

RESEARCH ARTICLE

Automated Log Statement Using Source-Code Metrics

SE-JIN KIM^{ID} AND CHAN-GUN LEE^{ID}

Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Chan-Gun Lee (cglee@cau.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government Ministry of Science and ICT (MSIT) under Grant RS-2024-00342059, and in part by the LG Electronics.

ABSTRACT Logging is essential in modern software development for testing, debugging, monitoring, and maintaining applications, but manually inserting log statements is time-consuming and inconsistent. While automated log generation methods exist, their effectiveness is often limited when applied to methods with high structural complexity, leading to inaccurate or irrelevant logs. To address this challenge, we propose a novel approach for automated log generation that proactively simplifies complex method before model training. By employing source-code metrics, including cyclomatic complexity, maintainability index, and lines of code, we identify methods that exceed predefined complexity thresholds and decompose them into smaller, more manageable blocks. This decomposition strategy allows a fine-tuned CodeT5+ model to learn from more focused contexts, significantly enhancing its ability to generate accurate logging statements. The proposed method achieved an overall accuracy of 25.23%, with a position accuracy of 99.87%, a level accuracy of 75.34%, and a message accuracy of 31.69%, represent a substantial improvement over baselines, including LEONID and ELogger. Our findings demonstrate that integrating a metrics-based code decomposition preprocessing step is a highly effective strategy for improving automated log generation, offering a scalable solution to enhance software maintainability.

INDEX TERMS Log generation, pretrained language model, software engineering, source-code metrics.

I. INTRODUCTION

In software development, logging is essential for testing, debugging, tracking, and maintaining applications [1]. Logs provide valuable information on the execution flow, performance, and potential problems in an application, enabling developers to maintain system reliability [2]. However, manually inserting logging statements is time-consuming and prone to inconsistencies in format, details, and placement [3]. These challenges arise from the following factors: differences in developers' styles and levels of understanding [4], the time-intensiveness of identifying appropriate logging points in complex and diverse code logic [5], and the difficulty of balancing development speed with debugging requirements [6]. Hence, this lack of standardization complicates log analysis and undermines collaborative development efforts.

The associate editor coordinating the review of this manuscript and approving it for publication was Mahmoud Elish^{ID}.

```
public static void close(Closeable closeable) {  
    if (closeable != null) {  
        try {  
            closeable.close();  
        } catch (Exception e) {  
            logger.info("Unable to close %s", closeable, e);  
        }  
    }  
}
```

FIGURE 1. Example of where to log, which log level and what to log [7].

Clear guidelines for where to log, which log level to use, and what to log must be established to address the challenges of manual logging, as illustrated in Fig. 1. Determining where to insert log statements is crucial for maximizing their diagnostic value. For example, the log statement is added after the catch block to capture critical failures during resource closure. Equally important is selecting the

appropriate log level(in this case, info) to ensure logs remain actionable and manageable. Finally, log messages benefit from being informative and contextually relevant. For example, the message “unable to close %s” provides clear information about the failure, and including the context (closeable and exception e) ensures that the log is meaningful for debugging and troubleshooting. Developers can create consistent and effective logs that enhance the reliability and maintainability of the system by following these principles.

Researchers have developed automated log generation techniques, increasingly employing deep learning to address problems, such as determining what to log [8], [9], [10], [11], [12], where to log [1], [13], [14], [15], [16], [17], and which to log [6], [18], [19], [20]. More recent approaches (e.g., LANCE [21] and LEONID [22]) employ large language models (LLMs; e.g., T5 [23]) to learn patterns between code features and logging practices to replicate developer decisions. Moreover, ELogger [24] applies a layered approach with GraphCodeBERT [25] to generate logs for specific code blocks, addressing the decision-making for log placement.

Despite these advancements, the existing methods face challenges. For instance, LEONID splits its models based on whether a method contains single or multiple logs, complicating its consistent application across methods without logs. Furthermore, ELogger restricts its code-block division to specific constructs, such as method declarations, loops, and try-catch blocks, limiting its ability to manage diverse and complex code structures. More generally, models struggle to maintain performance as code complexity increases, as the intricate control flows and data dependencies in such code can hinder their ability to capture the accurate context required for effective log generation.

These challenges point to a more fundamental limitation in prior works, because the intrinsic properties of the source code itself are not adequately leveraged. Approaches like ELogger engage with code structure only superficially, using fixed syntactic constructs like loops, method declarations, and try-catch blocks as simple triggers for division. Other methods bypass code characteristics altogether, focusing solely on fine-tuning a model on the relationship between a method’s text and its log statements. We argue that this overlooks the root cause of poor performance, because as code complexity increases, the context becomes too noisy for a model to learn effectively. Our work is the first to diagnose this core problem and propose a targeted solution. We directly utilize code complexity metrics not just as a feature, but as a strategic preprocessing step to simplify challenging methods, thereby creating a cleaner and more learnable context for the model.

This paper proposes a novel approach that integrates source-code metrics, specifically cyclomatic complexity (CC), lines of code (LOC), and maintainability index (MI), into the automated log generation to bridge this gap. The proposed method involves decomposing complex methods into simpler code blocks based on defined complexity thresholds. By fine-tuning a CodeT5+ model on this preprocessed

dataset, the proposed approach enhances the accuracy and contextual relevance of the generated logs, particularly in complex methods. Furthermore, the proposed approach is generalizable and scalable and is consistently applicable across diverse code structures without depending on specific constructs or scenarios.

We evaluated the proposed approach using a dataset of software methods and log statements. The proposed method achieves an overall accuracy of 25.23% in predicting the position, level, and message of log statements. Notably, the accuracy of the model in predicting log positions was 99.87%, demonstrating its effectiveness in determining where to insert logs. The accuracy of the level prediction was 75.34%, and the prediction of the message was 31.69%, indicating meaningful improvements in the relevance and quality of the log.

In summary, this paper makes the following contributions:

- We propose a novel approach integrating a source-code metrics analysis into the automated log generation process. By decomposing complex methods into simpler code blocks based on established source-code metrics, the proposed approach improves the accuracy and relevance of the generated logs.
- We provide a universally applicable solution that does not rely on specific code constructs or require separate models for various logging scenarios. The proposed method can be applied consistently across all methods, regardless of their structure or the number of required logs, offering a scalable and generalizable approach to automated log generation.

These contributions address the limitations of existing studies by enhancing the log generation accuracy and improving code maintainability by simplifying complex methods. The proposed approach offers developers a more efficient and reliable means of maintaining and debugging complex software systems.

The remainder of this paper is organized as follows. Section II reviews related work in automated log generation and source-code metrics. Section III details our proposed methodology, including the data collection and preprocess process, source-code metrics-based method decomposition, and the model training procedure. Section IV presents the experimental results and a comparative analysis against baseline models. Section V discusses potential threats to the validity of our study and outlines directions for future work. Finally, Section VI concludes the paper with a summary of our contributions.

II. RELATED WORK

A. LOG GENERATION

Early research on automated logging predominantly addressed this challenge by breaking it into distinct subproblems. These studies often focused on one of three fundamental considerations in isolation: what content to include in a log message, where to place the log statement in

the code, or which severity level to assign to it. The following sections review the principal approaches in each of these separate research streams before addressing the recent shift toward unified, end-to-end solutions.

1) WHAT TO LOG

What to log refers to the content and context of log messages, emphasizing their informativeness and relevance. Yuan et al. [8] introduced LogEnhancer, which recommends relevant variable values for each log statement, significantly reducing the search space for potential root failure causes. In addition, Liu et al. [9] addressed a similar problem using a customized deep learning network, which achieved a mean average precision of over 84%. Ding et al. [10] proposed LoGenText, a neural machine translation-based approach that summarizes code preceding a log statement into a concise, descriptive text. Such enhancements can be considered complementary to other methods because they offer more intelligible log content. Furthermore, Li et al. [11] studied software logging using topic models, revealing common themes that developers log and informing the design of automated log message generation techniques. Understanding logging intentions and patterns [12] and logging configuration practices [26] can guide tool design.

2) WHERE TO LOG

Where to log refers to identifying the optimal locations for log statements to maximize their diagnostic value. For example, Zhu et al. [13] introduced LogAdvisor, which recommends log insertion points and reached 60% accuracy when applied to code without logs. Similarly, Yao et al. [14] focused on central processing unit use monitoring in web-based systems, revealing that contextual understanding facilitates log placement decisions. Li et al. [15] proposed a deep learning framework that suggests logging locations at the code-block level, achieving up to 80% accuracy. Other researchers have explored log placement strategies in both open-source and industrial contexts [1], [16], [17], confirming that models trained on open-source data can be effective in corporate environments. Such efforts underscore the interplay between code structure, system context, and logging best practices.

3) WHICH TO LOG

Which to log focuses on selecting suitable log levels (e.g., error, warning, and info) to balance clarity and utility in log messages. Techniques have been developed to recommend log levels [6], [18], ensuring that critical events receive adequate attention without overwhelming developers. For instance, Mizouchi et al. [19] introduced PADLA, dynamically adapting log levels in the Apache Log4j framework. Moreover, Li et al. [20] proposed DeepLV, a model that applies syntactic and semantic features of source code to suggest optimal log levels, outperforming previous approaches. Prior studies have also examined which log level

developers should select under various circumstances [5], guiding automated methods towards decisions aligned with human judgment and practice.

4) GENERATING COMPLETE LOG STATEMENT

Recent approaches have begun to employ LLMs to streamline the entire logging process, moving beyond individual sub-tasks to provide more comprehensive solutions. For instance, LANCE [21] employs a T5-based LLM to generate complete log statements for Java methods, although it depends on the prior identification of methods requiring logging. In contrast, ELogger [24] integrates the decision-making for all four logging subtasks (i.e., whether to log, where to place logs, which log level to use, and what information to include) within a single model at the code-block level, ensuring a consistent and unified feature representation. Building on these advancements, LEONID [22] extends this capability further by enabling the insertion of multiple log statements when needed; however, managing highly complex methods remains a challenge, as higher complexity can hinder the ability of the model to capture the accurate context required for effective log generation. Furthermore, studies are applying log generation techniques to various other domains. Shu et al. [27] evaluated various language models, such as GPT-3.5-Turbo and a test-specific variant of LANCE, on their ability to generate complete log statements in the distinct context of test code. Li et al. [28] utilized models like DeepSeek and GPT-4o with Chain-of-Thought reasoning to generate log sequences specifically for creating log anomaly detection datasets.

The proposed approach is differentiated from previous methods by incorporating source-code metrics to guide the decomposition of complex methods into smaller code blocks. This strategy enables the proposed model to generate accurate and contextually appropriate logging statements even for highly complex code, achieving an overall accuracy of 25.23%.

B. SOURCE-CODE FEATURE-BASED LOGGING

Considerable research has focused on improving automated logging by employing the intrinsic characteristics of the source code. These approaches analyze code features to predict optimal logging placement and content. We categorize these studies based on the primary type of source-code characteristics they exploit, highlighting an opportunity for further research: employing complexity to guide log generation proactively.

1) SYNTACTIC AND STRUCTURAL FEATURES

One research direction focuses on applying detailed structural metrics for specific, critical contexts. For example, Lal and Sureka [29] developed LogOpt, a tool that employs 46 handcrafted metrics, including statement counts and nesting depth, to determine whether a “catch” block requires a log statement. This approach demonstrates that a deep

analysis of the code structure in a focused area can yield highly accurate predictions.

2) SEMANTIC AND CONTEXTUAL FEATURES

Other approaches employ a richer semantic context to improve recommendations. For example, Gholamian [30] combined a code clone analysis with natural language processing to recommend the location and content of log statements by finding semantically similar code snippets. In a mapping study, Bushong et al. [31] surveyed tools that create a semantic link between log entries and the source code that emits them, underscoring the importance of understanding the code meaning for effective logging.

3) LINK BETWEEN LOGGING AND COMPLEXITY

Large-scale empirical studies have consistently established a strong correlation between logging and code complexity. Lal et al. [32], [33] revealed this pattern at the file and exception-handling levels. Further, Shang et al. [34] linked extensive logging to a higher defect density. These seminal studies offer crucial evidence of this relationship, positioning complexity as an essential characteristic of logged code rather than a direct input for logging automation.

4) MODERN PREDICTIVE AND GENERATIVE MODELS

More recent work has shifted to build holistic models. For instance, Batoun et al. [35] proposed a deep learning model to predict the overall log density that a source file might require, moving beyond single-statement placement. Li et al. [36] benchmarked 11 LLMs for log generation, demonstrating that a richer code context improves performance and revealing that current models still have limitations. These modern approaches aim for more comprehensive solutions; however, they still primarily learn directly from unmodified code.

In summary, prior studies have applied static code features to guide automated logging. These studies have typically employed features—ranging from syntactic patterns to semantic embeddings—to predict log placement or content. Although code complexity is often identified as a critical correlational factor in these studies, it has not yet been employed as an actionable signal for the method decomposition preprocessing step. Our study directly addresses this gap. We introduce a novel approach, using explicit complexity thresholds to simplify code before synthesizing log statements, ensuring that even highly complex methods receive more targeted and precise logging generation.

III. METHOD

In this section, we propose a novel method to generate automated log statements by integrating a source-code metrics analysis into training. We collected Java projects from

GitHub, focusing on repositories with active development and standard logging practices. The methods were analyzed using CC, MI, LOC to identify complex methods exceeding the defined thresholds. We divided these complex methods into smaller, more manageable code blocks when thresholds were exceeded. We fine-tuned the CodeT5+ [37] model using this preprocessed dataset, with code excluding logging statements as input and the original code including logs as output. This approach enhances the ability of the model to generate accurate and contextually appropriate logging statements across code structures, addressing the limitations of existing methods.

A. SOURCE-CODE METRICS

This work applies well-established source-code metrics to measure code complexity quantitatively. These metrics provide objective criteria for identifying complex methods requiring decomposition. The primary metrics employed in the proposed approach are listed below:

- **Cyclomatic Complexity (CC):** Proposed by McCabe [38], CC measures the number of linearly independent paths through a control flow graph of a program. Higher values indicate higher structural complexity and potential maintenance difficulty [39].
- **Lines of Code (LOC):** This metric provides a basic measure of the size of method [40]. Although this metric is not a direct measure of structural complexity, larger code units often pose higher challenges concerning maintainability and comprehension [41].
- **Maintainability Index (MI):** The MI [42], [43] provides a single, holistic score for assessing the maintainability of software by integrating CC, Halstead volume [44], and LOC. We follow the formula used by Microsoft's Visual Studio [45], MI, as shown at the bottom of the page, where V denotes the Halstead volume. A higher MI value indicates better maintainability, whereas a lower value suggests the code is more complex and more challenging to maintain.

Initially, we considered including the Halstead volume as part of the complexity assessment. However, we were unable to find prior studies that established specific thresholds for the Halstead volume to define complex methods. Moreover, the range of the Halstead volume values varies significantly across methods, making it challenging to set a meaningful threshold. Hence, we excluded the Halstead volume from the complexity criteria.

B. DATA COLLECTION AND PREPROCESSING

Java projects were collected from GitHub repositories to build a comprehensive dataset for this study, following method similar to those in previous research, such as

$$MI = \max\left(0, \frac{(171 - 5.2 \ln V - 0.23 CC - 16.2 \ln LOC) \times 100}{171}\right)$$

research on LEONID and ELogger. The selection focused on repositories that are primarily written in Java, have at least 500 commits, involve at least 10 contributors, have at least 10 stars, and are not forks. These requirements were set to ensure that the repositories are active, well-maintained, and representative of real-world software development practices.

We performed preprocessing and filtering steps on the collected repositories to ensure the quality and relevance of the dataset. From the repositories that satisfied these conditions, further filtering was applied to include those that employ widely adopted logging libraries. We identified repositories that include dependencies for SLF4J (Simple Logging Facade for Java) [46] or Log4J [47] in their pom.xml or build.gradle files. We also ensured these were not forks to reduce duplicate code. This step ensured that the logging statements in the code follow standard logging conventions, which is crucial for analysis and model training. Applying this filter, yielded 2283 GitHub repositories.

We employed the Understand tool [48], a static code analysis platform, to process the selected repositories and extract all methods defined within the codebases, resulting in 24 943 104 methods. From these methods, we identified those containing logging statements using the standard log levels (fatal, error, warn, info, debug, and trace) provided by SLF4J and Log4J, excluding custom log levels.

We applied additional filtering to ensure the quality and relevance of the dataset. Moreover, we removed methods containing fewer than 10 tokens, as these were too short to provide meaningful context for log generation. We also excluded methods that could not be represented in ASCII encoding to avoid potential problems with non-standard characters that could affect the analysis and model training.

After these preprocessing steps, the resulting dataset comprised 766 696 methods with at least one logging statement. This curated set of methods then served as the input for the complexity-based decomposition process.

C. PRELIMINARY ANALYSIS AND MOTIVATION

Log generation involves generating code and inserting statements within methods, yet previous studies have focused on fine-tuning LLMs without source-code metrics. We conducted preliminary experiments to address this gap by applying the most commonly used source-code metrics (i.e., CC, LOC, and MI) to our dataset. Following thresholds established by prior research [38], [40], [45], we categorized methods as complex or non-complex based on the following metrics: CC (1–10 as non-complex and ≥ 11 as complex), LOC (0–250 as non-complex and ≥ 251 as complex), and MI (0–9 as complex and 10–100 as non-complex).

Table 1 illustrates the distribution of log counts across these metrics. Methods classified as complex frequently contain four or more logs, demonstrating a strong correlation between complexity and log count. These observations motivated us to employ source-code metrics for log generation, targeting complex methods for decomposition into simpler code

blocks. This approach ensures more effective log placement while addressing the challenges of high-complexity methods.

The proposed approach integrates source-code metrics analysis into the log generation process. We measured the source-code metrics of methods using the established metrics CC, LOC and MI and define the thresholds to determine whether a method is considered complex. If a method exceeded these thresholds, we decomposed it into simpler code blocks. Unlike ELogger, this decomposition approach was not limited to specific constructs but applied to all methods, offering a more general and scalable solution.

By breaking down complex methods into simpler code blocks, the proposed approach enhances log generation for intricate functionalities, ensuring logs are relevant and effective. This strategy improves the relevance of logs and enhances the performance of machine learning models in log prediction, allowing them to learn more effectively from less complex structures.

D. CODE BLOCK DIVISION BASED ON SOURCE-CODE METRICS

From the dataset constructed in Section III-B, we analyzed each method to measure its source-code metrics and identified complex methods that required decomposition into smaller code blocks. As mentioned, we employed three widely recognized metrics in software engineering, CC, LOC and MI, to quantify the complexity. These metrics capture distinct aspects of the code complexity and maintainability, offering a comprehensive basis for decomposition decisions. We applied the following thresholds to determine whether a method is complex:

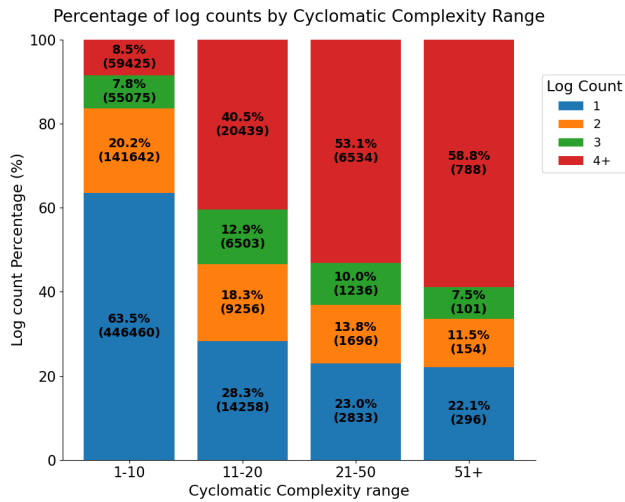
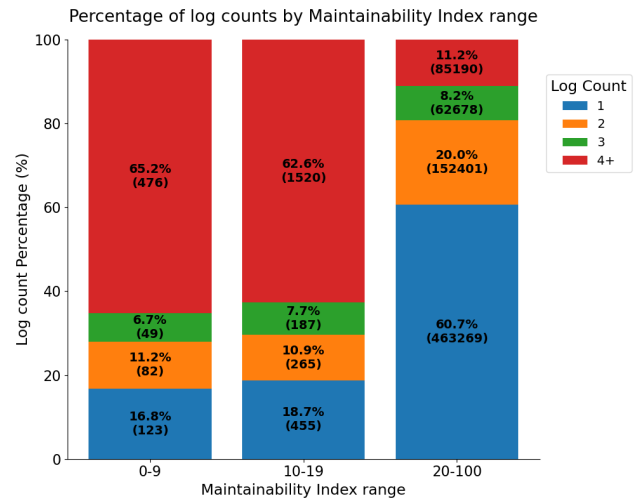
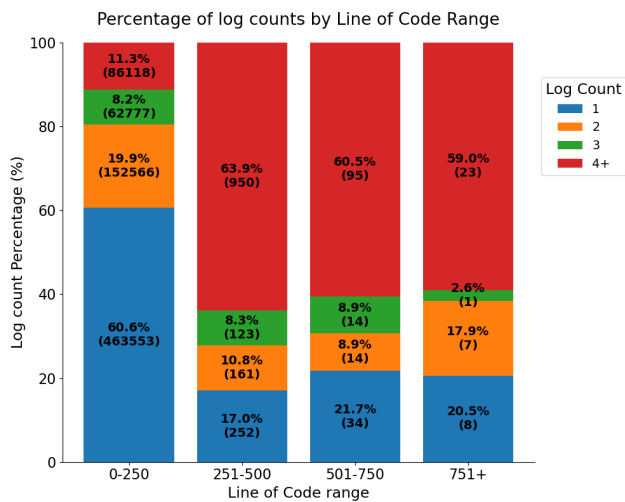
- CC > 10: Methods exceeding this value are considered highly complex and potentially difficult to understand and maintain.
- LOC > 250: Methods longer than this are often too lengthy and may encompass multiple responsibilities.
- MI ≤ 19 : A value at or below this threshold indicates poor maintainability, suggesting that the code is complex and challenging to modify.

The thresholds for CC, LOC, and MI were determined by analyzing their relationship with logging patterns, as shown in Figures 2, 3, and 4. For CC (Fig. 2), methods with low complexity (1–10) are considered easy to maintain, and they predominantly contained a single log (63.5%), with only 8.5% having four or more logs. However, as complexity increased to 11–20 (classified as difficult to maintain), methods with four or more logs rose significantly to 40.5%. For complicated methods (21–50), over half contained four or more logs (53.1%). For methods deemed impossible to maintain (51+), this proportion increased further to 58.8%. Thus, a threshold of CC > 10 was set.

Similarly, for LOC (Fig. 3), methods shorter than 250 lines (considered easy to maintain) generally included a single log (60.6%). As the length increased to between 251 and 500 lines (categorized as challenging to maintain) the

TABLE 1. Distribution of log counts in methods across source-code metrics.

		Log Count			
		1	2	3	4+
Cyclomatic Complexity	non-Complex	61.2%(460718)	20%(150898)	8.2%(61578)	10.6%(79864)
	complex	22.9%(3129)	13.6%(1850)	9.8%(1337)	53.7%(7322)
Line of Code	non-Complex	60.6%(463553)	19.9%(152566)	8.2%(62777)	11.3%(86118)
	complex	16.8%(123)	11.2%(82)	6.7%(49)	65.2%(476)
Maintainability Index	non-Complex	60.5%(463724)	19.9%(150898)	8.2%(62866)	11.3%(86710)
	complex	17.5%(294)	10.8%(182)	8.2%(138)	63.5%(1068)

**FIGURE 2.** Distribution of log counts by cyclomatic complexity range.**FIGURE 4.** Distribution of log counts by maintainability index range.**FIGURE 3.** Distribution of log counts by lines of code range.

proportion of methods with four or more logs rose to 63.9%. This trend persisted for very difficult (501–750) and impossible to maintain methods (≥ 751), where the majority contained multiple logs, highlighting the need for additional logging in lengthy methods. Thus, a threshold of LOC > 250 was set.

Finally, for the MI (Fig. 4), methods with high maintainability (20–100), classified as easy to maintain, typically

require less frequent logging, with 60.7% having only one log. In contrast, methods with an MI between 10 and 19 (categorized as challenging to maintain) increased with four or more logs (62.6%), whereas those with an MI ≤ 9 (considered very difficult to maintain) reached 65.2%, reflecting their complexity and poor maintainability. Therefore, a threshold of MI ≤ 19 was set.

These thresholds provide a systematic basis for identifying complex methods that require decomposition into smaller, more manageable code blocks to facilitate effective logging.

We divided methods exceeding the established threshold into smaller, less complex code blocks by reading the method line by line while incrementally calculating the cumulative CC and LOC for the current block. If the CC exceeded 10 or the LOC surpassed 250, the current block was marked, and a new block was initiated. For example, a method with a CC exceeding 10 is divided into two code blocks (Code Blocks 1 and 2 in Fig. 5). This process was repeated until the end of the method to ensure that all resulting code blocks adhered to the predefined complexity thresholds.

Although we employed MI to identify complex methods at the method level, we did not apply it when dividing the code blocks. Calculating the MI requires the Halstead volume, which depends on accurately counting operators and operands. In code blocks extracted from the methods, distinguishing operators and operands is challenging due to the lack of full context and potential code fragmentation.

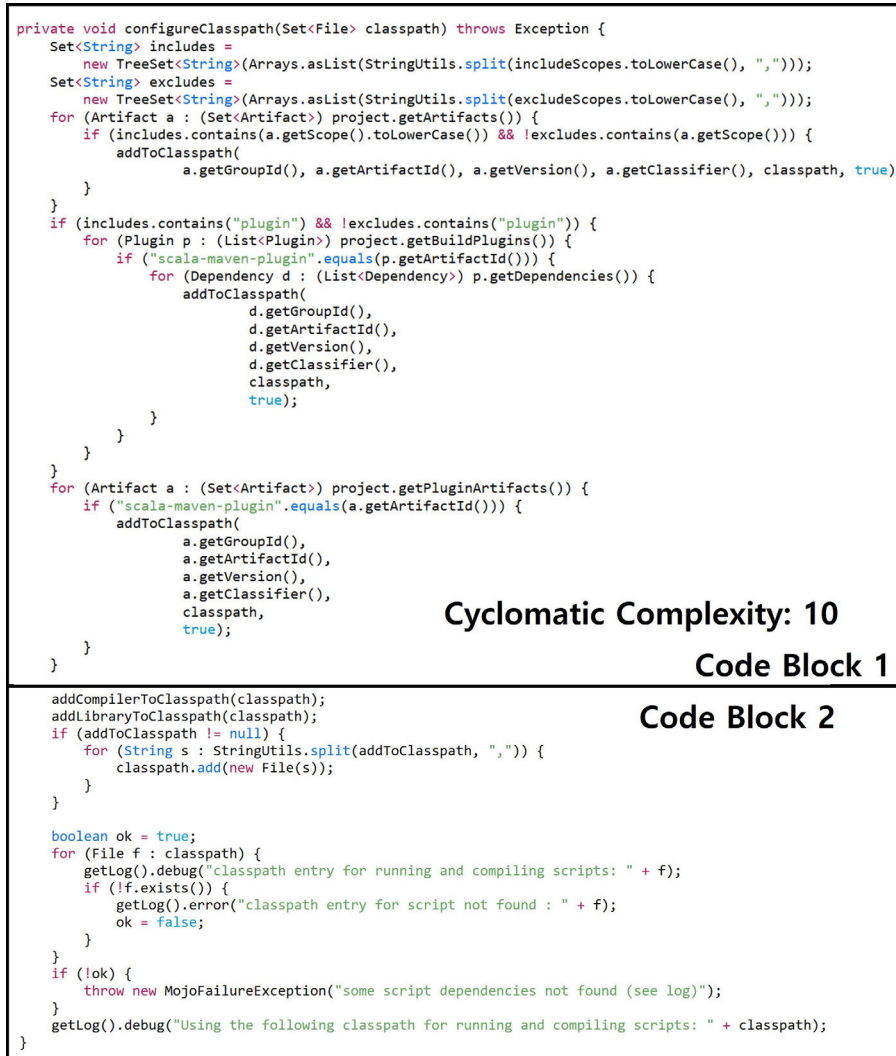


FIGURE 5. Example of a method split into code blocks based on complexity thresholds [49].

TABLE 2. Summary of the dataset.

Metric	Value
Data Collection	
Initial GitHub Repositories	2,283
Total Methods Extracted	24,943,104
Preprocessing	
Methods with Logs (Before Decomposition)	766,696
Final Dataset Size (After Decomposition)	917,835

Thus, we could not reliably compute the Halstead volume or MI for individual code blocks and relied solely on CC and LOC for code-block division.

By dividing complex methods into smaller code blocks based on these metrics, we aimed to create manageable units of code that are more flexible and for which the proposed log generation model could generate accurate logging statements. This approach addresses the problem that complex methods may encompass multiple functionalities or intricate logic, making it challenging to generate appropriate logs that capture all relevant details. Code blocks that did not contain any logs were excluded from the dataset to ensure that all

included blocks were relevant for log generation analysis. For the model training phase, each code block in this curated dataset had its logging statements removed to serve as the input, while the original code containing the logs served as the ground-truth output. Through this decomposition process, the final dataset comprised 917 835 code blocks and methods for analysis and model training, as summarized in Table 2.

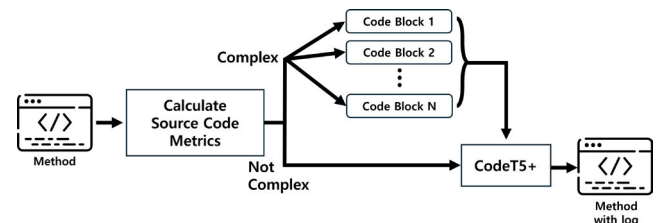


FIGURE 6. Overview of the model training pipeline.

E. MODEL TRAINING

In this phase, the CodeT5+ model was fine-tuned using the code blocks and their corresponding logging statements

obtained in Section III-D. The CodeT5+ model was selected as the foundation for our approach due to its proven capabilities in code-related generative tasks. As a high-performance LLM, CodeT5+ is built on a flexible encoder-decoder Transformer architecture. It was pretrained on a vast, multilingual corpus of source-code, as well as code-adjacent natural language text such as comments and documentation. This allows the model to develop a deep, intrinsic understanding of programming language syntax and structure.

The task of log generation requires a model to excel at both code comprehension and natural language generation. First, the model understands the semantic context of a code block to determine an appropriate logging point and identify relevant variables. Subsequently, it generates a log message that is coherent and contextually appropriate for the code block. CodeT5+'s pre-training on a mixture of code and text makes it particularly well-suited for this hybrid task, distinguishing it from models trained solely on natural language. This dual capability is crucial for generating log statements that are not only syntactically correct but also semantically relevant to the code's intent.

Fig. 6 illustrates the overall training pipeline. The process begins with calculating source-code metrics, where complex methods are segmented into simpler code blocks. These blocks are paired with their corresponding logging statements to create a dataset to fine-tune the CodeT5+ model. This focus on simpler code structures is intended to enhance the ability of the model to generate accurate and contextually relevant logging statements.

Training was conducted on an Nvidia H100 graphics processing unit. The learning rate was set to 5×10^{-5} , and a weight decay of 0.05 was applied to prevent overfitting. The batch size was 32, and the model was trained for 30 epochs.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

This section presents a comprehensive evaluation to validate the effectiveness of the proposed method. The analysis was structured around two principal research questions (RQs) addressing a comparison of the proposed approach against state-of-the-art baselines and an ablation study examining the influence of its core components.

A. EVALUATION METRICS

We employed several metrics commonly used in natural language processing and code generation tasks to evaluate the performance of the automated log statement generation approach. These metrics offer a comprehensive assessment of the ability of the model to generate accurate and contextually appropriate logging statements.

Accuracy measures the proportion of correct predictions made by the model over the total number of predictions. In our context, accuracy reflects the ability of the model to generate logging components correctly, including the insertion position, logging level, and log message content.

Higher accuracy indicates that the model is more effective at generating output that aligns with the ground truth.

1) CodeBLEU

The specialized CodeBLEU [50] evaluation metric was designed for programming language generation tasks. This metric extends the traditional BLEU metric by incorporating syntax and semantic features that are specific to the code, providing a more detailed assessment of the code generation quality.

The CodeBLEU combines four weighted components:

- The n-gram Match (BLEU score) measures the overlap of n-grams between the generated code and reference code, considering sequences of up to four.
- The weighted n-gram match assigns weights to code tokens based on their significance in the programming language syntax.
- The abstract syntax tree match compares the structural similarity by analyzing the abstract syntax tree representations of the generated and reference code.
- The data flow match evaluates the correctness of the data dependencies and variable usages in the code.

The overall CodeBLEU score is computed as the weighted sum of these components, offering a comprehensive assessment that accounts for the syntactic and semantic aspects of the code.

2) ROUGE-LCS

ROUGE-LCS (Recall-Oriented Understudy for Gisting Evaluation using Longest Common Subsequence) [51] is a metric that assesses the quality of the generated text by measuring the longest common subsequence between the generated output and the reference text. In this study, we employed the following three variants:

- ROUGE-LCS Precision: This variant is the ratio of the longest common subsequence length to the generated text length.
- ROUGE-LCS Recall: This variant is the ratio of the longest common subsequence length to the reference text length.
- ROUGE-LCS F-measure: This variant is the harmonic mean of ROUGE-LCS Precision and ROUGE-LCS Recall.

These metrics capture both the precision and recall aspects of the generated log messages, emphasizing the importance of both completeness and conciseness.

3) METEOR

METEOR (Metric for Evaluation of Translation with Explicit ORdering) [52] was initially developed for machine translation tasks. This metric that assesses the quality of the generated text by aligning it with the reference text at the word and phrase levels, considering exact word matches, stem matches, and synonym matches. The METEOR approach calculates a score based on the harmonic mean of the

precision and recall from these matches, placing a higher weight on the recall. An additional fragmentation penalty was applied to account for disordered or fragmented alignments, encouraging coherent and well-ordered outputs. This metric is suitable for this evaluation because it aligns well with human judgments by considering semantic similarities, providing a holistic assessment of the generated log messages.

B. RESEARCH QUESTIONS

This section presents the performance evaluation of the proposed method via comparison with the prior research. To ensure the reliability and robustness of this study, we re-implemented the method from the previous work. The following research questions guided this evaluation:

- **RQ 1)** How effective is integrating a source-code metric analysis and code-block division in improving the accuracy of generating automated log statements generation?
- **RQ 2)** How does the exclusion of specific source-code metrics (i.e., CC, MI, and LOC) affect the performance of the log generation model across log generation subtasks?

1) RESEARCH QUESTION 1

To assess the effectiveness of the proposed approach that integrates a source-code metric analysis and divides complex methods into code blocks, we conducted a series of experiments comparing the performance of the proposed model to baseline models and existing methods, including LEONID and ELogger. We focused on evaluating the accuracy, consistency, and contextual appropriateness of the generated log statements across various metrics, including position, level, and message accuracy.

The model was assessed on three subtask components: position, level and message accuracy. The experimental results in Table 3 reveal that the proposed approach significantly outperforms the baseline methods in predicting log statement characteristics. In Table 3, the symbols represent the accuracy of the predictions for each task. A check mark (✓) indicates a correct prediction, a dash (–) signifies that the prediction for that subtask was not considered for the score, and a cross (✗) indicates an incorrect prediction.

The proposed model achieved a position accuracy of 99.87%, significantly outperforming that of LEONID at 76.34% and ELogger at 80.61%. This high accuracy indicates that the integration of source-code metrics and the code-block decomposition process effectively identified optimal logging positions, regardless of the method structure or complexity. For predicting the correct logging level, the proposed model achieved an accuracy of 75.34%, which was higher than that of LEONID at 74.98% and ELogger at 72.95%. Dividing complex methods into smaller, more manageable blocks renders the context for determining appropriate log levels more distinguishable, enabling more relevant and context-specific log levels. Concerning message accuracy,

the proposed model reached 31.69%, an improvement over that of LEONID at 22.54% and ELogger at 27.54%. This improvement suggests that providing more focused code blocks enables generating more relevant log messages.

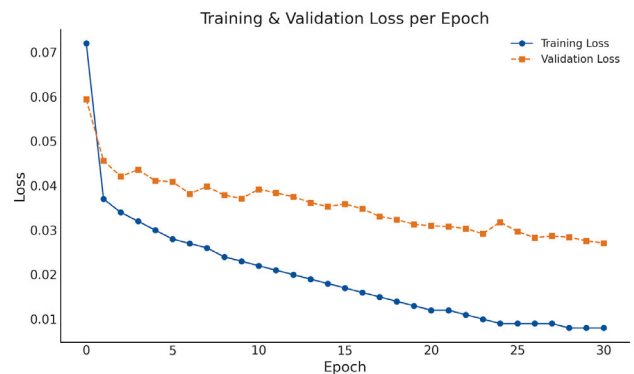


FIGURE 7. Training and validation loss.

Figure 7 validates the training process for the model that achieved these results. The figure plots the training and validation loss over 30 epochs, showing that the validation loss curve closely tracks the training loss. Both losses decreased steadily without significant divergence, which provides evidence that the model did not overfit the training data. Therefore, the high accuracy in predicting log position, level, and message can be attributed to the model's ability to robustly generalize, rather than simply memorizing the training set.

To further illustrate these results, Figs. 8 and 9 provide concrete examples of log statement predictions generated by the proposed model compared to ELogger and LEONID. In Fig. 8, the proposed model accurately predicted the position, log level, and message, fully matching the ground truth. In contrast, ELogger correctly predicted the position, but failed to generate an accurate log message, and LEONID missed the correct position and message. Similarly, in Fig. 9, our model again achieved a perfect match with the ground truth across all aspects. ELogger managed to predict the correct position but generated an imprecise message, whereas LEONID failed in both position and message prediction.

In certain instances, the proposed model produces log statements that deviate from the ground-truth logs yet offer more precise and informative messages. For example, as illustrated in Fig. 10, the ground-truth message states “Missing node for stat,” providing no additional details. In contrast, the model-generated log clarifies that the node may have been deleted or is otherwise unavailable, furnishing clearer diagnostic information. This observation underscores the capability of the model to approximate existing logs and enhance their clarity and utility.

To provide a more detailed evaluation of log level performance beyond simple accuracy, we conducted a log level distance analysis. We encoded the six log levels on an ordinal scale as follows: Trace (1), Debug (2), Info (3), Warn (4), Error (5), and Fatal (6). We then calculated the absolute

TABLE 3. Evaluation results for log generation: LEONID, ELogger, and Ours.

	Log generation subtask			LEONID	ELogger	Ours
	Position	Level	Message	Preds(%)	Preds(%)	Preds(%)
1 out of 3	✓	—	—	76.34	80.61	99.87
	—	✓	—	74.98	72.95	75.34
	—	—	✓	22.54	27.54	31.69
2 out of 3	✓	✓	—	62.18	53.64	75.27
	✓	—	✓	21.46	23.22	31.66
	✓	✓	✗	34.28	30.71	44.22
	✓	✗	✓	0.42	0.29	0.61
All	✓	✓	✓	21.45	19.55	25.23

<pre>try{ final JSONObject json = new JSONObject(line); if(json.has("error")){ logger.warn("Indexing stopping due to error:" +json); break loop; } final String id = json.getString("id"); state.writer.updateDocuments(new Term("id",id),docs); state.readerDirty=true; }catch(final JSONException e){ logger.error("JSON exception in changes loop",e); break loop; }</pre>	Ground-Truth	<pre>try{ final JSONObject json = new JSONObject(line); if(json.has("error")){ logger.warn("Indexing stopping due to error:" +json); break loop; } final String id = json.getString("id"); state.writer.updateDocuments(new Term("id",id),docs); state.readerDirty=true; }catch(final JSONException e){ logger.error("JSON exception in changes loop",e); break loop; }</pre>	Ours
<pre>try{ final JSONObject json = new JSONObject(line); if(json.has("error")){ logger.warn("Error encountered " +json); break loop; } final String id = json.getString("id"); state.writer.updateDocuments(new Term("id",id),docs); state.readerDirty=true; }catch(final JSONException e){ logger.error("JSON exception in changes loop",e); break loop; }</pre>	ELogger	<pre>try{ final JSONObject json = new JSONObject(line); if(json.has("error")){ break loop; } final String id = json.getString("id"); state.writer.updateDocuments(new Term("id",id),docs); state.readerDirty=true; }catch(final JSONException e){ logger.error("JSON exception in changes loop",e); log.error("Exception error",e); break loop; }</pre>	LEONID

FIGURE 8. Example of a correct log prediction (Case 1) [53].

<pre>if (now >= nextRefresh) { LOG.info("refreshing now because expiry is before next scheduled refresh time."); } else if (now < nextRefresh) { Date until = new Date(nextRefresh); LOG.info("TGT renewal sleeping until: {}", until.toString()); try { Thread.sleep(nextRefresh - now); } catch (InterruptedException ie) { LOG.warn("TGT renewal thread has been interrupted and will exit."); break; } }</pre>	Ground-Truth	<pre>if (now >= nextRefresh) { LOG.info("refreshing now because expiry is before next scheduled refresh time."); } else if (now < nextRefresh) { Date until = new Date(nextRefresh); LOG.info("TGT renewal sleeping until: {}", until.toString()); try { Thread.sleep(nextRefresh - now); } catch (InterruptedException ie) { LOG.warn("TGT renewal thread has been interrupted and will exit."); break; } }</pre>	Ours
<pre>if (now >= nextRefresh) { LOG.info("refreshing now because expiry is before next scheduled refresh time."); } else if (now < nextRefresh) { Date until = new Date(nextRefresh); LOG.info("Waiting for next refresh: {}", until); try { Thread.sleep(nextRefresh - now); } catch (InterruptedException ie) { LOG.warn("TGT renewal thread has been interrupted and will exit."); break; } }</pre>	ELogger	<pre>if (now >= nextRefresh) { LOG.info("refreshing now because expiry is before next scheduled refresh time."); } else if (now < nextRefresh) { Date until = new Date(nextRefresh); LOG.info("Sleeping for a while."); try { Thread.sleep(nextRefresh - now); } catch (InterruptedException ie) { LOG.warn("TGT renewal thread has been interrupted and will exit."); break; } }</pre>	LEONID

FIGURE 9. Example of a correct log prediction (Case 2) [54].

difference between the predicted level and the ground-truth level for each prediction. Figure 11 shows the distribution of these prediction distances. While the performance for correct predictions (Distance = 0) is comparable across models, our approach shows a clear advantage in error severity. Our model has a lower percentage of high-magnitude errors (Distance ≥ 3) compared to both baselines. This indicates that even when our model makes an incorrect level prediction, the error is more likely to be a minor one (i.e., closer in severity to the

correct level), demonstrating a more refined understanding of appropriate log levels.

We further compared the proposed method against LEONID and ELogger using CodeBLEU, ROUGE-LCS, and METEOR to evaluate the log statements quality beyond accuracy. As presented in Table 4, the proposed approach considerably outperformed the existing methods. The proposed approach achieved a CodeBLEU score of 60.39%, which was notably higher than that of LEONID

<pre>String statNode = Quotas.statPath(lastPrefix); node = getZKDatabase().getNode(statNode); StatsTrack currentStats; if (node == null) { LOG.error("Missing node for stat {}", statNode); return; }</pre>	<pre>String statNode = Quotas.statPath(lastPrefix); node = getZKDatabase().getNode(statNode); StatsTrack currentStats; if (node == null) { LOG.error("Failed to locate the stat node '{}'. " + "It may have been deleted or is otherwise unavailable.", statNode); return; }</pre>
Ground-Truth	Ours

FIGURE 10. Example of a generated log with improved information [54].

TABLE 4. Evaluation of generated log message quality: LEONID, ELogger, and Ours.

Metrics	LEONID(%)	ELogger(%)	Ours(%)
CodeBLEU-1	46.23	50.32	60.39
CodeBLEU-2	37.43	42.56	60.21
CodeBLEU-3	34.82	35.96	60.08
CodeBLEU-4	31.01	33.01	59.94
METEOR	54.31	51.45	77.61
ROUGH-LCS precision	45.23	61.58	97.74
ROUGH-LCS recall	41.95	58.74	83.12
ROUGH-LCS F-measure	43.53	60.13	87.94

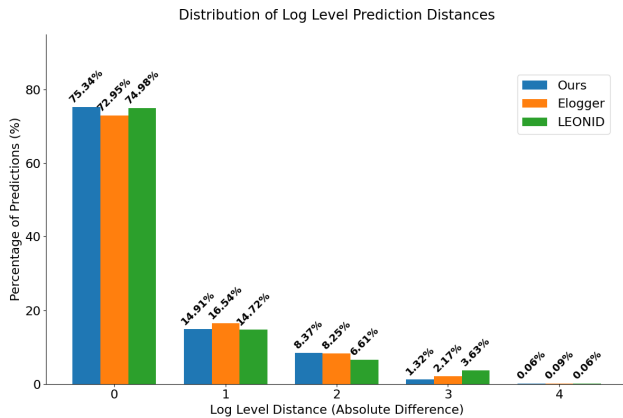


FIGURE 11. Distribution of absolute prediction distances for log level: Ours, ELogger, LEONID.

at 46.23% and ELogger at 50.32%. By splitting complex methods into smaller code blocks, the proposed model was better able to understand and generate code-specific logging, as evidenced by improvements in the abstract syntax tree and data flow matches. The proposed approach also achieved a ROUGE-LCS F-measure of 87.94%, significantly higher than that of ELogger at 60.13%. The enhanced recall and precision highlight the effectiveness of the proposed method in retaining relevant information and minimizing redundancy in generated log messages. The METEOR score of the proposed model was 77.61%, compared to that of LEONID at 54.31% and ELogger at 51.45%. By analyzing source-code metrics and using them as a basis for division, the proposed approach more effectively captured the semantics of the original methods, producing coherent and contextually appropriate logging.

The superior performance of the proposed model can be attributed to several factors. Unlike LEONID, which employs

the T5 model and requires pretraining on the programming language syntax, the proposed model applies the CodeT5+ model, which is pretrained on a significantly larger dataset and includes an enhanced understanding of source-code structures. The pretraining process in T5 may incorporate incorrect code patterns or erroneous syntax, affecting the quality of the generated logs. In contrast, CodeT5+ benefits from a more extensive and accurate dataset, improving precision and contextual understanding in log generation. Additionally, ELogger's restricted method of dividing code blocks, limited to specific constructs (e.g., try-catch blocks, loops, and method declarations) hinders its generalizability and effectiveness. The systematic approach of the proposed model to splitting complex methods into smaller, context-aware blocks offers a more versatile and robust framework for log generation, handling diverse and intricate coding scenarios.

These results demonstrate that integrating a source-code metrics analysis, combined with systematic code-block division, substantially enhances the overall accuracy and effectiveness of automated log generation. The improvements in position, level, and message accuracy across a wide variety of complex methods affirm the scalability and robustness of the proposed approach. Furthermore, by avoiding rigid, construct-specific division as in ELogger, the proposed method is more general and adaptable solution for log generation in real-world software systems.

2) RESEARCH QUESTION2

The exclusion of CC, LOC or MI impacts the predictive performance of the log generation model differently across subtasks in Table 5.

In the subtask that requires correctly predicting one of the three log attributes (position, level, or message), the exclusion

TABLE 5. Evaluation results of the ablation study for log generation: w/o CC, w/o MI, w/o LOC, and Ours. Note: w/o indicates ‘without’.

	Log generation subtask			w/o CC	w/o MI	w/o LOC	Ours
	Position	Level	Message	Preds(%)	Preds(%)	Preds(%)	Preds(%)
1 out of 3	✓	–	–	81.45	88.90	96.56	99.87
	–	✓	–	70.48	72.65	73.83	75.34
	–	–	✓	23.95	26.87	29.78	31.69
2 out of 3	✓	✓	–	68.33	71.12	73.83	75.27
	✓	–	✓	24.12	26.67	29.21	31.66
	✓	✓	✗	34.59	38.82	43.05	44.22
	✓	✗	✓	0.39	0.46	0.53	0.61
All	✓	✓	✓	18.85	21.21	23.57	25.23

of CC results in the greatest performance degradation. For the position attribute, accuracy decreases from 99.87% to 81.45%. Excluding MI reduces the accuracy to 88.90%, and excluding LOC results in a smaller reduction to 96.56%. For the level attribute, excluding CC reduces the accuracy from 75.34% to 70.48%. Excluding MI lowers the accuracy to 72.65%, and LOC exclusion reduces it to 73.83%. For the message attribute, excluding CC causes the most decline, from 31.69% to 23.95%. Excluding MI reduces the accuracy to 26.87%, and excluding LOC decreases the accuracy to 29.78%. These results reveal that CC is critical for predicting all attributes, with the message attribute being the most sensitive to its exclusion.

In the subtask that involves predicting combinations of two out of three attribute, such as position and level, position and message, or position and level with an incorrect message, the exclusion of CC causes the most significant performance decline. For position and level, the accuracy decreases from 75.27% to 68.33% when CC is excluded. Excluding MI lowers the accuracy to 71.12%, and LOC exclusion reduces it slightly to 74.26%. For position and message, CC exclusion reduces the accuracy from 44.22% to 34.59%. Moreover, MI exclusion lowers the accuracy to 38.82%, and LOC exclusion decreases it to 43.05%. For position and level with an incorrect message, the accuracy drops drastically when CC is excluded, from 31.66% to 24.12%. Excluding MI reduces the accuracy to 26.67%, and excluding LOC has a smaller effect, lowering the accuracy to 29.21%.

For the all attribute subtask, where all three log attributes must be correctly predicted, the performance degrades more drastically, highlighting the increased difficulty of the task. Removing CC reduces the overall accuracy from 25.23% to 18.85%, and excluding MI and LOC reduces the accuracy to 21.21% and 23.57%, respectively.

Across all subtasks, CC consistently exhibits the most influence on performance, particularly in tasks with higher complexity, such as the two out of three and all attributes, where combinations or full predictions are required. Additionally, MI contributes significantly, but its exclusion causes less severe degradation. Further, LOC has the least effect overall. These results emphasize the complementary roles of CC, MI, and LOC in supporting the predictive performance of the model, with CC being the most critical

metric for achieving accurate log predictions across all subtasks.

V. THREATS TO VALIDITY AND FUTURE WORK

A. THREATS TO VALIDITY

This section discusses the potential threats to the validity of this study.

1) INTERNAL VALIDITY

A threat to the internal validity stems from the composition of the final training dataset. After decomposing complex methods, we excluded any resulting code blocks that did not contain a log statement. This approach creates a potential selection bias, because the model is only trained on positive examples: code segments where a developer had already decided logging was necessary. Thus, the high performance of the model, particularly in predicting log positions, might have been influenced by the fact that the model never has to learn to distinguish between code that requires a log and code that does not.

Another potential threat is the threshold selection for method decomposition. Although these thresholds were empirically derived from analyzing trends in the dataset (as shown in Figures 2, 3, and 4), they may not be universally optimal. Different thresholds could lead to different decomposition results and, model performance.

2) EXTERNAL VALIDITY

External validity refers to the generalizability of these findings. The dataset comprises open-source Java projects from GitHub. Logging practices and code complexity patterns may differ in industrial, closed-source systems or in projects written in other programming languages (e.g., Python or C++). Therefore, the results may not be directly transferable to other contexts.

3) CONSTRUCT VALIDITY

Construct validity addresses whether the measurements accurately reflect the concepts they are intended to represent. A threat is employing CC, LOC, and MI as proxies for “code complexity.” Although these are well-established and widely used metrics, they primarily capture structural and size-based complexity. They may not fully represent other

facets (e.g., cognitive or domain-specific complexity) that could influence a developer's logging decisions.

B. FUTURE WORK

To address the limitations identified in this study, we propose the following directions for future work.

First, to overcome the selection bias in the dataset, we plan to construct a more balanced dataset that includes methods with and without existing log statements. This approach would allow training a more comprehensive model capable of generating logs and determining whether a given code block requires logging.

Second, to enhance the generalizability of the findings, we aim to extend this study to include a wider range of contexts. This extension involves creating new datasets from other popular programming languages (e.g., Python and C++) and analyzing proprietary code from industrial partners to validate this approach in different development environments.

Third, we aim to refine the definition and measurement of code complexity. We plan to investigate integrating additional metrics, such as the Halstead volume and cognitive complexity, to build a more detailed understanding of what makes code difficult to maintain. Furthermore, we aim to incorporate code smell detection to identify specific anti-patterns (e.g., long methods and large classes) that could serve as strong indicators for necessary logging.

Fourth, this study utilized a fixed set of standard hyperparameters for model training. A valuable direction for future work would be to conduct a comprehensive hyperparameter sensitivity analysis. Exploring a wider range of values for the learning rate, weight decay, batch size, and number of training epochs could potentially lead to further performance optimizations and provide deeper insights into the model's training dynamics.

VI. CONCLUSION

This study proposes a novel method for generating automated log statements by integrating source-code metrics analysis into the model training process. By analyzing Java methods using source-code metrics (CC, MI, and LOC) we identified complex methods and divided them into smaller, manageable code blocks. This decomposition allowed the fine-tuned CodeT5+ model to generate more accurate and contextually appropriate logging statements across code structures.

The experimental results demonstrated that this approach considerably enhances the accuracy and effectiveness of automated log generation. The model achieved an overall accuracy of 25.23%, indicating a substantial improvement in generating precise logging statements. By focusing on source-code metrics and adapting the training data, the proposed method offers a general and adaptable solution for log generation in real-world software systems, addressing challenges associated with complex code structures and contributing to more efficient software maintenance and debugging.

REFERENCES

- [1] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? An empirical study on logging practices in industry," in *Companion Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 24–33.
- [2] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2012, pp. 293–306.
- [3] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Softw. Eng.*, vol. 22, no. 4, pp. 1831–1865, Aug. 2017.
- [4] G. Rong, S. Gu, H. Zhang, D. Shao, and W. Liu, "How is logging practice implemented in open source software projects? A preliminary exploration," in *Proc. 25th Australas. Softw. Eng. Conf. (ASWEC)*, Nov. 2018, pp. 171–180.
- [5] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Softw. Eng.*, vol. 22, no. 4, pp. 1684–1716, Aug. 2017.
- [6] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, Feb. 2012.
- [7] *Eladmin*. Accessed: Oct. 19, 2025. [Online]. Available: <https://github.com/elunez/eladmin>
- [8] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Trans. Comput. Syst. (TOCS)*, vol. 30, pp. 1–28, Feb. 2012.
- [9] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should I log?" *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 2012–2031, Sep. 2019.
- [10] Z. Ding, H. Li, and W. Shang, "LoGenText: Automatically generating logging texts using neural machine translation," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2022, pp. 349–360.
- [11] H. Li, T.-H. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 2655–2694, Oct. 2018.
- [12] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "SMARTLOG: Place error log statement by deep understanding of log intention," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2018, pp. 61–71.
- [13] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 415–425.
- [14] 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 *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, Mar. 2018, pp. 127–138.
- [15] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? Studying and suggesting logging locations in code blocks," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2020, pp. 361–372.
- [16] J. Cândido, J. Haesen, M. Aniche, and A. van Deursen, "An exploratory study of log placement recommendation in an enterprise system," in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR)*, May 2021, pp. 143–154.
- [17] Z. Li, "Towards providing automated supports to developers on writing logging statements," in *IEEE/ACM 42nd Int. Conf. Softw. Engineering, Companion Proc. (ICSE-Companion)*, Oct. 2020, pp. 198–201.
- [18] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 102–112.
- [19] T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue, "PADLA: A dynamic log level adapter using online phase detection," in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, May 2019, pp. 135–138.
- [20] Z. Li, H. Li, T.-H. Chen, and W. Shang, "DeepLV: Suggesting log levels using ordinal based neural networks," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1461–1472.
- [21] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 2279–2290.
- [22] A. Mastropaolo, V. Ferrari, L. Pascarella, and G. Bavota, "Log statements generation via deep learning: Widening the support provided to developers," *J. Syst. Softw.*, vol. 210, Apr. 2024, Art. no. 111947.

- [23] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.
- [24] Y. Fu, M. Yan, P. He, C. Liu, X. Zhang, and D. Yang, "End-to-end log statement generation at block-level," *J. Syst. Softw.*, vol. 216, Oct. 2024, Art. no. 112146.
- [25] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. Kun Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," 2020, *arXiv:2009.08366*.
- [26] C. Zhi, J. Yin, S. Deng, M. Ye, M. Fu, and T. Xie, "An exploratory study of logging configuration practice in Java," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2019, pp. 459–469.
- [27] H. Shu, D. Wang, A. Mastropaolo, G. Bavota, and Y. Kamei, "An empirical study on language models for generating log statements in test code," *ACM Trans. Softw. Eng. Methodol.*, Aug. 2025.
- [28] X. Li, Y. Huo, C. Mao, S. Shan, Y. Su, D. Li, and Z. Zheng, "AnomalyGen: An automated semantic log sequence generation framework with LLM for anomaly detection," 2025, *arXiv:2504.12250*.
- [29] S. Lal and A. Sureka, "LogOpt: Static feature extraction from source code for automated catch block logging prediction," in *Proc. 9th India Softw. Eng. Conf.*, India, Feb. 2016, pp. 151–155.
- [30] S. Gholamian, "Leveraging code clones and natural language processing for log statement prediction," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 1043–1047.
- [31] V. Bushong, R. Sanders, J. Curtis, M. Du, T. Cerny, K. Frajtak, M. Bures, P. Tisnovsky, and D. Shin, "On matching log analysis to source code: A systematic mapping study," in *Proc. Int. Conf. Res. Adapt. Convergent Syst.*, Oct. 2020, pp. 181–187.
- [32] S. Lal, N. Sardana, and A. Sureka, "Two level empirical study of logging statements in open source Java projects," *Int. J. Open Source Softw. Processes*, vol. 6, no. 1, pp. 49–73, Jan. 2015.
- [33] S. Lal, N. Sardana, and A. Sureka, "Analysis and prediction of log statement in open source Java projects," in *Proc. Doctoral Consortium 13th Int. Conf. Open Source Syst. (OSS)* (Skövde University Studies in Informatics). Skövde, Sweden: University of Skövde, Jan. 2017, pp. 65–80.
- [34] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Softw. Eng.*, vol. 20, no. 1, pp. 1–27, Feb. 2015.
- [35] M. A. Batoun, M. Sayagh, and A. Ouni, "How much logs does my source code file need? Learning to predict the density of logs," in *Proc. 28th Int. Conf. Eval. Assessment Softw. Eng.*, Jun. 2024, pp. 140–149.
- [36] Y. Li, Y. Huo, Z. Jiang, R. Zhong, P. He, Y. Su, L. Briand, and M. R. Lyu, "Exploring the effectiveness of LLMs in automated logging generation: An empirical study," 2023, *arXiv:2307.05950*.
- [37] Y. Wang, H. Le, A. Deepak Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "CodeT5+: Open code large language models for code understanding and generation," 2023, *arXiv:2305.07922*.
- [38] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
- [39] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. 27, no. 1, pp. 42–52, 1984.
- [40] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: A software science validation," *IEEE Trans. Softw. Eng.*, vol. SE-9, no. 6, pp. 639–648, 1983.
- [41] L. C. Briand, J. Wüst, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proc. IEEE Int. Conf. Softw. Maintenance (ICSM)*, Oct. 1999, pp. 475–482.
- [42] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proc. Conf. Softw. Maintenance*, 1992, pp. 337–344.
- [43] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, Nov. 1993.
- [44] M. Halstead, *Elements of Software Science* (Operating and programming systems series). Amsterdam, The Netherlands: Elsevier, 1977.
- [45] Microsoft. (2024). *Maintainability Index Range and Meaning*. Accessed: Mar. 2024. [Online]. Available: <https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022>
- [46] *Simple Logging Facade for Java (SLF4J)*. Accessed: Oct. 19, 2025. [Online]. Available: <https://www.slf4j.org/>
- [47] Log4J. *Apache LOG4J*. Accessed: Oct. 19, 2025. [Online]. Available: <https://logging.apache.org/log4j/2.x/>
- [48] SciTools. *Scitools Website*. Accessed: Oct. 19, 2025. [Online]. Available: <https://scitools.com/>
- [49] *Scala-Maven-Plugin*. Accessed: Oct. 19, 2025. [Online]. Available: <https://github.com/davidB/scala-maven-plugin>
- [50] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: A method for automatic evaluation of code synthesis," 2020, *arXiv:2009.10297*.
- [51] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Proc. Text Summarization Branches Out*, 2004, pp. 74–81.
- [52] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. ACL Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 65–72.
- [53] rnewson. *Couchdb-Lucene*. Accessed: Oct. 19, 2025. [Online]. Available: <https://github.com/rnewson/couchdb-lucene>
- [54] apache. *Zookeeper*. Accessed: Oct. 19, 2025. [Online]. Available: <https://github.com/apache/zookeeper>



SE-JIN KIM received the B.S. and M.S. degrees in computer science and engineering from Chung-Ang University, Seoul, South Korea, in 2023 and 2025, respectively. He is currently a Software Developer.



CHAN-GUN LEE was born in Seoul, South Korea, in 1972. He received the B.S. degree in computer engineering from Chung-Ang University, Seoul, in 1996, the M.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 1998, and the Ph.D. degree in computer science from The University of Texas at Austin, Austin, TX, USA, in 2005. From 2005 to 2007, he was a Senior Software Engineer with Intel, Hillsboro, OR, USA. Since 2007, he has been a Professor with the Department of Computer Science and Engineering, Chung-Ang University. His research interests include software engineering and real-time systems. He was a recipient of Korea Foundation of Advanced Studies (KFAS) Fellowship, from 1999 to 2005.

...