PDLogger: Automated Logging Framework for Practical Software Development

Anonymous Author(s)

Abstract

Logs, critical for system monitoring, troubleshooting, and auditing, are widely used in security analysis and business analytics, making them an indispensable component of modern software systems. Although multiple automated log generation techniques have been proposed, their practical adoption in software development remains challenging. State-of-the-art solutions lack sufficient contextual awareness and are designed to insert only one log statement in a method, significantly limiting their real-world applicability. Specifically, we identify three inherent limitations with existing schemes: lack of multi-log generation capability, insufficient semantic dependency context and limited logging variable scope. To this end, we propose PDLogger, the first high-accuracy log statement generation approach that is applicable to real-world development scenarios. First, PDLogger leverages static analysis to extract semantic dependency information, enabling the generation of more closely reflect the original intent of the ground truth and less redundant log statements. Second, it introduces a block-type-based structured prompting strategy to query the model in a more targeted manner, which significantly reduces the false positive rate of log position prediction in the multi-log prediction setting. Finally, performing function-aware extension by incorporating function information into the variable list. The evaluation results demonstrate that PDLogger outperforms the state-of-the-art approach by 139.00% in log position precision, 69.20% in F1 score, 82.30% in logging level accuracy, 131.80% in variable precision, and 65.70% in BERTScore for log message generation. Furthermore, PDLogger consistently achieves strong performance across a variety of large language models, indicating its robust generalizability.

1 Introduction

With the growing complexity and scale of software systems, logging has evolved from optional to a widely recognized and indispensable mechanism for ensuring software reliability and integrity. However, unreasonable logging practices not only compromise the readability of logs but also influence system performance[60]. To address this issue, numerous automated logging schemes have been proposed to help developers insert log statements more effectively. A typical logging statement consists of four key components: position, level, message, and variable. Most prior techniques target only individual components (e.g., predicting log positions [18, 29, 51, 59, 60], levels [23, 30, 33], or messages [10, 15, 55]) and are not able to generate complete logs, making them unsuitable for real-world development.

Recent advances in large language models (LLMs) have attracted much attention in the field of automated logging. LLMs learn common logging patterns from massive code corpora, understand both natural language and code, and can infer a developer's intent from surrounding context. This makes them potentially capable of generating concise, informative messages that follow project-specific conventions, incorporate relevant variables, and assign appropriate severity levels. As a result, LANCE [37], the first end-to-end log generation approach, is built on an LLM named T5 [42] and is capable of inserting complete log statements into given code snippets. However, it relies solely on intra-method information, overlooking contextual information from both callers and callees. To address it, SCLogger [26] enhances log quality through static scope expansion, style adaptation, and context-aware prompt construction. Nevertheless, SCLogger fails to incorporate semantic dependency information, often generating superficial error descriptions that lack insight into the root cause. UniLog [48] leverages LLM with line-position-sensitive in-context learning prompts to predict a complete log without fine-tuning; however, it likewise fails to capture deeper semantic dependencies.

Limitations. To sum up, we identify three key limitations in existing state-of-the-art approaches, which will be further illustrated with real-world examples in Section 2.

First, none of the existing approaches, by design, handle multiple log generation scenarios within a method. In modern software systems, it is extremely common for a single method to contain multiple logging statements, which are essential for enhancing system observability and facilitating fault diagnosis. Our analysis of the LogBench-O benchmark [25] reveals 6,849 log statements across 3,870 methods, resulting in an average of 1.77 log statements per method. However, state-of-the-art automated logging techniques [26, 36, 48] are designed and assessed under the assumption that each method requires at most one log statement. This limitation severely hinders their applicability in real-world development scenarios. Therefore, it is imperative to develop techniques that support multi-log generation within a single method.

Second, state-of-the-art logging techniques lack sufficient semantic dependency context. Among the existing methods, only SCLogger [26] attempts to incorporate contextual information; however, its context is limited to a small subset of randomly selected callers and callees (within two hops). As a result, the retrieved context is often incomplete and semantically irrelevant or misleading, rendering low-quality log generation.

Third, all existing techniques suffer from a limited scope of logging variables. Recent studies, such as SCLogger, incorporate the member functions and attributes of the class that contains the given method into its log variable list. However, it overlooks the case where non-member functions or function expressions are used directly as logging variables. Our empirical study on 100 randomly sampled logs from Apache Hadoop [5] reveals that 13% of the logs utilize non-member functions or function expressions as variables. This observation highlights that the limited scope of log variable lists will likely result in the omission of critical variables, thereby reducing the expressiveness and diagnostic value of the logs.

Our Approach. To tackle these issues, we propose PDLogger, the first automated log generation technique that is designed to

be practical in real-world software development. It generates logging statements through three major phases, namely log position prediction, log generation, and refinement. In the log position prediction phase, we first extract the boundaries of code blocks within a given method and annotate their start and end positions to facilitate better recognition by LLM. Based on our empirical analysis of log placement principles, we design customized prompts tailored to different types of blocks. Finally, we query the LLM once for each prompt. This helps address the inability to perform multi-log prediction and mitigates the excessive false positives that arise in such scenarios. Then PDLogger leverages the LLM to predict log positions for different block types independently. The predicted positions may still contain false positives, which will be addressed in the deduplication step. In the log generation phase, based on the positions predicted in the first stage, our system applies static slicing techniques to generate backward slices, which can capture the precise semantic dependencies of the code around the logging positions.

Then, in the Variable Analysis and Function-aware Extension step, in addition to following SCLogger's approach to variable scope extension, we further expand the information of functions that can be used to generate log variables, thereby addressing the issue where log variables cannot be functions or function expressions. These dependencies and function information are then integrated into updated prompts, which are passed to LLM to generate complete log statements. After that, PDLogger using the log level refinement strategy to increase the accuracy of the level. In the deduplication step, PDLogger further identifies similar logs within specific log contexts and removes redundant ones. This reduces the number of false-positive logs in the program, thereby mitigating the risk of developers being misled and improving the utility of log information.

Unlike previous approaches[26, 37], we acquire a dataset that contains no logging statements. We constructed such a dataset using open-source Java projects from two different domains and conducted a comprehensive evaluation on it. The results show that PDLogger achieves the best performance across all metrics, except for recall in log position prediction. Specifically, PDLogger outperforms the baseline approach by 139.00% in log position precision, 69.20% in F1 score, 82.30% in logging level accuracy, 131.80% in variable precision, and 65.70% in BERTScore [57] for log message generation. Moreover, PDLogger consistently achieves strong performance across a variety of large language models, thus demonstrating its generalizability.

The contributions of this paper are summarized as follows:

- To the best of our knowledge, we present the first practically deployable log-statement generator. By embedding semantic-dependency context, PDLogger overcomes prior work that captures only surface-level signals.
- We design a novel prompt structure to improve prediction accuracy in multiple log generation scenarios. This structure can be integrated into other generative log generation methods and generalized to future LLM improvements.
- We investigate existing logs in real-world projects and summarize a deletion priority rule to reduce false positives. This rule is generalizable and can be applied to false-positive deduplication tasks in various log generation methods.
- We conduct a comprehensive evaluation of PDLogger on a dataset generated from public projects. The results show that PDLogger outperforms existing approaches and is adaptable to different LLMs.
- The source code of PDLogger is publicly available at https://github.com/qlbdsc/PDLogger to benefit both developers and researchers.

2 Motivation

In this section, we highlight the motivation of our research by listing three major limitations of the existing logging techniques.

2.1 Lack of Multi-log Generation Capability

Empirical evidence indicates that developers typically embed multiple log statements within a single method. However, state-of-the-art automated logging techniques [26, 36, 48] are designed and assessed under the assumption that each method requires at most one log statement. Although these techniques can, in principle, be extended to predicate multiple logs, such naive adaptations will likely introduce an excessive number of false positives.

```
private synchronized void pushToZK(byte[] newSecret, byte[] currentSecret,byte[] previousSecret) {
① LOG.info("Preparing to push data to ZooKeeper at path: {}", path);
② LOG.debug("Generated ZooKeeper data payload of length: {}", (newSecret != null &&
  currentSecret != null && previousSecret != null)? generateZKData(newSecret, currentSecret,
  previousSecret).length : "null input");
③ LOG.trace("Generated ZooKeeper data payload of length: {}", bytes != null ? bytes.length : 0);
  byte[] bytes = generateZKData(newSecret, currentSecret, previousSecret);
  try {
④   LOG.trace("Attempting to set ZooKeeper data with version: {}", zkVersion);
    client.setData().withVersion(zkVersion).forPath(path, bytes);
⑤   LOG.info("Successfully pushed data to ZooKeeper path: {}", path);
  } catch (KeeperException.BadVersionException bve) {
⑥   LOG.warn("Failed to push data to ZK due to version conflict (expectedversion: {})", zkVersion);
  } catch (Exception ex) {
⑦   LOG.error("Unexpected error while pushing data to ZK at path {}", path, ex);
  }
}
```

⑥, ⑦ are the ground truths ⑥ predicted by SCLogger ① - ⑦ predicted by SCLogger

Figure 1: Multi-Log Generation Issues

We illustrate this limitation with an example method from the ActiveMQ [1] project, shown in Figure 1. The function contains two logging statements written by the developer, which we treat as the ground truth and mark in red (6 and 7). The vanilla state-of-the-art technique, SCLogger, can only generate one log for the method (6), and thus fails to support multi-log generation. To accommodate multiple log cases, we adapt SCLogger [26] by revising its LLM prompt to elicit predictions of several log statements, resulting in a new variant that we designate as *Multi-SCLogger*. It generates seven logs — two true positives and five false positives (1-7), yielding 100% recall but only 28.6% precision. The results show that simply modifying existing techniques to support multi-log generation is ineffective in practice, as it leads to performance degradation and significant data redundancy. Consequently, we need a system that is specifically designed to handle multiple-log scenarios.

2.2 Insufficient Semantic Dependency Context

In modern software systems, a method's execution often depends on interactions with other methods [35]. Although approaches like

SCLogger [26] extend context from the method level by including randomly selected caller and callee methods within two hops, such random selection will likely fail to capture complete relevant information for log generation, resulting in inaccurate logs. To improve, it is crucial to analyze the precise contextual information by incorporating control and data dependencies, which reveal execution prerequisites and upstream influences vital for understanding the semantics and execution conditions of the target line [9].

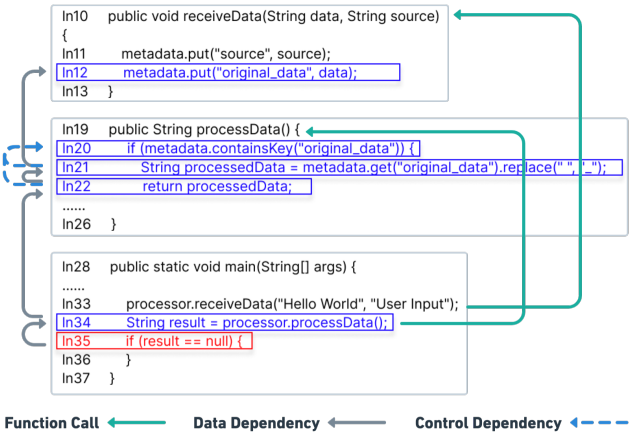


Figure 2: Semantic Dependency Tracking. The red-highlighted line marks the starting statement of the backward slice. Blue-highlighted lines denote statements reached through data and control dependency tracing.

Figure 2 highlights the importance of extracting statement-level semantic dependencies. When inserting a log after `if (result == null) (ln.35)`, SCLogger's random sampling may only capture the `receive_data` function (ln.10), offering little insight to infer the actual cause of the condition. In contrast, tracing control and data dependencies reveals that the condition at ln.35 stems from ln.34, which depends on `processor.process_data()` at ln.22, itself relying on earlier lines. Therefore, the root cause of the condition `result == null` is the absence of the value associated with the `original_data` key in `metadata`.

Without the dependency information of the target line, it is nearly impossible to infer that the condition holds due to the missing `original_data`. Moreover, if `DataProcessor()` is randomly selected during context expansion, the extracted information may be irrelevant to the log, reducing log utility. More critically, if a standard library function is chosen, its function body is typically inaccessible, offering no meaningful context. Therefore, to generate high-quality and contextually relevant log statements, it is necessary to obtain more precise contextual information from the statement level by leveraging semantic dependencies.

2.3 Limited Log Variable Scope

Log statements often contain variables to enhance log readability, improve debugging effectiveness, and increase the utility of automated analysis [41, 53]. For instance, `"LOG.info("Starting with address: ",getDatanodeAddress());"` includes a function call to retrieve and output the actual runtime address of the current node. The return value of this function enables developers to quickly identify key information, such as which specific node initiates the startup process. Prior studies [11, 12, 52] have shown that, compared to purely textual logs, developers prefer log statements that contain program state variables, as they facilitate issue localization and comprehension of system behavior. However, when considering potential logging variables, existing techniques (e.g., SCLogger [26]) focus only on member functions and local variables, but overlook cases where non-member functions and function expressions should also be considered.

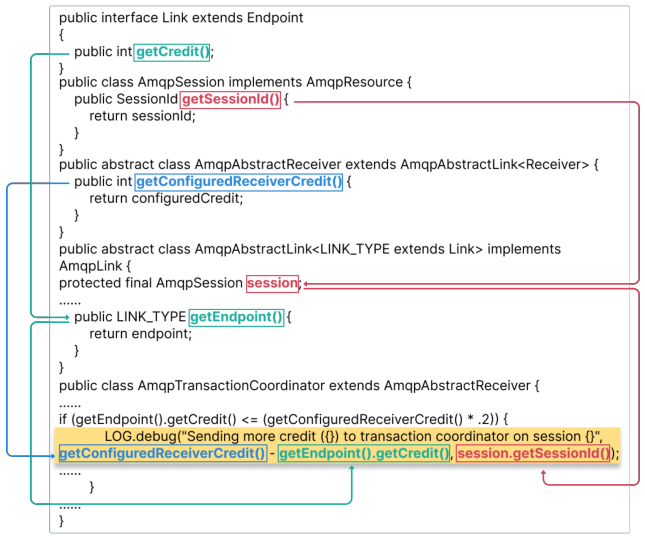


Figure 3: Log Variable Origins

Figure 3 illustrates an example from the *ActiveMQ* [1]project, where the log statement includes both a function call and arithmetic operations involving multiple functions as its variables. The log is intended to record the status of sending additional credit to a transaction coordinator within a particular session. To achieve this, the log message integrates information derived from non-member functions defined outside the current class. These pieces of information are essential for developers to understand the system's behavior regarding resource allocation and transaction coordination during message processing. Although such omissions may not significantly impact traditional variable-level metrics in log evaluation, they can substantially degrade the accuracy of log-based program analysis and anomaly detection. This limitation can be addressed by extending non-member functions extracted from semantic dependency contextual information into the variable list during log generation.

3 System Design and Implementation

3.1 Overview

To address the aforementioned limitations, we propose PDLogger, an automated LLM-based log generation framework that can be practically applicable to real-world software development.

Figure 4 illustrates the overview of our system, which takes a project codebase as input and outputs an enhanced project with log statements generated. The entire system consists of three major phases: log position prediction, log generation, and refinement.

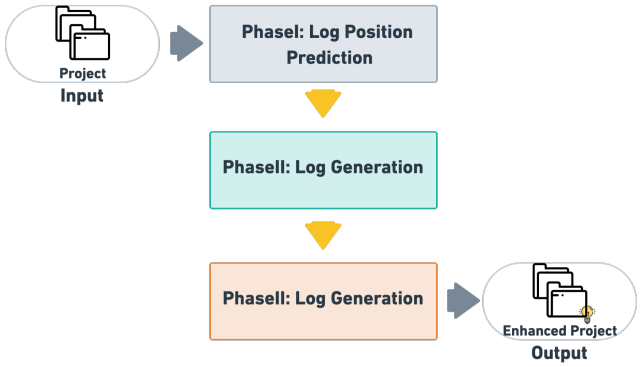


Figure 4: Overview of PDLogger

Given a project codebase, we generate logs for each method by following these three phases. Specifically, in the log position prediction phase, we identify the start and end positions of different code blocks, following a prior work [29] that identifies where log statements are most likely to exist. Then, we annotate the method with the block information to form a block-type-based structured prompt, which is fed into the LLM to predict log positions.

Then, in the log generation phase, we use the predicted log positions to perform backward slicing, incorporating both data and control dependencies information into the prompt. We then extract the variable list and enrich it with detailed function-level information. This enriched semantic context is combined with the context extracted in the previous phase, along with CoT [46] reasoning, to form a new prompt. This prompt is fed into the LLM to obtain the final logging statements.

Since the log level reflects the importance of a logging statement [14, 38], in the refinement phase, we subsequently conduct a level refinement to improve the accuracy of log level prediction. Finally, to reduce false positives, PDLogger introduces a unique deduplication process, producing a filtered set of logging statements. The final output is the final version of a project, containing the generated log statements with redundant entries removed.

3.2 Log Position Prediction

To extract a precise semantic dependency context, we need to find an accurate starting point. Therefore, the first step in PDLogger is to predict log positions. To do so, for the given project, we perform static analysis for each method and identify all code blocks. Then we annotate the method with the

Block-type-based Structured Prompt Construction. The block-type-based structured prompt construction step is designed to generate customized prompts for each type of code blocks of the target method. The primary goal of this step is to ensure high precision and a low false positive rate when predicting multiple log positions. Accurate position prediction is very important, because if the position prediction is wrong, the prediction of message, level, and variable will be meaningless. As shown in Figure 5, , PDLogger categorizes the blocks within a method into four types: branch blocks, try-catch blocks, loop blocks, and method definition blocks[29]. For each type, PDLogger counts the number of corresponding blocks, assigns unique identifiers, and marks the start and end lines of each block. Then a corresponding prompt is constructed for each block type and feeds into the LLM, with the number of queries corresponding to the number of such blocks with the target method.

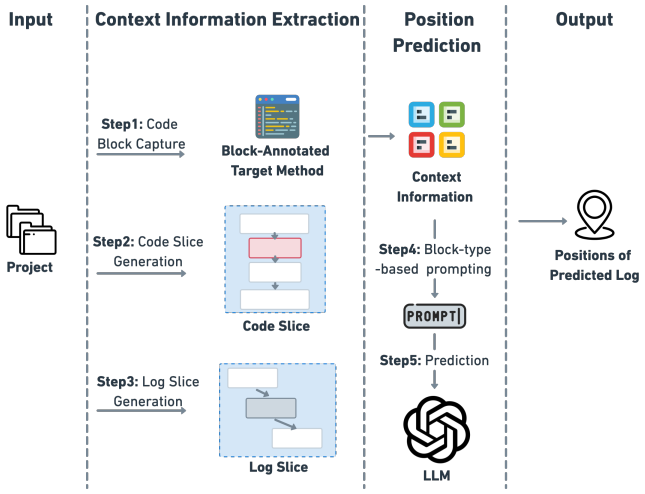


Figure 5: The Overview Workflow of Phase I

To enhance the precision of position in multi-log prediction while reducing the false positive rate of position, each prompt for different type of block incorporates a set of heuristic rules. These rules were derived through our in-depth review of prior studies [11, 17, 28, 31, 60] and our own empirical analysis of log placement principles, particularly the structural and semantic cues that govern when and where logs should be generated. Importantly, these block-specific prompting strategies can be readily generalized to other generative log synthesis frameworks. The four prompt templates have been made publicly available on our GitHub repository.

Position Prediction. To extract backward slices with accurate semantic dependencies, this step is designed to predict log positions. Following the design principles of SCLogger[26], we begin by extracting inter-method information, including both the code slice and the log slice. Notably, unlike SCLogger, we omit the variable list in this step, as it does not contribute to the accuracy of log position prediction and may even introduce noise.

As shown in Figure 5, the extracted inter-method information is then combined with the block-type-based prompt, and enhanced using a set of the heuristic rules. These components together form the prompt, which is subsequently fed into the LLM to obtain log position predictions.

3.3 Log Generation

Semantic-dependency Expansion. We define semantic dependency information as the combination of control dependencies and data dependencies. To extract such information, we generate backward slices based on the log positions predicted in the previous step. These slices are extracted using Joern[49], which integrates

syntactic, control-flow, and data-flow dependencies. A key advantage of Joern is its ability to directly analyze source code without packaging into JAR files.

As shown in Figure 6, based on the predicted log positions within the target method, we first map each position to its corresponding line number in the original Java source file and extract the state-

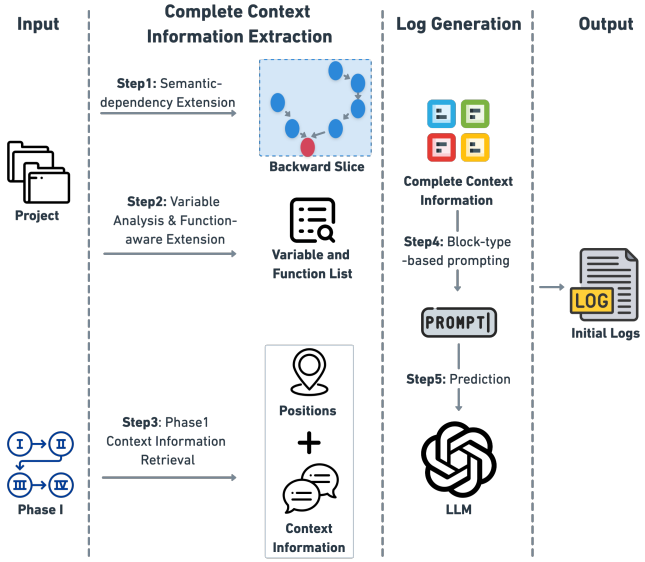


Figure 6: The Overview Workflow of Phase II

ments corresponding to the line numbers. However, a key challenge arises in practice: for certain statements, the generated backward slice is empty. To address this, PDLogger iteratively selects preceding lines until reaching beyond the start of the current block. Specifically, for branch blocks, if the predicted position falls within an else block and no suitable line is found before exceeding the start of the else block, we instead select the corresponding if condition statement as the starting point of the backward slice. This is based on the observation that the if condition and the else block typically express opposite branches of the same logical judgment, often sharing the same set of conditional parameters[45]. We then perform

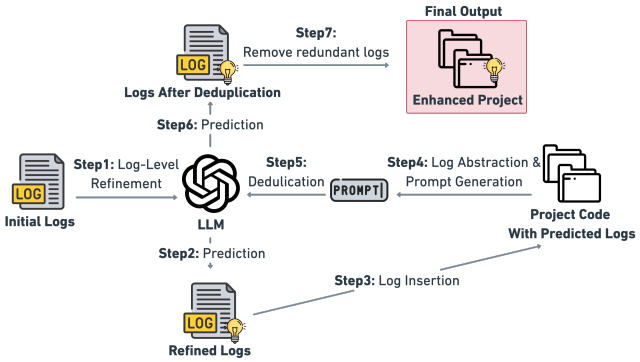


Figure 7: The Overview Workflow of Phase III

static analysis on the selected line to obtain the corresponding backward slice. To accommodate the input length limitations of LLMs, we constrain the backward slice to a maximum of seven hops. We define each "hop" as a traversal across a semantic dependency or control dependency edge.

Variable Analysis and Function-aware Extension. As highlighted in the motivation study, merely expanding the scope of accessible variables is insufficient to fully meet the requirements of logging practices, as certain log statements rely on the return values of functions as logging variables. Therefore, building upon SCLogger's approach[26] to expanding the variable scope, we further summarize the information of functions that are likely to appear as logging variables.

We first adopt the variable sets v defined in SCLogger, where $v \in V_p \cup V_m \cup V_c \cup V_s \cup V_i$[54]. Here, V_p denotes the set of parameters, V_m denotes the set of local variables, V_c denotes the set of class member variables, V_s denotes the set of static variables, and V_i denotes the set of inherited variables. Within the context of the target method, we define functions sets $f \in F_m \cup V_i \cup V_d \cup V_l \cup V_s$ as follows:

- F_m: The set of member methods defined in the current class.
- F_i: The set of methods inherited from the parent class.
- F_d: The set of default methods from implemented interfaces.
- F_l: The set of lambda expressions or function variables declared within the current method.
- F_s: The set of statically imported methods from the parent or current class.

During the prediction of logging statements, the model should attend not only to the variable sets, but also to function variables f that belong to one or more of the above-defined function sets. Consequently, the context provided for logging variable selection should encompass both variables and function-derived values, collectively denoted as $f \cup v$, which are treated as candidate logging variables and incorporated into the available variable list.

3.4 Refinement

Log Level Refinement. As shown in 7, we refine the initially predicted log levels before inserting the predicted log statements into the project. Log levels are essential for distinguishing the verbosity and severity of log messages, enabling developers to control log output across different environments, facilitate efficient debugging, and support runtime monitoring and alerting. Therefore, ensuring the accuracy of log level prediction is critical. Based on our investigation and [33, 50], we identify five key factors that contribute to log level determination: (1) the content of the log message, (2) the method in which the log resides, (3) the semantic explanation of the log message, (4) the function of the block containing the target log, and (5) the total number of lines in that block. According to this rule, we extract these five types of information along with the originally predicted log level and provide them as input to a large language model (LLM). The LLM is then tasked with deciding whether the original log level requires adjustment; if so, it outputs the adjusted log level, and if not, it recommends retaining the original one. Experimental results across different backbone models demonstrate that our proposed log-level refinement approach exhibits strong

generalization ability. The full prompt used in this process has been made publicly available in our GitHub repository.

Deduplication. The deduplication step is designed to reduce the false-positive rate of the predicted log statements, thereby mitigating the performance degradation that can arise from excessive logging. It proceeds in two steps. First, the predicted logs are inserted into the project; second, we construct a *Code Context Bundle*—defined as the target method together with the caller and callee methods within one hop—and eliminate redundant logs by examining whether the bundle already contains outputs that are semantically equivalent to the target log.

Log Insertion. To avoid the line-number shifts that would occur if logs were inserted in an arbitrary order, we first group the predicted logs by their corresponding Java file. For each file, we then insert all logs in descending order of line number, from the largest to the smallest. Note that we insert the predicted log statement at the line immediately following the predicted position, as the prompt explicitly instructs the model to insert the log one line after the predicted location. And some developers may split a single statement across multiple lines for readability or formatting purposes. To address this issue, we analyze the abstract syntax tree (AST) of the Java source code to accurately identify the termination point of the complete conditional expression, thereby enabling precise insertion of log statements. Finally we get a project augmented with the predicted logs.

Deduplication. To maximize coverage of potential semantic redundancies, we identify five situations in which a predicted log can be considered redundant; each is addressed below. (1) Overlap with throw messages. A log is removed if it conveys the same message as a throw statement, provided the exception is caught and its message printed, or it propagates uncaught to main(). For string-based exceptions, inter-procedural data-flow analysis traces variable origins to compare messages more precisely. (2) Contradictory Logs in an if-else Block. For logs in opposite branches of an `if-else` block, we retain the one with a higher level. If levels match, we keep the else-branch log, as execution of the `else` branch often signals an anomalous path of control. (3)Start–End Log Pairs. We identify pairs of logs that are logically sequential, where the first log denotes the start of a process (`start_log`) and the second denotes its completion (`end_log`). If a (`start_log`) is post-dominated by its (`end_log`), the former is removed. This rule excludes trace-level logs, which capture fine-grained execution. (4) Duplicate Semantics with Shared Variables. For semantically identical logs sharing a variable `var_common`, static analysis checks for reassignment between them. If reassigned, the earlier log is removed; otherwise, the later log is removed. Without shared variables, we retain the log nearer to meaningful code (e.g., method calls, exceptions). Applying these five language-agnostic rules yields the final Augmented Project, and they can be adapted to any Java project because they rely solely on static analysis and semantic comparison.

4 Evaluation

In this section, we conduct a comprehensive evaluation on the effectiveness of PDLogger. Specifically, we aim to answer four essential research questions.

- **RQ1:** How effective is PDLogger in the single-log generation scenario?
- **RQ2:** How effective is PDLogger in the multi-log generation scenario?
- **RQ3:** What is the impact of different components of PDLogger?
- **RQ4:** How generalizable is PDLogger on different backbone LLMs?

4.1 Experiment Setup

Dataset Selection. To evaluate PDLogger, we randomly select 3,113 log statements from two popular projects, Apache Hadoop 3.4.1 [5] and Apache ActiveMQ 5.18.7 [1]. These logs are distributed across 914 methods, with up to 13 log statements per method, representing real-world software development scenarios. For each sampled method, we extract the identifiers of its package, class, and method names, and locate all log statements that invoke two mainstream logging utilities, Log4j [4] and SLF4J [16]. We record each log statement, along with its corresponding line number in the source code, and use this information as the ground truth.

Baseline Techniques. In a single-log prediction setting, where exactly one log is randomly removed from each target method, the task is to recover this single log. We adopt SCLogger [26], the first contextualized logging-statement generation approach that exploits inter-method static contexts, as our primary baseline. We also include the first deep-learning-based complete log generation work, LANCE [37], and its successor, LANCE 2.0 [36], as baselines. Approaches that target only specific sub-components of log generation are not considered.

In the multi-log prediction setting, we adapt SCLogger by revising its LLM prompt to elicit predictions of several log statements, resulting in a new variant that we designate as *Multi-SCLogger* and serves as the baseline for the multi-log setting. To demonstrate the generalizability of our approach, we further conduct experiments on three mainstream LLMs: DeepSeek-V3 [7], LLaMA3-70B [2], and OpenAI o3-mini [39].

Log Position Evaluation Metrics. For single-log generation setting, consistent with [37], we use Position Accuracy (PA) to assess log-position prediction. A prediction is counted as correct (PA = 1) if the predicted line number deviates from the true line number by at most one and both lines are located within the same code block; otherwise, PA = 0. For the multi-log setting, we use precision, recall, and F1 score to evaluate the accuracy and to measure the proportion of false positives and true positives.

Log Level Evaluation Metrics. We employ L-ACC (level accuracy) and Average Ordered Distance (AOD) to measure log-level prediction, as in [25, 30, 33]. L-ACC represents the proportion of correctly predicted levels. AOD captures the ordinal distance between levels, acknowledging that log levels are not independent categories (e.g., *error* is closer to *warn* than to *trace*).

Log Variables Evaluation Metrics. We adopt precision, recall, and F1-score to evaluate the set of variables referenced in the generated log statements. For each generated log statement, we denote the set of variables predicted by the model as S_p, and the set of variables in the ground-truth log as S_g. The evaluation metrics are defined as

follows: Precision = $\frac{|S_p \cap S_g|}{|S_p|}$, Recall = $\frac{|S_p \cap S_g|}{|S_g|}$, F1 = $\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}}$. It is important to note that if a predicted variable has the same name as a ground-truth variable but differs in the usage of its member function, it is still considered an incorrect prediction.

Log Messages Evaluation Metrics. Following prior studies, we employ BLEU–K ($K = \{1, 4\}$) [40] and ROUGE–K ($K = \{1, L\}$) [32] to quantify surface-level similarity, and additionally adopt BERTScore to capture semantic similarity, because BLEU and ROUGE operate solely on K-gram overlap. BERTScore ranges from 0 to 1, with higher values indicating better quality.

Experimental Environment. The static analysis component of PDLogger is implemented with 4,293 lines of Java and shell code, combining Joern [49] and Eclipse JDT Core [6] to enable comprehensive analysis of Java source code. All experiments for PDLogger and the baselines are executed on a Linux machine running Ubuntu 22.04.5 LTS, equipped with an AMD EPYC 9354 32-Core Processor @ 3.8 GHz, 2 NVIDIA L40S GPUs (each with 46GB of memory), and 256 GB of RAM. We employ the public API to access O3-mini-20240131 and Deepseek-V3, while LLaMA3-70B-Chat-HF is executed locally. To ensure deterministic outputs and facilitate stability evaluation in log generation, we set the temperature to 0.

4.2 RQ1: Single-Log Generation Scenario

We first evaluate PDLogger's performance on a dataset constructed by randomly removing only one log statement per method, and compare it with several state-of-the-art baseline techniques. The evaluation results are presented in Table 1, with the best results for each metric highlighted in bold.

For log position, PDLogger outperforms all baselines. Specifically, it improves upon the best-performing baseline, SCLogger [26], by 29.5% from 0.417 to 0.54. This confirms that our proposed block-type-based prompting strategy is highly effective. In terms of log level prediction, PDLogger achieves the highest AOD score. Although its accuracy is slightly lower (0.609 vs. 0.613) than that of LANCE 2.0, it outperforms all other baselines. For log variable prediction, PDLogger surpasses all baselines in accuracy, recall, and F1 score, demonstrating that the function-aware extension indeed improves the accuracy of log variable prediction. Regarding the log message, PDLogger shows significant improvements over all baseline methods, surpassing the best baseline by 30.80% (0.467 vs. 0.37) in BERTScore. These results suggest that by extracting semantic dependency information, our system generates log messages that better match the project style and more accurately convey developer intent, making it more suitable for real-world development.

> Answer to RQ1: PDLogger can significantly outperform state-of-the-art techniques, demonstrating strong effectiveness in the single-log generation setting.

Case Study. Figure 8 presents a case study, with the developer-written log highlighted, to illustrate the effectiveness of PDLogger in single-log insertion tasks. All four tools correctly identify the appropriate log position. However, only PDLogger accurately captures the semantic dependencies, enabling it to generate a log message that explains the root cause of `result == null`. In contrast, SCLogger and LANCE 2.0 merely record the occurrence of the condition, while the message produced by LANCE lacks any diagnostic value. Notably, the original developer-written log also attempts to convey the root cause of `result == null`, further demonstrating that the log generated by PDLogger is more consistent with developer intent and can meet practical diagnostic requirements.

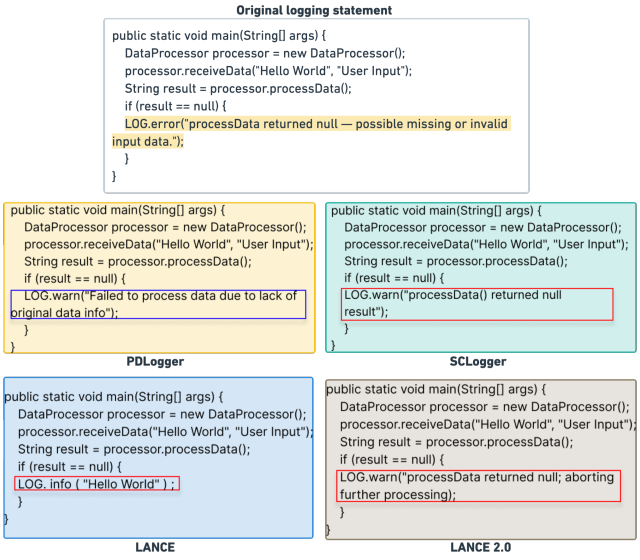


Figure 8: A Case Study in the Single-log Generation Task

4.3 RQ2: Multi-Log Generation Scenario

To evaluate PDLogger's practicality in real-world development, we assess its performance on a dataset where all log statements are removed from methods. The modified SCLogger (*Multi-SCLogger*) serves as the primary baseline.

As depicted in Table 2, PDLogger can vastly outperform *Multi-SCLogger* in all four log component generations. In particular, for log position prediction, we evaluate whether the predicted logs are neither excessive (causing system overhead) nor insufficient (providing too little information). Using precision, recall, and F1 score as metrics, we find that PDLogger surpasses SCLogger in both precision and F1 score. Specifically, PDLogger improves F1 score by 69.21% (0.621 vs. 0.367). This is primarily because *Multi-SCLogger* generates a substantial number of false positives, whereas PDLogger significantly reduces false positives by employing block-type-based prompting and refinement.

In terms of log level prediction, PDLogger surpasses SCLogger in both level accuracy (L-ACC) and AOD, with improvements of 82.2% and 26.5% respectively. These results indicate that semantic dependency information can also assist PDLogger in assessing the importance of log statements and predicting appropriate log levels. For log variable prediction, PDLogger achieves remarkable gains with an F1 score improved by a whopping 135% (0.657 vs. 0.28). Regarding log message generation, PDLogger significantly

Table 1: Results in the Single-Log Prediction Setting.

Model	Position	Logging Levels		Logging Variables			Logging Texts				
	PA	L- ACC	AOD	Precision	Recall	F1	BLEU-1	BLEU-4	ROUGE-1	ROUGE-L	BERTScore
SCLogger	0.417	0.573	0.807	0.535	0.613	0.571	0.474	0.201	0.354	0.34	0.341
LANCE	0.341	0.59	0.747	0.286	0.302	0.294	0.514	0.184	0.275	0.273	0.357
LANCE2.0	0.383	**0.613**	0.775	0.311	0.397	0.349	**0.544**	0.182	0.315	0.297	0.37
PDLogger	**0.54**	0.609	**0.815**	**0.589**	**0.638**	**0.607**	0.514	**0.235**	**0.442**	**0.42**	**0.467**

Table 2: Results in the Multiple-Log Prediction Setting.

Model	Position			Logging Levels		Logging Variables			Logging Texts				
	Precision	Recall	F1	L- ACC	AOD	Precision	Recall	F1	BLEU-1	BLEU-4	ROUGE-1	ROUGE-L	BERTScore
Multi-SCLogger	0.24	**0.784**	0.367	0.446	0.739	0.283	0.315	0.28	0.438	0.174	0.384	0.375	0.324
PDLogger	**0.575**	0.674	**0.621**	**0.813**	**0.935**	**0.656**	**0.657**	**0.657**	**0.57**	**0.315**	**0.487**	**0.469**	**0.537**

outperforms SCLogger, with a BLEU-4 score of 0.314 (45.4% higher) and a BERTScore of 0.544 (52.1% higher). These gains stem from PDLogger 's effective capture of semantic dependencies, allowing it to better capture developer intent and produce semantically aligned log messages, unlike SCLogger's heuristics-based random sampling of callers and callees.

> Answer to RQ2: By constructing block-type-based structured prompts and capturing semantic dependencies, PDLogger significantly outperforms *Multi-SCLogger* across all four dimensions: position, level, variable, and message, in a more practical multi-log generation scanrio.

Case Study. Figure 9 presents a case study illustrating the effectiveness of PDLogger in the multi-log generation scenario. The ground truths have been highlighted. For log position, both tools achieve 100% recall. However, PDLogger 's block-type-based prompting yields only one false positive (66.6% precision), compared to five from *Multi-SCLogger* (28.6% precision). Regarding the log message, for the first log, *Multi-SCLogger* generates a superficial message that fails to explain the root cause of the push data failure. In contrast, PDLogger captures the semantic dependencies and attributes the issue to a prior operation by another server, closely matching the ground truth. Moreover, PDLogger correctly predicts the log level and variable, while *Multi-SCLogger* does not. For the second log, while both tools generate accurate messages and levels, *Multi-SCLogger* mispredicts one variable (*path*).

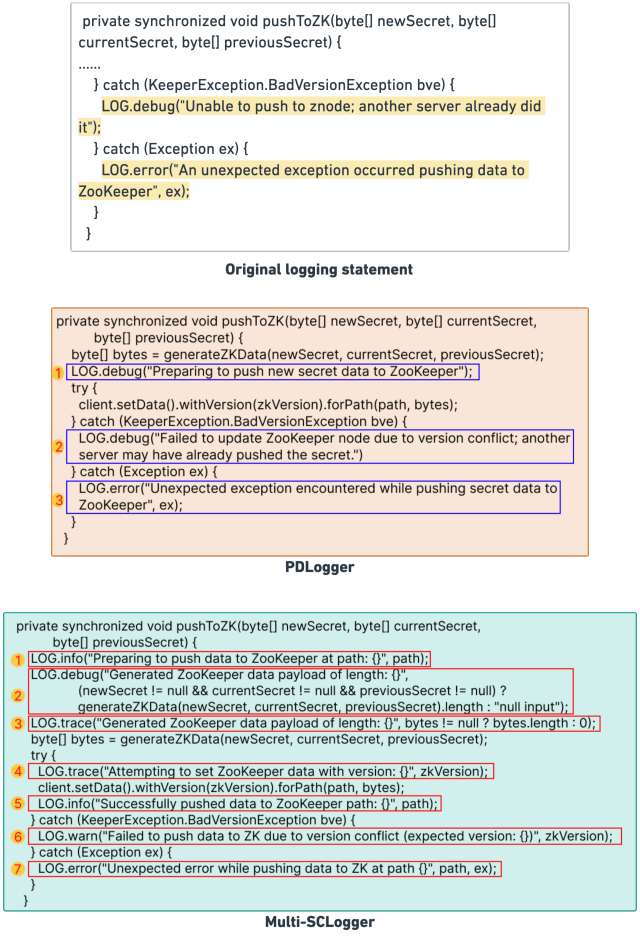


Figure 9: A Case Study in the Multi-Log Generation Scenario.

4.4 RQ3: Ablation Study

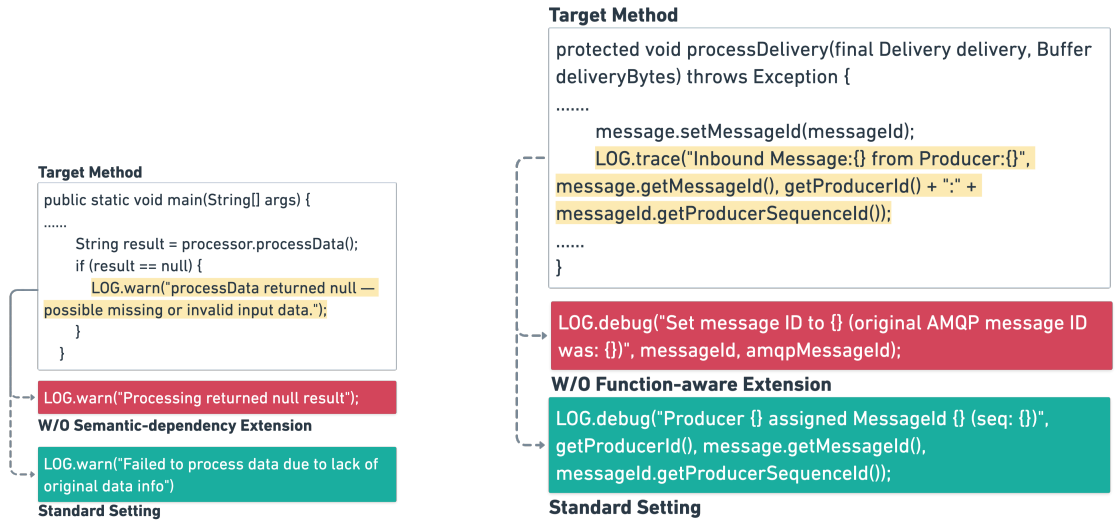
We further conduct an ablation study to demonstrate the effectiveness of five major design choices in PDLogger: block-type prompts, semantic-dependency, function-aware extensions, deduplication, and level refinement. We then create five variants, each with one design choice removed[1], and compare them against the full-fledged PDLogger.

As shown in Table 3, removing block-type-based prompting increases the number of false positive positions nearly fourfold. Leading to a sharp decline in precision (-58.3%) and level accuracy (-29.3%), primarily due to inaccurate slicing entry points, which result in uninformative or even misleading semantic slices. Removing the semantic dependency extension substantially degrades the quality of log messages. For example, BERTScore drops by 15.1%, indicating that the extension effectively contributes to generating

[1]As semantic slicing depends on block-type prompts, we use the first eligible line in the same block as the slicing entry when block-type prompting is removed.

Table 3: Ablation Study of PDLogger.

Model	Position			Logging Levels		Logging Variables			Logging Texts				
	Precision	Recall	F1	L- ACC	AOD	Precision	Recall	F1	BLEU-1	BLEU-4	ROUGE-1	ROUGE-L	BERTScore
PDLogger	**0.575**	0.674	**0.621**	**0.813**	**0.935**	**0.656**	**0.657**	**0.657**	**0.57**	**0.315**	0.487	0.469	0.537
w/o Block-type-based Structured Prompt Construction	0.246	0.718	0.366	0.575	0.798	0.443	0.492	0.466	0.442	0.143	0.332	0.317	0.473
w/o Semantic-dependency Extension	0.575	0.674	0.621	0.685	0.819	0.413	0.411	0.412	0.482	0.202	0.392	0.374	0.456
w/o Function-aware Extension	0.575	0.674	0.621	0.606	0.768	0.56	0.559	0.56	0.538	0.253	0.428	0.411	0.493
w/o Deduplication	0.489	**0.704**	0.573	0.813	0.935	0.656	0.657	0.657	0.572	0.314	**0.498**	**0.474**	**0.544**
w/o Log-Level Refinement	0.575	0.674	0.621	0.733	0.833	0.656	0.657	0.657	0.57	0.315	0.487	0.469	0.537



(a) Semantic-dependency Extension Removed **(b) Function-Aware Extension Removed**

Figure 10: Case Study of the Ablation Study

semantically more aligned log messages. Removing the Function-aware Extension further reduces variable F1 by 14.8%. Without deduplication, recall for position slightly increases by 4.3%, but precision drops by 17.59%, leading to an overall F1 score decrease of 8.38%, confirming its effectiveness in suppressing false positives. Lastly, excluding level refinement decreases level accuracy by 9.8%, showing its value in predicting appropriate log levels.

Case Study. Figure 10 presents two cases that demonstrate how PDLogger benefits from each phase of the framework. The highlighted lines denote the original log statements. As shown in Figure 10a, without capturing semantic dependency information, the variant fails to identify the root cause that leads to the conditional statement `if (result == null)` being true, and merely generates a generic log indicating that `result` is null. However, after incorporating semantic dependency expansion, PDLogger can understand that the underlying reason for `result` being null is due to a lack of original data info, thereby enabling the generation of a more informative log message that facilitates and accelerates system debugging during development.

In the case shown in Figure 10a, without applying function-aware extension to the target method, PDLogger cannot include relevant functions in the variable candidate list. With an explicitly provided function list, PDLogger autonomously selects appropriate functions as variables, thereby producing higher-quality logging statements, as shown in the green box.

PDLogger can leverage log level refinement to correct inappropriate log levels. For example, adjusting an improper debug level in `LOG.debug("Failed to process data due to lack of original data info")` to `error`, to alert developers to a potential failure rather than simply providing debugging information.

> Answer to RQ3: Ablation results indicate that each major design choice in PDLogger plays a critical role in the overall effectiveness.

4.5 RQ4: Generalizability Study

To evaluate PDLogger 's generalizability, we deploy it on three widely used and representative LLMs: OpenAI o3-mini, Deepseek-Chat, and LLaMA-3-70B-chat. Note that OpenAI o3-mini is the default backbone used in PDLogger.

As shown in Table 4, PDLogger consistently outperforms SCLogger across all tested models, demonstrating strong generalization capabilities. On average, PDLogger improves log position prediction F1 score by 108.4%, log level accuracy by 76.4%, log variable

Table 4: The Performance of PDLogger and SCLogger with Different Backbone Models.

Model	Approach	Position			Logging Levels		Logging Variables			Logging Texts				
		Precision	Recall	F1	L-ACC	AOD	Precision	Recall	F1	BLEU-1	BLEU-4	ROUGE-1	ROUGE-L	BERTScore
O3-mini	PDLogger	0.575	0.674	0.621	0.813	0.935	0.656	0.657	0.657	0.57	0.315	0.487	0.469	0.537
	SCLogger	0.24	0.784	0.367	0.446	0.739	0.283	0.315	0.28	0.438	0.174	0.384	0.375	0.324
	△	139%	-16.3%	69.2%	82.3%	19.6%	131.8%	108.5%	134.6%	30.1%	81%	26.8%	25.1%	65.7%
Llama-3-70b	PDLogger	0.548	0.646	0.593	0.729	0.853	0.605	0.588	0.597	0.482	0.243	0.408	0.394	0.485
	SCLogger	0.142	0.844	0.243	0.424	0.691	0.416	0.501	0.455	0.41	0.125	0.325	0.319	0.428
	△	285.9%	-23.5%	144.0%	71.9%	23.4%	45.4%	17.4%	31.2%	17.6%	94.4%	25.5%	23.5%	13.3%
Deepseek-chat	PDLogger	0.523	0.658	0.583	0.749	0.891	0.463	0.437	0.45	0.468	0.21	0.416	0.394	0.456
	SCLogger	0.16	0.982	0.275	0.428	0.68	0.44	0.49	0.464	0.423	0.137	0.359	0.349	0.355
	△	226.8%	-32.9%	112%	75%	31%	5.2%	-10.8%	-3.0%	10.6%	53.3%	15.9%	12.9%	28.5%

F1 score by 54.3%, and log message BERTScore by 60.3%. Furthermore, models with stronger comprehension capabilities, such as OpenAI o3 mini, exhibit greater performance when integrated with PDLogger.

> Answer to RQ4: PDLogger maintains high effectiveness in log generation when used with different LLMs, showcasing strong generalization capabilities across backbone models.

5 Discussion

Practical Implications. The adoption of an automated logging solution hinges on deployability. PDLogger automatically injects an appropriate number of high-quality log statements into projects without any logs— functionality absent from prior approaches. Developers input only their source code, and PDLogger outputs a predicted-log augmented project with no extra effort. This substantially reduces developers' workload and mitigates the common problem of "after-the-fact" log insertion. Thus, PDLogger integrates into workflows at a low cost and high level of automation, offering strong practical value.

Limitations. Two main limitations arise. (i) Evaluation uses mostly Java projects, leaving cross-language generality uncertain; yet our technique is language-agnostic and could transfer with moderate adaptation. (ii) The pipeline struggles when several logs should be placed in one block. Future work should explore reducing false positives in multi-log scenarios.

Threats to Validity. Given that PDLogger relies on large language models (LLMs) during processing, it raises data-leakage risks for proprietary code. Mitigations include asserting code copyright, running LLMs offline, and evaluating leakage risk before adoption in closed-source projects.

6 Related Work

Logging Practices. Software logs are indispensable for understanding system behaviour and diagnosing failures. According to [53], logs appear every 30 lines of code on average and improve fault diagnosis efficiency by 2.2×.

However, [11] finds that commits involving log insertions are rare in version-control history, suggesting that most logs are added retrospectively. Recent work [24] shows that large language models (LLMs) are effective for automatic log-statement generation. A key challenge is balancing log quantity: too many logs cause runtime overhead [8], while too few risk missing critical information [3], hindering diagnosis. To bridge this gap, this paper proposes the PDLogger.

Automated Logging Statements. Research on automated logging traditionally divides the task into predicting the log position, level, message and variable. For log-position prediction. A variety of methods seek to identify suitable insertion position [21, 29, 52, 60]. While [60] learns developers' habits from existing repositories, it does not incorporate rich contextual information. [29] observes that logging decisions depend on both syntactic structure and semantic context, yet its contextual modelling excludes inter-procedural or cross-method information.

For log-level recommendation, severity prediction is addressed by [30, 33], which employ machine-learning techniques to recommend appropriate log levels. For log-message generation, high-quality, context-aware messages are produced by [10, 13]. For log-variable selection, variable-selection strategies are explored in [34].

Although component-wise methods have advanced the field, they lack end-to-end pipelines. Recent holistic systems [26, 37, 47, 48, 56] generate complete logs but have limitations: [37] offers a Java method-level solution with limited context, and [26] extends static context but lacks semantic dependencies and variable handling. Existing end-to-end approaches remain impractical for real-world use. We propose a practical, deployable method that inserts appropriate high-quality logs into initially log-free projects.

7 CONCLUSION

In this paper, we introduce PDLogger, the first automatically log generation approach that is practically applicable to real-world software development. PDLogger incorporates semantic dependency information and variables within function scope into language models through block-type-based prompt construction. Moreover, it employs deduplication and level refinement strategies to ensure its usability in practical development scenarios. Experimental results demonstrate that PDLogger outperforms all baseline methods in overall performance and can be effectively adapted to a wide range of LLMs. We believe that PDLogger can enhance developer productivity and provide valuable insights for researchers in the field of automated log generation.

References

[1] 2025. Apache ActiveMQ: Version 5.18.7. https://activemq.apache.org/components/classic/download/.
[2] Meta AI. 2024. LLaMA 3. https://ai.meta.com/llama/. Accessed July 2025.
[3] Abhay Amar and Peter C. Rigby. 2019. Mining Historical Test Logs to Predict Bugs and Localize Faults. In *41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 140–151.

[4] Apache. 2023. Apache Log4j 2. https://logging.apache.org/log4j/2.x/.

[5] Apache. 2025. Apache Hadoop: Release 3.4.1. https://hadoop.apache.org/releases.html.

[6] Paul A. Cohen. 2003. Java AST and Compiler Reuse via the Eclipse JDT. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 123–136.

[7] DeepSeek. 2024. DeepSeek-Chat. https://github.com/deepseek-ai/DeepSeek-LLM. Accessed July 2025.

[8] Rui Ding, Huailin Zhou, Jian-Guang Lou, Hongyu Zhang, Qiang Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis. In *USENIX Annual Technical Conference (USENIX ATC 15)*. 139–150.

[9] Yuhang Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. TRACED: Execution-Aware Pre-Training for Source Code. In *46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–12.

[10] Zhaoning Ding, Ying Tang, Xiaoning Cheng, Heng Li, and Weiyi Shang. 2023. LogEnText-plus: Improving Neural Machine Translation Based Logging Texts Generation with Syntactic Templates. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–45.

[11] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 24–33.

[12] Yuhan Fu, Minrui Yan, Peng He, Chang Liu, Xiaoning Zhang, and Di Yang. 2024. End-to-End Log Statement Generation at Block Level. *Journal of Systems and Software* 216 (2024), 112–146.

[13] Saeid Gholamian. 2021. Leveraging Code Clones and Natural Language Processing for Log Statement Prediction. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1043–1047.

[14] Saeid Gholamian and Patrick A. S. Ward. 2021. What Distributed Systems Say: A Study of Seven Spark Application Logs. *arXiv preprint arXiv:2108.08395* (2021).

[15] Shenghui Gu, He Zhang, Wanggen Liu, and Guoping Rong. 2023. LoGenText-Plus: Improving Neural Machine Translation-based Logging Texts Generation with Syntactic Templates. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 32, 3 (2023), 61:1–61:33.

[16] C. Gulcu. 2023. SLF4J: Simple Logging Facade for Java. https://www.slf4j.org.

[17] Jiawei He, Mohit Rungta, Daniel Koleczek, Aman Sekhon, Feng-Xiang Wang, and Shafiq Hasan. 2024. Does Prompt Formatting Have Any Impact on LLM Performance? *arXiv preprint arXiv:2411.10541* (2024).

[18] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 61–71.

[19] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. 2023. LLMParser: A LLM-based Log Parsing Framework. *arXiv preprint arXiv:2310.01796* (2023).

[20] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. 2023. A Large-scale Benchmark for Log Parsing. *arXiv preprint arXiv:2308.10828* (2023).

[21] Sahil Lal, Nikhil Sardana, and Anil Sureka. 2016. LogOptPlus: Learning to Optimize Logging in `catch` and `if` Programming Constructs. In *40th IEEE Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 215–220.

[22] Heng Li, Zhilei Mao, and David Lo. 2016. Towards Just-in-Time Suggestions for Log Changes. *Empirical Software Engineering* 21, 2 (2016), 621–656.

[23] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which Log Level Should Developers Choose for a New Logging Statement? *Empirical Software Engineering* 22 (2017), 1684–1716.

[24] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, Lionel Briand, and Michael R. Lyu. 2023. Exploring the Effectiveness of LLMs in Automated Logging Generation: An Empirical Study. *arXiv preprint arXiv:2307.05950* (2023).

[25] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, and Michael R. Lyu. 2023. Exploring the Effectiveness of LLMs in Automated Logging Generation: An Empirical Study. *arXiv preprint arXiv:2307.05950* (2023).

[26] Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jing Liu, Jiang Huang, Jiazhen Gu, Pinjia He, and Michael R. Lyu. 2024. Go Static: Contextualized Logging Statement Generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 609–630.

[27] Zhenhao Li, Tse-Hsun Chen, Jingwei Yang, and Weiyi Shang. 2021. Studying Duplicate Logging Statements and Their Relationships with Code Clones. *IEEE Transactions on Software Engineering* (2021), 2476–2494.

[28] Zhenhao Li, Tse-Hsun Chen, Jing Yang, and Weiyi Shang. 2023. Are They All Good? Studying Practitioners' Expectations on the Readability of Log Messages. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–12.

[29] Zhenhao Li, Tse-Hsun Peter Chen, and Weiyi Shang. 2020. Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 361–372.

[30] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiyi Shang. 2021. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. 1461–1472.

[31] Zhenhao Li and Weiyi Shang. 2014. The Game of Twenty Questions: Do You Know Where to Log?. In *11th Working Conference on Mining Software Repositories (MSR)*. 146–156.

[32] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Workshop on Text Summarization Branches Out*. 74–81.

[33] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. 2022. TeLL: Log Level Suggestions via Modeling Multi-Level Code Block Information. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 27–38.

[34] Zihan Liu, Xiaoning Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Sen Li. 2019. Which Variables Should I Log? *IEEE Transactions on Software Engineering* (2019), 2012–2031.

[35] Robert C. Martin. 2002. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, NJ.

[36] Antonio Mastropaolo. 2023. LANCE2.0. https://github.com/antonio-mastropaolo/automating-logging-acitivities.

[37] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using Deep Learning to Generate Complete Log Statements. In *44th International Conference on Software Engineering (ICSE)*. 2279–2290.

[38] Eduardo Mendes and Fabio Petrillo. 2021. Log Severity Levels Matter: A Multivocal Mapping Study. *arXiv preprint arXiv:2109.01192* (2021).

[39] OpenAI. 2024. o3-mini. https://platform.openai.com/. Accessed July 2025.

[40] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *40th Annual Meeting of the Association for Computational Linguistics (ACL)*. 311–318.

[41] Kunal Patel, José Faccin, Ameél Hamou-Lhadj, and Iury Nunes. 2022. The Sense of Logging in the Linux Kernel. *Empirical Software Engineering* 27, 6 (2022), 153.

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[43] Guoping Rong, Shenghui Gu, He Zhang, Dong Shao, and Wanggen Liu. 2018. How Is Logging Practice Implemented in Open Source Software Projects? A Preliminary Exploration. In *25th Australasian Software Engineering Conference (ASWEC)*. 171–180.

[44] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers*. 214–224.

[45] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems* 13, 2 (1991), 181–210.

[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *arXiv preprint arXiv:2201.11903* (2022).

[47] Xiaoxue Xie, Zhiyuan Cai, Sheng Chen, and Jing Xuan. 2024. FastLog: An End-to-End Method to Efficiently Generate and Insert Logging Statements. In *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 26–37.

[48] Jiaxu Xu, Zhiqing Cui, Yihao Zhao, Xin Zhang, Sheng He, Peng He, Lin Li, Yu Kang, Qiang Lin, Yuan Dang, and Shrinivasan Rajmohan. 2024. Unilog: Automatic Logging via LLM and In-Context Learning. In *46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–12.

[49] Fermin Yamaguchi, Nicolai Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *33rd IEEE Symposium on Security and Privacy (S&P)*. 590–604.

[50] Jie Yang, Yuan Dang, and Dongmei Zhang. 2018. Predicting Logging Levels Using Software Development Features. In *24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2074–2083.

[51] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 127–138.

[52] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 293–306.

[53] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing Logging Practices in Open-Source Software. In *34th International Conference on Software Engineering (ICSE)*. 102–112.

[54] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. *ACM Transactions on Computer Systems* 30, 1 (2012), 4.

[55] He Zhang, Shenghui Gu, Guoping Rong, and Wanggen Liu. 2020. Automatically Generating Descriptive Texts in Logging Statements: How Far Are We?. In *18th Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes*

in Computer Science, Vol. 12470). Springer, 311–331.

[56] Hongyu Zhang, Dasen Yu, Liuzhe Zhang, Guoping Rong, Yiyang Yu, Hong Shen, Hongbo Zhang, Dong Shao, and Haoyu Kuang. 2024. LoGFiLM: Fine-Tuning a Large Language Model for Automated Generation of Log Statements. *arXiv preprint arXiv:2412.18835* (2024).

[57] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating Text Generation with BERT. In *8th International Conference on Learning Representations (ICLR)*.

[58] Yang Zhang, Xiaosong Chang, Lining Fang, and Yifan Lu. 2023. DeepLog: Deep-Learning-Based Log Recommendation. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 88–92.

[59] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 565–581.

[60] Jieming Zhu, Peng He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*, Vol. 1. 415–425.