Evaluating Large Language Models for Docstring-to-Code Alignment

UNDERGRADUATE HONOURS THESIS
COMPUTER SCIENCE, FACULTY OF SCIENCE
ONTARIO TECH UNIVERSITY
OSHAWA, ON, CANADA

Rabia Chattha

SUPERVISOR:
Cristiano Politowski
Ken Pu

April 16, 2026

Abstract

Large Language Models (LLMs) have demonstrated strong capabilities in software engineering tasks, particularly in code understanding and generation. However, their ability to correctly associate documentation with the exact code it describes remains an open challenge. This thesis investigates the problem of docstring-to-code alignment by framing it as a structured matching task.

In this work, source code is transformed by replacing documentation comments with unique slot identifiers while maintaining a list of candidate comments. Given this input, models are required to assign each slot identifier to the correct comment identifier under strict one-to-one constraints.

We evaluate multiple modern LLMs across a dataset of real-world repositories using three key metrics: exact-match accuracy, valid one-to-one mapping rate, and non-empty prediction rate. Results show that while some models achieve strong performance, inconsistencies remain in output completeness and reliability.

This work highlights both the potential and limitations of LLMs in reasoning about fine-grained relationships between code and documentation, and provides a structured framework for future evaluation.

Contents

1 Introduction **1**

2 Background **2**
2.1 Docstrings and Code Documentation 2
2.2 Large Language Models for Code 3
2.3 Limitations of Existing Work 4

3 Approach **5**
3.1 Overview 5
3.2 Comment Masking Pipeline 6
3.3 Candidate Comments 7
3.4 Prompt Construction 7
3.5 Pipeline Overview 8

4 Evaluation **9**
4.1 Metrics 9
4.2 Results 10
4.3 Accuracy Comparison 11
4.4 Discussion 12

5 Conclusions **13**
5.1 Summary & Conclusions 13
5.2 Limitations 13
5.3 Future Work 14

List of Figures

1 Docstring-to-code alignment pipeline showing data preprocessing, prompt construction, model inference, and evaluation stages. 8
2 Accuracy comparison across evaluated models. 11

List of Tables

1 Overall model performance across all evaluated repositories. 10

1 Introduction

Documentation is a fundamental component of modern software development. It enables developers to understand existing systems, collaborate effectively, and maintain code over time. Among the most widely used forms of documentation are docstrings and block comments, which provide natural language descriptions of code behavior, parameters, and intended functionality. Despite their importance, documentation is often inconsistent or outdated, particularly in large and evolving codebases.

Maintaining alignment between documentation and implementation is a non-trivial task. As software evolves, developers may update code without modifying the corresponding comments, leading to discrepancies that reduce trust in the documentation. This creates challenges not only for human developers but also for automated tools that rely on accurate documentation.

Recent advances in Large Language Models (LLMs) have significantly improved performance on code-related tasks such as generation, summarization, and translation [2, 3, 4]. These models are capable of understanding both natural language and programming constructs, making them promising candidates for tasks involving documentation analysis. However, most existing work focuses on generating documentation rather than verifying or aligning existing comments with code.

This thesis explores the problem of docstring-to-code alignment by reformulating it as a structured matching problem. Instead of asking a model to generate text, we require it to assign each masked code location to the correct documentation comment. This formulation enables precise evaluation and provides deeper insight into the model's reasoning capabilities.

2 Background

2.1 Docstrings and Code Documentation

Docstrings and block comments serve as an essential interface between human understanding and machine-executable code. They provide developers with contextual information about how a particular piece of code is intended to function, including its purpose, inputs, outputs, and any important assumptions. In many modern programming languages, such as Python and JavaScript, docstrings follow structured conventions that allow them to be automatically parsed and used in documentation generation tools.

Despite their importance, maintaining accurate and up-to-date documentation remains a persistent challenge. In practice, developers often modify code without updating the associated comments, leading to inconsistencies that can reduce trust in the documentation. These inconsistencies can make it more difficult for developers to understand the system, especially when working with unfamiliar codebases or collaborating in large teams.

From a research perspective, the relationship between code and documentation is particularly interesting because it involves aligning two different representations of the same concept: a formal, structured representation (code) and a natural language description (comments). This alignment requires both syntactic understanding and semantic reasoning, making it a challenging task for automated systems.

In this thesis, docstrings are treated not simply as supplementary text, but as structured entities that can be systematically analyzed and matched to specific regions of code. By transforming documentation into identifiable units, we are able to evaluate how well models can recover the relationships between code and its corresponding descriptions.

2.2 Large Language Models for Code

Large Language Models (LLMs) have emerged as powerful tools for understanding and generating both natural language and source code. These models are typically based on transformer architectures introduced by Vaswani et al. [5] and are trained on large-scale datasets that include a mixture of textual and programming data. As a result, they are able to capture patterns that connect code constructs with their corresponding natural language descriptions.

In recent years, LLMs have demonstrated strong performance on a variety of software engineering tasks, including code completion, summarization, translation, and bug detection [3, 4]. Their ability to reason about programming logic and produce human-readable explanations makes them particularly well-suited for tasks involving code documentation.

However, most existing applications of LLMs focus on generative tasks, where the model is asked to produce new text based on input code. While this is useful, it does not directly address the problem of verifying or aligning existing documentation. In contrast, the task explored in this thesis requires models to select the correct documentation from a predefined set, which introduces additional constraints and requires more precise reasoning.

Another important consideration is that LLMs often operate probabilistically, meaning their outputs can vary depending on prompt design and sampling parameters. This can lead to inconsistencies, especially in tasks that require strict adherence to formatting or structural constraints. Understanding these limitations is crucial when evaluating their performance in structured tasks such as docstring-to-code alignment.

2.3 Limitations of Existing Work

Although there has been significant progress in applying Large Language Models to software engineering tasks, relatively little work has focused on evaluating their ability to align existing documentation with code. Most prior research has concentrated on generating documentation from code or retrieving relevant code snippets from textual queries. While these tasks demonstrate the expressive capabilities of LLMs, they do not require precise matching between predefined elements.

One key limitation of generative approaches is that they are difficult to evaluate objectively. Generated documentation may be partially correct or semantically similar to the ground truth, making it challenging to measure performance using strict metrics. In contrast, the matching task considered in this thesis allows for exact evaluation, as each slot has a clearly defined correct answer.

Another limitation of existing work is the lack of constraints. Many LLM-based systems allow models to generate outputs freely, without enforcing structural requirements such as one-to-one mappings. This can result in outputs that are difficult to interpret or use in practice. By introducing explicit constraints, the approach in this thesis ensures that model outputs are both meaningful and usable.

Finally, prior studies often rely on simplified datasets or synthetic examples that may not fully capture the complexity of real-world codebases [1]. In this work, we address this limitation by using actual repositories, which introduces variability in coding style, structure, and documentation quality. This makes the evaluation more realistic and provides deeper insights into model behavior.

3 Approach

3.1 Overview

The overall approach of this thesis is to transform the problem of docstring-to-code alignment into a structured matching task that can be evaluated in a controlled and reproducible manner. Rather than asking models to generate documentation, we require them to identify the correct association between existing code locations and a predefined set of candidate comments.

This transformation is achieved through a multi-stage pipeline that begins with raw source code and ends with a set of model predictions that can be directly compared to ground truth mappings. Each stage of the pipeline is designed to isolate a specific aspect of the problem, allowing for more precise analysis of model behavior.

At a high level, the pipeline consists of comment extraction, masking, prompt construction, model inference, and evaluation. These stages are interconnected, and the quality of each step can significantly influence the final results. For example, inaccuracies in the masking process can propagate through the pipeline, while poorly constructed prompts can lead to ambiguous or invalid model outputs.

By structuring the problem in this way, we are able to evaluate not only whether models can understand the relationship between code and documentation, but also whether they can operate under strict constraints and produce consistent, interpretable outputs. This makes the approach particularly suitable for benchmarking and comparative analysis across different models.

3.2 Comment Masking Pipeline

The comment masking pipeline is a central component of the proposed system, as it transforms raw source code into a structured format suitable for evaluation. The process begins by scanning each source file and identifying all block comments and docstrings based on language-specific patterns. This includes multi-line comments in languages such as Python, JavaScript, and C-style languages, ensuring that a wide range of repositories can be processed consistently.

Once identified, each comment is extracted and assigned a unique identifier referred to as a `comment_id`. In parallel, a corresponding `slot_id` is generated to represent the location of that comment within the code. The original comment text is then replaced with a placeholder token containing the slot identifier, effectively masking the documentation while preserving the structure and readability of the source file.

In addition to masking, the pipeline records metadata such as the start and end line numbers of each comment. This information is useful for analysis and debugging, although it is not directly exposed to the model during inference. The result of this process is a modified version of the source code in which all documentation has been replaced by identifiable slots, along with a structured mapping between slot identifiers and their corresponding comment identifiers.

This transformation serves two key purposes. First, it prevents the model from directly relying on the original documentation, forcing it to infer relationships based on code semantics. Second, it converts the problem into a controlled matching task, enabling precise evaluation of model predictions.

3.3 Candidate Comments

The candidate comments component defines the pool of possible matches that the model can select from when assigning documentation to code slots. For each file, all extracted comments are collected and stored along with their corresponding identifiers. These comments form a closed set, meaning that the correct answer for each slot must be chosen from this list.

This formulation introduces an important constraint that distinguishes the task from generative approaches. Instead of producing new text, the model must reason about the relationship between the code and each candidate comment, selecting the one that best fits the given context. This requires the model to compare multiple options and make a decision based on both semantic similarity and structural cues.

One challenge associated with this setup is that candidate comments may be similar in content or style, especially in larger files. This increases the difficulty of the task, as the model must distinguish between closely related descriptions. Additionally, the number of candidates can vary across files, introducing variability in task complexity.

Despite these challenges, using a fixed set of candidate comments provides significant advantages for evaluation. It allows for precise measurement of correctness and ensures that outputs are directly comparable across models. Furthermore, it aligns well with real-world scenarios, where developers often need to match existing documentation to specific parts of a codebase.

3.4 Prompt Construction

Prompt construction is a critical component of the proposed approach, as it defines how the task is presented to the model. The prompt must provide sufficient context for the model to understand

the problem while also enforcing strict constraints on the output format.

Each prompt is constructed by combining three key elements: a list of slot identifiers, a list of candidate comments with their corresponding identifiers, and the masked source code. These elements are presented in a structured format, along with explicit instructions that guide the model's behavior. The instructions emphasize requirements such as using only the provided identifiers, assigning each slot exactly once, and producing valid JSON output.

Designing effective prompts requires careful consideration of both clarity and specificity. If the prompt is too vague, the model may produce inconsistent or incomplete outputs. On the other hand, overly complex prompts can increase the likelihood of formatting errors. Striking the right balance is essential for achieving reliable performance.

Another important aspect of prompt construction is consistency across models. By using a standardized format, we ensure that all models are evaluated under the same conditions, allowing for fair comparisons. This also makes it easier to identify differences in performance that are attributable to the models themselves rather than variations in input.

Overall, prompt construction plays a central role in bridging the gap between the raw data and the model's reasoning process. It directly influences both the quality and reliability of the predictions, making it a key area of focus in this work.

3.5 Pipeline Overview

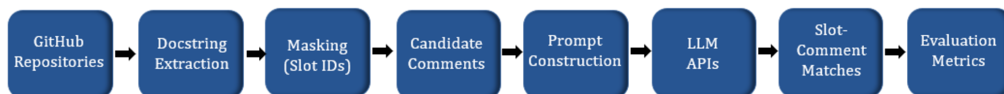

Figure 1: Docstring-to-code alignment pipeline showing data preprocessing, prompt construction, model inference, and evaluation stages.

Figure 1 illustrates the complete workflow used in this thesis. The process begins with GitHub repositories, where documentation comments are extracted and replaced with slot identifiers. Candidate comments are then compiled and used to construct a structured prompt, which is passed to LLM APIs. The resulting slot-comment matches are evaluated using the defined accuracy and consistency metrics.

4 Evaluation

4.1 Metrics

To evaluate model performance in a comprehensive and meaningful way, three complementary metrics are used. Each metric captures a different aspect of the task, allowing us to assess not only correctness but also reliability and consistency.

Exact-Match Accuracy measures the percentage of slot assignments where the predicted `comment_id` exactly matches the ground truth. This metric provides a direct indication of how well the model understands the relationship between code and documentation. However, accuracy alone is not sufficient, as it does not account for structural validity or completeness of the output.

Valid One-to-One Mapping Rate evaluates whether the model produces structurally valid outputs at the file level. A prediction is considered valid if every slot is assigned exactly one unique comment and no comment is used more than once. This metric is particularly important because the task imposes strict constraints, and violating these constraints can render otherwise correct predictions unusable in practice.

Non-Empty Prediction Rate measures the proportion of slots for which the model produces

a non-empty prediction. This metric captures output completeness and robustness. In some cases, models may fail to produce predictions for certain slots due to formatting errors or uncertainty, which can significantly impact usability.

Together, these metrics provide a balanced evaluation framework. Exact-match accuracy captures semantic correctness, valid mapping rate captures structural consistency, and non-empty prediction rate captures completeness. By analyzing all three metrics, we gain a more nuanced understanding of model performance.

4.2 Results

Model	Accuracy (%)	Valid Mapping (%)	Non-Empty (%)
DeepSeek Chat v3	71.82	89.86	80.44
DeepSeek v3.2	61.79	92.75	68.78
Claude Sonnet 4.5	63.27	88.41	68.20
Claude Sonnet 4.6	59.82	89.86	61.30
Gemini 3 Flash	57.52	84.06	60.31
Gemma 4 31B	56.94	85.51	60.15
Gemma 3 12B	43.14	76.81	58.42
Gemini 2.5 Flash	34.76	71.01	36.24

Table 1: Overall model performance across all evaluated repositories.

Table 1 presents the overall results for each evaluated model. DeepSeek Chat v3 achieved the highest exact-match accuracy at 71.82%, showing the strongest overall ability to correctly pair slot IDs with their corresponding comment IDs. DeepSeek v3.2 achieved the highest valid mapping rate at 92.75%, meaning it was especially strong at producing structurally valid one-to-one mappings. Gemini 2.5 Flash had the lowest overall accuracy and non-empty prediction rate, suggesting that it struggled more with both correctness and output completeness. Overall, the table shows that

stronger performance is not only about accuracy, but also about whether the model can consistently return complete and valid mappings.

4.3 Accuracy Comparison

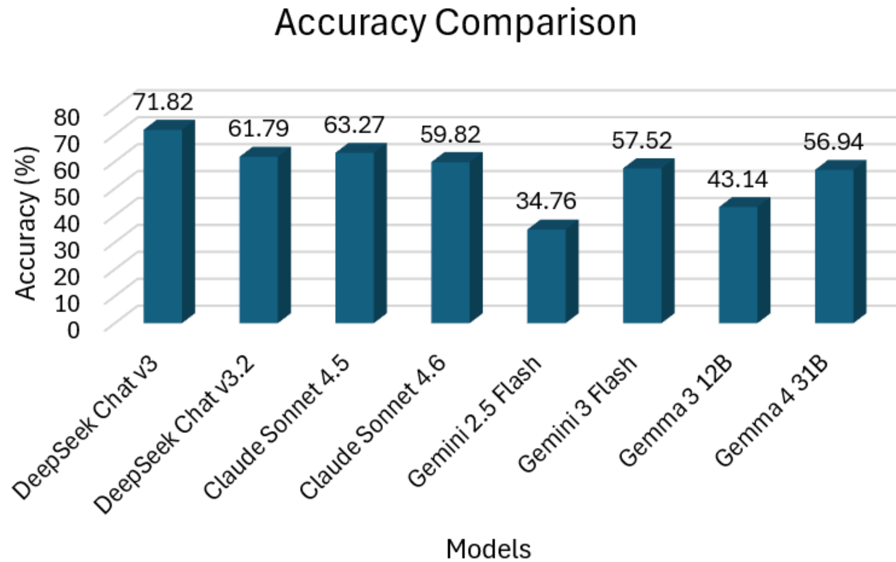

Figure 2: Accuracy comparison across evaluated models.

Figure 2 provides a visual comparison of exact-match accuracy across the evaluated models. The chart makes the performance differences easier to see than the table alone. DeepSeek Chat v3 stands out as the strongest model in terms of accuracy, followed by Claude Sonnet 4.5 and DeepSeek v3.2. The lower accuracy of Gemini 2.5 Flash and Gemma 3 12B suggests that not all models handle the structured matching task equally well, even when they are capable of general code understanding.

4.4 Discussion

The results demonstrate that modern Large Language Models are capable of performing the docstring-to-code alignment task with a reasonable degree of success, but their performance varies significantly depending on both the model architecture and the characteristics of the input data.

Among the evaluated models, DeepSeek Chat v3 achieves the highest overall accuracy, indicating a strong ability to reason about the relationship between code and natural language descriptions. Its high non-empty prediction rate further suggests that it is able to consistently produce complete outputs, which is critical for practical applications. Similarly, models such as Claude Sonnet 4.5 and DeepSeek v3.2 show strong performance across multiple metrics, highlighting the effectiveness of newer architectures.

However, the results also reveal several important limitations. First, performance varies considerably across repositories, suggesting that certain types of code are inherently more challenging for LLMs to interpret. Factors such as code complexity, naming conventions, and lack of contextual clues may contribute to this variability.

Second, some models struggle with enforcing the one-to-one mapping constraint, even when they demonstrate reasonable accuracy. This indicates that while models may understand individual relationships, maintaining global consistency across multiple assignments remains difficult.

Finally, the non-empty prediction rate highlights issues related to output formatting and robustness. In some cases, models fail to produce valid outputs for all slots, which reduces their effectiveness in real-world scenarios.

Overall, these findings suggest that while LLMs have made significant progress in understanding code, there is still room for improvement in tasks that require structured reasoning and strict

constraint satisfaction.

5 Conclusions

5.1 Summary & Conclusions

This thesis presented a structured framework for evaluating the ability of Large Language Models to align documentation comments with the code they describe. By reformulating the problem as a constrained matching task, we were able to move beyond traditional generative evaluations and instead focus on precise, interpretable measurements of model performance.

The proposed approach involves transforming source code through a masking pipeline, constructing structured prompts, and evaluating model outputs using a set of complementary metrics. This framework enables a detailed analysis of both correctness and reliability, providing insights that are not easily captured by standard benchmarks.

The experimental results demonstrate that modern LLMs are capable of performing this task with a moderate to high degree of accuracy. In particular, newer models show clear improvements over earlier versions, indicating that advancements in model architecture and training data are contributing to better code understanding capabilities.

5.2 Limitations

Despite these promising results, several limitations remain. The evaluation is based on static code analysis and does not account for runtime behavior or execution semantics. Additionally, model performance is sensitive to prompt design and output formatting, which can introduce variability.

Another limitation is the diversity of the dataset. While multiple repositories are included, they may not fully represent all types of real-world codebases. Certain domains or programming styles may present additional challenges that are not captured in this study.

5.3 Future Work

Future work can build upon this framework in several ways. One potential direction is to expand the dataset to include a wider range of programming languages and codebases, improving the generalizability of the results. Another direction is to refine prompt construction techniques to improve consistency and robustness across models.

Additionally, integrating traditional static analysis methods with LLM-based approaches may help address some of the observed limitations. By combining symbolic reasoning with learned representations, it may be possible to achieve more reliable and accurate alignment.

Finally, this work can be extended to real-world applications, such as automated documentation validation tools or developer assistance systems. These applications could help ensure that documentation remains accurate and useful over time, ultimately improving software quality and maintainability.

References

[1] Miltiadis Allamanis et al. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 2018.

[2] Tom Brown et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 2020.

[3] Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Baptiste Roziere et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[5] Ashish Vaswani et al. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.