Automated and Context-Aware Code Documentation Leveraging Advanced LLMs

Swapnil Sharma Sarker
Ahsanullah University of Science and Technology
Dhaka, Bangladesh
swapnilsharmasarker@gmail.com

Tanzina Taher Ifty
George Mason University
Fairfax, Virginia, USA
tifty@gmu.edu

Abstract

Code documentation is essential to improve software maintainability and comprehension. The tedious nature of manual code documentation has led to much research on automated documentation generation. Existing automated approaches primarily focused on code summarization, leaving a gap in template-based documentation generation (e.g., Javadoc), particularly with publicly available Large Language Models (LLMs). Furthermore, progress in this area has been hindered by the lack of a Javadoc-specific dataset that incorporates modern language features, provides broad framework/library coverage, and includes necessary contextual information. This study aims to address these gaps by developing a tailored dataset and assessing the capabilities of publicly available LLMs for context-aware, template-based Javadoc generation. In this work, we present a novel, context-aware dataset for Javadoc generation that includes critical structural and semantic information from modern Java codebases. We evaluate five open-source LLMs (including LLaMA-3.1, Gemma-2, Phi-3, Mistral, Qwen-2.5) using zero-shot, few-shot, and fine-tuned setups and provide a comparative analysis of their performance. Our results demonstrate that LLaMA 3.1 performs consistently well and is a reliable candidate for practical, automated Javadoc generation, offering a viable alternative to proprietary systems.

1 Introduction

Code documentation is a crucial part of software development that bridges the gap between developers, end-users, and future maintainers of a software system. While the fundamental purpose of documentation is to guarantee that the code is comprehensible and accessible, it also acts as a crucial tool for promoting collaboration, boosting productivity, and decreasing technical debt (Dvivedi et al., 2024). Without heavily depending on the original code writers, developers can understand the complexities of a code-base, troubleshoot problems, and make well-informed changes with the help of well-structured documentation. A study involving software maintainers highlighted the importance of documentation, revealing that 94.03% agreed that source code documentation is crucial for object-oriented artifacts (de Souza et al., 2005).

However, creating and maintaining such documentation is expensive and time-consuming (Khan and Uddin, 2022). Many developers fail to document their code consistently or neglect it altogether, leading to technical debt. This is often due to a lack of time, unclear guidelines, or the assumption that the code is self-explanatory (Uddin and Robillard, 2015; Forward and Lethbridge, 2002). Also, developers working in larger teams often lack a standardized approach to documenting code, which causes inconsistency in the style and quality of documentation, which confuses collaborators (Dragan et al., 2006; Parnas and Clements, 1986). In order to tackle these issues, developers over the years have often turned to template-based documentation solutions like Javadoc, TSDoc, and OpenAPI specifications, etc., which helped to bring consistency in the style of documentation (Uddin and Robillard, 2015; Stylos and Clarke, 2007; Horning, 2001). But the inconsistency persists when it comes to the details of the documentation. Some documentation is overly detailed and some is brief, which impacts the readability of the documentation (Buse and Weimer, 2010; Treude et al., 2011). Moreover, large projects make it particularly challenging to manually mark and document code snippets. Furthermore, the documentation often becomes outdated as the development continues maybe because of new features being developed or requirement changes (Fluri et al., 2007; Rastkar et al., 2010).

Code and dataset are available at https://github.com/ineffablekenobi/Documentation-generation-using-LLM

Proceedings of the 18th International Natural Language Generation Conference, pages 486–498
October 29–November 2, 2025.

Since one of the main areas where programmers value automation is documentation, an automated solution is therefore quite desirable (McBurney et al., 2017).

The introduction of Large Language Models (LLM) has revolutionized the field of software development (Khan and Uddin, 2022; Dvivedi et al., 2024; Kneidinger et al., 2024). Although these models are trained on huge corpus of data from diverse sources, they can be used effectively for tasks like code completion (Husein et al., 2025), code generation (Paik and Wang, 2021; Destefanis et al., 2023), project planning (Barcaui and Monat, 2023) and documentation generation (Khan and Uddin, 2022; Dvivedi et al., 2024; Kneidinger et al., 2024). Recently, automated code documentation generation has been in the center of attention in language research, and quite a few advancements have been made in this field. Summarization techniques have already been implemented and evaluated (Khan and Uddin, 2022) in many different languages with the help of popular datasets like CodeSearchNet (Husain et al., 2020), CoDesc (Hui et al., 2024). Both text-to-text and LLMs have been used in implementing these solutions. Other studies also focused on comprehensive comparisons between multiple LLMs and evaluated their performance in documentation generation over multiple programming languages (Dvivedi et al., 2024). Also, Pandey et al. have explored agent-based approaches (i.e. github copilot) for documentation generation (Pandey et al., 2024). Moreover, models like GPT-4 (OpenAI, 2023) have been evaluated in template-based documentation such as Javadoc generation (Kneidinger et al., 2024). However, many of these high-performing solutions rely on proprietary, closed-source models or APIs, presenting significant challenges for organizations. Data privacy remains one of the major concerns. Additionally, limitations in customization for specific documentation standards, potential latency issues, restrictive rate limits, ongoing costs, and reliance on external providers hinder their adoption where control and security are crucial. Consequently, while potential is clear, there remains a gap in understanding the capabilities of these models in template-based documentation generation task. To date, no studies have focused on evaluating these open source models for task like Javadoc generation, which require adherence to specific formats and contextual understanding.

Evaluating and fine-tuning publicly available LLMs for Javadoc generation requires a suitable dataset. While large code datasets like CodeSearchNet (Husain et al., 2020) and CoDesc (Hui et al., 2024) exist, containing millions of code snippets, they are primarily designed for code summarization, making them ill-suited for generating structured, template-based documentation. Furthermore, these datasets lack contextual information (such as class or package context needed for accurate Javadoc tags) and have limited coverage of modern Java features like lambdas and reactive programming constructs such as Mono and Flux, which are extensively used in Java projects. Finally, there are currently no datasets available for template-based documentation generation tasks like Javadoc. This gap highlights the need for a new dataset specifically for training and evaluating models in Javadoc generation that includes contextual information and modern Java features.

This paper makes the following contributions to address these gaps:

- Introduction of a new, context-aware dataset for Javadoc generation, covering methods, lambdas, and modern Java features, curated from multiple public codebases.
- Application of automated and manual filtering techniques to ensure the quality and relevance of the dataset.
- Systematic evaluation and fine-tuning of five publicly available LLMs (LLaMA-3.1, Gemma-2, Phi-3, Mistral, Qwen-2.5) on the proposed Javadoc generation task.
- A comparative analysis of model performance across zero-shot, few-shot, and fine-tuned settings, providing insights into their capabilities for automated documentation.

2 Related Works

Numerous studies have been conducted on code documentation, starting with conventional rule-based techniques and advancing to **pre-LLM** AI models like LSTM and early Transformer-based methods. Ahmad et al. (Ahmad et al., 2020) evaluated the Transformer model, which learns code representation for summarization through a self-attention mechanism. To summarize C# code snippets, CODE-NN, an LSTM-based model with an attention mechanism, was proposed by Iyer et al. (Iyer et al., 2016). An early neural attention model

for code summarization was presented by Allamanis et al. (Allamanis et al., 2016). It incorporates a dual attention mechanism and convolutional features into a recurrent encoder-decoder architecture. However, they were constrained by low flexibility, poor generalization, limited memory, and an insufficient understanding of the content of the code.

Modern LLMs are increasingly applied to automated code documentation, yet existing studies reveal critical limitations. For instance, Khan et al. (Khan and Uddin, 2022) used Codex for multi-language documentation, achieving a modest BLEU score of 20.6, while Diggs et al. (Diggs et al., 2024) developed specific prompting strategies and evaluation rubrics for generating comments in legacy systems. Similarly, Geng et al. (Geng et al., 2024) focused on satisfying developer goals by pre-training models with code-comment pairs. Despite these efforts, performance remains a key issue, with Kneidinger et al. (Kneidinger et al., 2024) demonstrating that even a powerful proprietary model like **GPT-4 produces unsatisfactory results** for class-level documentation. Furthermore, a major gap persists: existing research has almost exclusively used proprietary models, overlooking the application of **open-source LLMs specifically for Javadoc generation**. This leaves developers who require flexible and transparent solutions without a viable alternative to paid, closed-source APIs.

Agent-based approaches have also shown promise in this area. For instance, REPOAGENT is an open-source system that excels at repository-level documentation, though it is limited to Python projects and lacks template support (Luo et al., 2024). Similarly, commercial agents like GitHub Copilot have demonstrated significant efficiency gains, saving up to 50% of the time developers spend on documentation tasks (Pandey et al., 2024). Although LLM-powered agents have strong capabilities, there are still significant obstacles to overcome before they can be used in the real world. Data security is still a top priority, particularly when managing private or confidential data without the right protections. Another challenge is customization, since many LLMs find it difficult to adjust to domain-specific requirements without prompt engineering or fine-tuning. Furthermore, delay can impact usability, especially in interactive environments where users anticipate prompt responses. Deploying dependable and safe AI-driven bots requires addressing these constraints.

Several **existing datasets** are relevant to code documentation generation, but possess limitations for our specific focus on template-based Javadoc. For instance, Hasan et al. introduced **CoDesc** (Hui et al., 2024), a large dataset containing over 4.2 million Java methods paired with natural language descriptions. Despite its size, CoDesc suffers from noise and inconsistencies, lacks necessary contextual information for Javadoc generation, does not provide template-based documentation, and offers limited coverage of modern Java constructs. Similarly, **CodeXGLUE-CONCODE** (Iyer et al., 2018) provides Java code snippets and natural language descriptions but shares identical drawbacks when considering template-based documentation needs. **CodeSearchNet** (Husain et al., 2020), introduced by Husain et al., covers multiple programming languages but also exhibits issues like noise, potential duplicate entries, a lack of modern Java features, and a primary focus on function-level summaries rather than structured documentation. While **The Stack** (Kocetkov et al., 2022) represents a massive collection (over 6TB) of permissively licensed source code across many languages, it is a general code corpus and is not specifically curated or structured for the task of documentation generation.

3 EXPERIMENT

This section presents our experimental setup and the corresponding model results.

3.1 DATASET

The data collection was guided by several principles to ensure a comprehensive and novel dataset. We prioritized **diversity** by selecting repositories with varied coding styles and documentation patterns, focusing on projects with **permissive open-source licenses** (e.g., MIT, Apache 2.0, GPL 3.0) to allow for analysis and redistribution. Projects were selected for their high **Javadoc prevalence** and significant **contribution activity**, indicating established documentation practices and wide community adoption. Furthermore, the dataset ensures broad **framework and library coverage**, including tools like Project Reactor and Spring Boot, and incorporates codebases utilizing **modern Java features** such as lambdas, generics, and stream APIs to reflect current language usage. Our data was sourced from the following repositories, which were selected to meet the requirements mentioned above.

Our data processing pipeline (Figure 1) be-

Table 1: Public repositories used in the dataset

Index	Repository Name
1	CitiesAPI (Nurislom373, 2025)
2	Database-api (Heliumdioxid, 2025)
3	Discord4J (Discord4J, 2025)
4	htmldoclet4jdk8 (WinRoad-NET, 2025)
5	JDA (discord jda, 2025)
6	Jestures (thedevstone, 2025)
7	Milenage (brake, 2025)
8	project-tracking-system-backend-app (SelimHorri, 2025)
9	SavageFactions (SavageLabs, 2025)
10	termenu (AugustoRavazoli, 2025)

gins with scripts identifying .java files containing Javadoc comments (/** ... */) in the selected repositories (Table 1). Files lacking Javadoc are discarded. From the remaining files, we used a series of regular expressions to parse and extract Javadoc comments, method and class declarations, and package information from the source files. This automated process was followed by syntactic validation to ensure structural integrity. (The specific regular expressions are detailed in Appendix B). Essential contextual information, like package and enclosing class names, is captured alongside each code-Javadoc pair, yielding an initial set of roughly 5128 entries.

These pairs then undergo **automated filtering** using the same patterns to perform syntactic validation. This step verifies the structural integrity of both the Javadoc comments (ensuring proper format) and the code snippets (checking conformance to basic Java syntax). This efficient pattern-based check filters out invalid or incomplete entries, significantly improving dataset quality before manual review.

After automated data filtering, we have conducted a **thorough review** ensuring correctness and quality of the data. To ensure dataset quality, we instructed annotators to remove entries containing documentation that was faulty, out-of-context, irrelevant or included personal information, and therefore did not accurately describe the corresponding code. Four volunteers, two software engineers, and two academic researchers, generously assisted with the manual verification process. To determine the trustworthiness threshold, we randomly selected 20 samples, distributing 10 of them among four participants who had achieved a 90% trustworthiness score (Price et al., 2020). The degree of agreement

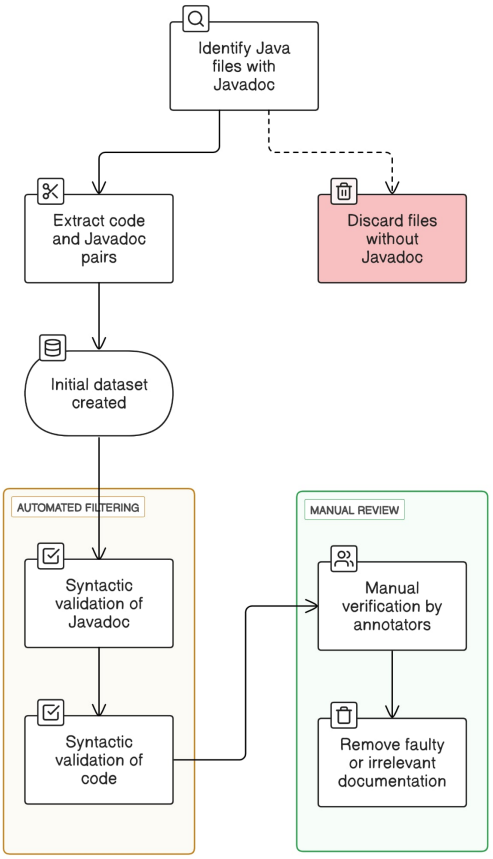

Figure 1: Data Collection and Filtering Pipeline

among annotators was assessed using Fleiss' kappa score (Fleiss, 1971), resulting in a value of 0.66, which indicates a substantial level of agreement and helps ensure annotation quality.

Initially, we have collected 5128 rows of data, which were later filtered based on correctness and relevance. After automated and manual filtering, the dataset contained 3,614 high-quality code-documentation pairs along with their package information. The distribution across training, validation, and test splits is detailed in Table 2.

Table 2: Distribution of filtered dataset splits

Dataset Type	Number of Samples
Train	2,778
Validation	140
Test	696
Total	3,614

Each entry in our final, filtered dataset consists of three key components: the Java code snippet (e.g., a method), its complete Javadoc documentation,

and the corresponding package context to aid in understanding dependencies. A concrete example is provided in Appendix A.

3.2 Models

To achieve optimal outcomes, we fine-tuned five advanced Large Language Models (LLMs) on our dataset: LLaMA-3.1, Gemma-2, Phi-3, Mistral, and Qwen-2.5. Each model employs a decoder-only Transformer architecture with distinct optimizations: **LLaMA-3.1-8B** utilizes SwiGLU activation and Rotary Positional Embeddings (RoPE) (Vavekanand and Sam, 2024); **Gemma-2-9B** (Team et al., 2024) and **Phi-3.5-Mini-Instruct** (3.8B) (Abdin et al., 2024) feature RMSNorm, logit soft-capping, and alternating local/global attention; **Mistral-7B-v0.3** (Jiang et al., 2023) incorporates sliding window attention and grouped-query attention; and **Qwen-2.5-Coder-3B** (Hui et al., 2024) includes optimized attention mechanisms and enhanced fine-tuning for long-text generation and instruction following.

3.3 Prompt Engineering

Prompts are queries written in a compatible template, so that a model can comprehend what our request is and how it should address the task. Although the specific structure may vary depending on the model, the overall design principles of the prompts are quite similar. Prompts have parts like roles, context, input, etc. (Kıcıman et al., 2024). In our study, we designed three distinct types of prompt for different evaluation procedures. Our base prompt format included clear instructions and marked inputs. For **zero-shot prompting**, we used this base prompt without additional examples. In **one-shot prompting**, we added a single example to the prompt template, while for **few-shot prompting**, we carefully handpicked three examples from our dataset to guide the model toward more accurate and desirable responses.

3.4 Evaluation Metrics

To assess the quality of the generated documentation, we employed a set of standard, well-established metrics. We used the BLEU score (Papineni et al., 2002) to measure the precision of n-gram overlap between the generated and reference documentation. Additionally, we used several variants of the ROUGE score (R-1, R-2, R-L, and R-Lsum) (Lin, 2004) to evaluate content overlap by assessing recall on unigrams, bigrams, and the longest common subsequence. Detailed definitions and formulas for these metrics can be found in Appendix C.

3.5 Parameter Efficient Training

To fine-tune the large language models efficiently under resource constraints, we employed Low-Rank Adaptation (LoRA) (Hu et al., 2021), a parameter-efficient training technique. LoRA significantly reduces the number of trainable parameters by freezing the pre-trained weights and injecting smaller, trainable low-rank matrices into the Transformer layers.

We configured LoRA with $\alpha = 16$ to effectively control the influence of low-rank updates on the original weights, balancing responsiveness with parameter stability. Additionally, gradient checkpointing was enabled to reduce memory consumption during back-propagation, allowing for larger batch sizes and efficient GPU memory usage (Daniel Han and team, 2023). Table 3 illustrates the number of trainable parameters for each model.

Table 3: Model and number of trainable parameters

Model	Trainable Parameters
LLaMA-3.1-8B	41,943,040
Mistral-7B-v0.3	41,943,040
Qwen-2.5-Coder-3B	29,933,568
Gemma-2-9B	54,018,048
Phi-3.5-Mini-Instruct	29,884,416

We tuned several key hyperparameters, such as the learning rate and weight decay, to ensure stable model convergence. The final training configuration is provided in Appendix D. The use of linear schedulers ensures improved convergence and the weight decay was introduced to prevent possible over-fitting of the models. We always saved the best model (based on validation performance) to ensure that even if overfitting occurs, the selected evaluation model (the best checkpoint) is not affected. All of these models except for Gemma-2-9B were trained on a single P100 GPU in a Kaggle environment. Gemma-2-9B was trained on a single A100 GPU in Google Colab.

All models were trained for 5 epochs using a 'steps' evaluation strategy. As shown in Fig. 2, the Gemma-2-9B model showed clear signs of overfitting, with its validation loss increasing after 180 steps and remaining significantly higher than other models, possibly due to model complexity or in-

sufficient data. This did not affect the final results, as the best-performing checkpoint was saved. In contrast, LLaMA-3.1-8B's validation loss was the most consistent, stabilizing after 200 steps and suggesting it had reached a point of saturation where further training offered minimal benefit.

Finally, Fig. **??** shows the growth of the BLEU score as a function of the number of steps. It is evident that **LLaMA-3.1-8B** and **Mistral-7B-v0.3** performed the best on the validation set, demonstrating consistent performance growth in parallel with the number of steps. Our most efficient model, **Phi-3.5-Mini-Instruct**, was on par with these models and even outperformed larger models like **Gemma-2-9B**, though the difference was not substantial.

3.6 Evaluation

To assess the performance of the models, we have relied on well-known evaluation metrics, including BLEU and ROUGE scores. The same set of evaluation metrics were consistently implemented to evaluate all the models and then compared to analyze the results and discuss their effectiveness in the documentation generation task.

Firstly, we have focused on the zero-shot evaluation method. In this evaluation technique, we assess the performance of pretrained models without providing them with any examples. The model is provided with simple contexts that tell them their role and the expected output. As shown in Table 4, **Qwen-2.5-Coder-3B** showed exceptional performance, outperforming its more complex counterparts in this specific task. The Qwen-2.5 Coder was trained on coding-related tasks like this, making it more suitable for documentation generation. **LLaMA-3.1-8B** has also shown similar performance despite being a general-purpose model.

In one-shot and few-shot settings (Table 4), **Qwen-2.5-Coder-3B** showed substantial performance gains in the few-shot learning evaluation process by establishing a significant lead over the other models. This illustrates the effect of improved prompts, as the models could use multiple examples as a reference before generating outputs. However, while prompt optimization plays a key role, the training data used to train these models have a significant impact on their understanding of performing a specific task. Therefore, our evaluation results should not be viewed as definitive indicators of a model's overall capability from a design perspective, but can be considered as reflections of how well a model adapts to this specific task.

After the fine-tuning process, we re-evaluated the models. All models demonstrated substantial performance improvements compared to their pre-fine-tuning results. However, **LLaMA-3.1-8B** emerged as the top performer, followed by **Mistral-7B-v0.3**.

We can see the ROUGE score progression across different evaluation stages illustrated in Fig. 4 (showing ROUGE-Lsum). One interesting observation is **Mistral-7B-v0.3**'s lower ROUGE scores after fine-tuning compared to some other models, despite its strong BLEU score. This suggests that **Mistral-7B-v0.3** prioritizes exact replication of patterns learned from fine-tuning data (high BLEU), rather than effectively capturing broader contextual meaning (relatively lower ROUGE). This might imply overfitting to the fine-tuning data structure or a lesser degree of generalization in paraphrasing.

While observing the performance over different stages of evaluation (Fig. 3 and Fig. 4), it is evident that **Qwen-2.5-Coder-3B** consistently outperformed every model during the zero-shot, one-shot, and few-shot evaluation stages. However, after fine-tuning, its performance relative to others (especially LLaMA-3.1-8B) was no longer the best. This demonstrates how the pre-training data helped **Qwen-2.5-Coder-3B** excel initially. After fine-tuning, its performance may have plateaued or the fine-tuning dataset might not have aligned perfectly with its pre-training distributions, causing it to shift away and not maintain its lead.

Meanwhile, **Phi-3.5-Mini-Instruct**, which previously performed the worst across all earlier evaluations, showed exceptional improvement after fine-tuning. Recall that we mentioned earlier that performance in the initial stages doesn't necessarily represent a model's ability to capture information; this observation serves as evidence of that claim. **Phi-3.5-Mini-Instruct**'s pre-training data might not have provided enough exposure to the documentation generation task, which could explain its weaker performance in the earlier evaluation stages, but it adapted well during fine-tuning.

Additionally, **Gemma-2-9B** had the lowest BLEU score after fine-tuning compared to LLaMA and Mistral. However, its ROUGE scores were quite high, particularly R1 and R2, indicating strong content overlap even if exact phrasing (BLEU) differed. This suggests that the model focused on capturing the broader context and se-

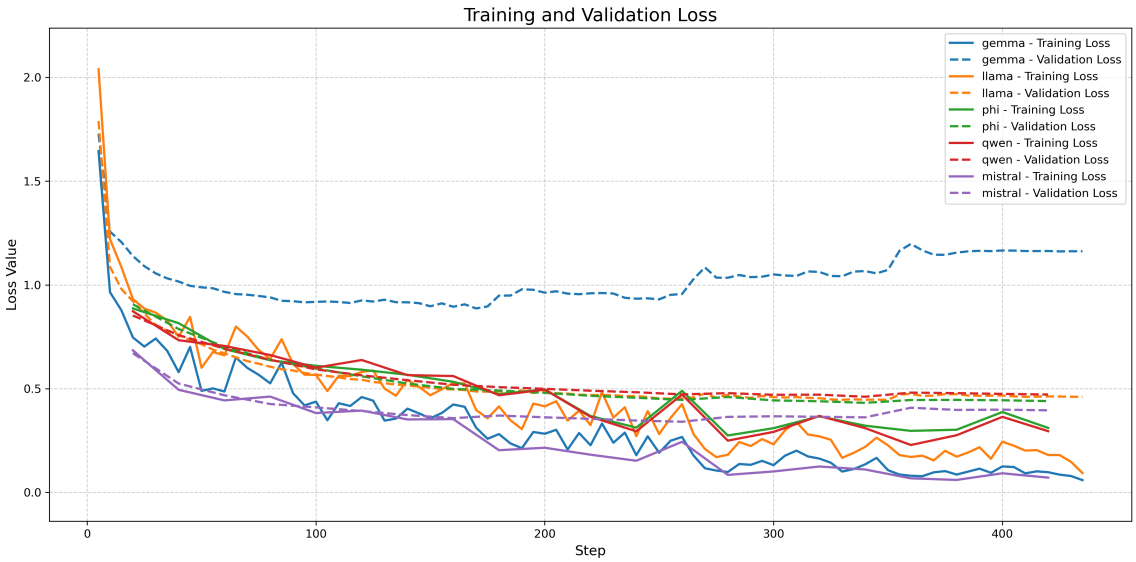

Figure 2: Training and validation loss plot

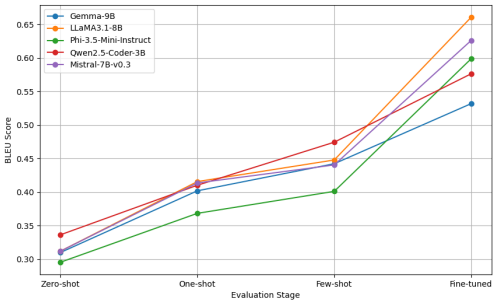

Figure 3: BLEU score progression over evaluation stages (Zero-shot, One-shot, Few-shot, Fine-tuned)

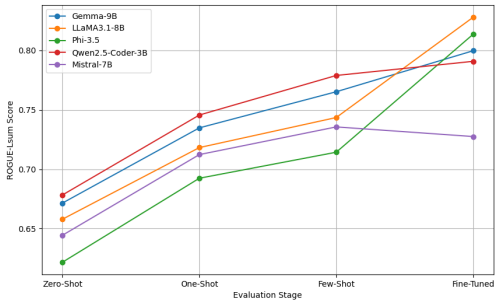

Figure 4: ROUGE-Lsum score progression over evaluation stages (Zero-shot, One-shot, Few-shot, Fine-tuned)

mantics rather than simply replicating the reference text structure, which is a positive sign for generating diverse but relevant documentation.

4 Conclusion

We introduced a new dataset consisting of reference documentation for methods, lambdas, packages, and class references, designed to provide richer context for fine-tuning publicly available models for Javadoc-style generation. Additionally, we trained models such as LLaMA-3.1-8B, Gemma-2-9B, Phi-3.5-Mini-Instruct, Mistral-7B-v0.3, and Qwen-2.5-Coder-3B on our dataset. Furthermore, we assessed the performance of each model across four different evaluation stages (zero-shot, one-shot, few-shot, and fine-tuned) and measured their effectiveness using BLEU and ROUGE evaluation metrics. Finally, we provided a comprehensive analysis of their performance, highlighting how pre-training influences initial capabilities and how fine-tuning on a targeted documentation generation dataset affects their performance, with LLaMA-3.1-8B showing consistently strong results after fine-tuning.

5 Limitations and Risks

This study has several limitations and risks. **Resource constraints** limited our dataset size, prevented the fine-tuning of larger model variants, and restricted hyperparameter exploration. The **dataset diversity** was also insufficient, particularly regarding template-based formats such as TSDoc and JSDoc. Furthermore, our methodology introduced **fine-tuning risks**, including potential model bias

Table 4: Evaluation results across different settings (Zero-shot, One-shot, Few-shot, and Fine-tuned)

Setting	Model	BLEU	R1	R2	RL	RLsum
Zero-shot	Gemma-2-9B	0.3098	0.5522	0.2552	0.4491	0.5429
	LLaMA-3.1-8B	0.3118	0.5734	0.2667	0.4696	0.5490
	Phi-3.5-Mini-Instruct	0.2953	0.5230	0.2261	0.4381	0.5029
	Qwen-2.5-Coder-3B	**0.3362**	0.5620	0.2770	0.4627	0.5431
	Mistral-7B-v0.3	0.3118	0.5104	0.2221	0.4342	0.5037
One-shot	Gemma-2-9B	0.4018	0.6270	0.2905	0.5055	0.6140
	LLaMA-3.1-8B	**0.4156**	0.6428	0.2966	0.5211	0.6222
	Phi-3.5-Mini-Instruct	0.3682	0.6016	0.2745	0.4961	0.5830
	Qwen-2.5-Coder-3B	0.4101	0.6457	0.3243	0.5263	0.6294
	Mistral-7B-v0.3	0.4135	0.6200	0.2815	0.5147	0.6141
Few-shot	Gemma-2-9B	0.4422	0.6780	0.3522	0.5524	0.6715
	LLaMA-3.1-8B	0.4478	0.6677	0.3422	0.5440	0.6579
	Phi-3.5-Mini-Instruct	0.4010	0.6279	0.2942	0.4968	0.6217
	Qwen-2.5-Coder-3B	**0.4743**	0.6852	0.3774	0.5708	0.6783
	Mistral-7B-v0.3	0.4404	0.6514	0.3294	0.5424	0.6444
Fine-tuned	Gemma-2-9B	0.5318	0.8023	0.6734	0.7782	0.7997
	LLaMA-3.1-8B	**0.6606**	0.9301	0.7213	0.8125	0.8279
	Phi-3.5-Mini-Instruct	0.5987	0.8156	0.6947	0.7986	0.8136
	Qwen-2.5-Coder-3B	0.5763	0.7936	0.6737	0.7676	0.7908
	Mistral-7B-v0.3	0.6260	0.7288	0.6372	0.7164	0.7275

from the training data and a degradation of the model's general-purpose performance. A critical operational risk is the potential for the models to **generate factually incorrect or misleading documentation (hallucinations)**, which could introduce bugs if trusted by developers without verification.

6 Acknowledgment

We express our gratitude to the contributors and maintainers of the open-source models and repositories that have facilitated this research. Specifically, we acknowledge Google for providing **Gemma-2-9B** under the Google AI Model License, Meta for **LLaMA-3.1-8B** under the Llama 3.1 Community License, Microsoft for **Phi-3.5-Mini-Instruct** under the MIT License, Alibaba Qwen Team for **Qwen2.5-Coder-3B** under the Qwen Research License, and Mistral AI for **Mistral-7B-v0.3** under the Apache 2.0 License.

Additionally, we thank the volunteers who contributed to the manual verification of the dataset. Finally, we acknowledge the developers of PyTorch, Unsloth, and other open-source libraries used in this study, including Evaluate, Rouge Score, TensorBoard, and gdown, for enabling efficient experimentation and evaluation.

7 Data availability

To facilitate reproducibility and further research, the curated dataset and the code used for model fine-tuning and evaluation are made publicly available. During the anonymous review period, they can be accessed at the following repository: https://anonymous.4open.science/r/automated-documentation-generation-using-llm. Upon acceptance, the material will be made available via a persistent public repository under a MIT License, CC-BY 4.0 License. As this work utilizes publicly available codebases as its source, our curated and filtered dataset is provided in accordance with open data sharing requirements.

8 Ethics Statement

This research follows the principles of the ACM Code of Ethics. Our main goal is to support the software development community by creating and testing open-source tools for automated documentation. We aim to help developers work more efficiently and make software easier to maintain (ACM Code 1.1).

We understand that our work could have negative effects, and we have taken steps to reduce these risks (ACM Code 1.2, 2.5). The biggest risk is that our models could generate documentation that is wrong or confusing (a phenomenon known as "hal-

lucination"). If developers trust this documentation without checking it, it could lead to software bugs, security issues, or a poor understanding of the code. For this reason, we believe these models should be used to assist developers, not to replace them. It is essential that a human developer always reviews the final output.

To be fair and protect privacy (ACM Code 1.4, 1.7), we built our dataset using only public repositories with permissive open-source licenses. We carefully checked the data by hand to find and remove any personal or sensitive information. We also know that the original training data may contain biases, which our models could learn and repeat. While our filtering helps, we acknowledge that the risk of spreading these biases is a limitation of our work.

By making our dataset and code publicly available, we aim to be honest and trustworthy (ACM Code 1.3). This allows the community to be transparent and enables others to reproduce, critique, and build upon our research.

References

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. PMLR.

AugustoRavazoli. 2025. Termenu. https://github.com/AugustoRavazoli/termenu. Accessed: 2025-01-27.

André Barcaui and André Monat. 2023. Who is better in project planning?generative artificial intelligence or project managers? *Project Leadership and Society*, 4:100101.

brake. 2025. Milenage. https://github.com/brake/milenage. Accessed: 2025-01-27.

Raymond P.L. Buse and Westley R. Weimer. 2010. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558.

Michael Han Daniel Han and Unsloth team. 2023. Unsloth. GitHub repository, accessed 2025-01-27.

Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*.

Giuseppe Destefanis, Silvia Bartolucci, and Marco Ortu. 2023. A preliminary analysis on the code generation capabilities of gpt-3.5 and bard ai models for java functions. *Preprint*, arXiv:2305.09402.

Colin Diggs, Michael Doyle, Amit Madan, Siggy Scott, Emily Escamilla, Jacob Zimmer, Naveed Nekoo, Paul Ursino, Michael Bartholf, Zachary Robin, et al. 2024. Leveraging llms for legacy code modernization: Challenges and opportunities for llm-generated documentation. *arXiv preprint arXiv:2411.14971*.

discord jda. 2025. Jda. https://github.com/discord-jda/JDA. Accessed: 2025-01-27.

Discord4J. 2025. Discord4j. https://github.com/Discord4J/Discord4J. Accessed: 2025-01-27.

Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2006. Reverse engineering method stereotypes. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 24–34. IEEE.

Shubhang Shekhar Dvivedi et al. 2024. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*.

Joseph L. Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382.

Beat Fluri, Michael Wursch, and Harald C. Gall. 2007. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79.

Andrew Forward and Timothy C. Lethbridge. 2002. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, page 26–33, New York, NY, USA. Association for Computing Machinery.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13.

Heliumdioxid. 2025. Database api. https://github.com/Heliumdioxid/database-api. Accessed: 2025-01-27.

Jim Horning. 2001. Software fundamentals: collected papers by david l. parnas. *SIGSOFT Softw. Eng. Notes*, 26(4):91.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *Preprint*, arXiv:1909.09436.

Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. 2025. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92:103917.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*, pages 2073–2083. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Junaed Younus Khan and Gias Uddin. 2022. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*.

Magdalena Kneidinger, Markus Feneberger, and Reinhold Plösch. 2024. Using gpt-4 for source code documentation. *WiPiEC Journal - Works in Progress in Embedded Computing Journal*, 10(2).

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code. *Preprint*.

Emre Kıcıman, Robert Ness, Amit Sharma, and Chenhao Tan. 2024. Causal reasoning and large language models: Opening a new frontier for causality. *Preprint*, arXiv:2305.00050.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*.

Paul W. McBurney et al. 2017. Towards prioritizing documentation effort. *IEEE Transactions on Software Engineering*, 44(9):897–913.

Nurislom373. 2025. Citiesapi. `https://github.com/Nurislom373/CitiesAPI`. Accessed: 2025-01-27.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Incheon Paik and Jun-Wei Wang. 2021. Improving text-to-code generation with features of code graph on gpt-2. *Electronics*, 10(21).

Ruchika Pandey, Prabhat Singh, Raymond Wei, and Shaila Shankar. 2024. Transforming software development: Evaluating the efficiency and challenges of github copilot in real-world projects. *arXiv preprint arXiv:2406.17910*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

David Lorge Parnas and Paul C. Clements. 1986. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257.

Ilan Price, Jordan Gifford-Moore, Jory Flemming, Saul Musker, Maayan Roichman, Guillaume Sylvain, Nithum Thain, Lucas Dixon, and Jeffrey Sorensen. 2020. Six attributes of unhealthy conversations. In *Proceedings of the 6th Workshop on Abusive Language Online (ALW)*, pages 114–124. Association for Computational Linguistics.

Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2010. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 505–514, New York, NY, USA. Association for Computing Machinery.

SavageLabs. 2025. Savagefactions. `https://github.com/SavageLabs/SavageFactions`. Accessed: 2025-01-27.

SelimHorri. 2025. Project tracking system backend app. `https://github.com/SelimHorri/project-tracking-system-backend-app`. Accessed: 2025-01-27.

Jeffrey Stylos and Steven Clarke. 2007. Usability implications of requiring parameters in objects' constructors. In *29th International Conference on Software Engineering (ICSE'07)*, pages 529–539.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*.

thedevstone. 2025. Jestures. https://github.com/thedevstone/Jestures. Accessed: 2025-01-27.

Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. 2011. How do programmers ask and answer questions on the web?: Nier track. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 804–807.

G. Uddin and M. P. Robillard. 2015. How api documentation fails. *IEEE Software*, 32(4):68–75.

Raja Vavekanand and Kira Sam. 2024. Llama 3.1: An in-depth analysis of the next-generation large language model.

WinRoad-NET. 2025. htmldoclet4jdk8. https://github.com/WinRoad-NET/htmldoclet4jdk8. Accessed: 2025-01-27.

Appendix

A Dataset Example

Below is an example entry from our curated dataset, illustrating the structure which includes the source code snippet, its corresponding Javadoc documentation, and the package context.

Component

Package:

`discord4j.core.object`

Code:

```
public Optional<Snowflake> getBotId
    () {
    return data.botId().toOptional()
                .map(Snowflake::of);
}
```

Documentation:

```
/**
 * Gets the id of the bot this role
 * belongs to, if present.
 *
 * @return The id of the bot this
    role
 * belongs to, if present.
 */
```

Table 5: A sample data entry from the curated dataset.

B Regular Expressions for Data Extraction

The regular expressions used to parse Java source files for Javadoc comments, package declarations, class/interface/enum declarations, and method/constructor signatures are detailed in Figure 5.

```
Javadoc:
/\*\*([\s\S]*?)\*/

Class:
(public|protected|private)?
  \s*(class|interface|enum)\s+(\w+)

Method:
(public|protected|private|static|final
|abstract|synchronized|native)?
\s*(?!class|interface|enum)
(?:<[^>]+>)?\s*([\w.<>?\[\],\s]+)
\s+(\w+)
\s*\((([\s\S]*?))\)
\s*(throws\s+[\w\s,]+)?
\s*(\{|;)

Package:
package\s+[^\s]+\s*;
```

Figure 5: Regular Expressions for Extracting Javadoc Comments, Classes, Methods, and Packages from Java Source Code.

C Evaluation Metrics

This section provides detailed definitions of the evaluation metrics used in our study.

C.1 BLEU Score

BLEU (Bilingual Evaluation Understudy) is an automated metric for evaluating machine translation by measuring n-gram overlap between a generated text g and a reference text r. It incorporates a brevity penalty to discourage overly short outputs. Higher BLEU scores indicate a closer alignment with human-quality translations (Papineni et al., 2002).

C.2 ROUGE Score

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics used for evaluating text summarization and machine translation. It compares a generated summary g with one or more reference summaries r by measuring overlap in units such as unigrams, bigrams, and the longest common subsequence (LCS) (Lin, 2004).

Notation

- $\text{Count}_{\text{match}}(u)$: Number of times a unigram u from reference r appears in the generated summary g.
- $\text{Count}(u', r)$: Total occurrences of unigram u' in the reference summary r.
- $\text{Count}_{\text{match}}(b)$: Number of bigrams b from reference r that match in g.
- $\text{Count}(b', r)$: Total occurrences of bigram b' in the reference summary r.

- $\text{LCS}(g, r)$: Length of the Longest Common Subsequence between generated summary g and reference r.
- L_r: Total number of words in the reference summary r.
- C_{LCS}: Cumulative LCS over all sentence pairs between g and r.
- W_r: Total word count across all sentences in the reference summary r.

ROUGE-1 (R1) Evaluates unigram overlap:

$$\text{R1} = \frac{\sum_{u \in \text{Unigrams}(r)} \text{Count}_{\text{match}}(u)}{\sum_{u' \in \text{Unigrams}(r)} \text{Count}(u', r)} \tag{1}$$

ROUGE-2 (R2) Evaluates bigram overlap:

$$\text{R2} = \frac{\sum_{b \in \text{Bigrams}(r)} \text{Count}_{\text{match}}(b)}{\sum_{b' \in \text{Bigrams}(r)} \text{Count}(b', r)} \tag{2}$$

ROUGE-L (RL) Calculates LCS normalized by reference length:

$$\text{RL} = \frac{\text{LCS}(g, r)}{L_r} \tag{3}$$

ROUGE-Lsum (RLsum) Evaluates summary-level LCS similarity:

$$\text{RLsum} = \frac{C_{\text{LCS}}}{W_r} \tag{4}$$

D Fine-Tuning Hyperparameters

D.1 LoRA Configuration

We employed Low-Rank Adaptation (LoRA) for parameter-efficient fine-tuning. LoRA freezes the pre-trained model weights and injects trainable rank decomposition matrices into the Transformer architecture. The weight update is defined as:

$$W = W_0 + \Delta W = W_0 + BA \tag{5}$$

where $W_0 \in \mathbb{R}^{d \times k}$ is the original weight matrix, and $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$ are the trainable low-rank matrices, with rank $r \ll \min(d, k)$.

For our experiments, we set the rank to $r = 16$. The LoRA adaptation was applied to the following projection layers:

$$\begin{aligned} \mathcal{T} = \{&q_{\text{proj}}, k_{\text{proj}}, v_{\text{proj}}, o_{\text{proj}}, \\ &\text{gate}_{\text{proj}}, \text{up}_{\text{proj}}, \text{down}_{\text{proj}}\} \end{aligned} \tag{6}$$

A scaling factor $\alpha = 16$ was used to moderate the magnitude of the weight updates.

D.2 Training Configuration

Table 6 provides an example of the training configuration used for the LLaMA-3.1-8B model. Similar hyperparameter settings were used for the other models, with minor adjustments where necessary.

Table 6: Example training configuration (LLaMA-3.1-8B)

Configuration	Value
Batch Size (Training)	8
Batch Size (Validation)	2
Gradient Accumulation Steps	4
Optimizer	AdamW
Learning Rate	2×10^{-4}
Evaluation Strategy	steps
Evaluation Steps	5
Linear Scheduler	Yes
Weight Decay	0.01
Epochs	5