Defects4Log: Benchmarking LLMs for Logging Code Defect Detection and Reasoning

Xin Wang
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
xwang496@connect.hkust-gz.edu.cn

Zhenhao Li*
York University
Toronto, Canada
lzhenhao@yorku.ca

Zishuo Ding*
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
zishuoding@hkust-gz.edu.cn

***Abstract*—Logging code is written by developers to capture system runtime behavior and plays a vital role in debugging, performance analysis, and system monitoring. However, defects in logging code can undermine the usefulness of logs and lead to misinterpretations. Although prior work has identified several logging defect patterns and provided valuable insights into logging practices, these studies often focus on a narrow range of defect patterns derived from limited sources (e.g., commit histories) and lack a systematic and comprehensive analysis. Moreover, large language models (LLMs) have demonstrated promising generalization and reasoning capabilities across a variety of code-related tasks, yet their potential for detecting logging code defects remains largely unexplored.**

In this paper, we derive a comprehensive taxonomy of logging code defects, which encompasses seven logging code defect patterns with 14 detailed scenarios. We further construct a benchmark dataset, *Defects4Log*, consisting of 164 developer-verified real-world logging defects. Then we propose an automated framework that leverages various prompting strategies and contextual information to evaluate LLMs' capability in detecting and reasoning logging code defects. Experimental results reveal that LLMs generally struggle to accurately detect and reason logging code defects based on the source code only. However, incorporating proper knowledge (e.g., detailed scenarios of defect patterns) can lead to 10.9% improvement in detection accuracy. Overall, our findings provide actionable guidance for practitioners to avoid common defect patterns and establish a foundation for improving LLM-based reasoning in logging code defect detection.

***Index Terms*—logging code, defects, large language models**

I. Introduction

Logging is a fundamental mechanism in software systems, generating runtime execution records (i.e., logs) that provide developers with valuable insights into system behavior and state [1], [2]. These logs are essential for a wide range of software engineering tasks, including debugging, performance analysis, and system monitoring [3]–[10]. A logging code typically consists of three main components: a logging level that indicates the severity and importance of the event, a static text message (i.e., logging text) that describes the event, and dynamic variables that provide additional context about the event. The information captured in the generated logs helps developers better understand and analyze the behavior of their code during execution.

*Corresponding authors.

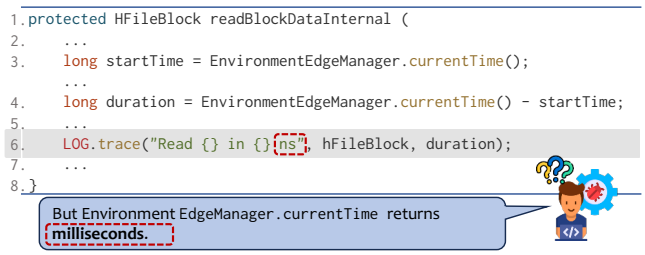

Fig. 1: An issue report from HBase (HBASE-27099) highlighting an incorrect logging text on line 6.

However, prior studies [8], [11]–[13] have shown that poorly written logging code (i.e., logging code with defects) can significantly reduce the usefulness of logs, or even mislead developers. As shown in Fig. 1, an HBase issue report [14] describes a defect in the logging code: while the actual return value is measured in milliseconds, the logging text written by developers incorrectly indicates the unit as nanoseconds. This example illustrates how inconsistencies between the semantic meaning of static logging text and its surrounding code context can mislead developers, potentially leading to incorrect assessments of system performance.

Given the severe consequences of logging code defects, several studies have explored their characteristics and developed tools for detecting such defects [11]–[13], [15]–[17]. While these works provide valuable insights into logging practices, they often focus on a narrow range of defect patterns derived from limited sources (e.g., commit) and lack a systematic and comprehensive analysis of logging code defects. Moreover, most existing detection approaches are rule-based and rely primarily on syntactic and structural analysis, which often overlook the critical factors such as data dependencies that influence runtime behavior. As a result, these techniques are limited to detecting relatively simple and predefined defect patterns. In contrast, large language models (LLMs) have shown strong generalization and reasoning capabilities across various code-related tasks [18]–[23], yet their potential for retroactively detecting logging code defects remains unexplored.

To address these limitations, we present the first comprehensive investigation of logging code defects based on three data

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00161

sources: (1) a systematic literature review, (2) issue reports from tracking systems, and (3) logging-related commits mined from open-source projects. Through this process, we derive a comprehensive taxonomy comprising seven logging code defect patterns, including previously undocumented ones, such as performance issues caused by logging in hot code paths and sensitive information leakage through logging credential data, along with 14 detailed scenarios where these defects commonly occur. This taxonomy provides a structured foundation for understanding the diverse nature of logging defects. Based on this categorization, we construct the ***Defects4Log*** benchmark, containing 164 real-world logging code defects. We then systematically evaluate the capabilities of multiple open-source and closed-source LLMs in detecting these defects, exploring the influence of contextual information, domain-specific knowledge, and prompting strategies. Finally, we provide an in-depth analysis of how LLMs' reasoning aligns with developer expectations, highlighting both strengths and areas for improvement.

Through our benchmarking results, we find that detecting defect patterns in logging code remains challenging for LLMs when only the source code is provided as input without additional guidance. In this setting, the average accuracy across various models ranges from 12.4% to 41.7%. This challenge is particularly evident when identifying semantic inconsistencies between logging code and its corresponding code context, where a majority of the average accuracy drops to 0.0%. This difficulty is due to the complex nature of the task, which goes beyond basic code understanding. LLMs must capture program behavior, track runtime variables, interpret the intent behind the logging text, and reason about their alignment with the surrounding code. Effective detection also requires awareness of project-level logging practices to ensure consistency. These aspects require a deeper, holistic understanding of the source code and its runtime behavior, which current LLMs often lack.

Nonetheless, we find that incorporating domain knowledge (i.e., detailed scenario descriptions of defect patterns) considerably improves LLMs' performance, yielding an absolute improvement of up to 10.9%. On the other hand, while structural signals from inter-procedural flow (e.g., data and control flow) provide valuable context, current LLMs still struggle to leverage this information effectively. In some cases, the inclusion of these information leads to limited or even degraded performance, underscoring the models' difficulty in capturing nuanced relationships between logging statements and overall program logic. Furthermore, our in-depth analysis of LLM-generated reasoning reveals frequent misalignments between the models' explanations and their final predictions, indicating limitations in reasoning consistency and reliability.

We summarize the contributions of this paper as follows:

- We derive a comprehensive taxonomy of logging code defects from multiple sources, including seven defect patterns and 14 detailed scenarios. These findings provide practical insights for practitioners aiming to enhance logging practices by avoiding common defect patterns.
- We construct ***Defects4Log***, a real-world benchmark dataset for evaluating the effectiveness of LLMs in logging code defect detection. The replication package of this paper is available [24].
- We evaluate the effectiveness of several LLMs on the ***Defects4Log*** benchmark. Experimental results show the effectiveness of our taxonomy and reveal a notable gap between LLMs and human developers. Our findings highlight key limitations and point to future directions for improving LLM-based reasoning in logging code defect detection.

II. Background and Related Work

A. Improving Logging Code

Prior work has explored logging defects at varying levels of scope. Some studies have investigated general logging-related issues by analyzing issue reports from two projects [12], while others have focused on specific aspects such as "how-to-log" practices [11], temporal inconsistencies between logging and surrounding code [16], duplicate logging code [13], and readability issues [17]. These efforts typically rely on mining commit histories [11], [13], [16] or conducting developer surveys [17]. More recently, Zhong et al. [25] proposed a tool, LogUpdater, which identifies logging defects based on types mined from commit histories, primarily targeting inconsistency and readability issues.

Our work complements prior studies by introducing a comprehensive taxonomy of logging defects with 14 representative scenarios, derived from multiple sources. We also benchmark LLMs on these patterns to assess their detection and reasoning capabilities.

B. Logging Code Generation

Prior research on logging code generation has focused on three key questions: where to log, what level to use, and what message to write. To address where to log, researchers have developed models, often based on deep learning, to analyze code features and recommend optimal logging locations [26], [27]. To determine the appropriate logging level, deep learning and LLMs have also been used to analyze code context and suggest the correct severity [28], [29]. Most recently, for generating the logging statement itself, approaches have employed neural machine translation and LLMs to automatically generate log messages and code based on source code context [2], [30]–[32].

All these works focus on the automatic generation of logging code, and our derived taxonomy of logging defects offers guidance for avoiding common anti-patterns. Meanwhile, few have investigated whether the generated logging code exhibits defects, where our taxonomy can serve as a useful reference.

C. Log Analysis

Log analysis transforms raw logs into structured information to support tasks such as log parsing [33], [34], anomaly detection [35], [36], fault localization [37], and performance diagnosis [38]. Regarding log parsing, researchers have developed various techniques, including parallel systems for scalability,

online methods for streaming data, and self-supervised learning models for higher accuracy [39]–[41]. Despite their varied goals, all of these approaches rely on the availability of high-quality logs. For anomaly detection, researchers have developed automated systems that transform unstructured log messages into structured patterns to identify execution anomalies, often employing machine learning [42], [43]. In the field of fault localization, studies have focused on analyzing log classifications and sequences to create contextualized information that guides programmers to the source of an error [44], [45]. Within the domain of performance diagnosis, prior work leverages event logs and machine learning to model system behavior and detect performance issues, particularly in domains like HPC [5], [7].

Our work contributes at this foundational layer by focusing on detecting defects in logging code. By improving logging quality at the source, we help strengthen the effectiveness of downstream log analysis tasks.

III. Methodology

In this section, we present our methodology for developing a taxonomy of logging code defects, constructing a benchmark dataset of real-world defect instances, and outlining our framework for evaluating the effectiveness of LLMs in detecting these defects. The overall workflow is illustrated in Fig. 2.

A. Deriving the Taxonomy of Logging Code Defects

To uncover defect patterns in logging code, we conduct a comprehensive analysis of existing literature, issue tracking systems, and commit histories. These sources are strategically selected because they offer complementary perspectives: (1) literature provides established defect classifications, (2) issue reports highlight real-world problems encountered by users and developers, and (3) commit histories reveal defects actively addressed in code along with their fixes. Our objective is to systematically identify and categorize common patterns of logging code defects and to investigate their detailed scenarios.

1) Academic Literature: We perform a structured literature review using a three-phase methodology. (1) We begin with a search for relevant papers published from 2010 to 2025, on logging code defects using keywords (e.g., "logging issues", "logging anti-patterns") from major digital libraries (i.e., IEEE Xplore, ACM Digital Library, SpringerLink). (2) We then manually screen the retrieved papers to identify those explicitly addressing logging code defects. (3) To broaden the search scope, we apply a snowballing strategy by reviewing both the references cited by the selected papers and the papers that cite them. This process yields 23 relevant papers.

2) Issue Tracking Systems: We then collect logging-related defects from issue tracking systems (e.g., JIRA and GitHub). In this process, we base our study on five open-source Java projects: Hadoop, HBase, Hive, Yarn, and ActiveMQ. We select these subject projects because they are widely used in diverse domains (e.g., message brokering, distributed computing, resource management, and databases), are actively maintained, and have been studied in prior research [12], [13], [16], [17]. We access the issue reports through their respective APIs, which provide structured data including issue titles, descriptions, comments, and associated code changes.

We search for issue reports using keywords (e.g., "log", "logging") within issue titles and descriptions, and collect 2,526 logging-related reports. Then, we sample these issues with a 95% confidence level and 5% confidence interval, resulting in 334 candidate issue reports. All sampled reports are manually examined by the authors to ensure they describe actual logging code defects. During this process, reports unrelated to logging code defects are filtered out, and 295 issue reports remain for further analysis.

3) Commit History: We also mine software repositories from GitHub to identify commits involving logging code modifications. The process begins with collecting complete commit histories from selected repositories (i.e., Hadoop, HBase, Hive, Camel, and ActiveMQ). To identify relevant commits, we retain 328 commit summaries and descriptions after filtering with relevant keywords and excluding unrelated terms (e.g., "login", "dialog"). Each filtered commit is then manually reviewed to confirm its relevance to logging code defects, and the associated code modifications are extracted for further analysis.

4) Manual Examination of Logging Code Defects: After gathering data from the three sources above, we conduct a detailed manual analysis of the extracted logging-related artifacts to identify and categorize defect patterns. This analysis also involves summarizing the potential scenarios in which each pattern may occur. Similar to prior studies [46], [47], the first author initially analyzes the descriptions and code examples from academic papers, issue reports, commit messages, and associated code changes to derive a set of preliminary defect patterns. These patterns are then reviewed and refined by the second author. In cases of disagreement, a third author participates in the discussion until consensus is reached.

B. Defects4Log Dataset Construction

Based on our prior manual examination of defect patterns, we construct ***Defects4Log***, a benchmark for evaluating LLMs in detecting logging code defects. Table I presents the distribution of defects in ***Defects4Log***. In total, the dataset comprises 164 real-world logging defect instances across seven detect patterns, collected from commit histories and issue reports. We discuss each pattern in detail in Section V.

Each instance is labeled with a specific defect pattern based on commit message or issue description, and is accompanied by detailed scenario explanations and relevant contextual information extracted from the source code. Although the instances are curated from five software systems, the identified defect patterns and our data collection methodology are broadly applicable.

C. Framework for Benchmarking LLMs in Detecting Logging Code Defects

In this section, we introduce our framework for evaluating the extent to which LLMs can assist developers in detecting

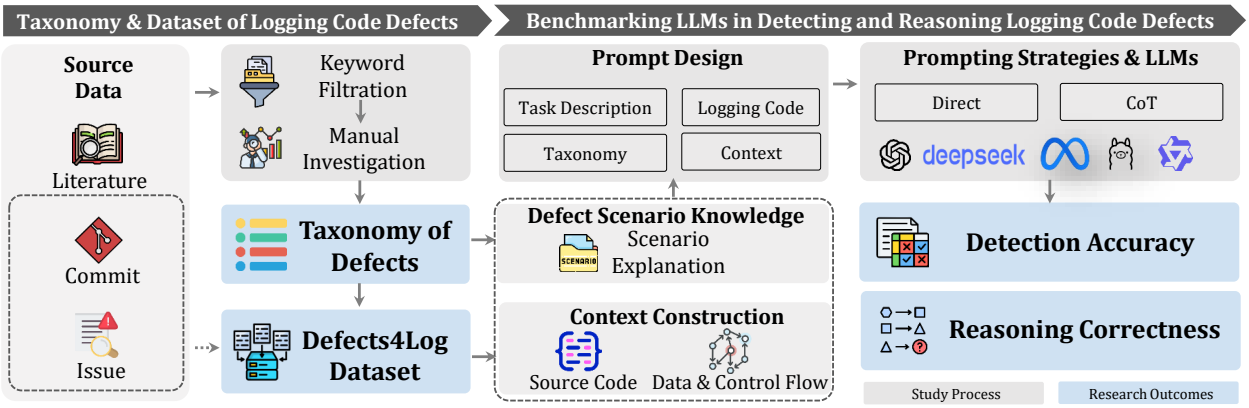

Fig. 2: An overview of our study.

TABLE I: Statistics of ***Defects4Log***.

Abbreviation	Defect Pattern	Number
RD	Readability Issues	39
VR	Variable Issues	22
LV	Logging Level Issues	26
SM	Semantics Inconsistent with Context	19
SS	Sensitive Information	23
IS	Insufficient Information	28
PF	Performance Issues	7
	Total	164

logging code defects. As shown in Fig. 2, the framework consists of four components: *Context Construction*, *Defect Scenario Knowledge*, *Prompt Design*, and *Prompting Strategy*. We describe each component in detail below.

1) Context Construction: Contextual information has been widely used to enhance the reasoning capabilities of LLMs in various code-related tasks [19], [48]–[52]. For logging code, context is especially critical, as logging code is closely tied to the execution behavior of surrounding code and the runtime state of relevant variables. Accurately interpreting logging code therefore requires a clear understanding of its surrounding context and its relationship to overall program behavior.

In our framework, we focus on three types of contextual information: (1) direct context, referring to the source code of the function that contains the logging code; and two forms of inter-procedural flow context: (2) control flow, which captures the execution paths leading to the logging code; and (3) data flow, which traces how data influences the variables referenced within the logging code.

Source code of the function. The primary type of context for logging code is the source code of the function in which the logging code resides. This context has been widely utilized in prior work on logging-related tasks [30]–[32], [53]. It captures the immediate surroundings of the logging code, such as declarations, that directly interact with or affect the logging code. Therefore, we include this function-level code as the basic form of contextual information in our framework.

Control flow. We also consider the program's control flow, which captures the potential execution sequences and decision points. This can help isolate the specific conditions or paths under which a logging code is executed by filtering out irrelevant code paths. In our framework, control flow context is defined by two specific paths relative to the logging code: a backward path, capturing the code executed before the logging code, and a forward path, representing the code executed afterward. The scope of these paths is controlled by a hyperparameter that limits the maximum length traced in each direction from the logging code.

Data flow. We also incorporate data flow context, as it directly impacts the runtime state of the variables captured in logging code. To construct this context, we first identify all variables referenced in the logging code. We then perform a backward data-flow analysis, starting from each usage and tracing dependencies in reverse to their definitions. The analysis proceeds until it reaches a data source, such as constant assignments, method parameters, or return values from function calls. In the case of method invocations, the analysis can either stop at the call site or be extended into the callee function. This process allows us to recover the computation chain that shapes the runtime state of each logged variable.

2) Defect Scenario Knowledge: Although LLMs show strong performance across diverse tasks, they often struggle with domain-specific issues without explicit contextual guidance [54], [55]. To systematically assess this, our benchmark incorporates defect scenario knowledge into the prompting process. Specifically, we provide scenario names and their corresponding explanations, which define the characteristics of each defect pattern and serve as structured guidance for evaluating the LLMs' ability to leverage such knowledge.

3) Prompt Design: We design several prompt templates to evaluate LLMs under different input configurations. Each template is built around a core structure that includes the task description (i.e., decide from *"no defect"*, or any one of the defect patterns), the source code context (i.e., the function containing the logging code), the target logging code,

and the taxonomy of defect patterns. Variants of this base template additionally include inter-procedural flow context (i.e., control flow and data flow), domain knowledge (i.e., scenario descriptions), or both. These prompts are intended to guide the LLM in predicting the correct defect pattern and generating an explanation of its reasoning process.

4) Prompting Strategy: We experiment with two prompting strategies: **Direct** prompting, and **Chain of Thought (CoT)** prompting [56]. Both strategies are applied using the prompt templates described in the previous section.

IV. Experiment Setup

A. Research Questions

We study the following research questions:

- **RQ1:** How prevalent are logging code defects in open-source projects?
- **RQ2:** How is the effectiveness of different LLMs in detecting logging code defects?
- **RQ3:** How can LLMs correctly reason the detected logging code defects?

B. Evaluation Metrics

The detection task is formulated as a multi-class classification problem, where LLMs are expected to determine whether the given input corresponds to one of the defect patterns or is non-defective. A detection result is considered correct if the identified pattern matches the ground truth. We compute the accuracy for each defect pattern individually and report the macro-average accuracy. This enables a more fine-grained evaluation of the model's performance across different patterns of logging defects.

$$\text{Accuracy}_{\text{pattern}} = \frac{\text{Number of Correctly Detected Instances}}{\text{Total Instances of the Pattern}}$$

C. Studied LLMs and Implementation Details.

In our experiments, we study four widely used LLMs, including both open-source and closed-source ones. For open-source models, we select `Qwen2.5-72B` [57], `DeepSeek-R1` [58], and `Llama3.3` [59]. For the closed-source model, we choose `GPT-4o` [60].

All experiments are conducted on a Linux server running Ubuntu 20.04.6 LTS, equipped with an AMD 32-core processor, 1TB RAM, and eight NVIDIA A6000 GPUs. To extract inter-procedural flow information from the source code, we use SciTools Understand [61].

V. Results

In this section, we discuss the results of each RQ.

A. *RQ1: Prevalence of Logging Code Defects*

Through our comprehensive investigation (cf. Section III-A), we develop a taxonomy consisting of seven logging code defect patterns and 14 detailed scenarios in which these defects commonly occur. Table II presents an overview of these patterns. In the following, we elaborate on each defect pattern, describe its corresponding scenarios with the number of instances in ***Defects4Log*** inside *"()"*, and provide real-world examples.

TABLE II: Taxonomy of Logging Code Defects in ***Defects4Log*** (RQ1)

Defect Patterns	Scenario Name
RD: Readability Issues	**RD-1:** Complicated domain-specific terminology **RD-2:** Non-standard language used **RD-3:** Poorly formatted or unclear messages
VR: Variable Issues	**VR-1:** Incorrect variable value logging **VR-2:** Placeholder–value mismatch
LV: Logging Level Issues	**LV-1:** Improper verbosity level
SM: Semantics Inconsistent	**SM-1:** Wrong unit or metric label **SM-2:** Message text does not match the code **SM-3:** Misused variables in the message
SS: Sensitive Information	**SS-1:** Credentials logged in plain text **SS-2:** Dumping whole objects without scrubbing
IS: Insufficient Information	**IS-1:** Insufficient information
PF: Performance Issues	**PF-1:** Logging on hot path. **PF-2:** Costly string operations

Readability Issues (RD). This defect pattern includes issues in logging code that reduce the clarity of the produced log messages, making them harder for practitioners to interpret and for automated tools to parse.

Defect Scenarios:

- **RD-1** *Complicated domain-specific terminology (2):* Complex domain terms that may reduce message clarity for broader audiences.
- **RD-2** *Non-standard language used (15):* Grammatical errors or informal phrasing that affect message professionalism and clarity.
- **RD-3** *Poorly formatted or unclear messages (22):* Poorly structured messages that impede comprehension and hinder debugging.

Real-world Example: HBASE-24367 [62] presents a case where the average elapsed time is logged in nanoseconds, which one comment describes as non-intuitive or even "annoying". The issue reporter also notes that such high precision is unnecessary for averages. In the fix for similar issues, developers convert the time unit to milliseconds, with the commit message stating "log elapsed timespan in a human-friendly format". Detecting such issues requires LLMs to reason about return value ranges via call chains and interpret the semantics of logging text (e.g., recognizing that "average" implies high precision is unnecessary in such situations).

```
// HBASE-24367
LOG.info("{} average execution time: {} ns.", getName(),
   (long)(timeMeasurement.getAverageTime()));
```

Variable Issues (VR). This defect pattern refers to logging incorrect or mismatched variables that may cause confusion or mislead developers during debugging.

Defect Scenarios:

- **VR-1** *Incorrect variable value logging (3):* Logging a null or raw address instead of the actual value that may cause crashes or mislead developers.
- **VR-2** *Placeholder–value mismatch (19):* Mismatched placeholders and variables that lead to confusing or misleading outputs.

Real-world Example: A commit in AutoMQ [63] illustrates a clear case of a placeholder-argument mismatch. As shown in the listing below, the logging code contains three placeholders, but only two arguments are provided, resulting in the omission of the epoch value. The issue has been resolved by adding the missing epoch variable. Detecting such patterns requires models to accurately align placeholders with their corresponding arguments, while fixing them demands deep contextual reasoning to infer the developer's intent and identify the appropriate variable from the surrounding code.

```
// AutoMQ-Commit-e6f8ca8
log.trace("tryAppend(nodeId={}, epoch={}): the given node id
   does not match the current leader id of {}.", nodeId,
   leader.nodeId());
```

Logging Level Issues (LV). This pattern captures issues where the assigned logging level is inconsistent with the severity or importance of the event.

Defect Scenarios:

- **LV-1** *Improper verbosity level (26):* The verbosity level that does not accurately reflect the seriousness of the message.

Real-world Example: Issue HDFS-15045 [64] demonstrates an inappropriate logging level where an exception is logged using `INFO`, as shown in the code below. Using the `INFO` level for an exception is risky because the message can be easily overlooked. A developer on the issue report confirmed this risk, noting that clients often set their levels to `WARN`. In this common scenario, the logs for this exception would be completely hidden, preventing users from understanding the reason for a critical pipeline failure. Detecting this issue requires the model to go beyond syntax and understand implicit software engineering conventions (e.g., avoiding logging exceptions at the `INFO` level), as well as reason about the severity of the system behavior captured by the corresponding logging code.

```
// HDFS-15045
LOG.info("Exception in createBlockOutputStream " + this, ie);
```

Semantics Inconsistent (SM). This pattern encompasses logging code that is semantically inconsistent with the actual behavior or state of the surrounding code.

Defect Scenarios:

- **SM-1** *Wrong unit or metric label (1):* Incorrect units or labels that may mislead interpretation.
- **SM-2** *Message text doesn't match the code's action or state (12):* The log message that misrepresents the actual code behavior or logic.
- **SM-3** *Misused variables in the message (6):* Incorrect variables that may mislead users.

Real-world Example: Issue YARN-6951 [65] demonstrates a semantics inconsistent defect where the logging text contradicts the actual program logic. As shown in the listing below, the code logs "Resource handler chain enabled = true" when the `resourceHandlerChain` is `null`, which is misleading because `null` indicates an inactive or non-existent state. A developer has confirmed in the issue report that this was incorrect and clarified that the logs should display "true" only when the chain is not `null`. To detect it, a model must understand the natural language semantics of the word "enabled" and recognize its logical contradiction with the programming convention where null implies a disabled state, a reasoning process that requires aligning natural language with code semantics.

```
// YARN-6951
LOG.debug(\"Resource handler chain enabled = \" +
   (resourceHandlerChain == null));
```

Similarly, Fig. 1 shows a case of inconsistency [14] where the variable `duration` returns a value in milliseconds, while the logging text incorrectly labels it as "ns" (nanoseconds). Although this problem appears similar to the earlier example [62] categorized under Readability Issues, which also involves a unit correction, it reflects a completely different defect pattern. While the previous case affects log clarity for human readers, this example concerns the semantic correctness of the logged information. Detecting it requires the model to trace the data flow and reason about the true unit of the variable, highlighting the need for a deeper understanding of program behavior.

Sensitive Information (SS). This defect pattern captures scenarios where generated logs expose private or confidential data, a critical issue that has been overlooked in prior research on logging defects.

Defect Scenarios:

- **SS-1** *Credentials logged in plain text (7):* Logging code that exposes private or confidential data (e.g., password, token) in plain text during system execution, potentially leading to security and privacy risks.
- **SS-2** *Dumping whole objects / configs without scrubbing (16):* Sensitive information may be embedded within objects or configuration data, which can be inadvertently exposed if not properly scrubbed.

Real-world Example: Issue HIVE-20796 [66] demonstrates the risk of logging potentially sensitive information. As shown in the listing below, the variable `confVal` is logged directly, which poses a security risk because it can contain a JDBC URL. A developer on the issue report has elaborated on this risk, explaining that such URLs can embed sensitive credentials like passwords and therefore must be "masked out" before being logged. Detecting this defect requires the model not only to recognize what types of information are considered sensitive, but also to infer the potential contents of generic variables like `confVal`, which may hold confidential data. Additionally, it must understand the security principle of data masking, a process that extends beyond code syntax and draws on real-world security practices to properly address such issues.

```
// HIVE-20796
LOG.debug("Overriding {} value {} from jpox.properties with
   {}", varName, prevVal, confVal);
```

Insufficient Information (IS). This pattern encompasses logging code that lacks essential details needed to understand or diagnose events.

Defect Scenarios:

- **IS-1** *Insufficient information (28):* This issue typically arises when an error occurs and developers need more contextual information for diagnosis, but the logs lack key error details or essential event context.

Real-world Example: Issue HDFS-17310 [67] illustrates the logging code with insufficient context for effective debugging. As shown in the listing below, the original error logging code states that a plan submission failed but provides no specific identifiers. To make troubleshooting more convenient, as noted by the developer, the logging code has been enhanced by adding crucial details: the `planFile` and `planId`. Detecting such defects is challenging, as it requires the model to assess the diagnostic quality of logs and reason about whether they lack sufficient contextual information for effective troubleshooting.

```
// HDFS-17310
LOG.error("Disk Balancer - Executing another plan, submitPlan
   failed.");
```

Performance Issues (PF). This defect pattern refers to logging code that may negatively affect system performance due to where or how they are executed.

Defect Scenarios:

- **PF-1** *Logging on hot path (6):* Logging in tight loops or latency-sensitive code that impacts system performance.
- **PF-2** *Costly string operations (1):* Costly string concatenation or formatting that increases CPU or memory usage.

Real-world Example: Issue HIVE-25794 [68] reports a performance issue caused by using string concatenation inside a frequently executed `LOG.debug()` call. In the issue report, a developer mentions that this leads to memory pressure when processing a large number of files, as the string construction occurs even when the logging level is set to `INFO`. The issue has been fixed by replacing the concatenation with a parameterized logging approach using placeholders. Detecting such issues requires reasoning about the cost of string operations and their impact in performance-critical code.

```
// HIVE-25794
LOG.debug("Found spec for " + path + " " + otherPart + " from
   " + pathToPartInfo);
```

RQ1 Summary: The taxonomy of ***Defects4Log*** summarizes seven patterns of logging code defects and 14 real-world defect scenarios. We find that the summarized logging code defects are prevalent in real-world projects, with some defects (e.g., Readability Issues) appearing more commonly in our studied projects.

B. *RQ2: Effectiveness of LLMs on Defects4Log*

In this RQ, we aim to benchmark the effectiveness of current LLMs in detecting logging code defects.

Approach. Using the dataset constructed in RQ1 and the accuracy metric defined in Section IV, we evaluate LLM performance across three key dimensions: (1) the choice of LLM, (2) the prompting strategy used, and (3) the influence of different types of input context information.

Table III presents the results of this RQ for each pattern. The *Setting* column distinguishes between `Direct` (i.e., direct prompting) and `CoT` (i.e., chain-of-thought prompting) with different input contexts. The modifier `+K` indicates the inclusion of knowledge of defect scenarios, and `+I` indicates the inclusion of inter-procedural information. For each setting, we report individual results per pattern and the averaged accuracy across patterns. For each pattern, the highest accuracy is highlighted in **bold and red**. The prompting strategy with the highest average accuracy is highlighted in red in its cell, while the lowest is highlighted in blue.

Comparison of LLMs. Overall, `Deepseek-R1` consistently achieves the best performance across the majority of prompting settings. Specifically, its `Direct+K` setting yields the highest average accuracy (i.e., 47.6%), with the highest results on patterns such as VR (i.e., 68.2%), and SS (65.2%). Its `Direct+K+I` setting follows with an average accuracy of 45.6%. `GPT-4o` performs moderately well, with its best average accuracy of 27.3% under `CoT+K`. `Llama3.3` achieves high accuracy in RD (i.e., 69.2%) under `Direct+K+I` and `CoT+K+I` (69.2%) but almost completely ineffective in some of the patterns (e.g., 0.0% in SM and PF). `Qwen2.5-72B` achieves the lowest average accuracy (i.e., 12.4%) in the setting of `Direct`.

Comparison of Prompting Strategies. We find that CoT does not obviously outperform `Direct` across different models and patterns. For instance, in `GPT-4o`, the average accuracy of `CoT+K+I` is 26.2%, only slightly higher than `Direct+K+I` (i.e., 23.9%), while some CoT variants (e.g., `CoT+K` = 27.3%) perform worse than their direct settings (e.g., `Direct+K` = 28.4%). A similar trend is observed in `Llama3.3`, where `Direct+K` (i.e., 18.2% on average) outperforms all CoT-based settings. Even in the best-performing model, `Deepseek-R1`, the improvement from CoT prompting is marginal and sometimes reversed. For example, `Direct+K` achieves a higher average (i.e., 47.6%) than `CoT+K` (45.0%). These results suggest that `CoT` prompting may not provide noticeable improvements for tasks involving detecting logging code defects, especially when the models already have sufficient capabilities to reason directly from enriched input. Instead, it is possible that `CoT` may introduce unnecessary verbosity or distract from the precise detection of defects related to logging code.

Impact of Input Context. As shown in Table III, we also report the relative change in average performance when augmenting the prompt with detailed knowledge of pattern scenarios (`+K`) and/or inter-procedural information (`+I`). Overall, adding detailed knowledge of pattern scenarios (`+K`) tends to

TABLE III: Accuracy (%) of LLMs on ***Defects4Log*** (RQ2).

Model	Setting	RD	VR	LV	SM	SS	IS	PF	*Average*
GPT-4o	*Direct*	46.2	9.1	3.8	0.0	34.8	14.3	14.3	17.5
	Direct+K	41.0	36.4	15.4	0.0	52.2	39.3	14.3	28.4 (+10.9)
	Direct+I	46.2	0.0	3.8	0.0	43.5	14.3	28.6	19.5 (+2.0)
	Direct+K+I	41.0	27.3	7.7	0.0	52.2	39.3	0.0	23.9 (+6.4)
	CoT	43.6	13.6	3.80	0.0	47.8	14.3	14.3	19.6
	CoT+K	41.0	27.3	15.4	5.3	52.2	**50.0**	0.0	27.3 (+7.7)
	CoT+I	43.6	4.5	19.2	5.3	47.8	7.1	14.3	20.2 (+0.6)
	CoT+K+I	41.0	27.3	7.7	15.8	52.2	39.3	0.0	26.2 (+6.6)
Llama3.3	*Direct*	51.3	0.0	3.8	0.0	52.2	3.6	0.0	15.8
	Direct+K	61.5	4.5	0.0	0.0	43.5	17.9	0.0	18.2 (+2.4)
	Direct+I	53.8	0.0	0.0	0.0	47.8	3.6	0.0	15.0 (-0.8)
	Direct+K+I	**69.2**	4.5	0.0	0.0	34.8	7.1	0.0	16.5 (+0.7)
	CoT	51.3	0.0	3.8	0.0	47.8	3.6	0.0	15.2
	CoT+K	48.7	4.5	0.0	0.0	30.4	14.3	0.0	14.0 (-1.2)
	CoT+I	46.2	0.0	0.0	0.0	52.2	7.1	0.0	15.1 (-0.1)
	CoT+K+I	**69.2**	4.5	0.0	0.0	34.8	7.1	0.0	16.5 (+1.3)
DeepSeek-R1	*Direct*	56.4	22.7	**19.2**	31.6	**65.2**	39.3	57.1	41.7
	Direct+K	56.4	**68.2**	15.4	42.1	**65.2**	28.6	57.1	**47.6** (+5.9)
	Direct+I	48.7	22.7	15.4	47.4	56.5	42.9	57.1	41.5 (-0.2)
	Direct+K+I	46.2	40.9	**19.2**	**52.6**	56.5	32.1	71.4	45.6 (+3.9)
	CoT	53.8	13.6	**19.2**	21.1	56.5	32.1	57.1	36.2
	CoT+K	53.8	**68.2**	7.7	0.0	56.5	28.6	**100.0**	45.0 (+8.8)
	CoT+I	43.6	22.7	7.7	42.1	52.2	32.1	71.4	38.8 (+2.6)
	CoT+K+I	46.2	40.9	**19.2**	**52.6**	56.5	32.1	57.1	43.5 (+7.3)
Qwen2.5-72B	*Direct*	23.1	9.1	7.7	0.0	21.7	10.7	14.3	12.4
	Direct+K	46.2	9.1	3.8	0.0	34.8	14.3	14.3	17.5 (+5.1)
	Direct+I	35.9	0.0	3.8	0.0	17.4	3.6	28.6	12.8 (+0.4)
	Direct+K+I	46.2	9.1	3.8	0.0	26.1	17.9	14.3	16.8 (+4.4)
	CoT	30.8	9.1	3.8	0.0	21.7	17.9	28.6	16.0
	CoT+K	59.0	13.6	0.0	0.0	21.7	14.3	0.0	15.5 (-0.5)
	CoT+I	38.5	4.5	3.8	0.0	26.1	3.6	14.3	13.0 (-3.0)
	CoT+K+I	46.2	9.1	3.8	0.0	26.1	17.9	14.3	16.8 (+0.8)

consistently improve the accuracy across various prompting settings and LLMs. For example, `GPT-4o`'s average accuracy increases from 17.5% for `Direct` to 28.4% for `Direct+K`, and `Deepseek-R1`'s performance improves from 41.7% for `Direct` to 47.6% for `Direct+K`. Similarly, `Qwen2.5-72B` has an improvement from 12.4% to 17.5% for `Direct+K`. This indicates that providing LLMs with concrete scenarios of the logging code defect patterns may help them better understand the meaning of those defects and perform the detection.

In contrast, incorporating inter-procedural information (`+I`) yields mixed results. While it enhances performance in some patterns (e.g., improved PF in `Deepseek-R1` under `Direct+K+I`), it does not provide consistent gains. In some cases, the inclusion of inter-procedural information leads to performance drops, such as in `Llama3.3` where `Direct+I` (i.e., 15.0%) performs worse than `Direct` (i.e., 15.8%). These observations suggest that although `+I` provides useful contextual cues, its effectiveness depends on how well the underlying LLMs can interpret and leverage static code structures. Without proper alignment, it may introduce additional noise rather than benefit the reasoning process.

RQ2 Summary: We find that `Deepseek-R1` generally outperforms other LLMs (e.g., `Direct+K` achieves an average accuracy of 47.6%). In terms of prompting strategies, `CoT` does not demonstrate clear improvements over `Direct` prompting. Additionally, including detailed knowledge of pattern scenarios (`+K`) leads to noticeable performance gains, while incorporating inter-procedural information (`+I`) shows mixed effects and sometimes decreases performance.

C. ***RQ3:*** *Capability of Reasoning the Results*

In RQ2, we focus on the final prediction, specifically, whether the LLMs can correctly detect the defect patterns in logging code. However, relying solely on the final answer may overlook valuable insights embedded in the model's reasoning process. This is critical because, even if an LLM correctly detects a defect, it may fail to provide a coherent or accurate explanation of why the code is defective. Such reasoning gaps can lead to misunderstandings or negatively affect downstream tasks, such as defect fixing, thereby limiting the model's practical utility in assisting developers. Therefore, in this research question,

we investigate the LLM's ability to generate accurate and meaningful reasoning behind its predictions.

Approach. To answer this RQ, we first select cases generated by the best-performing LLM, `Deepseek-R1` with `Direct` prompt augmented by domain knowledge (i.e., `Direct+K`), and the worst-performing LLM, `Qwen2.5-72B` with the `Direct` prompt (i.e., `Direct`), as they represent two distinct performance extremes. We then manually evaluate the reasoning provided by each model by comparing it against the ground truth, which is derived by the authors based on code changes, issue descriptions, developer discussions, and commit messages, as well as the surrounding source code. Two authors independently review and label the reasoning correctness, achieving substantial agreement with a Cohen's Kappa score of 0.88 [69].

Table IV presents the performance of two LLMs in understanding defect patterns by evaluating both the correctness of their predicted patterns and the accompanying reasoning. Results are reported for each defect pattern (e.g., RD, VR) along with the overall average of percentage. In this table, ✓ and ✗ denote correct and incorrect results, respectively.

Comparison of LLMs. Overall, `DeepSeek-R1 & Direct+K` considerably outperforms `Qwen2.5-72b & Direct` in both defect pattern detection and reasoning correctness. `Deepseek-R1` achieves an average "Pattern Correct & Reason Correct" (PCRC) rate of 45.7%, whereas `Qwen2.5-72B` attains 11.6% average PCRC. This indicates a substantial difference in their ability to both correctly identify defect patterns and justify their predictions. The majority of instances for `Deepseek-R1` fall into "Pattern Incorrect & Reason Incorrect" (PIRI) with an average of 53.0%, while `Qwen2.5-72B` has a much higher average PIRI rate of 82.9%. This is reasonable as incorrect reasoning typically leads to incorrect final predictions.

Reasoning Results by Defect Pattern. `DeepSeek-R1 & Direct+K` achieves strong PCRC results for specific defect patterns, with 69.6% on VR and 65.2% on SS, but performs poorly on LV (11.5%). In comparison, `Qwen2.5-72B & Direct` has its best PCRC on PF (28.6%) and RD (28.2%), but fails completely on VR and SM (0.0% PCRC), with SM showing a 100.0% PIRI. Both models struggle with LV, where `Qwen2.5-72B` reaches 3.8% PCRC.

Observations on Reasoning Accuracy and Errors. `DeepSeek-R1` demonstrates strong reasoning capability, with an average PCRC of 45.7% and an average "Pattern Correct & Reason Incorrect" (PCRI) of 1.2%. Notably, `Deepseek-R1` shows no instances (0.0% average) of "Pattern Incorrect & Reason Correct" (PIRC). In contrast, `Qwen2.5-72B & Direct` shows lower reasoning accuracy at 82.9%, with a slightly higher average PCRI of 2.4%. This model also has a 3.0% average PIRC rate, with higher occurrences on VR (18.2%) and PF (14.3%) defect patterns.

Case Study. We present four illustrative examples corresponding to the correctness in defect pattern detection and the associated reasoning provided by LLMs.

Pattern Correct & Reason Correct. Issue YARN-8907 [70] exemplifies a semantic inconsistent defect where a logging code at the end of a test case incorrectly prints `START`, misleadingly signaling the beginning of a test that has already completed. A developer has fixed this by changing `START` to `END`. The LLM successfully detects this semantics inconsistent issue in the logging code, demonstrating its ability to identify logical contradictions by aligning the natural language of the logging code with the behavioral context of the surrounding code.

```
// YARN-8907
LOG.info("--- START: testNotAssignMultiple ---");
```

Pattern Correct & Reason Incorrect. The code in HDFS-15197 [71] illustrates a case where the LLM identifies the correct defect pattern but for the wrong reason. Specifically, the model correctly flags the use of the `INFO` level for logging an exception as a potential issue related to the logging level. It recommends raising the level to `WARN` or `ERROR` to improve visibility. However, the developer's actual fix lowers the logging level to `DEBUG`, citing concerns over excessive log volume in scenarios involving frequent `ObserverRetryOnActiveException`. This case highlights a subtle challenge: although the LLM's reasoning appears intuitively valid, as many developers might assume that exceptions should be logged at higher severity levels, it fails to align with the intended logging behavior and context understood by the developer. This discrepancy might arise from the LLM's lack of project-specific background knowledge and familiarity with the system's established logging practices.

```
// HDFS-15197
LOG.info("Encountered ObserverRetryOnActiveException from
  {}." + " Retry active namenode directly.",
  current.proxyInfo);
```

Pattern Incorrect & Reason Correct. This case highlights the result that correctly explains the reason for a defect but misclassifies its defect pattern. In the example from HDFS-14407 [72], the LLM accurately detects that improper string concatenation causes the {} placeholder to remain unsubstituted in the final log message. However, it labels the issue as a general string concatenation problem, instead of **VR-2** *Placeholder–value mismatch*, which refers to mismatches between placeholders and variables that lead to confusing or misleading log output. In the actual fix, the developer removes the string concatenation and correctly supplies the variable as an argument. This case illustrates a key limitation: without domain-specific knowledge, an LLM's reasoning may not align with the defect categorizations used by developers and may struggle to generate precise fix suggestions.

```
// HDFS-14407
LOG.warn("checkAllVolumes timed out after {} ms +
  maxAllowedTimeForCheckMs);
```

Pattern Incorrect & Reason Incorrect. This example [73] demonstrates a complete failure in the LLM's analysis, where both the reasoning and the prediction are incorrect. The model classifies the logging code as correct and reports no issues. However, the code actually contains a **SM-3** *Misused Variables*

TABLE IV: Evaluation of LLM Reasoning: Correctness of Defect Pattern Detection and Reasoning (RQ3)

Model & Setting	Pattern	Reason	RD	VR	LV	SM	SS	IS	PF	*Average*
DeepSeek-R1 & Direct+K	✓	✓	55.3	69.6	11.5	42.1	65.2	28.6	57.1	45.7
	✓	✗	2.6	0.0	3.8	0.0	0.0	0.0	0.0	1.2
	✗	✓	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	✗	✗	42.1	30.4	84.6	57.9	34.8	71.4	42.9	53.0
Qwen2.5-72B & Direct	✓	✓	28.2	0.0	3.8	0.0	17.4	3.6	28.6	11.6
	✓	✗	7.7	4.5	0.0	0.0	0.0	0.0	0.0	2.4
	✗	✓	0.0	18.2	0.0	0.0	0.0	0.0	14.3	3.0
	✗	✗	64.1	77.3	96.2	100.0	82.6	96.4	57.1	82.9

in the Message defect, where an incorrect variable is used in the logging code. Specifically, the logging references `tempTarget` instead of the intended target variable, `target`, which can mislead users. This issue is later fixed by a developer by replacing `tempTarget` with `target`. The LLM fails to detect the defect, likely due to its inability to accurately interpret variable usage within the surrounding code context.

```
//Camel Commit 0c1a589
log.trace("Deleting existing file: " + tempTarget);
```

RQ3 Summary: We find that even when LLMs correctly identify logging code defects, their accompanying reasoning is not always accurate or consistent with the code context. While correct predictions often align with valid reasoning, there are notable cases where the reasoning is incomplete or incorrect (e.g., a PIRI of 53.0% and 82.9%).

VI. Discussion

A. Implications

We summarize the implications of our study, where P1 - P3 are targeted at practitioners, while R1 - R2 provide insights for LLM researchers.

Implication P1: Actionable guidelines for better logging. Our findings provide practical takeaways for software practitioners aiming to improve their logging practices. Practitioners can leverage our logging code defect patterns and the detailed scenario knowledge as guidelines (i.e., the results of RQ1). These patterns and scenarios can serve as a checklist during code development and reviews, or be integrated into developer training materials, to proactively improve the quality logging code and reduce defects.

Implication P2: Importance of domain knowledge. Our findings in RQ2 show that enriching prompts with detailed scenario knowledge of defect patterns (i.e., `+K`) considerably improves LLM performance in detecting and reasoning the logging code defects. Practitioners can leverage this insight by incorporating detailed domain knowledge, such as concrete defect scenarios, explanations, and examples into prompt design or LLM fine-tuning pipelines.

Implication P3: Using LLM-generated reasoning with caution. The results of RQ3 show that LLMs may produce incorrect or misleading reasoning. When integrating LLMs into development workflows (e.g., for code review or log monitoring), it is important to verify both the detection results and their justifications before acting on them.

Implication R1: Better structural code understanding for LLMs. We find that incorporating inter-procedural information (i.e., `+I`), such as control flow and data flow, does not consistently improve LLM performance and even degrades it. This suggests that current LLMs, though proficient in processing natural language and source code, may lack the structural understanding required to analyze code semantics effectively. Prior studies have demonstrated the utility of static analysis in traditional defect detection, highlighting the importance of structural code information. Future LLM research may explore ways to enhance models' ability to interpret and utilize code structure, for example, by pretraining on program analysis artifacts, designing structure-aware representations, or integrating symbolic reasoning components.

Implication R2: Strengthening the LLMs' reasoning quality. Our RQ3 analysis reveals that even when LLMs correctly detect defect patterns in logging code, their accompanying explanations may be incomplete, inaccurate, or inconsistent with the actual code context. Conversely, LLMs can also produce logically sound reasoning for incorrect predictions, which indicates a gap between surface-level plausibility and genuine understanding. This discrepancy highlights the need to strengthen the reasoning capabilities of LLMs. Future research may explore techniques that align prediction outcomes with verifiable resoning process, or post-hoc verification mechanisms that assess the internal consistency of LLM-generated results.

B. Threats to Validity

Internal Validity. The validity of our taxonomy may be threatened by the diversity of our data sources. To mitigate this risk, we triangulate our findings through a comprehensive analysis of academic literature, issue tracking systems, and commit histories.

Since the dataset construction and the result analysis in this study involves a manual study, its validity can be influenced by the knowledge and experiences of the participants. To mitigate potential biases in such process, two authors independently perform the labeling, achieving substantial agreement, with an overall Cohen's Kappa value of 0.88.

External Validity. Our study investigate a specific set of four LLMs. While these represent a mix of open-source and closed-source models with varying capabilities, their performance in detecting and reasoning about logging defects may not be representative of all existing or future LLMs.

Since the ***Defects4Log*** benchmark is constructed using data from open-source Java projects, the number of studied subjects and programming languages may pose a threat to the study's validity. To mitigate this, careful consideration goes into the selection of subjects. These analyzed projects are well-known and have gained considerable attention from developers and researchers, based on the stars on GitHub and existing research papers [11], [12].

Although our analysis is based on large and widely-used Java systems, our findings on defect prevalence and LLM effectiveness may not generalize to projects from other domains or those written in different programming languages. Future studies could explore the generalizability of our results across these different contexts.

VII. Conclusion

In this paper, we present our comprehensive study on logging code defects, which identifies seven distinct patterns and constructs the ***Defects4Log*** benchmark with developer-verified real-world instances. Our evaluation of LLMs on ***Defects4Log*** reveals a notable gap between model predictions and human understanding, highlighting the limitations of current LLMs in reasoning about logging code and system runtime behavior. These findings underscore the value of our benchmark in guiding future research. Moreover, our work provides actionable insights for developers to avoid common defect patterns and lays a foundation for advancing LLM-based reasoning and defect detection in software logging.

References

[1] Y. Li, Y. Huo, Z. Jiang, R. Zhong, P. He, Y. Su, L. C. Briand, and M. R. Lyu, "Exploring the effectiveness of llms in automated logging statement generation: An empirical study," *IEEE Transactions on Software Engineering*, 2024.

[2] R. Zhong, Y. Li, G. Yu, W. Gu, J. Kuang, Y. Huo, and M. R. Lyu, "Beyond llms: An exploration of small open-source language models in logging statement generation," *arXiv preprint arXiv:2505.16590*, 2025.

[3] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: error diagnosis by connecting clues from run-time logs," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010.

[4] T. Barik, R. DeLine, S. Drucker, and D. Fisher, "The Bones of the System: A Case Study of Logging and Telemetry at Microsoft," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016.

[5] F. Milani and F. M. Maggi, "A comparative evaluation of log-based process performance analysis techniques," in *Business Information Systems*.

[6] A. R. Chen, "An empirical study on leveraging logs for debugging production failures," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 126–128.

[7] 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*, ser. ICSE-SEIP '20, 2020.

[8] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, 2021.

[9] 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.*, 2021.

[10] Z. Li, C. Luo, T.-H. Chen, W. Shang, S. He, Q. Lin, and D. Zhang, "Did we miss something important? studying and exploring variable-aware log abstraction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 830–842.

[11] B. Chen and Z. M. Jiang, "Characterizing and Detecting Anti-Patterns in the Logging Code," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.

[12] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Softw. Engg.*, 2018.

[13] Z. Li, T.-H. Chen, J. Yang, and W. Shang, "DLFinder: Characterizing and Detecting Duplicate Logging Code Smells," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[14] "HBASE-27099," https://issues.apache.org/jira/browse/HBASE-27099.

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

[16] Z. Ding, Y. Tang, Y. Li, H. Li, and W. Shang, "On the Temporal Relations between Logging and Code," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[17] Z. Li, A. R. Chen, X. Hu, X. Xia, T.-H. Chen, and W. Shang, "Are They All Good? Studying Practitioners' Expectations on the Readability of Log Messages," 2023.

[18] Y. Peng, A. D. Gotmare, M. R. Lyu, C. Xiong, S. Savarese, and D. Sahoo, "Perfcodegen: Improving performance of llm generated code with execution feedback," in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, 2025.

[19] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning runtime behavior of a program with llm: How far are we?" in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, 2025.

[20] Y. Peng, J. Wan, Y. Li, and X. Ren, "Coffe: A code efficiency benchmark for code generation," *arXiv preprint arXiv:2502.02827*, 2025.

[21] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code search is all you need? improving code suggestions with code search," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[22] X. Hu, F. Niu, J. Chen, X. Zhou, J. Zhang, J. He, X. Xia, and D. Lo, "Assessing and advancing benchmarks for evaluating large language models in software engineering tasks," *arXiv preprint arXiv:2505.08903*, 2025.

[23] B. Nouriinanloo and M. Lamothe, "Re-ranking step by step: Investigating pre-filtering for re-ranking with large language models," *arXiv preprint arXiv:2406.18740*, 2024.

[24] "Replicaton package," https://github.com/klsc749/Defects4Log, last accessed in August 2025.

[25] R. Zhong, Y. Li, J. Kuang, W. Gu, Y. Huo, and M. R. Lyu, "Logupdater: Automated detection and repair of specific defects in logging statements," *ACM Trans. Softw. Eng. Methodol.*, 2025.

[26] 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*, 2015.

[27] Z. Li, T.-H. Chen, and W. Shang, "Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[28] 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)*, 2021.

[29] Y. W. Heng, Z. Ma, Z. Li, D. J. Kim, and T.-H. Chen, "Benchmarking open-source large language models for log level suggestion," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2025, pp. 314–325.

[30] 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)*, 2022.

[31] 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.*, 2023.

[32] Y. Li, Y. Huo, R. Zhong, Z. Jiang, J. Liu, J. Huang, J. Gu, P. He, and M. R. Lyu, "Go static: Contextualized logging statement generation," *Proc. ACM Softw. Eng.*, 2024.

[33] Z. Ma, A. R. Chen, D. J. Kim, T.-H. Chen, and S. Wang, "Llmparser: An exploratory study on using large language models for log parsing," ser. ICSE '24, 2024.

[34] Y. Xiao, V.-H. Le, and H. Zhang, "Free: Towards more practical log parsing with large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 153–165.

[35] 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)*, 2023, pp. 497–509.

[36] Y. Xie, H. Zhang, and M. A. Babar, "Logsd: Detecting anomalies from system logs through self-supervised learning and frequency-based masking," *Proceedings of the ACM on Software Engineering*, no. FSE, pp. 2098–2120, 2024.

[37] 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*, pp. 2905–2919, 2021.

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

[39] 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*, 2018.

[40] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017.

[41] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-supervised Log Parsing," in *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track*, 2021.

[42] 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*, 2009.

[43] B. Debnath, M. Solaimani, M. A. G. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, and L. Khan, "LogLens: A real-time log analysis system," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*.

[44] D.-Q. Zou, H. Qin, and H. Jin, "UiLog: Improving log-based fault diagnosis by log analysis."

[45] W. Dobrowolski, K. Iwach-Kowalski, M. Nikodem, and O. Unold, "Log-based fault localization with unsupervised log segmentation."

[46] J. Chen, Z. Li, Q. Mao, X. Hu, K. Liu, and X. Xia, "Understanding practitioners' expectations on clear code review comments," *Proceedings of the ACM on Software Engineering*, no. ISSTA, pp. 1257–1279, 2025.

[47] J. Chen, Z. Li, X. Hu, and X. Xia, "Nlperturbator: Studying the robustness of code llms to natural language variations," *ACM Transactions on Software Engineering and Methodology*, 2024.

[48] S. Yun, S. Lin, X. Gu, and B. Shen, "Project-specific code summarization with in-context learning," 2024.

[49] V. Lomshakov, A. Podivilov, S. Savin, O. Baryshnikov, A. Lisevych, and S. Nikolenko, "ProConSuL: Project context for code summarization with LLMs," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2024.

[50] A. Khan, G. Liu, and X. Gao, "Code vulnerability repair with large language model using context-aware prompt tuning," 2024.

[51] C.-Y. Su, A. Bansal, Y. Huang, T. J.-J. Li, and C. McMillan, "Context-aware code summary generation," 2024.

[52] X. Zhang, Y. Zhou, G. Yang, T. Han, and T. Chen, "Context-aware code generation with synchronous bidirectional decoder," 2024.

[53] Z. Li, T.-H. Chen, J. Yang, and W. Shang, "Studying duplicate logging statements and their relationships with code clones," *IEEE Transactions on Software Engineering*, 2022.

[54] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large Language Models are Zero-Shot Reasoners," 2022.

[55] Y. Hu, K. Luo, and Y. Feng, "ELLA: Empowering LLMs for Interpretable, Accurate and Informative Legal Advice," 2024.

[56] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22, 2022, pp. 24 824–24 837.

[57] Qwen, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Tang, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu, "Qwen2.5 Technical Report," 2025.

[58] DeepSeek-AI, D. Guo, D. Yang, and et al., "DeepSeek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning."

[59] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, and et al., "The llama 3 herd of models."

[60] OpenAI, "GPT-4," https://openai.com/index/gpt-4/, 2023.

[61] Understand, "Understand: The Software Developer's Multi-Tool," https://scitools.com/, 2024.

[62] "HBASE-24367," https://issues.apache.org/jira/browse/HBASE-24367.

[63] "AutoMQ-Commit-e6f8ca8," https://github.com/AutoMQ/automq/commit/e6f8ca80cd1ae301caba7b9c3f0b1f2d90a51409.

[64] "HDFS-15045," https://issues.apache.org/jira/browse/HDFS-15045.

[65] "YARN-6951," https://issues.apache.org/jira/browse/YARN-6951.

[66] "HIVE-20796," https://issues.apache.org/jira/browse/HIVE-20796.

[67] "HDFS-17310," https://issues.apache.org/jira/browse/HDFS-17310.

[68] "HIVE-25794," https://issues.apache.org/jira/browse/HIVE-25794.

[69] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, 1960.

[70] "YARN-8907," https://issues.apache.org/jira/browse/YARN-8907.

[71] "HDFS-15197," https://issues.apache.org/jira/browse/HDFS-15197.

[72] "HDFS-14407," https://issues.apache.org/jira/browse/HDFS-14407.

[73] "Camel-Commit-0c1a589," https://github.com/apache/camel/commit/0c1a5897fe36866f56549511b73ad8db7c8fe32b.