significantly enhance the reliability and performance of complex software systems.

7.2.2 DevOps: Operation-guided Software Development

Software operations provide invaluable feedback that can guide the elimination of potential faults during the development phase, as illustrated in Figure 7.2. This insight underlies the "DevOps" paradigm, which aims to remove the barriers between traditionally siloed development and operation teams. This paradigm enables faster development of new products and easier maintenance of existing deployments, leading to the continuous delivery of reliable software versions. In our context, operation-guided fault removal involves leveraging operational data (such as logs) to develop program-fixing strategies for general bugs.

Our goal is to create intelligent development environments that not only diagnose runtime failures during operations but also use these failure cues to automatically remove corresponding faults from the source code. There have been exploratory

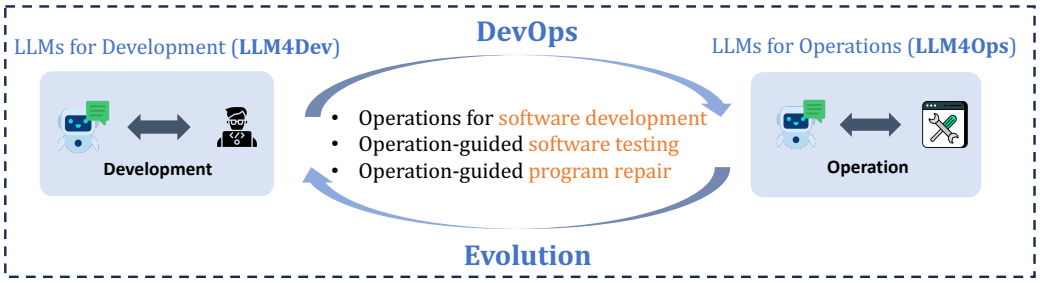


Figure 7.2: Future topics in DevOps paradigm.

studies on log-guided bug localization [213] and debugging production failures [214]. For instance, in log-guided program repair, the solution typically involves two steps: (1) Profiling program execution paths by matching log message sequences to determine where and how failures occur. (2) Repairing the bugs based on failure clues extracted from the log messages. This approach fosters a proactive fault-removal process, enhancing the efficiency and effectiveness of both development and operations by ensuring that real-world operational insights directly inform and improve software quality. The integration of such intelligent systems within the DevOps paradigm is a promising direction for future research and development.

7.2.3 LLM-powered Software Reliability Engineering Ecosystems

This research direction will continue to explore the potential of multi-agent large language models (LLMs) for software engineering. While LLMs have proven powerful in generating

code snippets for specific tasks, they can struggle with planning complex tasks and retaining long-term information. Recent studies suggest that LLM agents can think more like humans and plan ahead by breaking down big, complex tasks into smaller, more manageable steps.

Inspired by these advances in other fields, this research aims to embed LLMs into the software reliability engineering lifecycle. This involves exploring two main areas: (1) advancing LLMs for software engineering tasks, such as: refining LLM-based code generation tools, developing program testing algorithms, and creating frameworks for automatic code bug fixes; and (2) enhancing human-agent interaction, for example, developing responsible LLMs that not only execute human commands but also engage in a collaborative partnership with engineers, offering new ideas and insights. By integrating multi-agent LLMs into software reliability engineering, this direction seeks to enhance the overall efficiency and effectiveness of software development processes. The ultimate goal is to create intelligent systems that can autonomously handle various engineering tasks while collaborating seamlessly with human engineers, leading to more reliable and innovative software solutions.

□ **End of chapter.**

Appendix A

List of Publications

1. **Yintong Huo**, Yuxin Su, Hongming Zhang, Michael R. Lyu. *ARCLIN: Automated API Mention Resolution for Unformatted Texts.* Proceedings of 44th International Conference on Software Engineering (ICSE), 2022.

2. **Yintong Huo**, Yuxin Su, Michael R. Lyu. *LogVM: Variable Semantics Miner for Log Messages.* Proceedings of 33rd International Symposium on Software Reliability Engineering (ISSRE-W), 2022.

3. **Yintong Huo**, Yuxin Su, Baitong Li, Michael R. Lyu. *SemParser: A Semantic Parser for Log Analytics* Proceedings of 44th International Conference on Software Engineering (ICSE), 2023

4. **Yintong Huo**, Cheryl Lee, Yuxin Su, Shiwen Shan, Jinyang Liu and Michael R. Lyu. *EvLog: Identifying Anomalous Logs over Software Evolution.* Proceedings of 34th IEEE

International Symposium on Software Reliability Engineering (ISSRE), 2023

5. **Yintong Huo**, Yichen Li, Yuxin Su, Pinjia He, Zifan Xie, and Michael R. Lyu. *AutoLog: A Log Sequence Synthesis Framework for Anomaly Detection.* Proceedings of 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023.

6. Yun Peng, Shuzheng Gao, Cuiyun Gao, **Yintong Huo**, Michael R. Lyu. *Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors.* Proceedings of 44th International Conference on Software Engineering (ICSE), 2024.

7. Yichen Li, Yun Peng, **Yintong Huo**, and Michael R. Lyu. *Enhancing LLM-based Coding Tools Through Native Integration of IDE-Derived Static Context.* To appear in the IEEE/ACM International Conference on Software Engineering Workshop on Large Language Model for Code (ICSE'24-LLM4Code), 2024.

8. Yichen Li, **Yintong Huo**, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjie He, and Michael R. Lyu. *Go Static: Contextualized Logging Statement Generation.* To appear in the ACM International Conference on the Foundations of Software Engineering (FSE), 2024.

9. Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, **Yintong Huo**, Pinjie He, Jiazhen Gu, and Michael R. Lyu. *LILAC: Log Parsing using LLMs with Adaptive Parsing Cache.* To appear in the ACM International Conference on the Foundations of Software Engineering (FSE), 2024.

10. Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, **Yintong Huo**, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. *A Large-scale Evaluation for Log Parsing Techniques: How Far are We?* To appear in the ACM International Symposium on Software Testing and Analysis (ISSTA), 2024.

11. Cheryl Lee, Zhouruixin Zhu, Tianyi Yang, **Yintong Huo**, Yuxin Su, Pinjia He, and Michael R Lyu. *SPES: Towards Optimizing Performance-Resource Trade-Off for Serverless Functions.* To appear in the IEEE International Conference on Data Engineering (ICDE), 2024.

12. (In submission) Yichen Li, **Yintong Huo**, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, Lionel C. Briand, and Michael R. Lyu. *Exploring the Effectiveness of LLMs in Automated Logging Generation: An Empirical Study.*

Reference

[1] M. R. Lyu *et al.*, *Handbook of software reliability engineering*. IEEE computer society press Los Alamitos, 1996, vol. 222.

[2] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 130:1–130:37, 2021. [Online]. Available: https://doi.org/10.1145/3460345

[3] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 1. IEEE, 2015, pp. 415–425.

[4] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing, New York, NY, USA, May 17-19, 2004*. IEEE Computer Society, 2004, pp. 36–43. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICAC.2004.31

[5] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. Jordan, "Large-scale system problems detection by mining console logs," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-103, Jul 2009. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-103.html

[6] N. Zhao, H. Wang, Z. Li, X. Peng, G. Wang, Z. Pan, Y. Wu, Z. Feng, X. Wen, W. Zhang *et al.*, "An empirical investigation

of practical log anomaly detection for online service systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1404–1415. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3468264.3473933

[7] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 140–151. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00031

[8] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 207–218.

[9] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1245–1255, 2013.

[10] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[11] Y. Huo, Y. Su, C. Lee, and M. R. Lyu, "Semparser: A semantic parser for log analytics," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 881–893.

[12] Y. Huo, C. Lee, Y. Su, S. Shan, J. Liu, and M. Lyu, "Evlog: Evolving log analyzer for anomalous logs identification," *arXiv preprint arXiv:2306.01509*, 2023.

[13] Y. Huo, Y. Li, Y. Su, P. He, Z. Xie, and M. R. Lyu, "Autolog: A log sequence synthesis framework for anomaly detection," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 497–509.

[14] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 ninth IEEE international conference on data mining*. IEEE, 2009, pp. 149–158.

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

[16] R. 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)*. Ieee, 2003, pp. 119–126.

[17] G. Chu, J. Wang, Q. Qi, H. Sun, S. Tao, and J. Liao, "Prefix-graph: A versatile log parsing approach merging prefix tree with probabilistic graph," in *Proceedings of the 37th International Conference on Data Engineering, ICDE, Virtual Event, 2021*. IEEE, 2021, pp. 2411–2422. [Online]. Available: https://ieeexplore.ieee.org/document/9458609

[18] R. Vaarandi and M. Pihelgas, "Logcluster-a data clustering and pattern mining algorithm for event logs," in *2015 11th International conference on network and service management (CNSM)*. IEEE, 2015, pp. 1–7.

[19] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 859–864.

[20] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proceedings of the 24th IEEE International Conference on Web Services, ICWS, Honolulu, HI,*

USA, June 25-30, 2017, I. Altintas and S. Chen, Eds. IEEE, 2017, pp. 33–40. [Online]. Available: https://doi.org/10.1109/ICWS.2017.13

[21] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu, "Lilac: Log parsing using llms with adaptive parsing cache," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 137–160, 2024.

[22] Y. Xiao, V.-H. Le, and H. Zhang, "Stronger, faster, and cheaper log parsing with llms," *arXiv preprint arXiv:2406.06156*, 2024.

[23] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 807–817. [Online]. Available: https://doi.org/10.1145/3338906.3338931

[24] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-attentive classification-based anomaly detection in unstructured logs," in *International Conference on Data Mining, Sorrento, Italy, November 17-20, 2020*. IEEE, 2020, pp. 1196–1201. [Online]. Available: https://doi.org/10.1109/ICDM50108.2020.00148

[25] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 1285–1298. [Online]. Available: https://doi.org/10.1145/3133956.3134015

[26] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured

logs," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI, Macao, China, August 10-16, 2019*, S. Kraus, Ed. ijcai.org, 2019, pp. 4739–4745. [Online]. Available: https://doi.org/10.24963/ijcai.2019/658

[27] J. Liu, J. Zeng, X. Wang, K. Ji, and Z. Liang, "Tell: log level suggestions via modeling multi-level code block information," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 27–38.

[28] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?" *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 9, pp. 2012–2031, 2019.

[29] Z. Ding, H. Li, and W. Shang, "Logentext: Automatically generating logging texts using neural machine translation," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 349–360.

[30] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2279–2290.

[31] J. Wei, G. Zhang, J. Chen, Y. Wang, W. Zheng, T. Sun, J. Wu, and J. Jiang, "Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 452–468.

[32] G. Yu, P. Chen, P. Li, T. Weng, H. Zheng, Y. Deng, and Z. Zheng, "Logreducer: Identify and reduce log hotspots in kernel on the fly," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1763–1775.

[33] T. Li, Y. Jiang, C. Zeng, B. Xia, Z. Liu, W. Zhou, X. Zhu, W. Wang, L. Zhang, J. Wu *et al.*, "Flap: An end-to-end event log analysis

platform for system management," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1547–1556.

[34] M. Hahsler, M. Piekenbrock, and D. Doran, "dbscan: Fast density-based clustering with r," *Journal of Statistical Software*, vol. 91, pp. 1–30, 2019.

[35] E. Fix, *Discriminatory analysis: nonparametric discrimination, consistency properties*. USAF school of Aviation Medicine, 1985, vol. 1.

[36] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data mining and knowledge discovery*, vol. 15, no. 1, pp. 55–86, 2007.

[37] T. Zhang, H. Qiu, G. Castellano, M. Rifai, C. S. Chen, and F. Pianese, "System log parsing: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, Long Beach, CA, USA, December 4-9, 2017*, 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[39] Z. Chen, J. Liu, W. Gu, Y. Su, and M. R. Lyu, "Experience report: Deep learning-based system log analysis for anomaly detection," *arXiv preprint arXiv:2107.05908*, 2021.

[40] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security, CCS, Dallas, TX, USA, October 30 -*

November 03, 2017. ACM, 2017, pp. 1285–1298. [Online]. Available: https://doi.org/10.1145/3133956.3134015

[41] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. 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, pp. 305–316.

[42] J. Kim, V. Savchenko, K. Shin, K. Sorokin, H. Jeon, G. Pankratenko, S. Markov, and C.-J. Kim, "Automatic abnormal log detection by analyzing log history for providing debugging insight," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 2020, pp. 71–80.

[43] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ICSE)*, 2020, pp. 361–372.

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

[45] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 24–33.

[46] S. Lal, N. Sardana, and A. Sureka, "Logoptplus: Learning to optimize logging in catch and if programming constructs," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 215–220.

[47] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive

logging," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 293–306.

[48] Z. Li, H. Li, T.-H. Chen, and W. Shang, "Deeplv: Suggesting log levels using ordinal based neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1461–1472.

[49] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering (ESE)*, vol. 22, pp. 1684–1716, 2017.

[50] S. Dai, Z. Luan, S. Huang, C. Fung, H. Wang, H. Yang, and D. Qian, "Reval: Recommend which variables to log with pre-trained model and graph neural network," *IEEE Transactions on Network and Service Management (TNSM)*, 2022.

[51] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–28, 2012.

[52] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *Proceedings of the 7th International Working Conference on Mining Software Repositories, Cape Town, South Africa, May 2-3, 2010*, IEEE. IEEE Computer Society, 2010, pp. 114–117. [Online]. Available: https://doi.org/10.1109/MSR.2010.5463281

[53] H. Dai, H. Li, C. S. Chen, W. Shang, and T.-H. Chen, "Logram: Efficient log parsing using n-gram dictionaries," *CoRR*, vol. abs/2001.03038, 2020. [Online]. Available: http://arxiv.org/abs/2001.03038

[54] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proceedings of the 20th*

Conference on Information and Knowledge Management, UK, October 24-28, 2011. ACM, 2011, pp. 785–794. [Online]. Available: https://doi.org/10.1145/2063576.2063690

[55] M. Mizutani, "Incremental mining of system log format," in *International Conference on Services Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*. IEEE Computer Society, 2013, pp. 595–602. [Online]. Available: https://doi.org/10.1109/SCC.2013.73

[56] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2655–2694, 2018. [Online]. Available: https://doi.org/10.1007/s10664-018-9595-8

[57] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proceedings of the 26th Conference on Program Comprehension, Gothenburg, Sweden, May 27-28, 2018*. ACM, 2018, pp. 167–177. [Online]. Available: https://doi.org/10.1145/3196321.3196340

[58] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[59] R. He, W. S. Lee, H. T. Ng, and D. Dahlmeier, "An unsupervised neural attention model for aspect extraction," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017, pp. 388–397.

[60] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012*. ISCA, 2012, pp. 194–197. [Online]. Available: http://www.isca-speech.org/archive/interspeech_2012/i12_0194.html

[61] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *International Conference on Acoustics, Speech and Signal Processing, Vancouver, BC, Canada, May 26-31, 2013*. IEEE, 2013, pp. 6645–6649. [Online]. Available: https://doi.org/10.1109/ICASSP.2013.6638947

[62] M. Xuezhe and H. H. Eduard, "End-to-end sequence labeling via bi-directional lstm-cnns-crf," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, August 7-12, 2016*. The Association for Computer Linguistics, 2016. [Online]. Available: https://doi.org/10.18653/v1/p16-1101

[63] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," *CoRR*, vol. abs/1508.01991, 2015. [Online]. Available: http://arxiv.org/abs/1508.01991

[64] J. P. Chiu and E. Nichols, "Named entity recognition with bidirectional lstm-cnns," *Trans. Assoc. Comput. Linguistics*, vol. 4, pp. 357–370, 2016. [Online]. Available: https://transacl.org/ojs/index.php/tacl/article/view/792

[65] M. Shetty, C. Bansal, S. Kumar, N. Rao, N. Nagappan, and T. Zimmermann, "Neural knowledge extraction from cloud service incidents," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2021, pp. 218–227.

[66] J. Tabassum, M. Maddela, W. Xu, and A. Ritter, "Code and named entity recognition in stackoverflow," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, July 5-10, 2020*. Association for Computational Linguistics, 2020, pp. 4913–4926. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.443

[67] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1,

pp. 41–75, 1997. [Online]. Available: https://doi.org/10.1023/A:1007379606734

[68] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 200–211. [Online]. Available: https://doi.org/10.1145/3338906.3338916

[69] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, pp. 37–46, 1960. [Online]. Available: https://w3.ric.edu/faculty/organic/coge/cohen1960.pdf

[70] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Annual Conference on Neural Information Processing Systems, Lake Tahoe, Nevada, USA, December 5-8, 2013*, 2013, pp. 3111–3119. [Online]. Available: https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html

[71] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proceedings of LREC 2010 workshop New Challenges for NLP Frameworks, Valletta, Malta, May 22, 2010*. University of Malta, 2010, pp. 46–50. [Online]. Available: http://www.fi.muni.cz/usr/sojka/presentations/lrec2010-poster-rehurek-sojka.pdf

[72] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A large collection of system log datasets towards automated log analytics," *CoRR*, vol. abs/2008.06448, 2020. [Online]. Available: https://arxiv.org/abs/2008.06448

[73] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, "Logzip: extracting hidden structures via iterative clustering

for log compression," in *International Conference on Automated Software Engineering, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 863–873. [Online]. Available: https://doi.org/10.1109/ASE.2019.00085

[74] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. ACM, 2016, pp. 102–111. [Online]. Available: https://doi.org/10.1145/2889160.2889232

[75] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting execution logs to execution events for enterprise applications (short paper)," in *Proceedings of the Eighth International Conference on Quality Software, Oxford, UK, August 12-13, 2008,*. IEEE Computer Society, 2008, pp. 181–186. [Online]. Available: https://doi.org/10.1109/QSIC.2008.50

[76] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*. ACM, 2009, pp. 1255–1264. [Online]. Available: https://doi.org/10.1145/1557019.1557154

[77] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, Athens, Greece, August 12-15, 2018*. IEEE Computer Society, 2018, pp. 151–158. [Online]. Available: https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00037

[78] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*. ACM, 2016, pp. 191–198. [Online]. Available: https://doi.org/10.1145/2959100.2959190

[79] T. Osadchiy, I. Poliakov, P. Olivier, M. Rowland, and E. Foster, "Recommender system based on pairwise association rules," *Expert Systems with Applications*, vol. 115, pp. 535–542, 2019. [Online]. Available: https://doi.org/10.1016/j.eswa.2018.07.077

[80] Shilpika, B. Lusch, M. Emani, V. Vishwanath, M. E. Papka, and K. Ma, "MELA: A visual analytics tool for studying multifidelity HPC system logs," in *3rd IEEE/ACM Industry/University Joint International Workshop on Data-center Automation, Analytics, and Control, DAAC@SC, Denver, CO, USA, November 22, 2019*. IEEE, 2019, pp. 13–18. [Online]. Available: https://doi.org/10.1109/DAAC49578.2019.00008

[81] J. Liu, J. Huang, Y. Huo, Z. Jiang, J. Gu, Z. Chen, C. Feng, M. Yan, and M. R. Lyu, "Scalable and adaptive log-based anomaly detection with expert in the loop," *arXiv preprint arXiv:2306.05032*, 2023.

[82] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Müller, "Data mining-based analysis of HPC center operations," in *Proceedings of the 19th IEEE International Conference on Cluster Computing, CLUSTER, Honolulu, HI, USA, September 5-8, 2017*. IEEE Computer Society, 2017, pp. 766–773. [Online]. Available: https://doi.org/10.1109/CLUSTER.2017.23

[83] X. Zhang, Y. Xu, S. Qin, S. He, B. Qiao, Z. Li, H. Zhang, X. Li, Y. Dang, Q. Lin *et al.*, "Onion: identifying incident-indicating logs for cloud systems," in *Proceedings of the 29th ACM Joint Meeting*

on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1253–1263.

[84] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu, "Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs," in *International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, June 25-30, 2017*. IEEE Computer Society, 2017, pp. 447–455. [Online]. Available: https://doi.org/10.1109/CLOUD.2017.64

[85] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Online system problem detection by mining patterns of console logs," in *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*. IEEE Computer Society, 2009, pp. 588–597. [Online]. Available: https://doi.org/10.1109/ICDM.2009.19

[86] W. Meng, Y. Liu, S. Zhang, F. Zaiter, Y. Zhang, Y. Huang, Z. Yu, Y. Zhang, L. Song, M. Zhang *et al.*, "Logclass: Anomalous log identification and classification with partial labels," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1870–1884, 2021.

[87] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz, "Using finite-state models for log differencing," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2018, pp. 49–59.

[88] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Köprü, and T. Xie, "Groot: An event-graph-based approach for root cause analysis in industrial settings," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, ASE, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 419–429. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678708

[89] M. M. Lehman and J. F. Ramil, "Software evolution and software evolution processes," *Ann. Softw. Eng.*, vol. 14, no. 1-4, pp. 275–309, 2002. [Online]. Available: https://doi.org/10.1023/A:1020557525901

[90] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. N. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *J. Softw. Evol. Process.*, vol. 26, no. 1, pp. 3–26, 2014. [Online]. Available: https://doi.org/10.1002/smr.1579

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

[92] B. Chen and Z. M. Jiang, "Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, pp. 330–374, 2017.

[93] T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, and J. Xu, "An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services," in *Proceedings of the 24th IEEE International Conference on Web Services, ICWS, Honolulu, HI, USA, June 25-30, 2017*. IEEE, 2017, pp. 25–32. [Online]. Available: https://doi.org/10.1109/ICWS.2017.12

[94] X. Fu, R. Ren, S. A. McKee, J. Zhan, and N. Sun, "Digging deeper into cluster system logs for failure prediction and root cause diagnosis," in *Proceedings of the 16th IEEE International Conference on Cluster Computing, CLUSTER, Madrid, Spain, September 22-26, 2014*. IEEE, 2014, pp. 103–112. [Online]. Available: https://doi.org/10.1109/CLUSTER.2014.6968768

[95] X. Wang, X. Zhang, L. Li, S. He, H. Zhang, Y. Liu, L. Zheng, Y. Kang, Q. Lin, Y. Dang *et al.*, "Spine: a scalable log parser with feedback

guidance," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1198–1208.

[96] B. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan, "LOGAN: problem diagnosis in the cloud using log-based reference models," in *Proceedings of the 4th IEEE International Conference on Cloud Engineering, IC2E, Berlin, Germany, April 4-8, 2016*. IEEE, 2016, pp. 62–67. [Online]. Available: https://doi.org/10.1109/IC2E.2016.12

[97] Y. Huo, Y. Su, and M. Lyu, "Logvm: Variable semantics miner for log messages," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2022, pp. 124–125.

[98] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[99] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 492–504.

[100] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of the 19th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, Athens, Greece, August 23-28, 2021*. ACM, 2021, pp. 703–715. [Online]. Available: https://doi.org/10.1145/3468264.3468611

[101] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: sequence-to-sequence pre-training for learning source code representations," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2006–2018.

[102] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 401–412.

[103] L. Li, Z. Li, W. Zhang, J. Zhou, P. Wang, J. Wu, G. He, X. Zeng, Y. Deng, and T. Xie, "Clustering test steps in natural language toward automating test automation," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, Virtual Event, USA, November 8-13, 2020*. ACM, 2020, pp. 1285–1295. [Online]. Available: https://doi.org/10.1145/3368089.3417067

[104] M. Silic, G. Delac, and S. Srbljic, "Prediction of atomic web services reliability based on k-means clustering," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 2013, pp. 70–80. [Online]. Available: https://doi.org/10.1145/2491411.2491424

[105] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Semi-supervised log-based anomaly detection via probabilistic label estimation," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1448–1460.

[106] L. McInnes and J. Healy, "Accelerated hierarchical density based clustering," in *Proceedings of the 17th IEEE International Conference on Data Mining Workshops, ICDM Workshops, New Orleans, LA,*

USA, November 18-21, 2017. IEEE Computer Society, 2017, pp. 33–42. [Online]. Available: https://doi.org/10.1109/ICDMW.2017.12

[107] P. Cifariello, P. Ferragina, and M. Ponza, "Wiser: A semantic approach for expert finding in academia based on entity linking," *Information Systems*, vol. 82, pp. 1–16, 2019.

[108] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[109] S. Na, L. Xumin, and G. Yong, "Research on k-means clustering algorithm: An improved k-means clustering algorithm," in *2010 Third International Symposium on intelligent information technology and security informatics*. Ieee, 2010, pp. 63–67.

[110] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[111] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[112] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A large collection of system log datasets towards automated log analytics," *CoRR*, vol. abs/2008.06448, 2020. [Online]. Available: https://arxiv.org/abs/2008.06448

[113] A. J. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, 25-28 June 2007, Edinburgh, UK, Proceedings*. IEEE Computer Society, 2007, pp. 575–584. [Online]. Available: https://doi.org/10.1109/DSN.2007.103

[114] Q. Lin, H. Zhang, J. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering, ICSE, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. ACM, 2016, pp. 102–111. [Online]. Available: https://doi.org/10.1145/2889160.2889232

[115] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[116] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[117] "Hibench," Intel, 2021. [Online]. Available: https://github.com/Intel-bigdata/HiBench

[118] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[119] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015. [Online]. Available: http://arxiv.org/abs/1505.00853

[120] L. Ruff, N. Görnitz, L. Deecke, S. A. Siddiqui, R. A. Vandermeulen, A. Binder, E. Müller, and M. Kloft, "Deep one-class classification," in *Proceedings of the 35th International Conference on Machine*

Learning, ICML, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, ser. Proceedings of Machine Learning Research, vol. 80. PMLR, 2018, pp. 4390–4399. [Online]. Available: http://proceedings.mlr.press/v80/ruff18a.html

[121] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, "Log-based abnormal task detection and root cause analysis for spark," in *Proceedings of the 24th IEEE International Conference on Web Services, ICWS, Honolulu, HI, USA, June 25-30, 2017*, I. Altintas and S. Chen, Eds. IEEE, 2017, pp. 389–396. [Online]. Available: https://doi.org/10.1109/ICWS.2017.135

[122] L. Wang, N. Zhao, J. Chen, P. Li, W. Zhang, and K. Sui, "Root-cause metric location for microservice systems via log anomaly detection," in *2020 IEEE International Conference on Web Services (ICWS)*. IEEE, 2020, pp. 142–150.

[123] Y. Li, X. Zhang, S. He, Z. Chen, Y. Kang, J. Liu, L. Li, Y. Dang, F. Gao, Z. Xu *et al.*, "An intelligent framework for timely, accurate, and comprehensive cloud incident detection," *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 1–7, 2022.

[124] T. Jia, Y. Li, Y. Yang, G. Huang, and Z. Wu, "Augmenting log-based anomaly detection models to reduce false anomalies with human feedback," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 3081–3089.

[125] C. Wang, K. Wu, T. Zhou, G. Yu, and Z. Cai, "Tsagen: synthetic time series generation for kpi anomaly detection," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 130–145, 2021.

[126] V. Le and H. Zhang, "Log-based anomaly detection with deep learning: How far are we?" *CoRR*, vol. abs/2202.04301, 2022. [Online]. Available: https://arxiv.org/abs/2202.04301

[127] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia, "Assessing and improving the effectiveness of logs for the analysis of software faults," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2010, pp. 457–466.

[128] A. Pecchia and S. Russo, "Detection of software failures through event logs: An experimental study," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 31–40.

[129] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering (TSE)*, no. 3, pp. 216–226, 1979.

[130] Y. Smaragdakis, G. Balatsouras *et al.*, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[131] Apache, *SLF4J*, Aug. 2022, https://www.slf4j.org.

[132] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 18, no. 3, p. 206, 1992.

[133] B. A. Nejmeh, "Npath: a measure of execution path complexity and its applications," *Communications of the ACM*, vol. 31, no. 2, pp. 188–200, 1988.

[134] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[135] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," in *International Conference on Extending Database Technology*. Springer, 1998, pp. 467–483.

[136] N. Busany and S. Maoz, "Behavioral log analysis with statistical guarantees," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 877–887.

[137] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *Acm Sigplan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[138] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[139] M. Banerjee, M. Capozzoli, L. McSweeney, and D. Sinha, "Beyond kappa: A review of interrater agreement measures," *Canadian journal of statistics*, vol. 27, no. 1, pp. 3–23, 1999.

[140] Apache, *The Maven Repository*, Aug. 2022, https://mvnrepository.com/repos/central.

[141] *Storm*, Aug. 2022, https://storm.apache.org.

[142] *Flink*, Aug. 2022, https://flink.apache.org.

[143] *Kafka*, Aug. 2022, https://kafka.apache.org.

[144] S. Park, B. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: Achieving higher statement coverage faster," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[145] A. Utture, S. Liu, C. G. Kalhauge, and J. Palsberg, "Striking a balance: Pruning false-positives from static call graphs," 2022.

[146] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[147] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th*

International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 672–681.

[148] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP' 95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*. Springer, 1995, pp. 77–101.

[149] S. Gholamian, "Leveraging code clones and natural language processing for log statement prediction," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1043–1047.

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

[151] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[152] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[153] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[154] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[155] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[156] GitHub, "Github copilot: Parrot or crow? a first look at rote learning in github copilot suggestions." Mar 2023. [Online]. Available: https://github.blog/2021-06-30-github-copilot-research-recitation/

[157] ——, "Github copilot: Your ai pair programmer," Mar 2023. [Online]. Available: https://github.com/features/copilot

[158] Amazon, "Codewhisperer," Mar 2023. [Online]. Available: https://aws.amazon.com/cn/codewhisperer/

[159] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[160] M. R. I. Rabin, A. Hussain, M. A. Alipour, and V. J. Hellendoorn, "Memorization and generalization in neural code intelligence models," *Information and Software Technology (Inf. Softw. Technol.)*, vol. 153, p. 107066, 2023.

[161] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

[162] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain,

W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[163] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[164] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[165] OpenAI, "Chatgpt," Mar 2023. [Online]. Available: https://openai.com/blog/chatgpt/

[166] OpenAI., "Gpt-3.5," Mar 2022. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5

[167] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang *et al.*, "Gpt-neox-20b: An open-source autoregressive language model," *arXiv preprint arXiv:2204.06745*, 2022.

[168] EleutherAI., "Gpt-j," Mar 2022. [Online]. Available: https://huggingface.co/EleutherAI/gpt-j-6B.

[169] Y. Tan, D. Min, Y. Li, W. Li, N. Hu, Y. Chen, and G. Qi, "Can chatgpt replace traditional kbqa models? an in-depth analysis of the question answering performance of the gpt llm family," in *International Semantic Web Conference*. Springer, 2023, pp. 348–367.

[170] T. Goyal, J. J. Li, and G. Durrett, "News summarization and evaluation in the era of gpt-3," *arXiv preprint arXiv:2209.12356*, 2022.

[171] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[172] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen *et al.*, "A comprehensive capability analysis of gpt-3 and gpt-3.5 series models," *arXiv preprint arXiv:2303.10420*, 2023.

[173] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[174] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.

[175] LANCE, "Replication package of lance." Jan 2022. [Online]. Available: https://github.com/antonio-mastropaolo/LANCE#using-deep-learning-to-generate-complete-log-statements

[176] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[177] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[178] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet nl2code: A survey," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 7443–7464.

[179] CodeGeeX, "Codegeex," Mar 2023. [Online]. Available: https://models.aminer.cn/codegeex/blog/

[180] Tabnine, "Tabnine," Mar 2023. [Online]. Available: https://www.tabnine.com/

[181] Z. Ding, Y. Tang, X. Cheng, H. Li, and W. Shang, "Logentext-plus: Improving neural machine translation-based logging texts generation with syntactic templates," *ACM Trans. Softw. Eng. Methodol.*, sep 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3624740

[182] B. Chen and Z. M. Jiang, "Studying the use of java logging utilities in the wild," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 397–408.

[183] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[184] E. Quiring, A. Maier, K. Rieck *et al.*, "Misleading authorship attribution of source code using adversarial learning." in *USENIX Security Symposium (USENIX Security)*, 2019, pp. 479–496.

[185] Y. Li, S. Qi, C. Gao, Y. Peng, D. Lo, Z. Xu, and M. R. Lyu, "A closer look into transformer-based code intelligence through code transforma-

tion: Challenges and opportunities," *arXiv preprint arXiv:2207.04285*, 2022.

[186] Z. Li, C. Wang, Z. Liu, H. Wang, S. Wang, and C. Gao, "Cctest: Testing and repairing code completion systems," *arXiv preprint arXiv:2208.08289*, 2022.

[187] Z. Li, J. Tang, D. Zou, Q. Chen, S. Xu, C. Zhang, Y. Li, and H. Jin, "Towards making deep learning-based vulnerability detectors robust," *arXiv preprint arXiv:2108.00669*, 2021.

[188] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[189] H. Cheers, Y. Lin, and S. P. Smith, "Spplagiarise: A tool for generating simulated semantics-preserving plagiarism of java source code," in *2019 IEEE 10th International conference on software engineering and service science (ICSESS)*. IEEE, 2019, pp. 617–622.

[190] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of compilers*, vol. 19, p. 31, 2005.

[191] H. Zhang, Y. Pei, J. Chen, and S. H. Tan, "Statfier: Automated testing of static analyzers via semantic-preserving program transformations," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 237–249.

[192] JavaParser, "Javaparser," Mar 2019. [Online]. Available: https://javaparser.org

[193] Z. A. Khan, D. Shin, D. Bianculli, and L. Briand, "Guidelines for assessing the accuracy of log message template identification techniques," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1095–1106.

[194] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, "Prompting for automatic log template extraction," *arXiv preprint arXiv:2307.09950*, 2023.

[195] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu, "Do not give away my secrets: Uncovering the privacy issue of neural code completion tools," *arXiv preprint arXiv:2309.07639*, 2023.

[196] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics (ACL)*, 2002, pp. 311–318.

[197] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[198] S. Gao, X.-C. Wen, C. Gao, W. Wang, and M. R. Lyu, "Constructing effective in-context demonstration for code intelligence tasks: An empirical study," *arXiv preprint arXiv:2304.07575*, 2023.

[199] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *arXiv preprint arXiv:2310.11248*, 2023.

[200] OpenAI., "Openai embeddings," Aug 2023. [Online]. Available: https://platform.openai.com/docs/guides/embeddings

[201] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," *arXiv preprint arXiv:2308.01240*, 2023.

[202] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," *arXiv preprint arXiv:2302.00438*, 2023.

[203] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering (ASE)*, 2018, pp. 397–407.

[204] J. H. Dawes, D. Shin, and D. Bianculli, "Towards log slicing," in *International Conference on Fundamental Approaches to Software Engineering*. Springer Nature Switzerland Cham, 2023, pp. 249–259.

[205] J. Wieting, T. Berg-Kirkpatrick, K. Gimpel, and G. Neubig, "Beyond bleu: training neural machine translation with semantic similarity," *arXiv preprint arXiv:1909.06694*, 2019.

[206] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 141–151. [Online]. Available: https://doi.org/10.1145/3236024.3236068

[207] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 210–31 227.

[208] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[209] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," *arXiv preprint arXiv:2112.08633*, 2021.

[210] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[211] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *arXiv preprint arXiv:2205.11916*, 2022.

[212] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot' s code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[213] A. R. Chen, T.-H. Chen, and S. Wang, "Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2905–2919, 2021.

[214] A. R. Chen, "An empirical study on leveraging logs for debugging production failures," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 126–128. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00055