UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Ian Guelman

Assessing the Effectiveness of Large Language Models in Code Comment Generation

Belo Horizonte
2025

Ian Guelman

Assessing the Effectiveness of Large Language Models in Code Comment Generation

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Marco Tulio de Oliveira Valente
Co-Advisor: Prof. Dr. José Laerte Pires Xavier Júnior

Belo Horizonte
2025

2025, Ian Guelman.
Todos os direitos reservados.

Guelman, Ian.

G925a Assessing the effectiveness of large language models in code comment generation [recurso eletrônico] / Ian Guelman. – 2025.

1 recurso online (80 f. il.) : pdf.

Orientador: Marco Tulio de Oliveira Valente.
Coorientador: José Laerte Pires Xavier Júnior.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f. 74-80.

1. Computação - Teses. 2. Engenharia de software - Teses. 3. Inteligência Artificial - Teses. I. Valente, Marco Tulio de Oliveira. II. Xavier Júnior, José Laerte Pires. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*32 (043)

Ficha catalográfica elaborada por Célio Resende Diniz, bibliotecário CRB 6/2403 - Universidade Federal de Minas Gerais – ICEx.

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Assessing the Effectiveness of Large Language Models in Code Comment Generation

IAN GUELMAN

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Documento assinado digitalmente
MARCO TULIO DE OLIVEIRA VALENTE
Data: 11/12/2025 13:24:43-0300
Verifique em https://validar.iti.gov.br

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

Documento assinado digitalmente
JOSE LAERTE PIRES XAVIER JUNIOR
Data: 11/12/2025 17:00:39-0300
Verifique em https://validar.iti.gov.br

PROF. JOSÉ LAERTE PIRES XAVIER JÚNIOR - Coorientador
Instituto de Ciências Exatas e Informática - PUC Minas

Eduardo Magno Lages Figueiredo:03462440640
Digitally signed by Eduardo Magno Lages Figueiredo:03462440640
Date: 2025.12.11 13:54:33 -05'00'

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

Documento assinado digitalmente
JOAO EDUARDO MONTANDON DE ARAUJO FILH
Data: 11/12/2025 13:32:14-0300
Verifique em https://validar.iti.gov.br

PROF. JOÃO EDUARDO MONTANDON DE ARAUJO FILHO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 5 de dezembro de 2025.

Acknowledgments

Inicio meus agradecimentos relembrando as palavras de Martin Buber: *"Toda vida real é encontro."* Ao longo desta jornada, descobri que cada etapa, cada desafio e cada conquista só foram possíveis graças às pessoas que caminham comigo, oferecendo apoio, inspiração e sentido. Assim, reconheço que esta dissertação não nasce apenas do esforço individual, mas se torna real pelas pessoas e pelos encontros que deram forma e significado a este caminho.

מודה אני לפניך מלך חי וקיים

Agradeço:

À minha esposa, Ana Silvia, que está sempre ao meu lado, me apoiando nos momentos de incerteza e celebrando comigo cada pequena conquista. Seu carinho, paciência e companheirismo foram essenciais para esta jornada. Sou grato por dividir a vida com você.

Aos meus pais, Dayse e Alberto, pilares da minha formação e exemplos de dedicação e integridade. Obrigado por acreditarem em mim, por me ensinarem o valor do estudo e por me darem todo o suporte necessário para que eu pudesse chegar até aqui.

Aos meus familiares, em especial ao meu avô **Juscelino**, que sempre foi fonte de inspiração, sabedoria e afeto. A presença e o incentivo de todos vocês me deram forças nos momentos mais desafiadores. Agradeço por vibrarem pelo meu sucesso.

Aos meus amigos, que me apoiaram e proporcionaram momentos de descontração quando mais precisei. Em especial ao **Bruno Calvo**, cuja amizade me acompanha desde o maternal, e aos amigos que conheci durante o mestrado e que dividiram comigo essa caminhada: **Altino, Bruna, Helena e Matheus**. Cada conversa, risada e partilha tornaram este percurso mais leve.

Ao meu orientador, Prof. Dr. Marco Tulio Valente, e ao meu co-orientador, Prof. Dr. Laerte Xavier, pela orientação competente, pela confiança depositada no meu trabalho e pela paciência ao longo de todo o processo. Sou grato pelos ensinamentos, pela disponibilidade e pela oportunidade de aprender com profissionais que tanto admiro.

Aos membros do ASERG, pelas trocas de conhecimento, pelas discussões enriquecedoras e pelo ambiente colaborativo que contribuiu imensamente para meu crescimento acadêmico e pessoal.

Aos membros da banca, Prof. Dr. Eduardo Figueiredo e Prof. Dr. João Eduardo Montandon, pela disponibilidade em avaliar esta dissertação, pelas críticas construtivas e pelas contribuições que certamente aprimoraram este trabalho.

Ao DCC/UFMG e ao PPGCC, pela oportunidade de realizar este mestrado e pelo ambiente acadêmico estimulante, onde pude crescer, aprender e desenvolver minhas habilidades de pesquisa.

Aos docentes e funcionários da UFMG, que direta ou indiretamente contribuíram para a minha formação.

“*I [...] am rarely happier than when spending an entire day programming my computer to perform automatically a task that would otherwise take me a good ten seconds to do by hand.*”

(Douglas Adams)

Resumo

Comentários no código-fonte são essenciais para a legibilidade e a manutenibilidade de sistemas de software, contribuindo diretamente para a redução do esforço de desenvolvimento futuro. No entanto, muitos desenvolvedores negligenciam essa prática, considerando-a tediosa e demorada. Modelos de Linguagem de Grande Escala (LLMs, do inglês *Large Language Models*) vêm sendo aplicados em diferentes tarefas de engenharia de software, incluindo a geração automática de comentários em código-fonte. Alguns estudos recentes investigaram esse potencial; entretanto, poucos avaliaram de forma sistemática a qualidade da documentação produzida por essas ferramentas. Em grande parte dos casos, as avaliações recorrem exclusivamente a métricas quantitativas, que não capturam a adequação semântica nem a utilidade prática dos comentários. Quando incluem análises qualitativas, empregam critérios simplificados, muitas vezes ignorando aspectos mais sutis de qualidade. Além disso, diversas investigações são conduzidas com conjuntos de dados reduzidos, o que restringe a generalização dos resultados. Outro ponto relevante é que vários trabalhos priorizam a sumarização de código, em detrimento da documentação estruturada.

Este trabalho apresenta resultados de uma avaliação abrangente da documentação de código gerada por três LLMs amplamente utilizados: OpenAI GPT-3.5 Turbo, GPT-4o e DeepSeek-V3. Utilizando 415 trechos de código extraídos de repositórios populares no GitHub, foram gerados 1.245 Javadocs. A avaliação realizada combina métricas automatizadas de similaridade textual (BLEU, ROUGE-L e METEOR) com análises qualitativas baseadas em avaliação humana. Além disso, o estudo investiga como características estruturais do código influenciam a qualidade dos comentários gerados.

Os resultados indicam que 58,8% dos comentários gerados apresentam qualidade equivalente à dos originais, enquanto 27,7% os superam. Observou-se também que modelos mais recentes, como o GPT-4o e o DeepSeek-V3, produzem documentação mais completa. Curiosamente, embora o tamanho, a complexidade e as dependências do código sejam frequentemente associados a sua baixa qualidade, tais atributos demonstraram uma correlação fraca, porém positiva, com a qualidade dos comentários gerados pelos LLMs — sugerindo que maior contexto pode favorecer a automação dessa tarefa.

Palavras-chave: Documentação de Código. Comentários de Código-Fonte. Modelos de Linguagem Grande. GPT. DeepSeek.

Abstract

Source code comments are essential for ensuring software readability and maintainability, significantly reducing future development effort. However, developers often overlook this task, viewing it as tedious and time-consuming. Large Language Models (LLMs) have been applied to a variety of software engineering tasks, including automatic source code comment generation. Some recent studies have investigated this potential; however, few have systematically evaluated the quality of the documentation produced by these tools. In most cases, evaluations rely exclusively on quantitative metrics, which fail to capture the semantic adequacy and practical usefulness of the generated comments. When qualitative analyses are included, they often employ simplified criteria, overlooking more nuanced aspects of quality. Furthermore, many studies are conducted on small datasets, limiting the generalizability of their findings. Another relevant point is that plenty of the existing work focuses on code summarization rather than structured documentation.

This dissertation presents results from a comprehensive evaluation of code documentation generated by three widely used LLMs: OpenAI GPT-3.5 Turbo, GPT-4o, and DeepSeek-V3. Using 415 code snippets extracted from popular GitHub repositories, we generated a total of 1,245 Javadocs. Our methodology combines automated textual similarity metrics (BLEU, ROUGE-L, and METEOR) with human evaluation, and further explores how intrinsic code properties influence the quality of the generated documentation.

Results indicate that 58.8% of the generated comments match the quality of the original ones, while 27.7% surpass them. More recent models, such as GPT-4o and DeepSeek-V3, produce notably more complete and informative documentation. Interestingly, although attributes such as code size, complexity, and dependency count are typically associated with poorer code quality, our findings reveal a weak yet positive correlation between these factors and the quality of LLM-generated comments—suggesting that additional contextual information may actually support and enhance the automation of code documentation.

Keywords: Code Documentation. Source Code Comments. Large Language Models. GPT. DeepSeek.

List of Figures

List of Tables

List of Abbreviations and Acronyms

AI	*Artificial Intelligence*
BLEU	*Bilingual Evaluation Understudy*
CBO	*Coupling Between Objects*
DIT	*Depth of Inheritance Tree*
HA-Score	*Human Assessment Score*
LCOM	*Lack of Cohesion in Methods*
LLM	*Large Language Model*
LOC	*Lines of Code*
MCP	*Model Control Protocol*
METEOR	*Metric for Evaluation of Translation with Explicit Odering*
NLP	*Natural Language Processing*
NOC	*Number of Children*
RFC	*Response For a Class*
ROUGE	*Recall-Oriented Understudy for Gisting Evaluation*
UFMG	*Universidade Federal de Minas Gerais*
WMC	*Weighted Methods per Class*

Contents

Chapter 1

Introduction

1.1 Motivation

Software documentation refers to a comprehensive set of artifacts that describe the architecture, components, functionality, and/or usage of a system, serving developers, stakeholders, and end users [17]. It plays a critical role in software development by promoting clarity, improving reliability, and enhancing overall efficiency throughout the development lifecycle [13].

Among the various forms of documentation, source code comments are the most widely used and preferred method among practitioners [2, 30]. Comments are textual annotations added to the source code. While they have no functional effect on the program's execution, their main purpose is to convey contextual information that may not be readily inferable from the code itself [2, 17]. This supplemental information can assist other developers in understanding the code. Well-written comments contribute significantly to software readability, ease of maintenance, and reduced effort in future development [13, 33, 52].

Figure 1.1 presents an example of a code comment in Java. The comment is the section of text enclosed between /** and */, appearing directly above the method definition. Specifically, this is a form of structured comment, a specialized subset of comments that follow formalized conventions. Structured comments facilitate consistent documentation and tool integration, while also reducing the effort required to maintain technical information. While there are various standards for structured comments, the example shown uses Javadoc, which is the most extensively studied format in the literature [44].

Typically, code comments are written in natural language, and their structure is neither defined by the programming language's syntax nor verified by the compiler. As a result, the responsibility for writing and maintaining high-quality comments is left to the developers [44]. However, because commenting is often perceived as time-consuming, tedious, and disruptive to the coding process, many practitioners tend to overlook this important task.

```
/**
 * Registers the text to display in a tool tip. The text
 * displays when the cursor lingers over the component.
 *
 * @param text the string to display. If the text is null,
 * the tool tip is turned off for this component.
 */
public void setToolTipText(String text) {
```

Figure 1.1: Code Comment Example

While assessing code comment quality remains an open problem due to the lack of a clear definition [44], research interest in this area has continued to grow [44]. Among the commonly studied attributes, consistency stands out [1, 45, 62, 65], particularly because comments are not always updated alongside code, leading to inconsistencies [3, 62]. Some studies rely on metric-based approaches to evaluate comment quality [48, 49, 50], while others apply machine learning techniques [19, 57]; yet, manual assessment remains the most commonly used method [44].

Artificial Intelligence has been used for software engineering tasks for many years. Recent advancements in the field, particularly in Large Language Models (LLMs), have further accelerated its adoption. These models have been successfully applied to a range of activities, including code generation [15, 25], program repair [27, 28], incident management [23, 35], and code summarization [4].

LLMs can be leveraged to address the documentation problem, offering a compelling solution that helps practitioners bridge the gap between efficient coding and effective documentation through AI-driven automated source code comment generation [24, 30]. These models demonstrate potential for producing accurate code documentation [30]. Moreover, they align with practitioners' needs by generating useful and informative descriptions of code behavior, helping to alleviate the burden of writing comments, which many developers find tedious [24].

Although LLMs are already being applied to a wide range of software engineering tasks, studies specifically focused on their use for source code comment generation remain relatively limited. A few recent works [13, 38, 59] have begun to explore this area, employing those models to generate documentation, summarize code, and produce comments at different levels of granularity, often using structured formats like Javadoc.

However, several important gaps remain. Some of these studies rely exclusively on quantitative metrics, which are known to have limited ability to reflect the semantic adequacy and practical usefulness of generated comments [14, 30]. Others incorporate qualitative evaluation [38], but use simplistic criteria, offering little insight into more nuanced aspects of comment quality. In some cases, the evaluations are based on very small datasets [13], which restricts the generalizability of the results. Additionally, many studies prioritize code summarization rather than documentation, often limiting outputs and overlooking structured formats [59].

Despite recent progress, there remains a limited understanding of how effectively large language models can generate high-quality, and semantically meaningful source code comments. Existing studies offer only a partial view, leaving open questions about their performance when evaluated with richer criteria, larger datasets, and a clearer focus on documentation rather than summarization. These limitations suggest that there is room for studies that: (i) use a dataset with a large number of comments; (ii) combine quantitative and qualitative analyses; (iii) correlate the quality of the generated comments with properties of the source code as measured by well-known metrics; and (iv) focus on source code documentation, rather than summarization, using structured formats.

1.2 Objectives

In this master's dissertation we aim to evaluate the effectiveness of LLMs in generating structured source code comments, specifically in the form of Javadocs, by analyzing the quality of the generated documentation. To achieve this, we defined the following specific objectives:

- **Create a high-quality dataset of Javadoc comments**: The first objective is to construct a curated dataset consisting of real-world Java code (specifically classes and methods) paired with their corresponding Javadoc comments. Java was selected due to its mature documentation ecosystem and its widespread use in both industry and prior academic literature, which facilitates consistent and objective analysis. This dataset serves not only as the evaluation benchmark for this study but also as a reusable resource for future research on automated documentation generation.

- **Evaluate the quality of LLM-generated Javadocs using textual similarity metrics**: The second objective is to quantitatively assess the quality of Javadocs generated by LLMs. To this end, we will prompt the models to generate Javadoc comments for the code snippets in our ground-truth dataset and compare the generated comments to the original Javadocs that accompanied those same snippets. We plan to use established textual similarity metrics (BLEU, ROUGE-L, and METEOR) which capture different aspects of lexical, structural, and semantic overlap, enabling a comprehensive evaluation of the generated documentation.

- **Evaluate the quality of LLM-generated Javadocs through human assessment**: The goal is to complement the quantitative analyses with a qualitative evaluation based on human judgment. To this end, evaluators will compare the generated Javadocs with the original documentation in the ground-truth dataset, rating each based on how closely it aligns with the reference. This evaluation will be conducted using a Likert scoring scale specifically developed for this study. Our goal is to capture nuances that automated metrics may overlook and to provide a human-centered perspective on the effectiveness of LLM-generated documentation.
- **Investigate how the source code structure affect the quality of LLM-generated Javadocs**: The final objective is to explore how intrinsic characteristics of source code influence the quality of documentation generated by LLMs. To this end, we will compute well-known source code metrics for each snippet using static analysis tools. Then, correlate these metric values with the scores from the human evaluation to identify potential patterns or relationships, thereby shedding light on contextual factors that may enhance or hinder the performance of LLMs in documentation tasks.

1.3 Results Overview

In this dissertation, we performed three complementary studies on the quality of LLM-generated Javadocs. First, we conducted a quantitative evaluation using textual similarity metrics; second, we carried out a qualitative assessment through human evaluation; and finally, we employed static source-code metrics to investigate how intrinsic code properties influence documentation quality. From 415 code snippets, we generated a total of 1,245 Javadocs using the selected LLMs. Overall, 58.8% of the generated comments matched the quality of the original ones, while 27.7% surpassed them, highlighting the strong practical potential of LLM-assisted documentation. We observed that newer LLMs consistently delivered more complete, accurate, and contextually informative comments. The findings also showed that automated similarity metrics provide only limited insight into semantic quality, reinforcing the need for human-centered evaluation. Moreover, the analysis of code characteristics revealed weak but positive correlations between factors such as size, complexity, and dependency count and the quality of the generated documentation, suggesting that richer contextual structure can support more effective documentation generation.

1.4 Outline of the Dissertation

The remainder of this master's dissertation is organized as follows:

- **Chapter 2** presents the background that support the understanding of this master's dissertation and related work. We begin by introducing source code comments, particularly focusing on Javadoc, followed by an overview of Large Language Models, the core technology investigated in this research. We then describe the evaluation metrics adopted throughout the study and review prior work in the field, highlighting how our work extends and complements existing efforts.

- **Chapter 3** describes the methodology used to construct the dataset and configure the LLMs for our evaluation. We start by outlining the process of collecting code snippets, generating Javadocs using LLMs, and computing code metrics. We then detail the selected repositories, models, and prompt engineering strategies. Furthermore, we explain the curation of the ground truth and discuss potential threats to validity related to the dataset and model choices. This chapter establishes the experimental setup used across the subsequent evaluations.

- **Chapter 4** presents our first empirical study: a quantitative evaluation of LLM-generated Javadocs. We assess the quality of the generated documentation by comparing it to real-world Javadocs using established similarity metrics such as BLEU, ROUGE-L, and METEOR.

- **Chapter 5** presents our second empirical study: a qualitative assessment of LLM-generated Javadocs based on human evaluation. Two evaluators rated the similarity between the generated and original Javadocs using a novel evaluation scale we introduced. We analyzed these scores to assess the quality of the generated documentation and compared the results with those from the quantitative analyses.

- **Chapter 6** details our third empirical study: a quantitative investigation into how structural characteristics of source code impact the quality of documentation generated by LLMs. Using source code metrics such as complexity, size, and cohesion, we analyze their correlation with the human evaluation scores from the qualitative evaluation.

- **Chapter 7** summarizes the main contributions of this dissertation and reflects on its findings. We discuss the implications of our results, limitations of our approach, and outline directions for future work in the field of automated source code documentation generation using LLMs.

Chapter 2

Background and Related Work

2.1 Introduction

In this chapter, we discuss the background of this master's dissertation. We also present the state-of-the-art studies that relates to our work. For that, Section 2.2 introduces a fundamental type of software documentation: source code comments, highlighting Javadoc comments in particular. Section 2.3 provides an overview of Large Language Models, AI systems focused on understanding and generating human language. Section 2.4 outlines the evaluation metrics adopted in our study. Section 2.5 reviews relevant prior studies, positioning our work within the broader research landscape. Finally, Section 2.6 offers concluding remarks that summarize the key background elements and outline how they support the analyses presented in this study.

2.2 Source Code Comments

Documentation plays an important role in software development by promoting clarity, improving reliability, and enhancing overall efficiency throughout the development lifecycle [13]. It encompasses a wide range of artifacts designed to communicate information about a software system to various stakeholders, including developers, managers, and end-users [17]. These artifacts can describe what a software does, how to use it and how it operates [3]. Examples include Code Comments; User Manual; FAQ; Migration Guide; Release Note/Change Log; and API Reference [2].

Among the various forms of documentation, source code comments remain the most widely used and preferred method among practitioners [2, 30]. Comments are textual annotations embedded within the source code that have no functional effect on the program's execution or behavior. Their main purpose is to convey contextual information, such as design rationale, implementation logic, and intended usage, which may not be readily inferable from the code itself [2, 17].

This supplemental information can assist other developers in understanding the code, particularly in collaborative or long-term projects. Well-written comments contribute significantly to software readability, ease of maintenance, knowledge transfer, and reduced effort in future development tasks [13, 33, 52].

Within the category of code comments, structured comments represent a specialized subset that follows formalized conventions. These are typically used to annotate source code with metadata and descriptive information in a machine-readable format. Their primary purpose is to enable the automatic generation of external documentation, thereby extending the usefulness of in-code annotations for both internal and external stakeholders.

Structured comments offer several advantages, including improved consistency in documentation, seamless integration with development tools, and reduced manual effort in maintaining up-to-date technical documentation. Prominent examples include Javadoc[1] for Java, Pydoc[2] for Python, and Doxygen[3] for C and C++. Among these, Javadoc stands out as the most extensively studied in the literature [44], largely due to its widespread adoption and its influence on the design of other documentation systems.

In this dissertation, we focus on Javadoc comments, one of the supported comment types in Java. Java provides three primary mechanisms for embedding comments within source code: inline comments; multi-line comments; and Javadoc comments, a specialized form of multi-line comments originally intended for generating external documentation [43]. Among these, Javadocs are particularly significant due to their structured syntax and formal support within the Java ecosystem.

The Javadoc tool is the official API documentation generator for the Java programming language. It processes Javadoc comments to produce comprehensive, navigable HTML documentation directly from the source code. These comments begin with '/**' and end with '*/', and they are conventionally placed immediately before class, method, or field declarations that they describe [32]. The extracted documentation can include descriptions, parameter details, return types, cross-references, and other metadata relevant to the code's usage and behavior.

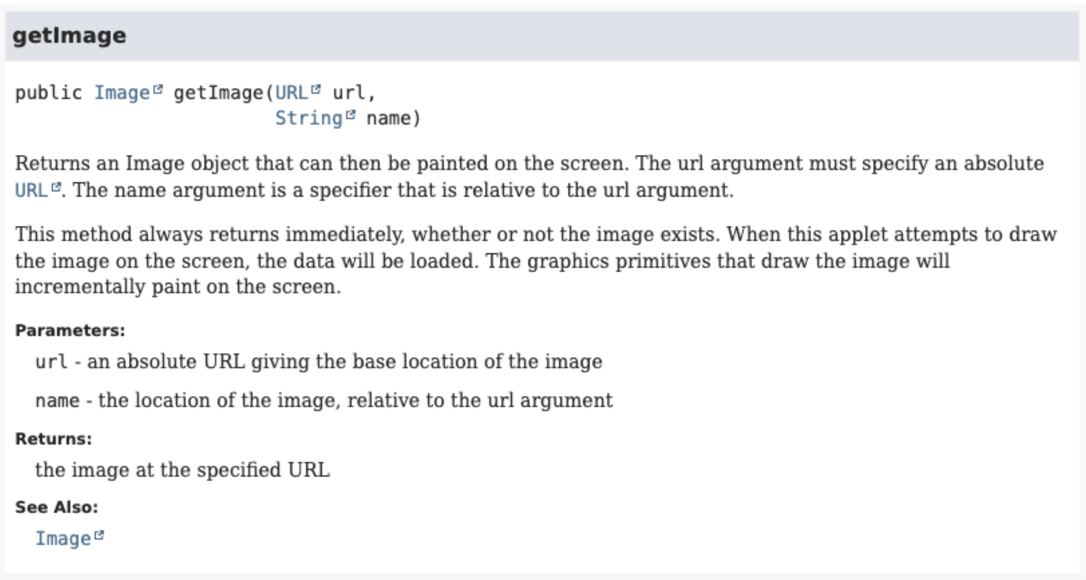
Figure 2.1 illustrates a typical example of a Javadoc comment applied to a method. The comment describes the method's purpose, its parameters, return value, and provides additional context regarding the method's behavior during execution. The use of tags and inline HTML demonstrates Javadoc's flexibility in producing rich, hyperlinked documentation outputs. The resulting documentation, rendered from the HTML output generated

[1] https://www.oracle.com/java/technologies/javase/javadoc-tool.html
[2] https://docs.python.org/3/library/pydoc.html
[3] https://www.doxygen.nl/

by the Javadoc tool, is shown in Figure 2.2.

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute <a href="#{@link}">{@link URL}</a
>. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Figure 2.1: Example of a method documented using Javadoc (source: Oracle [10])



getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an Image object that can then be painted on the screen. The url argument must specify an absolute URL. The name argument is a specifier that is relative to the url argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

url - an absolute URL giving the base location of the image

name - the location of the image, relative to the url argument

Returns:

the image at the specified URL

See Also:

Image

Figure 2.2: Rendered documentation generated from the Javadoc comment

Javadoc's influence extends beyond Java, having inspired similar documentation systems in other programming languages, such as Python's Pydoc and C++'s Doxygen. Its widespread adoption in both open-source and industrial contexts makes it a compelling subject of study—particularly in the context of automatic documentation generation and evaluation.

2.3 Large Language Models

Large Language Models (LLMs) are a class of Artificial Intelligence (AI) systems designed to understand and generate human language. These models are typically built on deep neural networks—particularly transformer architectures—and trained on massive volumes of textual data. Modern LLMs often contain hundreds of billions of parameters, allowing them to capture complex linguistic patterns and generate coherent, contextually appropriate text [8, 64]. As a result, LLMs are capable of producing fluent, coherent, and contextually relevant language across a range of language tasks, such as text generation, machine translation, and question answering [8, 16, 64].

Although originally developed for Natural Language Processing (NLP), LLMs have recently gained space in the domain of software engineering. Despite the syntactic and semantic differences between natural and programming languages, LLMs have demonstrated ability to process and generate code. This is largely due to their exposure to large-scale datasets containing both source code and natural language, allowing them to learn how code relates to its textual explanations [16].

This adaptability has led to the successful application of LLMs across a wide range of software engineering tasks. These include code generation [15, 25], where models produce source code from natural language specifications; program repair [27, 28], which involves suggesting fixes to faulty code; and incident management [23, 35], such as classifying and diagnosing software bugs. More recently, LLMs have also been explored for code summarization, a task that involves producing natural language explanations or documentation for existing code artifacts [4]. Automatically generated summaries have the potential to improve code comprehension, support maintenance, and reduce the manual effort required for documentation.

Thus, as LLM architectures and training methods continue to evolve, their performance on both natural and programming language tasks is expected to improve further. Their general-purpose capabilities position them as valuable tools in the software development pipeline, offering scalable solutions to language-driven tasks.

However, a relevant factor in the effectiveness of LLMs is the way in which users interact with them. Interacting effectively with LLMs requires a different approach from traditional programming. Instead of writing code to specify behavior, users influence LLM outputs through carefully crafted natural language inputs—known as prompts. The practice of designing and refining these inputs to achieve specific outcomes is known as prompt engineering [8, 16].

Prompt engineering has become a foundational technique in leveraging the capabilities of LLMs. Since LLMs are sensitive to input phrasing and contextual cues, even minor modifications to a prompt can significantly influence the relevance, accuracy, and overall quality of the generated output. As a result, crafting well-structured and semantically precise prompts is crucial for achieving optimal performance. This paradigm shift positions LLMs as a new kind of computation engine, where programming is done through language rather than code [4].

In software engineering, prompt engineering plays an important role in enabling LLMs to perform specialized tasks such as generating method summaries, producing documentation, or explaining code logic. By tailoring the prompt's structure and linguistic framing, practitioners can guide the model toward specific interpretations, control the level of abstraction, and fine-tune the granularity of the response. As such, prompt design has emerged as a critical skill for effectively applying LLMs in code-centric workflows.

To support these use cases, several prompting strategies have emerged as standard approaches when interacting with LLMs. The most common are zero-shot, one-shot, and few-shot prompting, each offering different levels of contextual guidance to the model. In zero-shot prompting, the model is provided only with a task description and is expected to produce a suitable response without any examples. One-shot prompting adds a single example to the prompt, offering a minimal template to illustrate the desired output. Few-shot prompting extends this concept by including multiple examples, aiming to reinforce a pattern and improve the model's understanding of the task structure. These strategies play a central role in shaping the model's behavior, particularly in domain-specific applications [30].

2.4 Evaluation Metrics

In this dissertation, we perform both quantitative and qualitative evaluations to investigate the quality of documentation generated by LLMs. The quantitative analysis relies on two complementary perspectives: (i) textual similarity metrics to measure how closely the generated comments align with human-written ones, and (ii) source code metrics to evaluate whether intrinsic properties of the code affect the quality of the generated documentation. This section introduces the metrics used in each of these perspectives.

2.4.1 Textual Similarity Metrics

Textual similarity refers to the commonality between two texts, typically measured in terms of lexical, structural, or semantic overlap. It can be defined as the degree of shared information between two sequences, where greater commonality implies higher similarity [61]. This concept is widely used in the field of NLP, particularly in tasks involving the evaluation of machine-generated text.

Several automated metrics have been proposed to assess textual similarity. These metrics are especially valuable in scenarios where system-generated outputs are compared against known references [12]. They offer a consistent, scalable, and reproducible framework for evaluation, enabling reliable and efficient quantitative analysis.

Such metrics are typically categorized according to the aspect of similarity they emphasize. Precision-based metrics, measure how much of the generated content matches the reference, favoring outputs that are concise and lexically aligned [12]. Recall-based metrics focus on how much of the reference is captured in the generated text, thus rewarding more comprehensive outputs [12]. Additionally, balanced metrics, aim to combine both precision and recall [47], addressing their individual limitations.

In this dissertation, we employ three widely used textual similarity metrics to evaluate the quality of generated Javadocs: BLEU [42], ROUGE-L [37], and METEOR [6]. These metrics provide complementary perspectives on how closely the generated documentation aligns with the reference one. Each metric is described below.

BLEU (Bilingual Evaluation Understudy) is a precision-focused metric that measures the overlap of n-grams between a candidate and reference text, emphasizing exact matches over broader semantic similarity. [47]. It has historically been a standard for evaluating machine translation and summarization systems [18, 51]. However, it has well-known limitations, for example, it struggles with evaluating semantically correct but lexically divergent outputs [18, 40, 47] (e.g., "returns the number of elements" x "gives the list size"). To mitigate this, we applied a smoothing function, as recommended by Khan and Uddin [30], which enhances BLEU's sensitivity to partial and low-frequency matches, especially relevant in shorter sequences like Javadoc comments.

ROUGE-L is part of the ROUGE (Recall-Oriented Understudy for Gisting Evaluation) family of metrics, which includes several variants. Among them, ROUGE-L—based on the Longest Common Subsequence—was selected due to its effectiveness in capturing sentence-level structure without requiring exact matches. ROUGE emphasizes the count of overlapping units such as n-grams, word pairs, and sequences, allowing it to measure how much of the reference content is preserved in the generated text, making it particularly suitable for evaluating the completeness and informativeness of the generated Javadocs.

METEOR (Metric for Evaluation of Translation with Explicit ORdering) is based on the general concept of unigram matching, combining precision, recall, and a customized score that considers the degree to which words are ordered correctly in the translation [47]. Unlike BLEU and ROUGE, METEOR incorporates stemming, synonymy, and paraphrasing, introducing semantic awareness into its evaluation. These characteristics make METEOR especially robust in capturing the meaning and intent of the generated text, aligning more closely with human judgments of quality.

Despite its limitations, BLEU remains a prevalent metric in the literature for historical and comparative purposes. Nevertheless, with the advent of LLMs and the increasing demand for semantically accurate text generation, recent studies have advocated for alternative metrics. In particular, METEOR and ROUGE-L have demonstrated stronger alignment with human judgment across a variety of natural language generation tasks [14, 21, 36, 47].

2.4.2 Source Code Metrics

Achieving high quality is a key goal in software engineering. Software quality can be broadly understood as the extent to which a system fulfills its specified requirements, encompassing multiple attributes such as maintainability, reliability, and complexity. Given its abstract nature, a common strategy for managing software quality is to decompose the concept into more concrete, measurable characteristics and evaluate these through tangible indicators [54]. Such indicators can be extracted from the analysis of the source code itself.

A systematic way to assess internal software quality is through source code metrics. These metrics map specific characteristics of the code into numerical values, enabling objective evaluation [41]. Source code metrics are typically collected through static analysis, a technique that examines the structure of source code without executing it. Static analysis tools parse and analyze the code to extract relevant information about its syntax, design, and interdependencies.

There are many metric suites designed to assess different aspects of source code. Among these, the Chidamber and Kemerer (CK) suite [9] stands out as one of the most widely adopted frameworks for analyzing object-oriented source code [58]. The CK metrics were specifically designed to capture key principles of object-oriented design and have been extensively used in both industry and academic research [7, 26, 39]. They provide valuable insights into code complexity, inheritance structures, class coupling, and cohesion factors.

In this dissertation, we employ a subset of the CK metrics, complemented by two additional measures. These metrics are used to capture different structural and lexical characteristics of the source code. Each metric is described below.

- **WMC (Weighted Methods per Class)**: Represents the combined complexity of all methods within a class. Higher WMC values indicate greater internal complexity.
- **CBO (Coupling Between Objects)**: Measures how many other classes a given class is coupled to. A high CBO suggests greater external dependency.
- **RFC (Response for a Class)**: Denotes the number of unique methods that can be executed in response to a message sent to a class. Higher RFC values imply more potential behaviors.
- **LCOM (Lack of Cohesion of Methods)**: Evaluates the degree to which methods in a class are related to one another. Low cohesion (i.e., high LCOM) indicates a more fragmented or dispersed class structure.
- **LOC (Lines of Code)**: Indicates the physical size of the code snippet, typically associated with the scale and scope of functionality.
- **Unique Words Quantity**: Measures the lexical diversity within the code snippet, which reflects the variety of identifiers and terms used.

2.5 Related Work

Previous studies in the literature evaluate the use of LLM in software development activities, including those directly related to source code documentation. In this section, we summarize the most relevant works, highlighting their goals and main limitations. Table 2.1 provides a structured comparison of these studies in chronological order, facilitating a clearer understanding of their contributions and differences.

One of the first relevant contributions was presented by Hu et al. [24], they conducted a mixed-methods study to investigate practitioners' expectations for automated comment generation. Their findings revealed a clear gap between the capabilities of existing tools and the needs of developers. In particular, they highlighted that commonly used evaluation metrics—such as BLEU—do not align well with practitioners' priorities, especially with regard to content adequacy and practical relevance. Developers emphasized that high-quality comments should ideally convey what the method does, how it does it, and why it is implemented in that way. However, the study does not consider newer language models that have greatly advanced the fluency, accuracy, and overall quality

Table 2.1: Summary of related work on LLMs for software documentation

Authors	Venue	Goal	Analyses	Limitations
Hu et al. [24]	ICSE 2022	Understanding practitioners' expectations for comment generation	Qualitative + Quantitative	Did not include modern LLMs, limited applicability to current models
Khan and Uddin [30]	ASE 2023	Automatic comment generation	Quantitative	Single metric (BLEU), no human evaluation, no baseline comparison
Evtikhiev et al. [14]	JSS 2023	Evaluation of text similarity metrics for code generation	Quantitative + Qualitative	Does not analyze documentation, which involves different linguistic and semantic aspects
Luo et al. [38]	Preprint 2024	Repository-level documentation automation	Quantitative + Qualitative	Binary preference only, possible subjectivity
Dvivedi et al. [13]	AIware 2024	Multi-level code documentation	Qualitative	Only 14 Python snippets, no comparison with original comments
Katzy et al. [29]	PROMISE 2025	Multilingual LLM-generated comments	Quantitative + Qualitative	Only open-source LLMs, excludes closed-source models common in industry
Sun et al. [59]	ICSE 2025	Source code summarization	Quantitative + Qualitative	Summaries limited to one sentence, no class-level or full documentation focus

of generated documentation. Consequently, certain limitations identified by the authors may no longer be as relevant in the context of modern LLM-based tools.

Following this early work, Khan and Uddin [30] evaluated the use of GPT-3 Codex for automatic comment generation across six programming languages, reporting that it outperformed previously evaluated LLMs with an average BLEU score of 20.6%. Their results suggest that GPT-generated documentation is comparable to human-written text in terms of readability and informativeness. However, their evaluation was entirely quantitative and relied exclusively on a single automated metric: BLEU. This metric has well-documented limitations, particularly its inability to adequately assess semantically correct but lexically diverse outputs [18, 40, 47]. Furthermore, their study did not include any form of qualitative assessment or human evaluation, which is widely regarded as the gold standard for evaluating natural language generation tasks.

In the context of assessing automatic metrics, Evtikhiev et al. [14] evaluated the suitability of six popular and widly used text similatiry metrics—BLEU, ROUGE-L, METEOR, ChrF, CodeBLEU, and RUBY—for assessing the quality of code generation models. By comparing the metrics' outcomes against human judgments, they demonstrated that none of these metrics reliably aligns with human preferences, especially when score differences are small. The study cautions against interpreting minor score variations as meaningful improvements and emphasizes the need for evaluation methods that better reflect semantic adequacy and human judgment. However, the study is limited to code generation tasks and does not explore documentation-related outputs, which may involve different linguistic and semantic properties.

Expanding the scope toward more comprehensive documentation approaches, Luo et al. [38] proposed RepoAgent, a framework that leverages LLMs to automate documentation at the repository level, enabling the generation and maintenance of high-level documentation as the codebase evolves. Their approach was evaluated using both quantitative metrics and qualitative methods, including human assessments. Notably, in two repositories, human evaluators preferred the generated documentation over the original in 70% and 91% of cases, respectively. However, their human evaluation relied on a binary preference model, in which each generated comment was judged solely as either better than or not better than the original.

In a subsequent effort to compare the capabilities of different LLMs, Dvivedi et al. [13] conducted a comparative analysis of multiple LLMs for code documentation, using a checklist-guided human evaluation. Notably, their study assessed documentation quality across multiple levels, including inline, function, and file-level comments. Their findings indicate that LLM-generated documentation frequently outperforms human-written content, and that closed-source models generally performed better than open-source alternatives. However, the analysis was limited to just 14 Python code snippets and did not include direct comparisons with the original developer-written comments, resulting in the absence of a reliable ground truth.

Exploring other contexts, Katzy et al. [29] conducted a qualitative and quantitative investigation into LLM-generated code comments in multilingual codebases. Their study highlights critical challenges in generating comments in five non-English languages, proposing a detailed taxonomy of errors and demonstrating that widely used evaluation metrics often fail to align with human judgment. The work offers insight into the linguistic limitations of current LLMs and the shortcomings of popular textual similarity metrics when used in multilingual scenarios. However, as their analysis focuses exclusively on open-source models, the findings may not fully reflect the behavior of newer or proprietary LLMs that dominate real-world development environments.

Finally, as a broader effort to evaluate LLM-based documentation, Sun et al. [59] present a large-scale study on LLM-based source code summarization, evaluating four models (GPT-3.5, GPT-4, CodeLlama-Instruct, and StarChat-β) across ten programming languages. Like our work, they employed BLEU, ROUGE-L, and METEOR for text similarity, and conducted a human evaluation. However, their study focuses primarily on function-level code summarization. Furthermore, their prompts request the generation of summaries limited to a single sentence. For example, the one-shot prompt used in their study is as follows: "Please generate a short comment *in one sentence* for the following function: code ."

2.6 Concluding Remarks

In this chapter, we introduced the key concepts of this master's dissertation and reviewed prior research aligned with our objectives. We outlined the role of source code comments—particularly Javadocs—in supporting software comprehension and maintainability. We also examined the capabilities of LLMs, highlighting their applications in software engineering and the importance of prompt engineering in enhancing output quality. Additionally, we presented key evaluation metrics, including textual similarity measures and source code metrics. Finally, we discussed relevant prior research to position our work within the existing literature.

In the following chapters, we build upon these foundations to conduct a multi-perspective evaluation of LLM-generated Javadocs. This includes quantitative analyses using textual similarity metrics, qualitative assessments through human evaluation, and source code analysis using established metric suites. Our objective is to assess the effectiveness of leveraging LLMs for automated Javadoc generation.

Chapter 3

Dataset and LLM Models

3.1 Introduction

In this master's dissertation, we assess the effectiveness of LLMs in generating structured source code comments, more specifically, Javadocs. To accomplish that, we evaluate the quality of Javadocs generated by such models. The assessment, presented in the following chapters, was conducted by comparing the generated Javadocs to real-world documentation for the same code snippets, using both quantitative (Chapter 4) and qualitative (Chapter 5) methods. We also examine how source code metrics influence the LLMs' ability to generate high-quality documentation (Chapter 6). The current chapter presents the definitions, procedures, and artifacts used throughout the entire evaluation, detailing the methodology followed and the construction of the dataset.

The remainder of this chapter is organized as follows. Section 3.2 describes the overall process to build our studied dataset, including Javadoc extraction, generation of code comments using LLMs, computation of source code metrics, and the analyses performed. Section 3.3 details the selected repositories. Section 3.4 explains the process adopted to define the ground truth, which serves as the baseline for the subsequent evaluations. Section 3.5 describes the process of generating the documentation, presenting the LLMs and the prompt engineering strategy employed. Finally, Section 3.6 discusses potential threats to validity associated with the dataset and model selection, and Section 3.7 presents concluding remarks.

3.2 Dataset Collection Procedures

Our dataset was built using a multi-stage approach. The overall collection procedure is illustrated in Figure 3.1. It was carried out mainly through an automated pipeline to ensure consistency and enhance reproducibility. We briefly describe the process below and explain it in more detail in the following sections.

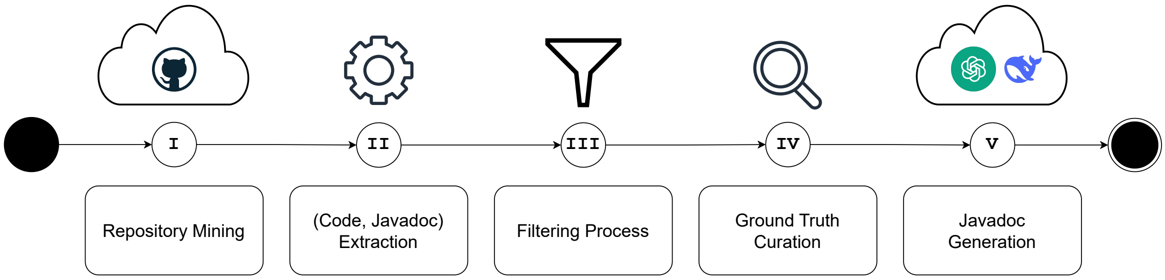


Figure 3.1: Dataset Collection Procedure

First, we manually selected the repositories to be analyzed. Specifically, we chose the top three Java repositories on GitHub that correspond to libraries or frameworks and had the highest number of stars. Then, we used a Python-based microservice, developed by us, to mine the contents of these repositories. This same microservice was also responsible for identifying Java code elements accompanied by Javadoc comments and extracting them along with their corresponding documentation. These extracted pairs are referred to as (Code, Javadoc) pairs throughout this dissertation.

After extracting all (Code, Javadoc) pairs, we applied a filtering step to remove any Javadocs that contained external references or were written in languages other than English. This filtering ensures linguistic consistency and that AI-driven comment generation relies solely on the context provided by the code itself. From the remaining pairs, we created a sample to serve as our ground truth. Each of these pairs was then manually inspected and validated to ensure the dataset contained only high-quality examples suitable for comparison and evaluation.

Finally, to generate new documentation for the curated code snippets, we employed a second Python-based microservice. This component was responsible for interfacing with the selected LLMs via their respective APIs. For each snippet, the microservice submitted a prompt requesting the generation of a Javadoc comment to document the given code. The resulting comments, produced by the LLMs, were then collected and stored. These generated comments enable a direct comparison with the original documentation in the subsequent analyses.

3.3 Repositories Characterization

In this study, we seek to analyze the quality of source-code comments generated by LLM. To accomplish that, we gathered code snippets along with their corresponding Javadocs from three popular Java-based repositories in GitHub. Java was selected as the target language—along with Javadoc—due to its structured syntax, long-standing popularity in both industry and academia, and the maturity of its documentation practices. Javadoc, in particular, offers a standardized and well-established format for code annotation, making it an ideal source for analyzing the relationship between implementation and documentation.

To reflect a broad range of programming paradigms and application domains, we selected three well-established and actively maintained repositories from the Java ecosystem: SPRING-BOOT, SPRING-FRAMEWORK, and GUAVA. Each of these projects plays a significant role in modern Java development and offers a rich corpus of code paired with relevant Javadoc documentation.

SPRING-FRAMEWORK [55] serves as the core foundation of the Spring ecosystem and offers a comprehensive programming and configuration model for Java applications. It includes essential features such as dependency injection, aspect-oriented programming, transaction management, and integration support for web applications, data access, and enterprise services.

SPRING-BOOT [56] is a widely adopted framework that supports the creation of stand-alone, production-ready applications based on the Spring platform. It promotes a convention-over-configuration philosophy, which reduces boilerplate code and simplifies application setup. By providing embedded servers and automatic configuration, it enables rapid development and deployment workflows.

GUAVA [20], developed and maintained by Google, is a versatile set of core libraries that augment the standard Java Development Kit. It provides advanced data structures, functional programming utilities, caching solutions, and concurrency support, among other features.

Table 3.1 summarizes the number of extracted (code, Javadoc) pairs from each repository, providing an overview of the amount of documentation available in each project. The SPRING-FRAMEWORK repository contains the largest number of documented elements, with more than 34,022 pairs extracted. The SPRING-BOOT repository contributes with 14,196 pairs, while GUAVA adds 12,471 pairs. Overall, the dataset comprises 60,689 (code, Javadoc) pairs.

Table 3.1: Extractions by Repository

Repository	Classes	Methods	Others	TOTAL
SPRING-FRAMEWORK	7,515	14,665	11,842	**34,022**
SPRING-BOOT	5,925	2,622	5,649	**14,196**
GUAVA	2,822	4,874	4,775	**12,471**
TOTAL	**16,262**	**22,161**	**22,266**	**60,689**

3.4 Ground Truth Curation

The qualitative analysis in this study relies on human evaluation, which is recognized as resource-intensive and time-consuming [52]. To address this, we selected a representative sample from the (code, Javadoc) pairs initially extracted from the studied repositories. This sample represents the ground truth of our study. It was used across all evaluation stages, both quantitative and qualitative, in order to ensure methodological coherence and to facilitate comparison of results. We decided to focus our evaluations on class- and method-level Javadocs, as these are typically the most informative and commonly used forms of documentation in Java projects

Prior to sampling, and to ensure that our ground-truth comments contained only code-derived context, we followed Khan and Uddin [30] and filtered out comments that: (i) were not in English; (ii) contained special tokens or external links (e.g., and `https://`); or (iii) included author metadata (i.e., `@author` tag). This ensures that AI-driven comment generation relies solely on code-derived context and that it preserves privacy. After this filtering, we determined the sample size using a 95% confidence level and a 5% margin of error, which yielded 142 class-level and 273 method-level Javadoc comments selected through uniform random sampling, 415 in total.

To ensure the quality and reliability of this subset, the selected comments were reviewed by the author of this dissertation and a colleague. Both evaluated the adequacy, clarity, and consistency of each comment with respect to its corresponding source code element. Both reviewers are experienced software engineers, with five and seven years of experience, respectively. Independently, each one assessed each comment, discarding and replacing any comment identified as inaccurate or misleading. During the filtering process, 27 comments (6.5% of the targeted 415) were replaced, resulting in a total of 442 comments analyzed to achieve the final sample of 415 valid entries.

As an illustrative example, Figure 3.2 shows a Javadoc comment—alongside an excerpt of the code it documents—that was excluded during this process. The comment begins with the phrase "this atrocity is here" which conveys the author's subjective opinion about the code's design rather than offering an objective description of its functionality.

Since this type of language is not directly informative about the code's behavior and could bias the evaluation of generated Javadocs, it was deemed unsuitable for inclusion in the curated dataset. As a result, this comment was subsequently replaced.

```
/**
 * This atrocity is here to let us report several of the elements in the stream
   if there were more
 * than one, not just two.
 */
private static final class ToOptionalState {
    static final int MAX_EXTRAS = 4;

    @CheckForNull Object element;
    List < Object > extras;
                                          [...]
    Object getElement() {
        if (element == null) {
            throw new NoSuchElementException();
        } else if (extras.isEmpty()) {
            return element;
        } else {
            throw multiples(false);
        }
    }
}
```

Figure 3.2: Example of Javadoc inconsistent with the source code

3.5 Javadoc Generation

In this section, we describe how the LLMs were used to generate Javadoc documentation for the code snippets in our ground-truth dataset. Section 3.5.1 presents the models employed in this process, while Section 3.5.2 explains how we built the prompt employed to generate the Javadocs with the LLMs.

3.5.1 Models Characterization

For our research, we leveraged three Large Language Models (LLMs), each selected based on specific attributes that contributed to a comprehensive and representative analysis. The selection was guided by a combination of practical and technical considerations. For example, we first considered the cost, availability and popularity of the models. In addition, performance benchmarks and architectural characteristics were also considered. Together, these models allowed us to explore a meaningful range of LLM capabilities. Next, we present each of the selected models.

OpenAI GPT-3.5 Turbo. This model was chosen primarily for its cost-effectiveness and widespread adoption at the time of the study. Among commercially available LLMs, GPT-3.5 Turbo offered one of the best trade-offs between performance and affordability, making it a preferred choice among developers and researchers working under budget constraints. Furthermore, as the earliest released model among those included in our evaluation, it serves as a practical lower bound. This baseline allows us to better contextualize the improvements introduced by newer architectures and training methods.

OpenAI GPT-4o. At the time of our study, GPT-4o represented the most advanced publicly available model from OpenAI that supported text-based reasoning. It was selected to provide insight into the upper bounds of current LLM capabilities within the OpenAI ecosystem. Its inclusion helps illustrate the potential of state-of-the-art models in addressing complex tasks and demonstrates what can be achieved when cost is less of a constraint. GPT-4o also offers a point of contrast to GPT-3.5 Turbo, showcasing the progression in model quality within the same family of architectures.

Deepseek-V3. In addition to OpenAI's models, we included Deepseek-V3, a prominent LLM developed outside the OpenAI ecosystem. At the time of the study, Deepseek-V3 had gained significant attention for its competitive performance in a variety of benchmarks, often matching or surpassing the capabilities of GPT-4-class models in specific tasks. The inclusion of Deepseek-V3 broadens the scope of our comparison and provides valuable insight into how alternative models fare in relation to dominant offerings. It also reflects a growing diversification in the LLM landscape, where multiple providers are achieving state-of-the-art results.

3.5.2 Prompt Engineering

For each (code, Javadoc) pair in our ground truth dataset, we extracted the source code and used it as input to the LLMs, prompting them to generate corresponding documentation as Javadoc comments. To guide the model effectively, we employed a one-shot strategy, wherein a single annotated example was provided alongside task-specific instructions. This approach was selected based on a series of preliminary experiments comparing zero-shot, one-shot, and few-shot prompting configurations. One-shot prompting, in alignment with findings from prior literature [30], consistently produced superior results. In contrast, expanding the prompt with additional examples (few-shot prompting) did not lead to significant performance gains. Regarding the temperature, we adopted the default value of 1 (on a 0–2 scale), which is also widely used by practitioners, as it provides a balanced trade-off between creativity and determinism while ensuring consistency and reproducibility.

The prompt template used for generation is illustrated in Figure 3.3. It was developed through multiple iterative refinements to enhance clarity, reduce ambiguity, and align the model's outputs more closely with conventional Javadoc standards. As part of the prompt design, we selected a representative example from each granularity level—class-level and method-level documentation. These examples were randomly chosen and excluded from the ground truth.

````
Context: Suppose you are a Java developer who needs to document some
[`methods' | `classes'] using source code comments following the Javadoc format
.
For example, for the following [`method' | `class']:
```java
[code example]
```
You should generate the following Javadoc comment:
[javadoc example]

Task: Generate a single comment (in Javadoc format) for the following
[`method' | `class']
```java
[extracted code]
```
In your answer, only include the suggested comment.
````

Figure 3.3: Prompt structure

3.6 Threats to Validity

In the context of an empirical research, it is crucial to acknowledge and address potential threats to validity [63]. In this section we address the ones that can potentially impact the following studies that depends on the dataset described in this chapter.

This master's dissertation focuses exclusively on Java methods and classes extracted from three open-source repositories, a scope that may pose potential threats to external validity due to the limited number of sources. To address this concern, the repositories were carefully selected to provide a diverse and representative overview of key aspects of Java development, including different coding styles, architectural patterns, and documentation practices. From each repository, a statistically significant sample of methods and classes was selected to ensure adequate variability and generalizability within the context of the study.

The Javadocs analyzed in this study may not fully capture the range of documentation practices used across all development contexts, which presents a potential threat to construct validity and may limit the broader applicability of our findings. To mitigate this concern, it is important to highlight that the selected repositories have been developed and maintained by thousands of contributors. This extensive and diverse contributor base increases the likelihood that the Javadoc styles and practices observed in the dataset are reflective of real-world, industry-relevant Java development standards.

Additionally, there is a possibility that the Javadocs comprising the ground truth do not consistently reflect high-quality documentation, which could compromise their suitability as a benchmark. To address this concern, all documentation comments were manually reviewed, and only those meeting predefined quality criteria were included in the ground truth. Through this process, 94% of the analyzed Javadocs were deemed to meet acceptable quality standards, highlighting the overall reliability of the dataset. Moreover, the repositories selected for this study are maintained by relevant organizations, further reinforcing the representativeness of the Javadocs used for evaluation.

The implementation of the Javadoc extraction process may also pose a threat to internal validity, as any inaccuracies or inconsistencies in the extraction logic could compromise the integrity of the dataset. To mitigate this risk, the extraction logic was carefully designed and implemented following rigorous software engineering practices, including testing across a diverse set of Java files. These validation efforts ensured that the extracted Javadocs accurately reflected the original source code, thereby minimizing the likelihood of systematic errors impacting the study's results.

Finally, a potential point of concern arises from the fact that the LLMs may have been exposed to the codebases and Javadocs under analysis, given that they are trained on publicly available open-source data. Such prior exposure could lead to data leakage, thereby inflating the performance of the models and compromising the validity of the evaluation. To mitigate this risk, we analyzed the project's commit history and restricted our ground-truth dataset to code and associated comments committed after the GPT-3.5 Turbo training cutoff date. This precaution ensures that the model had no access to the specific instances used in our evaluation, preserving the integrity and fairness of the assessment. We chose to use GPT-3.5 Turbo as our baseline model because it offered the best cost–benefit ratio, making it a practical and widely adopted option for developers and researchers working under budget constraints. We acknowledge that other models may have been exposed to portions of the Javadocs used in our study; however, we consider the likelihood and impact of such exposure to be minimal and not significant enough to compromise the integrity of our evaluation.

3.7 Concluding Remarks

This chapter lays the foundation for the evaluations presented in this dissertation. It begins by detailing the construction of the dataset, including the selection of repositories and the curation of the ground truth. It then explains how large language models were employed to generate new documentation, specifying the models used and the prompt strategies adopted. Finally, it discusses potential limitations related to both the dataset and model selection, providing context for the analyses developed in the following chapters.

Chapter 4

Quantitative Evaluation

4.1 Introduction

The adoption of AI, particularly LLMs, for software engineering tasks has grown in recent years. These models present a promising opportunity to support practitioners by bridging the gap between efficient coding and effective documentation [24, 30]. Nevertheless, despite recent progress, research that analyzes large datasets to evaluate LLMs'ability to generate high-quality documentation remains limited, leaving important questions unanswered, particularly focusing explicitly on documentation rather than code summarization. While code summarization provides a brief description of what the code does, documentation offers more comprehensive, purpose-oriented information that helps developers understand intent, usage constraints, and expected behaviors, making it more valuable for long-term maintainability and effective collaboration. Therefore, we seek to address this gap by evaluating the effectiveness of LLMs in generating code documentation.

In this chapter, we present the first evaluation conducted with the comments generated by LLMs for the codes in our ground-truth dataset (Chapter 3). In particular, we describe a quantitative analysis designed to investigate the extent to which Javadocs generated by LLMs resemble those originally authored in real-world codebases. The analysis aims to provide an objective assessment of documentation quality by measuring textual similarity between generated and reference Javadocs. To achieve this, we employed established textual similarity metrics—BLEU, ROUGE-L, and METEOR—that are widely used in natural language processing for evaluating text generation tasks.

The remainder of this chapter is organized as follows. Section 4.2 outlines the design of the quantitative study. Section 4.3 presents the results obtained, while Section 4.4 discusses the key findings and their implications. Section 4.5 addresses potential threats to the validity of the evaluation. Finally, Section 4.6 provides concluding remarks and summarizes the main insights drawn from the quantitative analysis.

4.2 Study Design

In order to assess the quality of LLM-generated Javadocs, we prompted each of the three selected LLMs—OpenAI GPT-3.5 Turbo, OpenAI GPT-4o, and Deepseek-V3—to generate a Javadoc to document each of the 415 code snippets that compose our ground-truth. This resulted in 1,245 generated Javadocs that serve as the basis for the qualitative analysis. We measure their quality by comparing them with the Javadocs that originally documented the same code snippets in the repository, which were also retrieved as part of our ground truth.

To evaluate the similarity between the generated Javadocs and their ground truth counterparts, we employed a set of well-established textual similarity metrics that are widely recognized in the fields of Natural Language Processing (NLP) and machine-generated text evaluation: BLEU [42], ROUGE-L [37] and METEOR [6]. While they share a common goal—measuring the closeness between a generated text and a reference–they differ in how they are computed and in the specific aspects of similarity they emphasize, such as n-gram overlap, sentence structure, or semantic alignment. By combining them, we enable a more comprehensive and nuanced analysis of the quality of the generated documentation.

To compute each of the textual similarity metrics, we relied on established open-source libraries. For BLEU, we used the same implementation employed by Khan and Uddin [30]; for ROUGE-L, we used Google's ROUGE-SCORE library;[1] and for METEOR, we used the implementation provided by NLTK.[2] These libraries are maintained by active developer communities and collectively have tens of thousands of GitHub stars, supporting their reliability.

BLEU is a precision-oriented metric that measures how much of the generated content matches the reference. ROUGE-L is recall-oriented and based on the Longest Common Subsequence, measuring how much of the reference content is preserved in the generated text. METEOR combines precision and recall, incorporating stemming, synonymy, and paraphrasing to introduce semantic awareness and capture whether the meaning and intent of the generated text align with the reference.

Accordingly, our study includes all three metrics: BLEU, to maintain compatibility with prior work, and ROUGE-L and METEOR, to provide a more accurate and representative assessment of LLM performance in documentation generation. Notably, this same set of metrics has been adopted in other recent studies [14, 29, 47].

[1] https://pypi.org/project/rouge-score/
[2] https://pypi.org/project/nltk/

4.3 Results

In this section, we present the quantitative results obtained for each LLM, focusing on their performance across the BLEU, ROUGE-L, and METEOR metrics. We compare the models individually and discuss how their generated Javadocs align with the reference documentation according to these measures.

4.3.1 OpenAI GPT-3.5 Turbo

Figure 4.1 presents the BLEU score results for GPT-3.5 Turbo. In particular, we observe that the model achieved a BLEU score median of 28.97% (Q1: 20.72%, Q3: 39.59%). When analyzing only class-level comments, the score drops to 19.85% (Q1: 15.88%, Q3: 25.17%). In contrast, for method-level comments, the model reached a BLEU score of 35.00% (Q1: 27.60%, Q3: 43.60%). A Mann–Whitney U test indicated that the difference between class- and method-level results was statistically significant ($p < 0.001$).

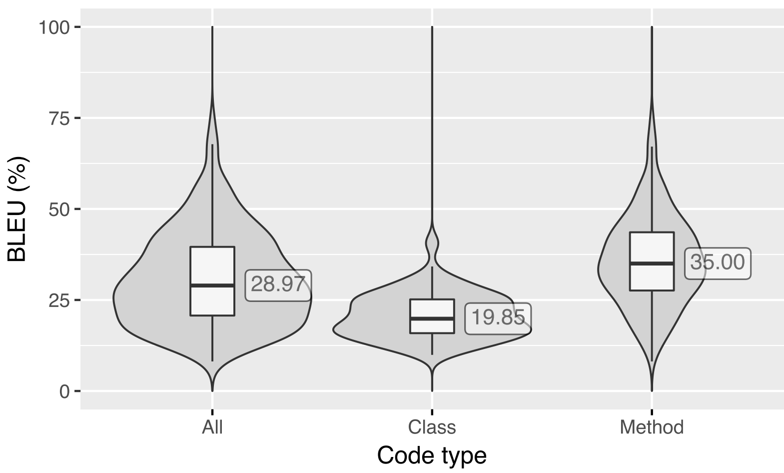


Figure 4.1: Smoothed BLEU Score GPT-3.5 Turbo

For ROUGE-L (Figure 4.2), GPT-3.5 Turbo attained an overall median score of 47.62% (Q1: 35.15%, Q3: 62.50%). Class-level comments recorded a median of 42.21% (Q1: 33.33%, Q3: 61.15%), while method-level comments achieved 50.00% (Q1: 36.51%, Q3: 63.64%). According to the Mann–Whitney U test, the observed difference between class- and method-level results reached statistical significance ($p = 0.022$).

The METEOR scores (Figure 4.3) follow a similar trend. The overall median score

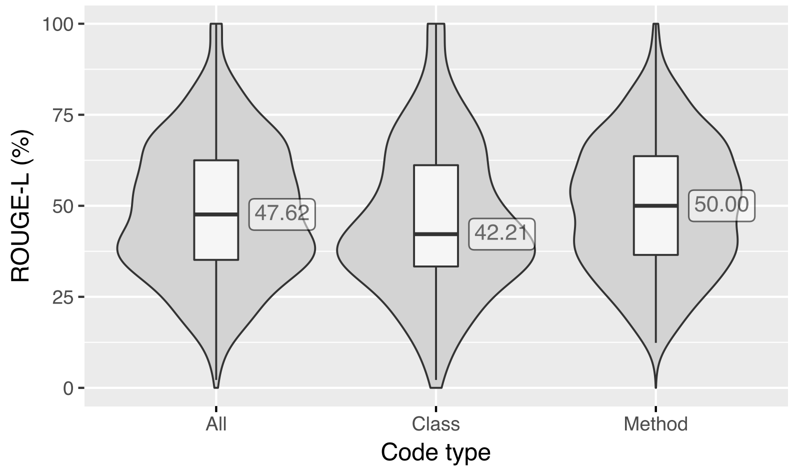


Figure 4.2: ROUGE-L GPT-3.5 Turbo

is 43.21% (Q1: 31.70%, Q3: 56.24%). For class-level documentation, the model reached 39.24% (Q1: 29.63%, Q3: 53.78%), while method-level comments scored 43.95% (Q1: 34.13%, Q3: 57.40%). Statistical analysis using the Mann–Whitney U test confirmed a significant difference between class- and method-level results ($p = 0.026$).

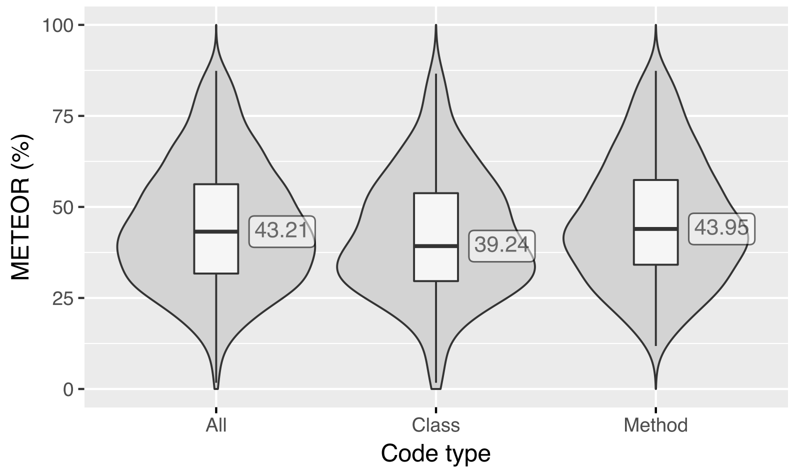


Figure 4.3: METEOR GPT-3.5 Turbo

In summary, GPT-3.5 Turbo demonstrated moderate performance across all textual similarity metrics. Its highest scores were observed in the method-level, indicating the model's relative proficiency in generating accurate and coherent documentation for localized code components. However, the substantial drop in scores for class-level comments, especially in BLEU (19.85%), reveals limitations in handling abstract, high-level documentation. The consistent gap between method-level and class-level performance highlights the model's reliance on concrete, narrow context to produce meaningful textual output.

4.3.2 OpenAI GPT-4o

In addition to GPT-3.5 Turbo, we also evaluated GPT-4o. Figure 4.4 presents its BLEU score results. Specifically, we observe that GPT-4o obtained a Smoothed BLEU score median of 26.28% (Q1: 13.54%, Q3: 40.51%) overall. Although slightly below GPT-3.5 Turbo overall, it showed stronger method-level performance with a score of 37.28% (Q1: 26.82%, Q3: 45.16%). However, it's class-level median score was worse, only 11.42% (Q1: 8.91%, Q3: 14.80%). A Mann–Whitney U test indicated that the difference between class- and method-level results was statistically significant ($p < 0.001$).

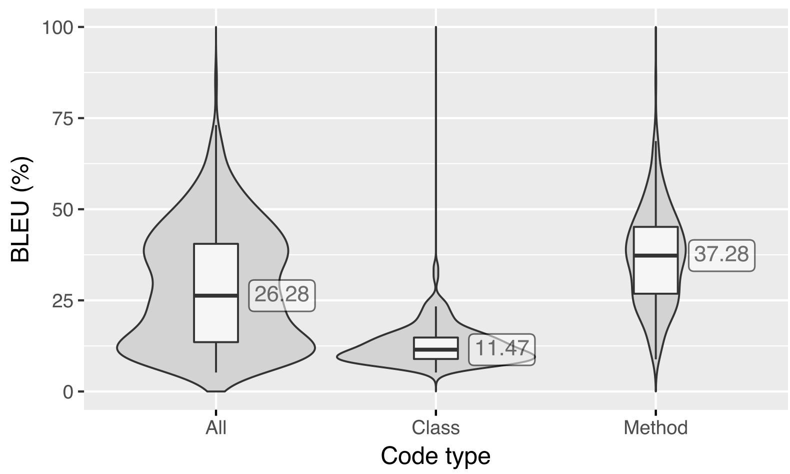


Figure 4.4: Smoothed BLEU Score GPT-4o

For ROUGE-L (Figure 4.5), GPT-4o demonstrated clear improvement, achieving an overall median of 52.17% (Q1: 38.10%, Q3: 65.52%). Class-level comments recorded 50.00% (Q1: 38.10%, Q3: 71.43%), while method-level comments slightly outperformed with 52.38% (Q1: 38.24%, Q3: 63.64%). Although the Mann–Whitney U test did not indicate a statistically significant difference between class- and method-level Javadocs ($p = 0.233$), the higher medians suggest that GPT-4o generated documentation with greater similarity to the ground-truth references.

METEOR scores for GPT-4o (Figure 4.6) reinforce this improvement. The overall score reached a median of 45.30% (Q1: 33.63%, Q3: 58.06%). The method-level score increased to 49.50% (Q1: 36.24%, Q3: 62.31%), while class-level comments received a lower score of 38.10% (Q1: 30.33%, Q3: 51.30%). This divergence illustrates the model's stronger capacity for semantic coherence at the method level, while its ability to abstract higher-level concepts remains limited. According to the Mann–Whitney U test, the observed difference between class- and method-level results reached statistical significance ($p < 0.001$).

In conclusion, we observe that GPT-4o showed mixed performance, with notable

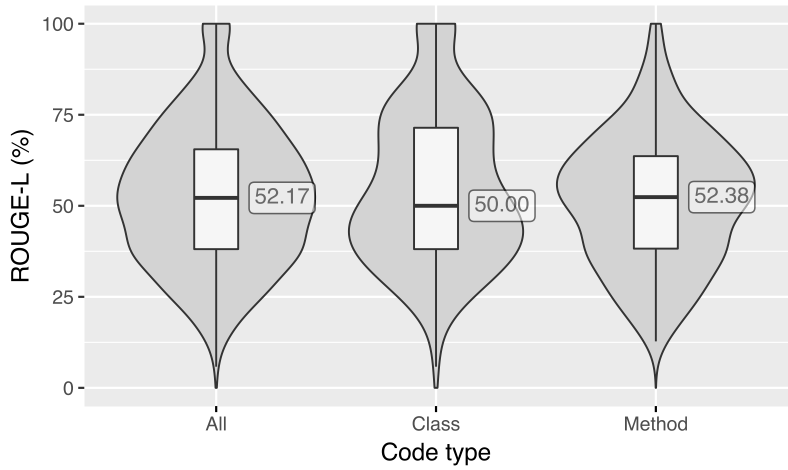


Figure 4.5: ROUGE-L GPT-4o

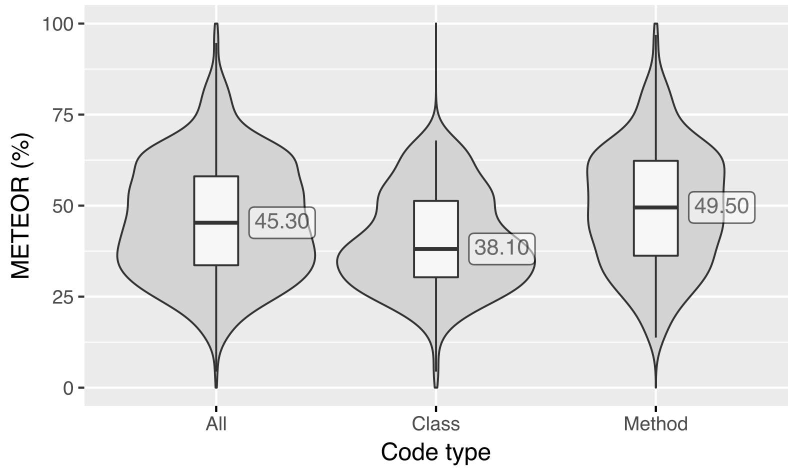


Figure 4.6: METEOR GPT-4o

gains over GPT-3.5 Turbo in ROUGE-L and METEOR, reflecting improvements in syntactic structure and semantic alignment. It achieved a higher method-level BLEU (37.28%), yet showed a lower class-level BLEU score (11.42%), suggesting a persistent challenge in summarizing abstract code constructs. These findings indicate that while GPT-4o generates more fluent and structurally coherent documentation than its predecessor, its ability to generalize at the class level remains limited.

4.3.3 Deepseek-V3

In addition to GPT-3.5 Turbo and GPT-4o, we also evaluated DeepSeek-V3. Figure 4.7 presents its BLEU score results. Notably, the model achieved a Smoothed BLEU score median of 25.00% (Q1: 15.21%, Q3: 37.35%), the lowest among all models. Its class-level performance stood at 13.83% (Q1: 10.59%, Q3: 17.07%), whereas method-level comments showed higher quality with a BLEU score of 33.67% (Q1: 23.58%, Q3: 44.45%). A Mann–Whitney U test indicated that the difference between class- and method-level results was statistically significant ($p < 0.001$).

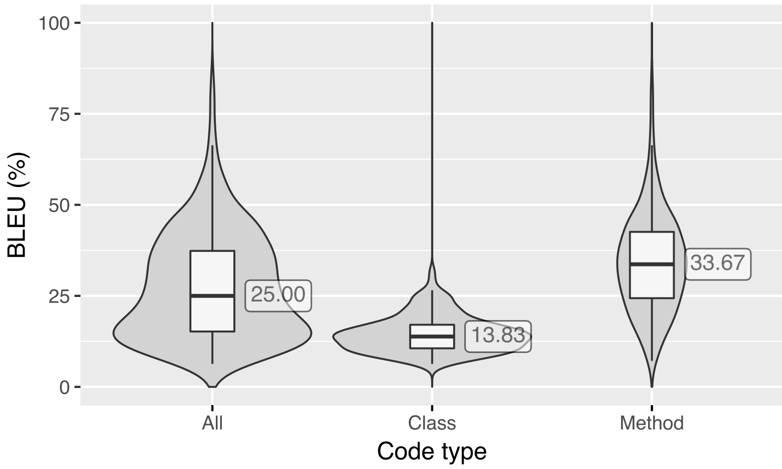


Figure 4.7: Smoothed BLEU Score Deepseek-V3

In terms of ROUGE-L (Figure 4.8), Deepseek-V3 outperformed both OpenAI models, with an overall median score of 53.33% (Q1: 40.00%, Q3: 68.21%). Class-level and method-level scores were 50.00% (Q1: 40.00%, Q3: 69.81%) and 54.55% (Q1: 39.62%, Q3: 67.86%), respectively. These results demonstrate superior ability to replicate the sequence and phrasing of the ground-truth comments. According to the Mann–Whitney U test, the observed difference between class- and method-level results reached statistical significance ($p < 0.001$).

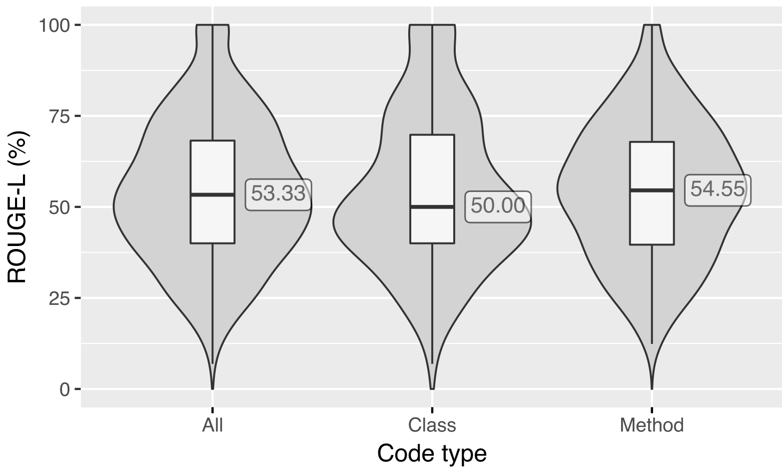


Figure 4.8: ROUGE-L Deepseek-V3

Finally, Deepseek-V3 also led in METEOR performance (Figure 4.9). The model achieved an overall median score of 46.48% (Q1: 35.55%, Q3: 59.99%), with class-level comments at 40.48% (Q1: 33.35%, Q3: 49.95%) and method-level at 50.19% (Q1: 37.50%, Q3: 62.00%). These results affirm Deepseek-V3's robustness in capturing both lexical similarity and semantic fidelity across varying levels of code abstraction. Statistical analysis using the Mann–Whitney U test confirmed a significant difference between class- and method-level results ($p < 0.001$).

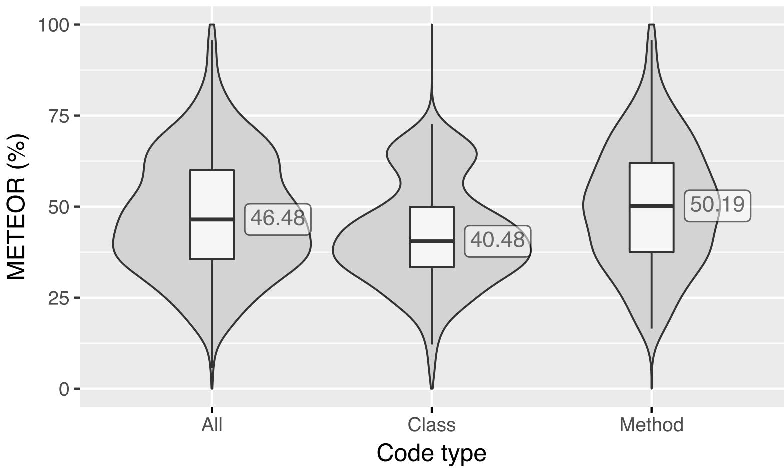


Figure 4.9: METEOR Deepseek-V3

In summary, Deepseek-V3 emerged as the most consistent and well-rounded performer across the board. Despite scoring slightly lower on overall BLEU compared to GPT models, it achieved the highest ROUGE-L and METEOR scores overall. It was also the top performer in both class-level and method-level METEOR, indicating superior capacity to capture both lexical patterns and semantic content in documentation.

4.4 Discussion

The quantitative evaluation revealed insights in the performance of the evaluated LLMs, alongside some unexpected discrepancies between the employed metrics. Across all models, method-level documentation consistently outperformed class-level documentation, reaffirming that LLMs perform more reliably when the context is narrowly scoped and semantically explicit.

A particularly noteworthy finding is the divergence in model rankings depending on the metric used. While BLEU scores suggest that GPT-3.5 Turbo, the oldest model in the comparison, slightly outperformed its successors, this trend is not supported by ROUGE-L and METEOR, which favor newer models. BLEU, by design, emphasizes strict n-gram overlap and exact phrase matching, which can disproportionately penalize paraphrased or semantically equivalent outputs. For example, GPT-3.5 Turbo achieved a higher overall BLEU median (28.97%) than GPT-4o (26.28%) and Deepseek-V3 (25.00%), which initially suggests superior performance. However, BLEU's rigidity likely understates the capabilities of newer models that produce more varied yet still valid phrasing.

In contrast, ROUGE-L and METEOR—metrics that are better suited to capturing structural resemblance and the preservation of meaning—presents a different behavior. Deepseek-V3 emerged as the overall top performer, achieving the highest median scores in both ROUGE-L (53.33%) and METEOR (46.48%). Notably, it also led in both class-level and method-level for METEOR, signaling its robust ability to capture the meaning and intent behind the code. GPT-4o also showed advancement over GPT-3.5 Turbo in these metrics, especially in ROUGE-L (52.17%) and method-level METEOR (49.50%).

These results suggest that BLEU may not be a reliable standalone indicator of documentation quality in this context, as it fails to account for the kind of semantic variation and syntactic flexibility characteristic of high-level documentation like Javadocs. Models like GPT-4o and Deepseek-V3, which leverage more advanced architectures and training paradigms, produce outputs that are more fluent, diverse, and semantically aligned.

In summary, while BLEU presents an anomalous picture by favoring GPT-3.5 Turbo, ROUGE-L and METEOR offer a more accurate reflection of real-world documentation quality. Deepseek-V3 stands out as the most balanced and effective model, combining strong lexical coverage with high semantic coherence. These findings reinforce the trend that newer models tend to generate higher-quality documentation, particularly when evaluated with metrics that capture more than surface-level similarity.

4.5 Threats to Validity

In addition to the threats to validity concerning the data collection and dataset construction procedures described in Section 3.6, this study also presents threats specific to the quantitative evaluation, as follows. The process of computing textual similarity metric scores may introduce a potential threat to internal validity, as it relies on external implementations of the evaluation algorithms. To address this, we employed widely accepted and rigorously tested implementations based on established techniques from the literature. Additionally, for BLEU, we applied a smoothing function to handle cases with short or sparse reference texts, as recommended in prior studies.

4.6 Concluding Remarks

In this initial study, we conducted a structured assessment of the lexical, structural, and semantic alignment between Javadocs generated by LLMs and the original documentation available in the codebase. While all models performed better on method-level documentation, substantial differences emerged depending on the evaluation metric used. BLEU highlighted the sensitivity to exact lexical overlap but also introduced misleading impressions about model effectiveness. In contrast, ROUGE-L and METEOR offered a more comprehensive view, emphasizing syntactic and semantic fidelity.

Among the models evaluated, Deepseek-V3 demonstrated the most consistent and well-rounded performance, making it the most effective model overall in this first assessment. These findings underscore the importance of using multiple, complementary metrics when evaluating documentation generation and highlight the evolving capability of modern LLMs to produce fluent and contextually accurate source code comments.

Chapter 5

Qualitative Evaluation

5.1 Introduction

Human evaluation is widely considered the gold standard for analysis due to its reliability and ability to account for nuanced judgments [47]. While automated metrics are useful in certain contexts for measuring textual similarity, their effectiveness in evaluating machine-generated texts has been increasingly questioned [18, 40, 47]. These metrics often struggle to capture semantic correctness, and contextual appropriateness. As such, although they offer valuable quantitative insights, automated metrics cannot fully replace human judgment [14, 52]. Therefore, we also employed human evaluation to assess the quality of LLM-generated Javadocs.

This chapter presents the qualitative evaluation conducted to assess the perceived quality of Javadocs generated by LLMs from a human-centered perspective. To this end, we designed and carried out a human evaluation in which expert annotators rated the similarity between the generated comments and the original documentation authored within the codebase. This analysis aims to complement the quantitative findings by capturing subjective judgments related to clarity, completeness, and semantic alignment—factors that are often difficult to quantify but critical to the practical utility of generated documentation. Additionally, the chapter discusses how the qualitative results relate to and contrast with the findings from the quantitative evaluation (Chapter 4), offering a more holistic view of LLM performance in this task.

The remainder of this chapter is organized as follows. Section 5.2 describes the design of the qualitative study. Section 5.3 reports the results obtained, presenting illustrative examples that clarify the rationale behind the scoring process. Section 5.4 compare the results from this chapter with the ones from the quantitative evaluation (Chapter 4). Section 5.5 discusses the implications of the findings. Section 5.6 addresses potential threats to the validity of the evaluation. Finally, Section 5.7 presents concluding remarks and summarizes the key insights derived from the human assessment.

5.2 Study Design

We compared each of the 415 Javadoc comments in our ground truth dataset with its corresponding version generated by OpenAI's GPT-3.5 Turbo. Due to the resource-intensive nature of human evaluation [52], only this model was evaluated. It was chosen because, at the time of analysis, it provided a cost-effective solution and was widely adopted by practitioners. Furthermore, since GPT-4o and Deepseek-V3 are more recent models, it is reasonable to expect that they possess enhanced capabilities. Therefore, if GPT-3.5 Turbo demonstrates strong performance, it provides a meaningful baseline, suggesting that the newer models are likely to perform equally well or better.

For the analysis, human evaluators assigned a similarity score to each generated Javadoc, reflecting its semantic and structural closeness to the original version. This evaluation method is adapted from Tran et al. [60] and is referred to in this dissertation as the Human Assessment Score (HA-Score). The HA-Score ranges from 1 to 4, with detailed evaluation criteria defined as follows:

- A score of 4 indicates that the GPT-generated comment is definitely equivalent and contains the same information as the original one;
- A score of 3 means that the generated comment is correct, even though it requires minor changes to be fully equivalent with the original one;
- A score of 2 denotes that the generated comment is usable, but major changes are required to make it equivalent to the original.
- A score of 1 implies that the GPT-generated comment is completely incorrect and cannot be used to replace the original one.

The scoring process was carried out by two evaluators, both of whom are experienced software engineers with five and seven years of experience. Their backgrounds include extensive authorship of technical documentation across a variety of software projects, including Javadoc authoring. The evaluation followed a two-phase procedure designed to enhance the reliability and objectivity of the results.

In the initial phase, each evaluator independently assessed 10% of the sample, assigning HA-Scores to the generated Javadocs. Following this, they discussed any discrepancies in their assessments and reached a consensus score for each instance. This preliminary step served to minimize individual biases and calibrate their evaluation criteria to ensure consistency.

In the subsequent phase, the evaluators applied the same procedure to the remaining 90% of the dataset. Each evaluator first scored the Javadocs independently, after which they reviewed together and resolved any differences to arrive at a final, agreed-upon score. Thus, every Javadoc was assessed by both evaluators.

A Cohen's Kappa value of 0.54 was achieved between both evaluators' classifications, indicating a moderate level of agreement [34]. Additionally, they ranked 73.5% of the comments with the same category, suggesting a reasonable consistency between their judgments. Figure 5.1 presents the HA-Score Agreement Matrix, which visualizes the frequency of aligned and divergent scores between the two evaluators. Notably, the highest concentration (216 instances) occurs at the (4,4) cell, where both evaluators agreed on the maximum quality score. This suggests a strong consensus in identifying well-written, highly informative comments. Lesser agreement is observed in lower score regions, with discrepancies appearing mostly between adjacent categories, which is expected in subjective evaluations and reflects nuanced differences in judgment rather than major disagreements.

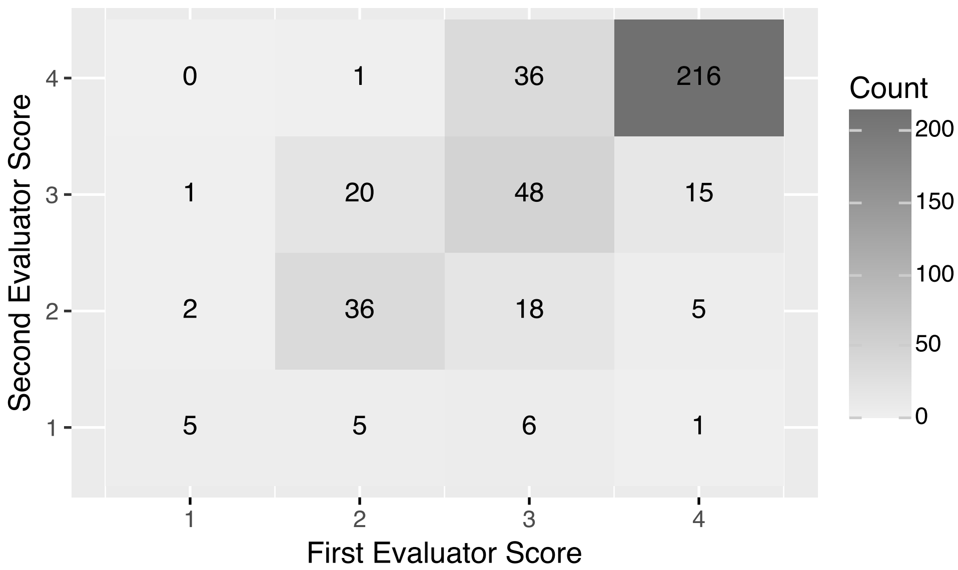


Figure 5.1: HA-Score Agreement Matrix

Interestingly, during the evaluation process, it was observed that certain generated Javadocs not only matched the original comments in terms of informational content—thus warranting a score of 4—but also surpassed them by providing additional relevant context. In such cases, the generated comments were considered to be of superior quality compared to the reference. To acknowledge these instances, we introduced a special distinction referred to as score 4+. Comments awarded a 4+ were separately flagged to highlight their exemplary quality. This distinction underscores the potential of LLMs not only to replicate existing documentation but, in some cases, to enhance and refine it beyond the original.

5.3 Results

In this section, we present the results of the qualitative evaluation, assessing how well GPT-3.5 Turbo performed in generating Javadocs through a human evaluation using the HA-Score. To enhance clarity, we also include one representative example for each score, showing the generated Javadoc alongside its corresponding original version and discussing the reasoning behind the assigned classification. Figure 5.2 presents the distribution of HA-Scores.

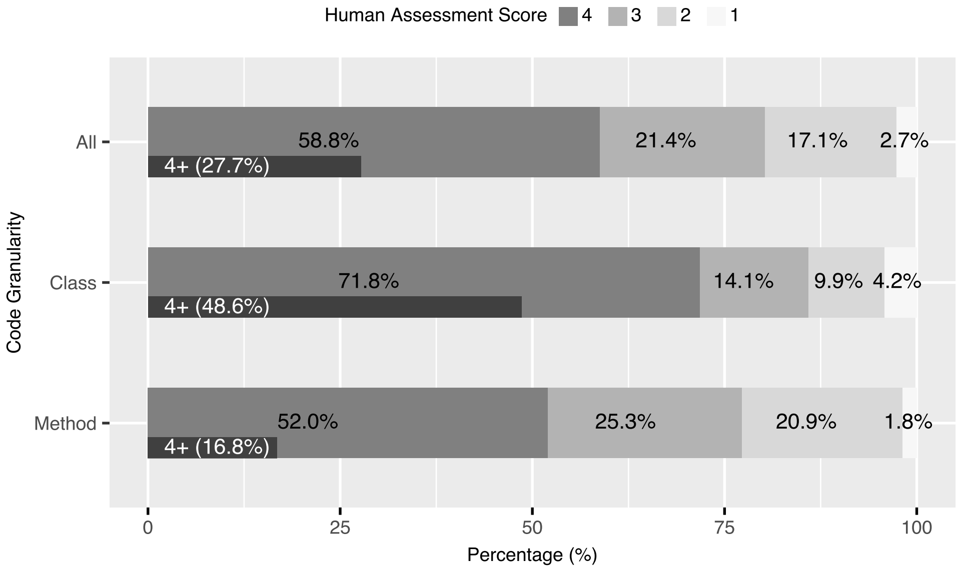


Figure 5.2: Human Assessment Score Distribution

Overall, the majority of generated comments (58.8%) received the highest score (4), indicating strong semantic and contextual alignment with the original documentation. Figure 5.3 presents and example where the generated and the original Javadoscs are nearly identical, differing only in subtle phrasing variations, thus justifying a score 4 for the generated comment. This score typically applies to Javadocs that are almost exact replicas of the originals, with only slight linguistic differences.

Approximately one-fifth of the generated Javadocs (21.4%) received a score of 3, indicating they require only minor revisions to achieve full equivalence with the original version. As an example, both Javadocs in Figure 5.4 describe the method's functionality: reversing an array's elements. However, the generated comment is concise, whereas the original elaborates on an equivalent method and its efficiency. We considered that the omissions are minor and do not hinder understanding of the method's purpose. Therefore, it receives a score 3.

A smaller portion of the generated Javadocs (17.1%) were judged usable but requiring substantial revisions (score 2). Figure 5.5 illustrates a case where both Javadocs

```
Original
/**
 * Register a {@link DeferredResultProcessingInterceptor} under the given key.
 * @param key the key
 * @param interceptor the interceptor to register
 */

GPT-Generated
/**
 * Register a {@link DeferredResultProcessingInterceptor} with a given key.
 *
 * @param key the key to associate with the interceptor
 * @param interceptor the interceptor to be registered
 */
```

Figure 5.3: Human Assessment Score—Example Score 4

```
Original
/**
 * Reverses the elements of {@code array}. This is equivalent to {@code
 * Collections.reverse(Longs.asList(array))}, but is likely to be more
 efficient.
 *
 * @since 23.1
 */

GPT-Generated
/**
 * Reverse the elements in the given array.
 * @param array the array to be reversed
 */
```

Figure 5.4: Human Assessment Score—Example Score 3

discuss a waiting time before shutdown. The original comment, however, provides additional details (e.g., behavior when tasks are submitted, and default wait periods). The generated version, lacking these important details, requires major additions to reach equivalence, thus warranting a score 2.

Only a small proportion (2.7%) were rated as completely unusable (score 1). As an example, while the original Javadoc in Figure 5.6 describes a method that only throws an exception, the generated one incorrectly claims that all map values will be replaced. Although the LLM was able to note that the exception is always thrown, this discrepancy justifies a score 1, indicating a fundamental misunderstanding of the method. This is a common trait among other Javadocs ranked similarly.

Among the comments rated with a score of 4, a notable subset was further distinguished with the 4+ designation—indicating outputs that were judged to be better than the original Javadocs. In total, 27.7% of all generated comments received this 4+

```
Original
/**
 * Configure the amount of time we'll wait before shutting down resources.
 * If a task is submitted during the {@code shutdownQuietPeriod}, it is
 guaranteed
 * to be accepted and the {@code shutdownQuietPeriod} will start over.
 * <p>By default, this is set to
 * {@link LoopResources#DEFAULT_SHUTDOWN_QUIET_PERIOD} which is 2 seconds but
 * can also be overridden with the system property
 * {@link reactor.netty5.ReactorNetty#SHUTDOWN_QUIET_PERIOD
 * ReactorNetty.SHUTDOWN_QUIET_PERIOD}.
 * @since 5.2.4
 * @see #setShutdownTimeout(Duration)
 */

GPT-Generated
/**
 * Set the duration of the quiet period before shutdown.
 * @param shutdownQuietPeriod the duration of the quiet period
 */
```

Figure 5.5: Human Assessment Score—Example Score 2

```
Original
/**
 * Guaranteed to throw an exception and leave the map unmodified.
 *
 * @throws UnsupportedOperationException always
 * @deprecated Unsupported operation.
 */

GPT-Generated
/**
 * Replaces all key-value pairs in the map by applying the specified function,
 always throwing UnsupportedOperationException.
 * @param function the function to apply (not used as the method always throws
 an exception)
 * @deprecated This method is deprecated and should not be called
 * @DoNotCall(""Always throws UnsupportedOperationException"")
 * @throws UnsupportedOperationException always thrown as this operation is not
  supported
 */
```

Figure 5.6: Human Assessment Score—Example Score 1

rating. This distinction highlights the model's ability to produce documentation that not only meets expected standards but, in many cases, surpasses the quality of the existing comments. For example, both Javadocs in Figure 5.7 describe an object that captures other objects to be closed when a ClosingFuture pipeline is completed. However, the generated comment is more comprehensive, offering extra details about the class, which enhances understanding from the documentation alone. This additional information merits the distinguished Score of 4+. Javadocs achieving this score consistently provide more valuable insights than the originals, enriching the reader's comprehension of the code.

```
Original
/**
 * An object that can capture objects to be closed later, when a {@link
 ClosingFuture} pipeline is
 * done.
 */

GPT-Generated
/**
 * A class that captures objects to be closed when a ClosingFuture pipeline is
 done.
 *
 * <p>Objects can be any AutoCloseable for the -jre flavor of Guava or
 Closeable for the -android flavor.
 * It is important to ensure compatibility with older SDK versions by passing
 objects that implement the necessary
 * interfaces at the oldest supported version.
 *
 * <p>The method is binary-compatible between flavors due to the parameter type
  erasure being Object.
 *
 * @see CloseableList
 */
```

Figure 5.7: Human Assessment Score—Example Score 4+

When analyzing the results by code granularity (Figure 5.2), it becomes clear that GPT-3.5 Turbo demonstrated stronger performance on class-level elements compared to method-level ones. For class-level Javadocs, 71.8% of the generated comments were rated as score 4, while 52.0% of method-level comments received the same score. These findings suggest that the model is particularly effective at generating high-quality documentation for higher-level constructs such as classes, which typically encapsulate broader responsibilities and more abstract semantic roles. Most notably, 48.6% of all class-level comments received the 4+ distinction—indicating that nearly half were judged to be of even higher quality than the corresponding original documentation. Nevertheless, despite the observed discrepancy, the model achieved strong results across both granularities, demonstrating its overall robustness and versatility in generating reliable Javadocs.

5.4 Quantitative vs Qualitative Evaluation

When comparing the qualitative results with the ones from the quantitative evaluation (Chapter 4), it becomes evident that the effectiveness of automatic metrics varies. While BLEU often fails to reflect the perceived quality of the generated comments, ROUGE-L and METEOR demonstrate stronger alignment with human-assigned scores, offering a more reliable approximation of documentation quality in this context.

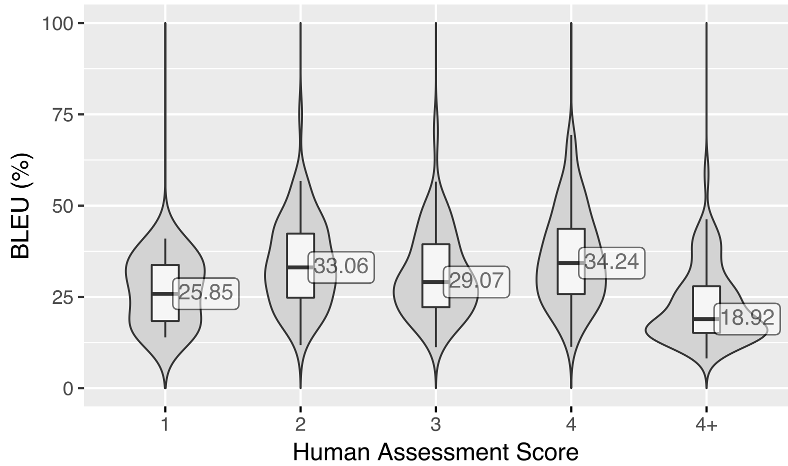


Figure 5.8: BLEU Score Distribution by HA-Score

Figure 5.8 illustrates the distribution of BLEU scores across different HA-Scores. At first glance, there appears to be no consistent upward trend in BLEU scores as HA scores increase. This observation is reinforced by a negative Spearman correlation coefficient of -0.28, suggesting an inverse relationship between the two measures. Notably, the BLEU score drops sharply for the highest HA-Score category (4+), which corresponds to Javadoc comments with the highest perceived quality. This counterintuitive result indicates that BLEU, as a lexical similarity metric, may not reliably capture the semantic richness or contextual appropriateness that human evaluators are able to recognize and reward.

Figure 5.9 presents a clear example where the BLEU metric does not align with human judgment. The Javadoc generated by GPT-3.5 Turbo for the given class received the highest rating in the human evaluation (HA-Score of 4+), indicating that it was considered not only accurate but also more detailed and informative than the original. Despite this, the BLEU score was notably low—just 9%. This example highlights a key limitation of metrics like BLEU: they focus heavily on exact word matching and tend to undervalue well-written outputs that use different wording to express the same or even more complete information.

In contrast to BLEU, ROUGE-L shows a stronger alignment with HA-Scores, as it

```
Original
/**
 * Processor used to register the hints.
 */

GPT-Generated
/**
 * A Processor class that handles the processing of bindable objects and their
 properties. It includes methods for handling constructors, properties, and
 nested types, as well as verifying parameter names and checking for nested
 configuration properties.
 */
```

Figure 5.9: Example of BLEU and HA-Score misalignment in class-level generated Javadoc

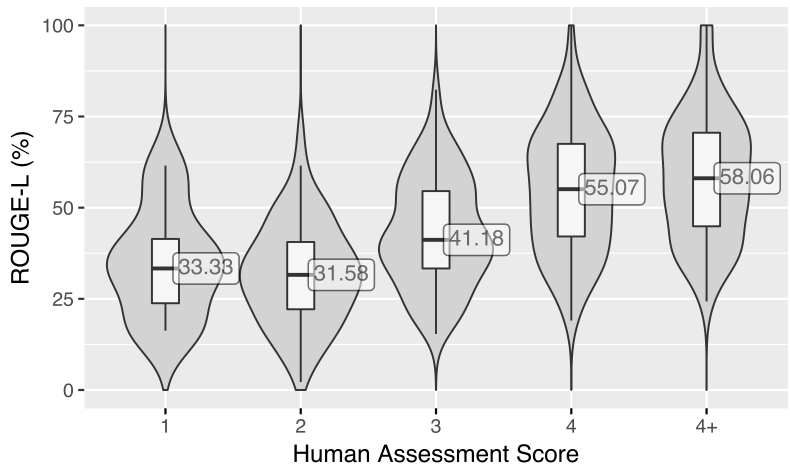


Figure 5.10: ROUGE-L Distribution by HA-Score

captures broader structural and content similarities. As shown in Figure 5.10, ROUGE-L scores tend to increase with higher HA-Scores, indicating a more consistent relationship between this metric and the human assessment. This is supported by a moderate positive Spearman correlation of 0.49, suggesting that ROUGE-L is more effective than BLEU at reflecting documentation quality in terms of content coverage and structural correspondence. This correlation indicates that ROUGE-L may provide a more robust basis for evaluating the effectiveness of generated Javadocs, especially when lexical variation is present.

Figure 5.11 illustrates an instance with strong agreement between the HA-Score and the ROUGE-L metric. The class-level GPT-generated Javadoc received the highest rating in both metrics—100% ROUGE-L and an HA-Score of 4+—indicating perfect agreement. The comment preserves the original intent while expanding it with additional, relevant information. In this case, the ROUGE-L score mirrors the human assessment, reflecting its ability to capture enhancements in content and structure that improve documentation quality, without penalizing lexical variation or added detail.

```
Original
/**
 * A collection of propagation factories.
 */

GPT-Generated
/**
 * Represents a collection of factories for creating propagations. This class
 provides methods to check if any factory requires a 128-bit trace ID,
 * if all factories support join, and to retrieve a list of propagations from
 the factories.
 */
```

Figure 5.11: Example of ROUGE-L and HA-Score high alignment in class-level generated Javadoc

Despite this alignment, the BLEU score for the same example was just 13%. This, once again, highlights BLEU's limitations in cases where high-quality outputs include rewording or additional detail. Since BLEU relies on surface-level lexical overlap, it may undervalue accurate and informative comments that differ in phrasing. This reinforces the need for evaluation metrics that better reflect semantic quality and real-world usefulness in documentation generation.

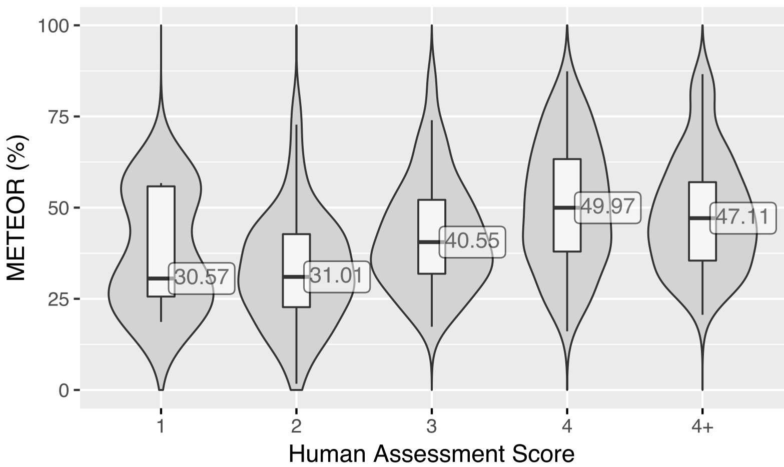


Figure 5.12: METEOR Distribution by HA-Score

Similar to ROUGE-L, METEOR scores generally increase as HA-Scores rise, as illustrated in Figure 5.12. While the alignment is not as pronounced, it still shows a positive correlation, with a Spearman coefficient of 0.30—indicating a moderate relationship. Although less sensitive than ROUGE-L in this context, METEOR captures certain aspects of semantic and structural similarity, likely due to its handling of synonyms and word stems. These characteristics make it a more flexible alternative to BLEU for evaluating generated documentation, even if its weaker alignment with HA-Scores.

5.5 Discussion

The qualitative evaluation demonstrates that the LLM consistently produces high-quality Javadoc comments, with the majority rated as either equivalent to or better than the original documentation. Notably, 58.8% of comments received the highest possible score, and an additional 27.7% were considered superior to the originals (score 4+). These results highlight the model's capacity not only to match but often exceed the quality of comments found in real-world codebases.

At the class level, we observe that the performance was especially strong: 71.8% of generated comments were rated as top-quality and nearly half were judged to exceed the quality of the reference documentation. This suggests that the LLM is particularly effective at summarizing broader code structures, possibly due to the richer semantic context provided by class declarations. Method-level results, while slightly lower, remained strong and reflect the model's versatility across different code granularities.

Interestingly, this contrasts with the quantitative evaluation, where method-level documentation received higher scores across most metrics. The divergence indicates that certain aspects present in high-quality class-level comments—such as abstraction and contextual breadth—may not be adequately captured by lexical similarity metrics, highlighting a limitation of relying solely on automated evaluation.

Moreover, this comparison between the qualitative and the quantitative results, highlighted that the effectiveness of automatic metrics varies. While BLEU often fails to reflect the perceived quality of the generated comments, ROUGE-L and METEOR demonstrate stronger alignment with human-assigned scores.

5.6 Threats to Validity

In addition to the threats to validity concerning the data collection and dataset construction procedures described in Section 3.6, this study also presents threats specific to the qualitative evaluation, described below.

The evaluation of Javadocs using the HA-Score may pose a threat to conclusion validity, as it relies on human judgment, which inherently introduces subjectivity and the potential for unintentional bias. To mitigate this threat, the evaluation was conducted by two experienced software developers that holds together more than a decade of professional experience. The evaluators followed a structured assessment process. Additionally, to further reduce individual bias, the evaluation included a crucial step in which the evaluators discussed their assessments to reach a consensus on the HA-Score for each Javadoc. Moreover, the assessments were conducted blindly with respect to the quantitative evaluation results, thereby ensuring an impartial and independent evaluation process.

5.7 Concluding Remarks

The qualitative evaluation revealed that the LLM consistently produces high-quality Javadocs, with many outputs rated as equivalent to or better than the existing documentation. This demonstrates the model's capability not only to replicate but in many cases enhance software documentation, reinforcing its potential as a practical tool for real-world development workflows.

When comparing these results with the quantitative evaluation, a clear distinction emerges among the metrics. BLEU, with its narrow focus on surface-level lexical overlap, frequently failed to reflect the quality of semantically rich and well-structured outputs. In contrast, ROUGE-L and METEOR demonstrated stronger alignment with human-assigned scores, particularly in cases involving paraphrasing or elaboration. ROUGE-L, in particular, showed slightly better correlation, highlighting its effectiveness in capturing structural and phrasal similarity. Yet, human assessment remains the most reliable standard for evaluating documentation quality.

Chapter 6

Source Code Assessment

6.1 Introduction

In this dissertation, we have already examined how effectively LLMs generate documentation from source code, through both quantitative (Chapter 4) and qualitative (Chapter 5) analyses. While these evaluations reveal how well the models perform, they do not explain why certain code snippets lead to better or worse documentation. Understanding whether characteristics of the source code itself influence generation quality is important for understanding the models' capabilities and limitations. Therefore, we present a complementary perspective, focusing on the relationship between source code properties and documentation quality.

In this chapter, we investigate how structural properties of source code influence the ability of LLMs in generating high-quality Javadocs. While previous chapters focused on evaluating the content of generated documentation, this analysis shifts attention to the code itself. By correlating source code metrics with the human-assigned HA-scores from the qualitative evaluation (Chapter 5), we aim to determine which code characteristics are associated with better or worse perceived documentation quality. To do so, we focused on complexity, size, coupling, cohesion, and lexical diversity.

The remainder of this chapter is organized as follows. Section 6.2 describes the study design. Section 6.3 presents the correlation results between source code metrics and documentation quality. Section 6.4 discusses key patterns and their implications. Section 6.5 outlines potential threats to validity. Finally, Section 6.6 summarizes the main insights and conclusions from this analysis.

6.2 Study Design

To investigate how the structural properties of source code influence the ability of LLMs to generate high-quality Javadocs, we conducted a static analysis on each code snippet in the ground truth dataset. The objective was to correlate specific code characteristics with the quality of the corresponding generated documentation, thereby gaining insights into the relationship between code complexity and documentation performance.

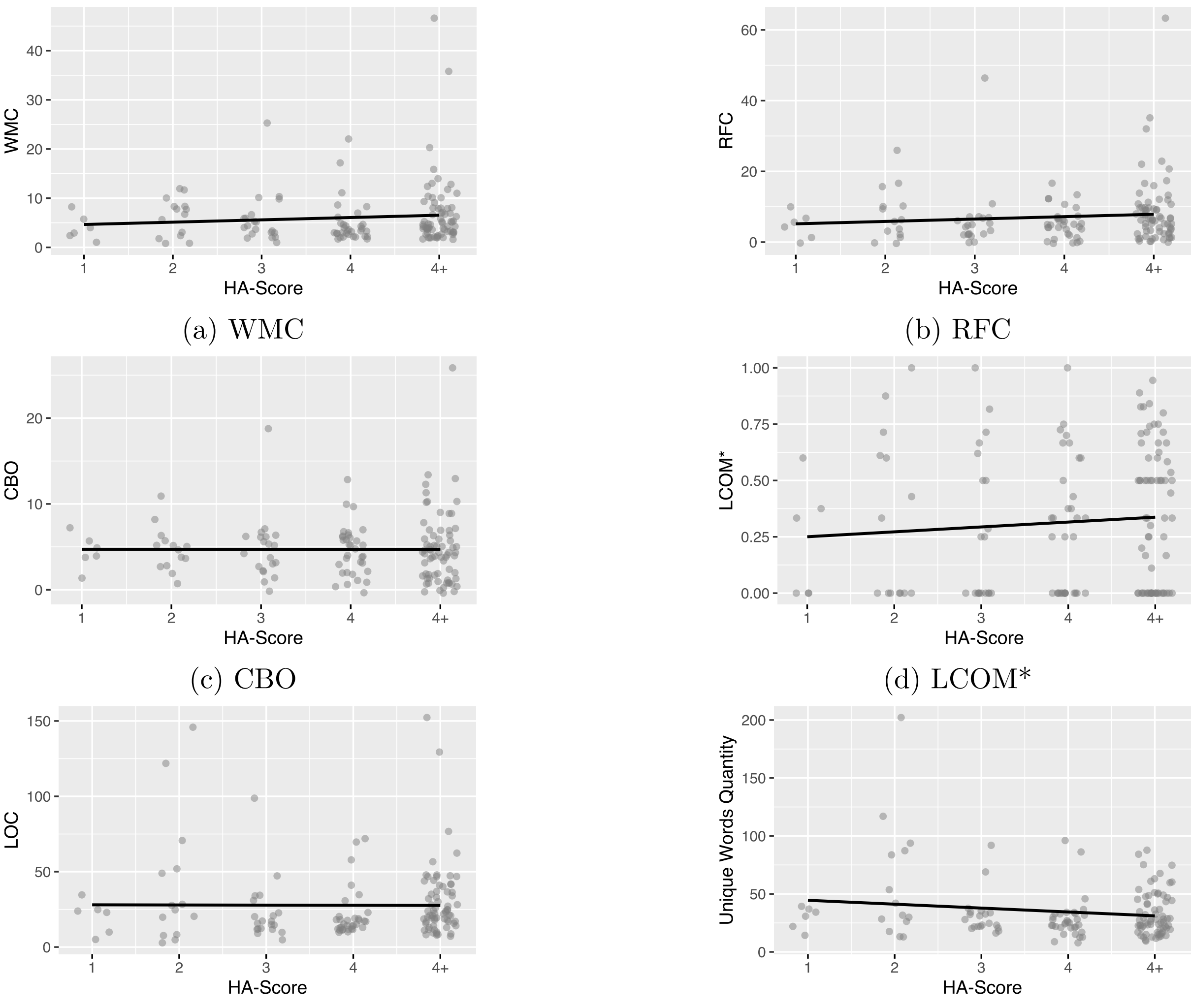
For this analysis, we employed the CK Tool [5], a widely used static analysis tool that computes a variety of source code metrics, including the well-established Chidamber and Kemerer (CK) suite [9]. From this set, we selected all CK metrics except Depth of Inheritance Tree (DIT) and Number of Children (NOC), as these are more reflective of class hierarchy, which was not directly exposed to the LLMs during inference. In addition to the CK metrics, we included two supplementary measures computed by the tool: Lines of Code (LOC) and Unique Words Quantity. These were chosen for their direct relevance to how LLMs process input, particularly in terms of token length and lexical diversity.

Thus, the final set of metrics analyzed is: WMC (Weighted Methods per Class), CBO (Coupling Between Objects), RFC (Response for a Class), LCOM (Lack of Cohesion of Methods), LOC (Lines of Code), and Unique Words Quantity.

For our analyses, we correlated each source code metric with the HA-score assigned during the Human Assessment phase (as described in Section 5.3) to the LLM-generated Javadoc corresponding to each code snippet. To perform this correlation, we employed Spearman's rank correlation coefficient [53], as it is well-suited for measuring the strength and direction of monotonic relationships between variables that may not be linearly related. This statistical approach enabled us to examine whether certain source code attributes are associated with higher or lower perceived quality in the generated documentation. Ultimately, this helps to better understand which structural characteristics in code may facilitate more accurate or complete Javadoc generation by LLMs.

6.3 Results

In this section, we present the results of the source code analysis. Building upon the dataset evaluated in Section 5.3, where the HA-Score was introduced, we examine how each of the selected metrics correlates with this score to better understand how the code's structural properties influence the quality of the generated Javadocs.



(a) WMC (b) RFC

(c) CBO (d) LCOM*

(e) LOC (f) Unique Words Quantity

Figure 6.1: Class-level metrics vs HA-Score (Spearman Coefficients)

Table 6.1 summarizes the obtained results. It reports the Spearman's correlation observed between each source code metric and the HA-Score, distinguishing between classes and methods. Statistically significant values are indicated in bold.

Table 6.1: Source Code Metrics Spearman's Correlation With HA-Score

	Correlation	
Metric	**Class**	**Method**
WMC	0.11	0.10
RFC	0.14	**0.14**
CBO	-0.06	0.07
LCOM*	**0.11**	—
LOC	**0.14**	0.07
Unique Words Quantity	-0.06	**-0.22**

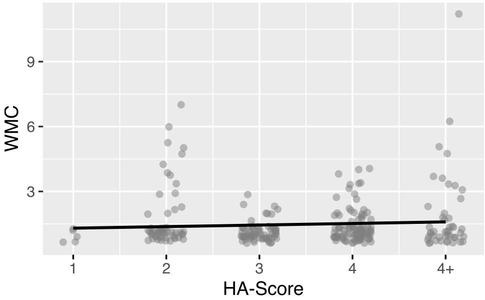
Across classes, the correlations between source code metrics and HA-Score remain generally weak, as illustrated in Figure 6.1. Nevertheless, a few tendencies can be observed. The strongest positive correlations appear for RFC and LOC (both Spearman 0.14), followed by LCOM* (Spearman 0.11). Although these values are small, they may indicate that larger or more behaviorally complex classes provide richer structural and semantic cues, which could help LLMs more easily infer the purpose and functionality of the class. In contrast, coupling (CBO) and lexical richness (Unique Words Quantity) show near-zero correlations, suggesting that external dependencies or vocabulary diversity at the class level do not meaningfully influence the model's ability to generate high-quality documentation. Overall, these results imply that class-level complexity and size offer only marginal benefits for LLM documentation quality.

A similar pattern emerges when examining methods, where correlations between structural metrics and HA-Score are also generally weak, as shown in Figure 6.2. Most complexity and size metrics exhibit weak positive correlations, suggesting that method-level structure have little effect on the LLMs' documentation performance. However, one metric stands out: Unique Words Quantity. Its moderate negative correlation with HA-Score (Spearman –0.22) suggests that methods with more focused and consistent vocabulary tend to receive higher documentation quality ratings. A plausible explanation is that excessive lexical variety may introduce ambiguity or make key conceptual features less noticeable, which makes it harder for the model to form a coherent summary. Conversely, methods with more focused and semantically coherent content may provide clearer contextual signals, enabling the model to generate more accurate documentation.

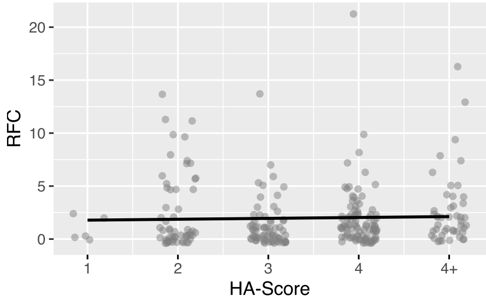
6.4 Discussion

The results obtained in this study might indicate that source code characteristics, especially those related to structural richness, are positively associated with the quality of LLM-generated documentation. Notably, LOC, RFC, and LCOM* showed positive correlations with HA-Score, with these effects being slightly stronger at the class level. Although these correlations remain weak, they subtly suggest that more elaborated and structurally articulated code elements may offer a clearer representation of their underlying intent. Such signals may help the model better infer the purpose and behavior of the code, thereby enabling it to generate more informative and accurate Javadocs.

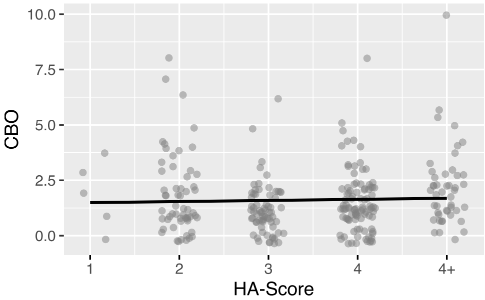
Figure 6.3 illustrates this behavior. The code snippet shows a very short (LOC: 3), simple (WMC: 1), and loosely coupled (RFC: 0) method. Yet, the LLM failed to generate a high-quality Javadoc for it. Compared to the original documentation, the generated version received an HA-Score of 1, the lowest possible rating, indicating that the two are not equivalent but rather opposites. This suggests that providing richer



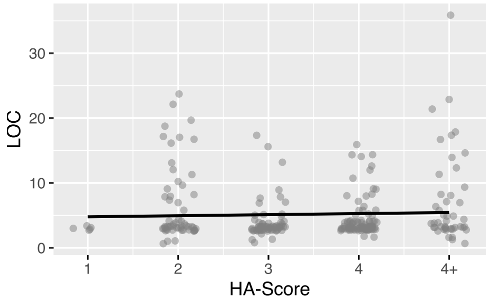
(a) WMC



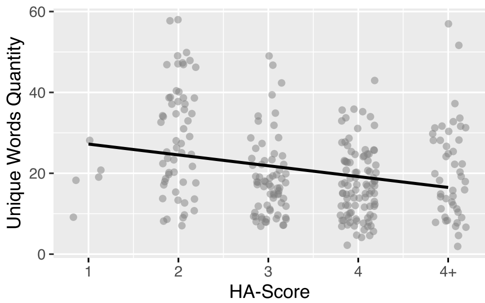
(b) RFC



(c) CBO



(d) LOC



(e) Unique Words Quantity

Figure 6.2: Method-level metrics vs HA-Score (Spearman Coefficients)

contextual information could have helped the LLM produce higher-quality documentation. Figure 6.4, on the other hand, shows the opposite case. The class is longer (LOC: 11), moderately complex (WMC: 2), and more tightly coupled (RFC: 8), thus offering the LLM richer structural and contextual cues. As a result, the generated Javadoc not only matched the original but surpassed it in quality, earning a high HA-Score of 4+.

However, it is important to note that these characteristic are generally viewed as undesirable from a software engineering standpoint, as they can reduce maintainability and increase the likelihood of defects. Their positive influence on documentation quality in this context reflects the LLM's ability to leverage additional context rather than indicating that such code structures are preferable in practice.

Conversely, the Unique Words Quantity metric shows a negative correlation, which is noticeably stronger at the method level, indicating that higher lexical diversity does not translate into better documentation quality. This suggests that varied or inconsistent terminology may obscure the semantic intent of the code, making it harder for the model

```
Original Javadoc
/**
 * Configure how unknown String keys will be handled.
 * @param fixedInput if false, an unknown key will be returned from {@link #
 intValue} as <code>-1</code>; if true,
 * the result will be undefined, and the resulting code will be faster
 */

LLM-Generated Javadoc
/**
 * Set whether the input for this method is fixed.
 * @param fixedInput true if the input is fixed, false otherwise
 */

Code Snippet
public void setFixedInput(boolean fixedInput) {
    this.fixedInput = fixedInput;
}
```

Figure 6.3: Short method (LOC: 3; WMC: 1; RFC: 0) with low-quality LLM-generated Javadoc (HA-Score: 1).

to generalize and produce coherent summaries.

In summary, these results highlight that LLMs benefit most from code that offers rich structural and contextual information. Size, complexity, and interaction patterns play a more substantial role in enabling high-quality comment generation, while cohesion and lexical diversity have limited or even negative influence. These findings underscore the importance of structural context over internal uniformity in documentation synthesis tasks. Nevertheless, it is important to emphasize that most correlations were weak and several were not statistically significant, meaning that these tendencies should be interpreted as suggestive patterns rather than strong or definitive effects.

6.5 Threats to Validity

In addition to the threats to validity concerning the data collection and dataset construction procedures described in Section 3.6, this study also presents threats specific to the source code evaluation, as follows. The process of computing source code metrics may introduce a potential threat to internal validity, as it depends on external tool implementations. To address this, we employed the CK tool [5], a widely known and actively maintained tool that has been repeatedly used in numerous studies [11, 22, 31, 46]. Its

```
Original Javadoc
/**
 * {@link Inflector} to for endpoint links.
 */

LLM-Generated Javadoc
/**
 * An inflector that maps container request contexts to responses using an
 endpoint links resolver.
 */

Code Snippet
private static final class EndpointLinksInflector implements Inflector <
ContainerRequestContext, Response > {

   private final EndpointLinksResolver linksResolver;

   private EndpointLinksInflector(EndpointLinksResolver linksResolver) {
      this.linksResolver = linksResolver;
   }

   @Override
   public Response apply(ContainerRequestContext request) {
      Map < String, Link > links = this.linksResolver
         .resolveLinks(request.getUriInfo().getAbsolutePath().toString());
      Map < String, Map < String, Link >> entity = OperationResponseBody.of(
      Collections.singletonMap("_links", links));
      return Response.ok(entity).build();
   }

}
```

Figure 6.4: Longer (LOC: 11), moderately complex (WMC: 2), and more tightly coupled (RFC: 8) class with high-quality LLM-generated Javadoc (HA-Score: 4+).

established presence in the literature and validated functionality provide confidence in the accuracy and reliability of the extracted metrics.

6.6 Concluding Remarks

The source code analysis conducted in this chapter revealed that certain structural properties of the code show weak correlation with the quality of documentation generated by LLMs. Metrics related to size and interaction displayed small positive correlations with HA-Scores, more noticeable at the class level, suggesting that code with slightly richer structural context may offer cues that help the model infer purpose and behavior. In contrast, cohesion and most method-level structural measures exhibited negligible correlations, while lexical diversity showed a modest negative association, indicating that higher vocabulary variety may hinder the model's ability to produce coherent summaries.

These findings suggest that LLMs benefit more from code that provides clear structural or contextual signals than from internally cohesive or lexically diverse code. However, it is important to recognize that the observed effects were generally weak, and several correlations were not statistically significant.

Chapter 7

Conclusion

In this master's dissertation we investigated whether LLMs can effectively generate code documentation, with a focus on code comments, specifically Javadocs. To accomplish this, we conducted a comprehensive empirical assessment composed of three interconnected studies. Together, these studies integrate quantitative, qualitative, and metric-centric perspectives.

The research began with the construction of a curated dataset extracted from three well-known and widely adopted open-source Java repositories: SPRING-BOOT, SPRING-FRAMEWORK, and GUAVA. From these sources, a statistically significant sample of 415 (code, Javadoc) pairs were carefully selected and refined to serve as the ground truth. This dataset was not only central to the experiments presented in this dissertation but also constitutes a reusable benchmark for future research in the area. Based on this dataset, we evaluated three state-of-the-art LLMs: GPT-3.5 Turbo, GPT-4o, and DeepSeek-V3. We prompted each model to generate Javadoc comments for every code snippet in the dataset, producing a total of 1,245 generated comments.

The evaluation was structured in three complementary stages. First, a quantitative study assessed the similarity between generated and reference documentation using established textual similarity metrics, namely BLEU, ROUGE-L, and METEOR. Second, a qualitative study involved human evaluators who rated the quality of the generated comments, allowing us to capture nuances that automated metrics cannot fully reflect. Finally, a source code assessment was conducted to analyze how intrinsic characteristics of the code, such as size, complexity, and coupling, influenced the quality of LLM-generated documentation.

7.1 Key Findings and Contributions

The main findings of this dissertation can be summarized as follows.

LLMs can produce high quality comments. Across the experiments, the generated comments frequently matched or even exceeded the quality of the original Javadocs. The quantitative evaluation indicated substantial lexical and structural alignment between the LLM-generated and the reference documentation, while the qualitative one revealed that a large proportion of generated comments are equivalent or better than the original ones.

Newer models generate more complete and informative documentation The comparison among models highlighted a clear progression in quality across generations of LLMs. While GPT-3.5 Turbo served as a strong baseline, both GPT-4o and DeepSeek-V3 consistently produced higher quality documentation. DeepSeek-V3, in particular, frequently outperformed the others, illustrating how rapid advancements in LLM design translate directly into improvements in applied software engineering tasks.

Textual similarity metrics alone are insufficient to evaluate LLM-generated content. A key finding of this dissertation is the limited reliability of automated textual similarity metrics. BLEU, for example, systematically underestimated the quality of outputs that were semantically correct but expressed with different lexical choices. ROUGE-L and METEOR correlated more closely with human evaluations but still failed to capture subtleties such as explanatory depth or contextual adequacy. The lack of semantic sensitivity and practical awareness in these metrics underscores the necessity of integrating human-centered assessments into the evaluation of AI-generated documentation.

Code structure influences the quality of generated documentation. Although software engineering standards typically favor smaller and less complex code elements to improve readability and maintainability, the results of this study revealed weak but positive correlations between code size, complexity, and the quality of generated comments. This suggests that the additional context available in richer code fragments might actually help LLMs produce more informative and accurate documentation.

Building on these findings, the main contributions of this work are summarized as follows:

We demonstrated the practical effectiveness of LLMs for software documentation. The results showed that LLMs can generate documentation not only comparable to human-written comments but also, in many cases, superior. This strengthens the case for integrating AI-assisted documentation tools into real-world development workflows.

We provided comparative insights into model performance. The evaluation of three different LLMs highlighted important differences across model generations and architectures. More advanced models, such as GPT-4o and DeepSeek-V3, delivered significant improvements in quality, setting expectations for practitioners who adopt cutting-edge tools.

We identified limitations in existing evaluation methods. The study revealed the shortcomings of relying exclusively on quantitative metrics. Concrete cases of misalignment between automated scores and human judgments emphasized the need for hybrid evaluation frameworks that combine the scalability of automated measures with the depth of structured human assessments.

We released an open dataset for future research. A curated dataset of (code, Javadoc) pairs was created and made publicly available (`https://doi.org/10.5281/zenodo.17653107`). This benchmark supports reproducibility, transparency, and continuity of research in automated documentation generation.

7.2 Future Work

Based on the results of this dissertation, which demonstrated the effectiveness of LLMs in generating high-quality code documentation, we identify several promising directions for future research.

A natural extension of this work is to broaden the dataset beyond Java to include projects developed in other programming languages. Such an expansion would enhance the generalizability of the findings and reveal language-specific challenges, such as syntax diversity and differing documentation conventions. Furthermore, extending the scope of documentation targets to encompass finer-grained code elements (e.g., statements, variable declarations, and packages) would enable a more comprehensive assessment of LLM capabilities in supporting software documentation.

Another promising direction is the development of an autonomous documentation agent capable of integrating seamlessly into real-world development workflows. Such an agent could leverage emerging protocols, such as the Model Control Protocol (MCP), to interact directly with code repositories, automatically generate or improve documentation, and submit these enhancements through pull requests. This line of research would bridge the gap between offline evaluation and practical deployment, making documentation automation directly actionable in software projects.

Future work may also investigate advanced prompting and fine-tuning strategies in combination with human-in-the-loop approaches. Techniques such as few-shot prompting, chain-of-thought reasoning, or retrieval augmentation could improve the accuracy and contextual richness of generated comments. When coupled with interactive frameworks in which developers collaborate with LLMs to iteratively refine documentation, these strategies could yield outputs that are both higher in quality and better aligned with real-world expectations.

Finally, subsequent research should address the long-term evolution of documentation in software projects. Mechanisms that ensure the continued relevance of LLM-generated comments as code evolves, including automated detection of outdated documentation and integration with version control systems, would contribute to the sustainability of automated approaches. Combining generation with robust change-detection pipelines could establish reliable workflows for maintaining accurate documentation over time.

References

[1] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on linguistic antipatterns affecting apis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 25–35, 2018.

[2] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. Software documentation: the practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 590–601, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE, 2019.

[4] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[5] Maurício Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[6] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005.

[7] Iqra Batool and Tamim Ahmed Khan. Software fault prediction using deep learning techniques. *Software Quality Journal*, 31(4):1241–1280, 2023.

[8] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3), March 2024.

[9] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6), 1994.

[10] Oracle Corporation. How to write doc comments for the javadoc tool. https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html, 2025.

[11] Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. Detecting bad smells with machine learning algorithms: an empirical study. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, page 31–40, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Samanta Dey, Venkatesh Vinayakarao, Monika Gupta, and Sampath Dechu. Evaluating commit message generation: to bleu or not to bleu? In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '22, page 31–35, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. A comparative analysis of Large Language Models for code documentation generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, 2024.

[14] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the BLEU: how should we assess quality of the code generation models? *Journal of Systems and Software*, 2023.

[15] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50(9):2254–2268, 2024.

[16] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, 2023.

[17] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. DocEng '02, page 26–33, New York, NY, USA, 2002. Association for Computing Machinery.

[18] Markus Freitag, David Grangier, and Isaac Caswell. BLEU might be guilty but references are not innocent. *arXiv preprint arXiv:2004.06063*, 2020.

[19] Davide Fucci, Alireza Mollaalizadehbahnemiri, and Walid Maalej. On using machine learning to identify knowledge in api reference documentation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 109–119, 2019.

[20] Google. *Guava: Google Core Libraries for Java*, 2011. Available in https://github.com/google/guava.

[21] Yvette Graham. Re-evaluating automatic summarization with BLEU and 192 shades of ROUGE. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 128–137. Association for Computational Linguistics, 2015.

[22] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. Pizza versus pinsa: On the perception and measurability of unit test code quality. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 336–347, 2020.

[23] Pouya Hamadanian, Behnaz Arzani, Sadjad Fouladi, Siva Kesava Reddy Kakarla, Rodrigo Fonseca, Denizcan Billor, Ahmad Cheema, Edet Nkposong, and Ranveer Chandra. A holistic view of ai-driven network incident management. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 180–188, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. Practitioners' expectations on automated code comment generation. ICSE '22. ACM, 2022.

[25] Dong Huang, Jie M. Zhang, Qingwen Bu, Xiaofei Xie, Junjie Chen, and Heming Cui. Bias testing and mitigation in llm-based code generation. *ACM Trans. Softw. Eng. Methodol.*, March 2025.

[26] Umar Iftikhar, Nauman Bin Ali, Jürgen Börstler, and Muhammad Usman. A tertiary study on links between source code metrics and external quality attributes. *Information and Software Technology*, 165:107348, 2024.

[27] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1646–1656, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 1646–1656, 2023.

[29] Jonathan Katzy, Yongcheng Huang, Gopal-Raj Panchu, Maksym Ziemlewski, Paris Loizides, Sander Vermeulen, Arie van Deursen, and Maliheh Izadi. A qualitative investigation into llm-generated multilingual code comments and automatic evaluation metrics. In *Proceedings of the 21st International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '25, page 31–40, New York, NY, USA, 2025. Association for Computing Machinery.

[30] Junaed Younus Khan and Gias Uddin. Automatic code documentation generation using GPT-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, 2023.

[31] Aleksandar Kovačević, Jelena Slivka, Dragan Vidaković, Katarina-Glorija Grujić, Nikola Luburić, Simona Prokić, and Goran Sladić. Automatic detection of long method and god class code smells through neural source code embeddings. *Expert Systems with Applications*, 204:117607, 2022.

[32] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, SIGDOC '99, page 147–153, New York, NY, USA, 1999. Association for Computing Machinery.

[33] Hans-Alexander Kruse, Tim Puhlfürß, and Walid Maalej. Can developers prompt? a controlled experiment for code documentation generation. In *International Conference on Software Maintenance and Evolution*, ICSME, 2024.

[34] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.

[35] Pedro Las-Casas, Alok Gautum Kumbhare, Rodrigo Fonseca, and Sharad Agarwal. Llexus: an ai agent system for incident management. *SIGOPS Oper. Syst. Rev.*, 58(1):23–36, August 2024.

[36] Alon Lavie. Evaluating the output of machine translation systems. In *Proceedings of the 9th Conference of the Association for Machine Translation in the Americas: Tutorials*, 2010.

[37] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, 2004.

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

[39] Zaira Hernandez Martinez, Patricia Martínez Moreno, José Antonio Vergara Camacho, and Gerardo Contreras Vega. Study on the use of defect metrics in the software development process. flaws and vulnerabilities. In *2023 IEEE International Conference on Engineering Veracruz (ICEV)*, pages 1–7, 2023.

[40] Nitika Mathur, Timothy Baldwin, and Trevor Cohn. Tangled up in BLEU: Reevaluating the evaluation of automatic machine translation evaluation metrics. *arXiv preprint arXiv:2006.06264*, 2020.

[41] Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *40th Annual Meeting of the Association for Computational Linguistics*, 2002.

[43] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. Classifying code comments in java software systems. *Empirical Software Engineering*, 24(3):1499–1537, 2019.

[44] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software*, 195:111515, 2023.

[45] Inderjot Kaur Ratol and Martin P. Robillard. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 112–122, 2017.

[46] Simone Romano, Fiorella Zampetti, Maria Teresa Baldassarre, Massimiliano Di Penta, and Giuseppe Scanniello. Do static analysis tools affect software quality when using test-driven development? In *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '22, page 80–91, New York, NY, USA, 2022. Association for Computing Machinery.

[47] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. Reassessing automatic evaluation metrics for code summarization tasks. In *29th Symposium on the Foundations of Software Engineering*, FSE, 2021.

[48] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427, 2017.

[49] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018.

[50] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[51] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. On the evaluation of neural code summarization. In *44th International Conference on Software Engineering*, ICSE, page 1597–1608, 2022.

[52] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, 2019.

[53] Charles Spearman. The proof and measurement of association between two things. 1961.

[54] Diomidis Spinellis. *Code quality: the open source perspective*. Adobe Press, 2006.

[55] Spring. *Spring Framework*, 2008. Available in https://github.com/spring-projects/spring-framework.

[56] Spring. *Spring Boot*, 2013. Available in https://github.com/spring-projects/spring-boot.

[57] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92, 2013.

[58] R. Subramanyam and M.S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[59] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source Code Summarization in the Era of Large Language Models . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1882–1894, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[60] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does BLEU score work for code migration? In *27th IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 165–176, 2019.

[61] Jiapeng Wang and Yihong Dong. Measurement of text similarity: A survey. *Information*, 11(9), 2020.

[62] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 53–64, 2019.

[63] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering.* Springer, Germany, 2012.

[64] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50, 2025.

[65] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.