LogGen : Intelligent Logging Statement Generation for Java Program Files

Yogesh Hasabe*
yhasabe@ncsu.edu
NC State University
Raleigh, NC, USA

Jay Patel*
jhpatel9@ncsu.edu
NC State University
Raleigh, NC, USA

Ashaka Mehta*
atmehta@ncsu.edu
NC State University
Raleigh, NC, USA

Sagar Dama*
sudama@ncsu.edu
NC State University
Raleigh, NC, USA

Abstract

In software development, logging statements are essential because they capture important runtime data, facilitating system observability, debugging, and maintenance. But figuring out where and how to log efficiently is still a constant struggle because too much logging overwhelms developers and resources, while too little logging hides problems. In order to forecast the best logging position and produce high-quality, contextually relevant logging statements, this paper introduces LogGen, an intelligent, automated solution for Java programs that makes use of Large Language Models (LLMs). LogGen first finds appropriate locations and then, without changing the underlying code structure, produces relevant log messages. The system performs better than current techniques when measured using metrics like BLEU and ROUGE. By enhancing logging practices, LogGen improves software maintainability, observability, and developer productivity, marking a significant step forward in automated code instrumentation.

Keywords

Logging automation, Large Language Models, Java programs, log generation

1 Introduction

Software development has reached a critical juncture where logging logging procedures have a big influence on system observability, maintenance, and overall program quality. Recent studies have repeatedly emphasized the significance of strategic logging, such as the groundbreaking work by Zhang et al. [6] and the thorough investigation by Li et al. [7]. They provide an essential theoretical framework for our research and further highlight the importance of integrating file-level programming environments to improve logging techniques.

The current landscape of logging is fraught with challenges. Modern software systems frequently struggle with two extremes: insufficient logging obscuring system behavior or excessive logging that overwhelms developers and system resources. Existing large language models (LLMs) like GPT, Claude, and Llama, while powerful, often need to catch up in generating logging statements. This limitation is corroborated by the research of Vishwakarma et al. [3], which demonstrates the complexity of automated code generation. Existing log analysis tools such as Dynatrace and Splunk provide robust post-logging insights. Yet, there still needs to be a critical gap in intelligent, proactive logging statement generation, as highlighted by the work of Chen et al. [8].

LogGen addresses these challenges by focusing on what research has identified as the three fundamental questions of high-quality logging: What to log, Where to log, and How to log. Drawing from the comprehensive findings presented by Gholamian and Ward [5], our approach precisely predicts optimal logging positions and generates contextually appropriate logging statements. The project aims to significantly improve

software maintainability and developer productivity by ensuring that logging statements are informative, strategically placed, and crucially do not alter the underlying code structure. This approach aligns with the emerging paradigm of context-aware software engineering tools that leverage advanced AI techniques.

The potential impact of LogGen extends across multiple domains of software engineering. Software developers will benefit from reduced manual logging overhead. DevOps engineers can achieve more precise system monitoring, quality assurance teams can more effectively trace bugs and errors, and software security teams can implement more robust observability mechanisms. By solving the intricate challenges of precise logging position prediction and high-quality statement generation while maintaining code integrity, LogGen represents a significant advancement in automated logging technology.

Ultimately, LogGen is not merely a tool but a paradigm shift in how we approach software logging. By leveraging advanced AI techniques and a deep understanding of software engineering principles, we aim to transform logging from a mundane, error-prone task to an intelligent, context-aware process that genuinely enhances software quality and developer efficiency. The research underpinning this project, including studies on logging practices and context-aware code analysis, provides a robust theoretical foundation for our innovative intelligent logging statement generation approach.

The remainder of this paper is organized as follows. We begin by surveying the current state-of-the-art techniques used in the industry, focusing on automated logging solutions. Next, we detail our proposed approach, including the methodology for dataset generation, the selection of the LLM model for experimentation, and the experimental setup. We then present the experimental results, comparing LogGen's performance to existing solutions, and conclude with a discussion on limitations and directions for future work. This structured flow provides a comprehensive overview of the development and evaluation of our intelligent logging solution.

1.1 Literature Review

The study by Zhu et al. [18] proposes LogAdvisor, a machine learning-based framework designed to assist developers in determining optimal logging locations in source code. By analyzing large datasets of logging practices from industrial and open-source projects, the system learns contextual features such as structural, textual, and syntactic characteristics of code snippets. LogAdvisor uses these features to train predictive models, achieving high accuracy in suggesting where to log during new development. Its evaluation of 19.1 million lines of code and over 100,000 logging statements demonstrates its ability to replicate expert logging decisions effectively. Despite its strengths, LogAdvisor focuses on where to log, leaving aspects of what to log as future work. The framework handles logging location choices; it does not handle recording material comprehensively, leaving it up to the developer. Its adaptability to C# codebases may also limit its applicability to other programming languages.

LoGenText, proposed by Ding et al. [4], is an automated approach that generates logging texts by translating related source code into short textual descriptions using neural machine translation models. The authors evaluate LoGenText on ten open-source Java projects, comparing it to a baseline approach that reuses logging texts from similar code snippets. LoGenText achieves BLEU scores of 23.3 to 41.8 and ROUGE-L scores of 42.1 to 53.9, outperforming the baseline by a significant margin. The study also explores the impact of incorporating context information, such as the abstract syntax tree (AST) structure and post-log code, finding that these additions can improve performance. A human evaluation involving 42 participants confirms the quality of the logging texts generated by LoGenText. However, the results are influenced by subjective human ratings and fixed hyper-parameter configurations, which may only generalize well with further fine-tuning.

Despite the fact that earlier recording techniques have demonstrated encouraging outcomes across a range of subtasks, they do not consider the logging tasks as a whole. Xu et al. [17] proposed UniLog is an innovative automatic logging framework that leverages the in-context learning (ICL) paradigm of large language models (LLMs) to generate appropriate logging statements. Unlike previous methods focused on individual subtasks or required extensive fine-tuning, UniLog offers an end-to-end solution that simultaneously determines logging positions, predicts verbosity levels, and generates log messages. The framework utilizes Codex, an LLM pre-trained on programming language tasks, as its backbone and employs a specially designed

line-aware prompt format. UniLog can generate logging statements with only five demonstration examples in the prompt without model tuning. It can also enhance its performance using just 500 random samples through a warmup mechanism. Evaluated on a dataset of 12,012 code snippets from 1,465 GitHub repositories, UniLog achieved state-of-the-art performance in automatic logging, with 76.9% accuracy in selecting logging positions, 72.3% accuracy in predicting verbosity levels, and a 27.1 BLEU-4 score in generating log messages. However, it has certain drawbacks since evaluation may be vulnerable to data leaking because of the overlap between the 2022 dataset and Codex's pretraining data (up to May 2020). Furthermore, performance variability may be introduced by randomness in sample selection and inference, necessitating several experiments to settle results.

Mastropaolo et al. [9] proposed LANCE, a machine-learning technique to automate Java program logging tasks. To solve logging practice issues, including deciding where to insert logs, choosing the right log level, and producing insightful log messages, this system uses the Text-to-Text Transfer Transformer (T5) model. In order to give the model Java-specific information, the authors pre-trained and optimized T5 on a dataset of more than 6.8 million Java methods using tasks like masked token prediction and log statement location. According to the evaluation, LANCE could select the appropriate log level 66.2% of the time, generate complete log statements with insightful messages in 15.2% of cases, and correctly predict logging locations 65.9% of the time. Despite its promising results, there were certain limitations, such as modifications to non-log code and the significant delay caused by regenerating entire methods when introducing logs. These problems affect efficiency and dependability.

A two-stage method that creates comprehensive logging statements for Java applications and predicts ideal logging sites is shown in the work by Xie et al. [16]. Key problems with the previous method, LANCE, like slowness and inadvertent changes to non-log code, are addressed with FastLog. FastLog guarantees quicker and more accurate log creation without changing the program's substance by using a Seq2Seq model to generate entire logging statements and token-level classification to anticipate exact insertion sites. Despite these advancements, the method's limitation to method-level actions makes it unsuitable for whole-code file-logging tasks. Its reliance on pre-trained models, like PLBART, which may not function well if not appropriately tailored for certain scenarios, raises questions about a potential obstacle to wider adoption in a variety of contexts.

2 Approach

Our approach is driven by a set of key questions that guide our exploration and evaluation. Our aim is to check the effectiveness of the state-of-the-art LLM models to accurately assist in predicting the logging position and then generate high-quality logging statements at those positions. To address this, we propose a novel automated log generation pipeline that leverages the capabilities of large language models (LLMs) in a two-stage design as shown in the Figure 1. The pipeline focuses on identifying ideal positions for logging and generating high-quality and contextually relevant logging statements. Stage One: Logging Position Prediction The first stage employs an LLM to analyze the input code and predict the optimal positions for inserting logging statements. The model identifies locations where logs are likely to enhance observability and debugging. These positions are marked with a placeholder, `<FILL_ME>`, which serves as an indicator for the subsequent stage. By narrowing down the insertion points in this stage, we set the foundations for generating specific and meaningful logs in the next phase.

Stage Two: Logging Statement Generation

In the second stage, the annotated code from Stage One, now containing `<FILL_ME>` placeholders, is passed through another LLM tasked with generating the actual log statements. This model uses the surrounding context of each placeholder to generate logs that are not only syntactically accurate but also semantically rich.

For example, generated logs can include key variable values, flow details, or error messages that align with the code functionality. These logs are intended to adhere to best practices, ensuring clarity, brevity, and relevance to software debugging and monitoring needs.

2.1 Dataset Generation

To create a robust dataset for our automated log generator project, we selected 10 Java repositories from GitHub that meet the following criteria:

Popularity: Repositories must not be forked and must

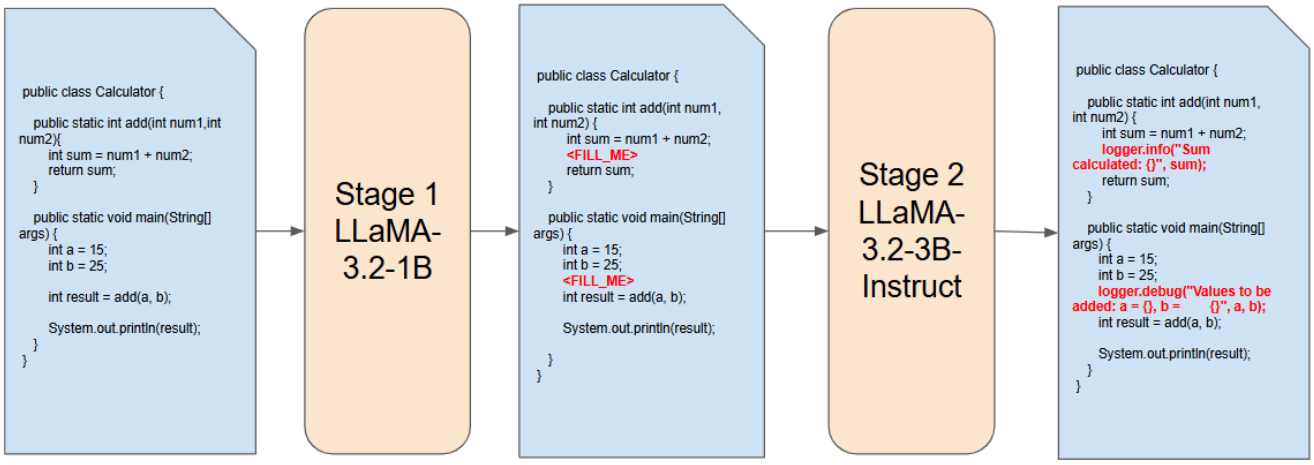

Figure 1: Key Design

have at least 100 stars, ensuring quality and community engagement.

Logging Dependency: Repositories must incorporate a Java logging library, such as Log4j, to align with the focus of our project.

Recency: Repositories must have been created between September 1, 2021, and May 1, 2024, to ensure the inclusion of modern coding practices and frameworks.

From these repositories, approximately 3,300 Java files were scraped and consolidated into our final dataset. The dataset was split into training, validation, and testing sets in a 60:20:20 ratio to support the development and evaluation of the two-stage pipeline. This distribution ensures that the model is trained on a large portion of the data while preserving enough unseen samples for validation and testing, thereby preventing over-fitting and ensuring generalization. Again the logic to generate the dataset for state 1 and stage 2 is different for both use cases.

2.1.1 Stage 1. The main task of Stage 1 is to take code as input (without any logs) and accurately identify the line positions in the code where logs need to be added. For our Stage 1 dataset, we created a CSV file with the following columns: **Original Code**: The original code with pre-processing done to remove unwanted empty lines and import statements.

Input Code: The original code with all logging statements completely removed

Output Label 1: This output label is an array of 1's and 0's. A 1 indicates that the next token marks the start of a logging statement, while 0 indicates non-logging content.

Output Label 2: This output label is an array of 1's and 0's. Since it's not feasible to create a single token for the entire logging statement, we use a series of 1's to represent a continuous logging statement. This helps the model better understand tokenization and how tokens are processed.

Output Label 3: Inspired by previous state-of-the-art solutions such as Fast-Log, we created a similar output label. It consists of 1's, 0's, and -1000's, where: 0 indicates a non-logging token, 1 marks the starting token of a logging statement, and -1000 marks the rest of the logging statement tokens after the token labeled with 1. This labeling technique helps the model focus on the position of the start of a logging statement, assigning a higher weight to the start token (1) and a lower weight to the rest of the logging tokens (-1000), ensuring better input context and output accuracy.

2.1.2 Stage 2. Stage 2 dataset is less complicated and relatively simple in comparison to stage 1. Stage 2 is driven by the output generated by the stage 1 as the logging position generated by stage 1 is used to modify the original code without logs by adding `<FILL_ME>` placeholders at those positions.Following are the columns of the csv file for stage 2

Input Code: Original code with logs replaced by <FILL_ME> placeholders.
Label: Original code with original logging statements.

3 Models Considered

3.1 CodeT5

CodeT5 [14], based on the T5 (Text-to-Text Transfer Transformer) architecture, is fine-tuned for code-related tasks, excelling in understanding and generating programming language structures. Trained on large datasets like CodeSearchNet, it can handle a variety of code-related tasks, including code summarization, code completion, defect detection, and code translation. CodeT5 treats these tasks as text-to-text problems, enabling it to generate meaningful code comments, identify potential errors, and suggest code completions based on contextual understanding.

Its ability to bridge natural language and programming syntax makes it invaluable for use cases such as generating documentation, writing test cases, and refactoring code. Additionally, it supports multiple programming languages, making it adaptable for polyglot codebases. CodeT5's versatility and fine-tuning capabilities make it a powerful tool for enhancing productivity in software development workflows.

3.2 Llama-3.2-1B

The LLaMA-3.2-1B model [10], part of the Large Language Model Meta AI (LLaMA) family, is a compact yet powerful transformer-based architecture designed for efficiency and versatility across a variety of tasks. With 1 billion parameters, it strikes an ideal balance between computational efficiency and robust performance. Its architecture employs advanced mechanisms like rotary positional embeddings and optimized attention layers, enabling precise understanding and generation of both structured and unstructured data.

LLaMA-3.2-1B demonstrates strong capabilities in text-related tasks such as summarization, translation, and question answering, as well as code-related tasks like code completion, syntax error detection, and debugging. Its training on diverse datasets, including large-scale programming language corpora, allows it to capture nuanced syntax and logical patterns, making it adept at understanding programming context. The model is particularly valuable in bridging natural language with programming languages, supporting applications like documentation generation, automated testing, and intelligent code suggestions. Its lightweight design ensures effective deployment even in resource-constrained environments, making it an accessible yet powerful tool for both researchers and developers.

3.3 LLaMA-3.3-3B-Instruct

The LLaMA-3.3-3B-Instruct model [10] is a variant of the LLaMA architecture, specifically fine-tuned for instruction-based tasks. With 3 billion parameters, it leverages instruction-following techniques to enhance its ability to generate high-quality responses based on user-provided instructions. This makes it highly effective for a variety of tasks, particularly in the realm of code generation, code translation, and debugging. It can generate code snippets, translate code between different languages, and identify errors with high accuracy. Additionally, the model excels in generating natural language explanations of code, which is useful for documentation, summarization, and teaching purposes.

LLaMA-3.3-3B-Instruct's ability to follow detailed instructions allows for precise control over its output, making it highly adaptable to specific programming needs. Its versatility in handling both text and code-related tasks, combined with multi-language support, positions it as a valuable tool for software development, improving productivity in diverse coding environments and for various developer skill levels.

3.4 CodeLlama

CodeLlama [13] is a specialized model built by Meta, designed specifically for code-related tasks. Built on the foundation of the LLaMA architecture, CodeLlama is fine-tuned on large-scale datasets containing programming languages, enabling it to handle a wide range of coding challenges. With its architecture optimized for both understanding and generating code, CodeLlama excels at tasks like code completion, debugging, code summarization, and code translation.

The model is highly effective in generating high-quality code snippets, suggesting improvements, and identifying errors in code. Its ability to understand programming syntax and logic enables it to generate meaningful explanations of code and assist in software documentation. Additionally, CodeLlama's multi-language

support makes it versatile for projects involving diverse programming languages. Its fine-tuned instruction-following capabilities allow developers to use natural language prompts to generate contextually appropriate and efficient solutions, enhancing both productivity and code quality.

4 Experimental Design

A series of experiments were conducted, each incorporating incremental modifications to the solution architecture. The performance of each experiment was carefully observed, and potential limitations were identified through thorough analysis. These insights informed iterative adjustments, enabling us to refine the approach in subsequent experiments to enhance performance.

4.1 Stage 1

Initially, we trained CodeLlama as the stage 1 model. In this setup, the input comprised the entire Java code file, while the output was a sequence of labeled tokens. A label of 1 indicated that a log statement should follow a token, whereas 0 indicated otherwise. However, one major challenge in fine-tuning CodeLlama was its constrained attention window, which was not suitable for our problem as a code file contains large number of tokens, prompting us to explore alternatives, including GPT.

As the next step, we fine-tuned GPT, beginning with GPT-3.5-turbo on a dataset containing 3,000 samples. While GPT demonstrated better performance than CodeLlama, fine-tuning large token sets with GPT proved to be significantly more expensive. Despite its effectiveness, the high cost led us to explore other open-source options.

In a subsequent experiment, we fine-tuned the Llama-3.2-1B model for stage 1, using a similar methodology to that employed with CodeLlama and GPT. Llama-3.2-1B outperformed CodeLlama, primarily due to its extended attention span, offering a more cost-effective solution while maintaining strong performance.

4.2 Stage 2

In stage 2, we fine-tuned the CodeT5 model. This model was fine-tuned to replace the `<FILL_ME>` tokens working as log-statement placeholders with contextually relevant log statements. We observed that CodeT5 hallucinated while producing results.

So, in our next experiment we used the Llama-3.2-3B-Instruct model without fine tuning using the same approach as CodeT5. It performed comparatively better than CodeT5. So, we fine-tuned this model using a small amount of data to prevent any loss in its original efficiency and at the same time make it capable enough to generate high quality logging statements.

5 Evaluation Metrics

5.1 Stage 1: Accuracy in Predicting Logging Positions

In Stage 1, the accuracy metric assesses the model's ability to predict logging positions within Java files correctly. Accuracy at the file level demonstrates the model's capacity to comprehend logging patterns holistically, ensuring that the generated log placements align with expected behavior. This metric is a pivotal indicator of how well the model generalizes logging conventions across diverse Java files.

5.2 Stage 2: BLEU and ROUGE Metrics

Stage 2 of the evaluation leverages BLEU [12] and ROUGE [2] metrics to assess the quality of log statements generated across multiple Java files. These metrics quantify the overlap between generated log placements and reference logs, providing insights into fluency, adequacy, and precision.

BLEU Score: The BLEU score measures n-gram overlap between generated logs and reference logs, considering brevity penalties for shorter predictions. A higher BLEU score indicates better alignment and relevance to the reference. ROUGE-1 and ROUGE-L Metrics

ROUGE-1 Precision: Evaluates the fraction of correctly generated unigrams among all generated tokens.

ROUGE-1 Recall: Measures how many reference unigrams are present in the generated text.

ROUGE-1 F-Measure: Provides a harmonic mean of precision and recall for unigrams.

ROUGE-L Precision: Focuses on the longest common subsequences (LCS) to capture structure-level similarity.

ROUGE-L Recall: Measures recall based on LCS alignment between generated and reference logs.

ROUGE-L F-Measure: Combines precision and recall for LCS, highlighting sequence alignment quality.

These metrics provide a multi-faceted evaluation of the system's ability to generate logs that closely resemble human-written logs, ensuring semantic correctness and contextual relevance.

In addition to assessing the quality of generated logging statements using BLEU & ROUGE scores, we evaluated the similarity/closeness of the logging levels produced by our Stage 2 model. During dataset creation, we extracted both the logging statements and their corresponding log levels from the original files. The top three logging levels observed across most Java files in our dataset were Error, Warn, and Info.

To measure performance, we mapped the textual similarity of the generated logs using BLEU scores and compared the generated log levels with those in the dataset. The BLEU scores were significantly higher compared to state-of-the-art methods like FastLog. This improvement can be attributed to two factors:

Context Awareness: Our model effectively captures a broader code context, enhancing the relevance of generated logs.

Advanced Base Model: Unlike FastLog, which relies on PLBART, our model leverages a more advanced language model for log generation.

6 Experimental Results

6.1 Stage 1: Log Position Prediction Evaluation

In Stage 1, the LLaMA-3.2-1B model achieved an accuracy of 36.84% in predicting Java code logging positions at the file level. This highlights its ability to discern logging patterns across a broader file context, albeit with a performance gap compared to state-of-the-art methods. For instance, FastLog and LANCE achieved 58.84% and 48.69% accuracy, respectively. However, our approach uniquely tackles the challenge of file-level granularity, as opposed to the method-level focus of existing methods, making direct comparisons less straightforward.

A direct comparison of accuracies of the state of the art approach and our approach is not feasible because the dataset used for FastLog differs significantly from the dataset generated in our process. While FastLog's dataset involves mapping labels to individual words, our approach focuses on file-level logging, with labels mapped to entire sentences.

6.2 Stage 2: Log Generation Evaluation

The performance of the LLaMA-3.2-3B Instruct model in Stage 2 was evaluated using BLEU, ROUGE-1, and ROUGE-L metrics across multiple Java files. The average scores on a scale from 0 to 1 are summarized in Table 1.

Metric	Average Score
BLEU	0.6464
ROUGE-1 Precision	0.8696
ROUGE-1 Recall	0.9116
ROUGE-1 F-Measure	0.8859
ROUGE-L Precision	0.8177
ROUGE-L Recall	0.8556
ROUGE-L F-Measure	0.8262

6.3 Key Observations

BLEU Score: Achieved a moderate score of 0.6464, indicating strong lexical alignment between the generated and reference logs.

ROUGE-1 Performance: High precision (0.8696) and recall (0.9116) values demonstrate the relevance and contextual appropriateness of the generated logs.

ROUGE-L Performance: F-Measure of 0.8262 reflects good structural similarity in generated logs, slightly trailing behind ROUGE-1 metrics but still indicative of robust performance.

6.4 Comparative Analysis

In Stage 2, our approach outperformed the state-of-the-art method FastLog in terms of log generation quality:

- **BLEU Score:** Our model achieved 0.6464 compared to FastLog's 0.3343.
- **ROUGE-1 Recall:** 0.9116 in our experiment versus 0.5991 in FastLog.
- **ROUGE-L Recall:** 0.8556 for our method compared to 0.5953 for FastLog.

6.5 Key Advantages

- **Model Selection:** Leveraging the LLaMA-3.2-3B Instruct model provided greater robustness and efficiency compared to the PLBART model used in FastLog.

- **File-Level Context [7]:** By operating at a file-level granularity rather than a method-level, our model benefited from a broader contextual understanding, enabling more accurate log generation and placement.

These results underscore the potential of our file-level approach for high-quality logging statement generation, demonstrating clear advantages over existing methods in terms of generated output quality.

7 Discussion and Future Work

7.1 Discussion

In our evaluation of automated log generation using the LLaMA-3.2 models, several noteworthy observations emerged. The preprocessing approach implemented in this project is language-agnostic, allowing it to be adapted for other programming languages such as C and C++. This flexibility ensures the scalability of the framework beyond Java, addressing the diverse needs of multi-language software systems.

Our experiments revealed promising results in leveraging the advanced capabilities of the LLaMA-3.2-3B Instruct model for log generation tasks. However, certain challenges were identified. For instance, while the model performed well in identifying broader contextual patterns, it occasionally struggled with precise placement of logs in complex code structures, especially in cases involving nested loops or method chains. Additionally, maintaining the semantic consistency of generated logs with the surrounding code context requires further refinement.

When comparing file-level granularity with method-level approaches, our findings suggest that the former provides a richer context for log placement but also introduces challenges related to increased computational overhead and the need for advanced context parsing. Addressing these challenges could significantly enhance the usability of file-level models in production environments.

7.2 Future Work

Several directions can be explored to extend the scope and impact of this project:

- **Fine-Tuning for Specialized Tasks:** Fine-tuning the CodeLlama model specifically for log generation tasks can leverage its advanced code modeling capabilities to achieve higher accuracy and contextual relevance in automated log placement. Incorporating domain-specific datasets into the fine-tuning process may further enhance performance.
- **Prompt Engineering:** Advanced prompt engineering techniques, such as Chain of Thought (CoT) [15] and Tree of Thought strategies, could be employed to better capture the nuanced requirements of log generation. These approaches can improve the model's ability to reason through complex code structures and produce more precise log placements.
- **Representation Enhancements:** Employing abstract syntax trees (ASTs) [11] or Code2Vec [1] embeddings to represent code in the training data can offer better insights into the structural and functional aspects of the code. Such representations may enable more context-aware log generation.
- **Optimization of Preprocessing Pipelines:** The language-agnostic nature of the preprocessing pipeline can be optimized to handle domain-specific quirks of various programming languages, ensuring seamless adaptation and improved performance.
- **Broader Test Case Coverage:** Future work could also involve integrating genetic algorithms to explore diverse distributions of test cases. This approach may result in broader coverage and a more comprehensive evaluation of the model's capabilities in automated log placement.

Addressing these areas has the potential to improve the robustness, accuracy, and adaptability of automated log generation systems, thereby supporting developers in managing logging practices across diverse software ecosystems.

8 Conclusion

In the realm of automated log generation, our investigation demonstrated the effectiveness of leveraging large language models (LLMs) for accurately predicting and generating log statements within Java code. The model

consistently achieved high recall and precision in placing relevant logs at the required positions, showcasing its effective learning and adaptability to complex code structures and diverse contexts.

The BLEU score of 0.6464 highlighted a strong lexical alignment between the generated and reference logs while also indicating room for improvement in achieving closer lexical similarity. This underscores the potential to refine the model further, particularly in cases requiring nuanced context understanding and precise word choices.

Our findings emphasize the critical role of incorporating broader file-level contexts, which significantly enhance the model's ability to understand logging patterns and generate relevant logs. Additionally, the evaluation revealed that maintaining semantic coherence and contextual relevance in the generated logs remains an area that can benefit from continued optimization.

This work further highlights the potential of combining advanced modeling techniques, such as fine-tuning, with innovative evaluation metrics like BLEU and ROUGE. These approaches provide deeper insights into the model's strengths and areas for improvement. Future research can focus on augmenting the training dataset with additional contextual information, employing advanced embeddings like ASTs or Code2Vec, and refining the language-agnostic pre-processing pipeline to achieve even greater adaptability and performance across programming languages.

The insights gained from this study establish a robust foundation for advancing automated log generation and integrating LLMs into practical software development workflows. By addressing the identified challenges and exploring the suggested enhancements, this approach holds significant promise for improving logging practices and overall code quality in real-world software systems.

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[2] Marcello Barbella and Genoveffa Tortora. 2022. Rouge Metric Evaluation for Text Summarization Techniques. *SSRN Electronic Journal* 14 (2022).

[3] Juan Cruz-Benito, Sanjay Vishwakarma, Francisco Martin-Fernandez, and Ismael Faro. 2021. Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches. *AI* 2, 1 (2021), 1–16.

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

[5] Sina Gholamian and Paul AS Ward. 2021. A comprehensive survey of logging in software: From logging statements automation to log mining and analysis. *arXiv preprint arXiv:2110.12489* (2021).

[6] Shenghui Gu, Guoping Rong, He Zhang, and Haifeng Shen. 2022. Logging practices in software engineering: A systematic mapping study. *IEEE Transactions on Software Engineering* 49, 2 (2022), 902–923.

[7] 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).

[8] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. LLMParser: An Exploratory Study on Using Large Language Models for Log Parsing *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 99, 13 pages. https://doi.org/10.1145/3597503.3639150

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

[10] Author Name1 and Author Name2. 2024. Meta Title of the Paper. *Journal Name or Conference* (2024). https://liweinlp.com/wp-content/uploads/2024/07/meta.pdf Accessed: 2024-12-03.

[11] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*. 1–5.

[12] Adwait Ratnaparkhi. 1996. A maximum entropy model for part-of-speech tagging. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 133–142. https://doi.org/10.3115/1073083.1073135

[13] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[14] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[15] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[16] Xiaoyuan Xie, Zhipeng Cai, Songqiang Chen, and Jifeng Xuan. 2024. FastLog: An End-to-End Method to Efficiently Generate

and Insert Logging Statements. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 26–37.

[17] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[18] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 415–425.