A Comparative Analysis of Large Language Models for Code Documentation Generation

Shubhang Shekhar Dvivedi*
shubhang20474@iiitd.ac.in
IIIT Delhi
New Delhi, India

Vyshnav Vijay*
vyshnav20157@iiitd.ac.in
IIIT Delhi
New Delhi, India

Sai Leela Rahul Pujari*
sai20401@iiitd.ac.in
IIIT Delhi
New Delhi, India

Shoumik Lodh*
shoumik20407@iiitd.ac.in
IIIT Delhi
New Delhi, India

Dhruv Kumar
dhruv.kumar@iiitd.ac.in
IIIT Delhi
New Delhi, India

Abstract

This paper presents a comprehensive comparative analysis of Large Language Models (LLMs) for generation of code documentation. Code documentation is an essential part of the software writing process. The paper evaluates models such as GPT-3.5, GPT-4, Bard, Llama2, and Starchat on various parameters like Accuracy, Completeness, Relevance, Understandability, Readability and Time Taken for different levels of code documentation. Our evaluation employs a checklist-based system to minimize subjectivity, providing a more objective assessment. We find that, barring Starchat, all LLMs consistently outperform the original documentation. Notably, closed-source models GPT-3.5, GPT-4, and Bard exhibit superior performance across various parameters compared to open-source/source-available LLMs, namely LLama 2 and StarChat. Considering the time taken for generation, GPT-4 demonstrated the longest duration, followed by Llama2, Bard, with ChatGPT and Starchat having comparable generation times. Additionally, file level documentation had a considerably worse performance across all parameters (except for time taken) as compared to inline and function level documentation.

CCS Concepts

• **General and reference** → **Evaluation**; • **Computing methodologies** → **Natural language generation**; • **Software and its engineering** → **Documentation**.

Keywords

Code documentation, Large Language Models

ACM Reference Format:

Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A Comparative Analysis of Large Language Models for Code Documentation Generation. In *Proceedings of 1st ACM International Conference on AIware (AIware 2024).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*All authors contributed equally to this research.

AIware 2024, July 2024, Porto de Galinhas, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Code documentation refers to the process of describing the functionality, purpose, and usage of software code through accompanying text, comments, or annotations. High-quality code documentation is extremely important [22] as it serves as a vital communication tool for developers, aiding in the understanding, maintenance, and collaboration of software projects. Effective code documentation not only enhances code readability and comprehension but also allows software maintenance, debugging, and future development efforts. Code documentation plays an indispensable role in software engineering, promoting clarity, reliability, and efficiency in the development life-cycle.

In the field of artificial intelligence, Large Language Models (LLMs) are revolutionizing how we handle information and develop software [29]. These models, built on massive data sets and sophisticated algorithms, have moved beyond mere convenience to become vital in the software creation process, offering unprecedented assistance in code generation, debugging, and indeed, code documentation [20]. As software grows more complex and development speeds up, our dependence on these digital assistants is increasing. This makes it crucial to comprehensively assess their effectiveness and accuracy.

Code documentation generation using LLMs [14, 28] has the potential to speed up the software development process, as creating documentation is often considered as a tedious and time consuming process by most programmers [30]. LLMs can play a very crucial role in bridging this gap if they are able to create and maintain the code documentation in a consistent and efficient manner. To maintain the standards, the LLMs must be able to summarize the code, explain the code, and also be able to format the code into an easily understandable, legible and usable form.

Existing work on generating code documentation using LLMs has focused on studying a single LLM across multiple languages [20], comparing LLM generated documentation with student generated documentation and documentation outsourced from paid services [15, 21]. Yet, there is no existing study which provides a comprehensive comparison of various LLMs for generating code documentation across multiple paradigms like comparing across different documentations levels, comparing between open source

and close source LLMs etc. This comparison is important because it can help the software developers to select the best LLM for the code documentation based on their requirements and constraints. This paper precisely bridges this research gap and presents a comprehensive comparitive analysis of LLMs for generation of code documentation. We evaluate models such as GPT-3.5 [16], GPT-4 [27], Bard [25], Llama2 [33], and Starchat [23] on various parameters like Accuracy, Completeness, Relevance, Understandability, Readability and Time Taken for different levels of code documentation (inline level, function level and file level). In order to minimize subjectivity, we use a checklist-based system providing a more objective assessment.

In terms of research questions, this paper aims to find answers to the following research questions:

- **RQ1:** How do popular LLMs compare on their ability to generate various levels of code documentations?
- **RQ2:** How does the performance of LLMs compare with the human generated documentation for same codebases?
- **RQ3:** How does the the performance of closed/private sourced LLMs compare with LLMs whose sources are publicly available?

Our results indicate that all LLMs (except StarChat) consistently outperform the original documentation generated by humans. Our evaluation also reveals that closed-source models such as GPT-3.5, GPT-4, and Bard exhibit superior performance across various parameters compared to open-source/source-available LLMs, namely LLama 2 and StarChat. In terms of the time taken for generating the code documentation, GPT-4 is the slowest followed by Llama2, Bard, with ChatGPT and Starchat having comparable generation times. Finally, file level documentation had a considerably worse performance across all parameters (except for time taken) as compared to inline and function level documentation.

This paper aims to pave the path for future research and development, pinpointing areas where LLMs excel and identifying gaps that require attention in the realm of automating the task of code documentation using LLMs.

2 Related Work

Code documentation generation has been a subject of considerable research in the previous years, especially since the rise of LLMs [15, 20, 21] and the rapid improvement in code comprehensibility softwares using AI based techniques [24, 32].

Existing approaches to automatic generation of code documentation use data publishing frameworks [17] or transformer-based approaches [26]. A recent study done by Zhu et al [34] compares the ability of creating code summaries using Information Retrieval methods and Deep Learning methods (including one open-source LLM, i.e. StarCoder [23]).

Several studies have been done on the ability of LLMs to perform tasks related to code documentation [18, 24, 29, 32]. Su et al [32] propose using LLMs for key information extraction from the source code. Macneil et al [24] propose creating code summaries from code in the context of CS education. Geng et al [18] study multi-intent comment generation capabilities (like code functionality, how to use the code etc.) of LLMs. There have also been attempts of studying conversational uses of LLMs as a coding assistant, being able to translate, explain, and complete code as it is being written [29].

With regards to code documentation generation using LLMs, Khan et al [20] studied the performance of GPT-3 across 6 programming languages. They found that Codex (a GPT-3 based model pretrained on both natural and programming language) outperformed the state of the art documentation generation techniques present till 2022. Empirical studies have also been conducted to compare LLM generated code documentation with student generated documentation [21] which found that LLM generated documentations were significantly easier to understand and more accurate. Alizadeh et al compared the performance of ChatGPT (GPT-3.5), with some open source LLMs (like HugginChat and FLAN) and documentation generated by human based paid services (MTurk) [15]. They found that the LLMs outperform MTurk and the open source LLMs demonstrate competitive potential against ChatGPT in certain tasks.

All of the above studies have consistently demonstrated promising outcomes in the realm of automatic code documentation and associated tasks.

However, in comparison to the above mentioned existing research, our study offers new insights from multiple perspectives: (1) We present a comprehensive analysis of multiple LLMs (both closed-source and open-source LLMs). (2) Our study analyzes multiple code documentation levels (inline level, function level, class level). Thus, our study further enriches the understanding of LLM capabilities in code documentation.

3 Methodology

3.1 Levels of documentation

To conduct a thorough analysis of the LLMs' capabilities in generating code documentation, we decided to categorise the documentations into four distinct categories/levels [28]:

- Inline level documentation
- Function level documentation
- Class level documentation
- Folder level documentation

Each level represents a different scope and complexity of documentation, providing insights into how well each LLM handles varying degrees of detail and abstraction in technical writing.

3.1.1 ***Inline Level:*** It involves the generation of comments within the code, explaining specific lines or blocks of code. Inline documentation is crucial for clarifying complex or non-obvious parts of the code, enhancing readability and maintainability. It can also be very helpful in explaining a piece of code line by line to new coders. This will help evaluate the LLMs' ability to produce concise, relevant, and helpful inline comments that enhance understanding without cluttering the code.

3.1.2 ***Function level:*** At the function level, the focus shifts to documenting individual functions or methods. Good function-level documentation explains the purpose, parameters, return values, exceptions, and usage examples of functions. This assessment will measure how well each LLM can summarise the functionality and provide clear, accurate descriptions for function usage.

3.1.3 ***Class level:*** Class level documentation provides an overview of a class, its purpose, and its interactions with other classes. It often includes descriptions of key attributes and methods within the class. This level of documentation is critical for understanding the design and architecture of the software.

3.1.4 ***Folder Level:*** Folder Level Documentation is the highest level of documentation involving creating overviews for entire folders or modules, summarising the functionalities and interactions of multiple classes or files within them. We did not perform the analysis on folder level documentation because of technological limitations in most of the LLMs (except for GPT-4) which could not handle multiple file inputs at the same time.

3.2 Dataset

A set of 14 python code snippets were selected from a list of well documented publicly available repositories [1]. These selections were made to ensure a comprehensive analysis across different levels of documentation, including inline, function, and file levels.

For inline level documentation, our analysis included the following code repositories:

1 - verify.py from Bitcoin [3],
2 - clip_ops.py from TensorFlow [12], and
3 - conftest.py [4], test_functions.py [11], and basic_association.py from SQLAlchemy [8].

For function level documentation, we focused on functions like:

1 - update_prediction_data() from the Reddit archive [5],
2 - test_flush_no_pk() from SQLAlchemy [10],
3 - fit() from Scikit-Learn [7], and
4 - check_multisig() from Bitcoin [3].

File level documentation analysis utilized files from:

1 - bench_glmnet.py from Scikit-Learn [6],
2 - compat.py from TensorFlow [13],
3 - message-capture-parser.py from Bitcoin [2], and
4 - dict_of_sets_with_default.py [9], and basic_association.py from SQLAlchemy [8].

The selection of codebases for analysis was based on their popularity and the perceived quality of documentation within the software development community. We chose codebases known for their widespread use and reputation for excellent documentation, as indicated by discussions and recommendations in online forums, technical communities, and software development platforms. Focusing on codebases with well-regarded documentation allows for better assessment of code documentation generation methods. This approach ensures credibility and relevance in the evaluation process, strengthening the validity and usefulness of the research findings.

These code snippets were then individually run through all the LLMs. There were a total of 84 documentations (14 code snippets * 6 documentations each = 84 documentations) and each of them were evaluated on all the metrics (namely accuracy, completeness, relevance, understandability, readability and time taken) and the evaluated data was then collected for further analysis [1].

3.3 Evaluation parameters and metrics

We carefully picked evaluation criteria and their measures to thoroughly analyse the database. Our goal was to choose the metrics in such a manner such that the evaluation of the documentations would be thorough and cover all the major aspects of a documentation. Importantly, we wanted our assessment to be as objective as possible, so the metrics we chose are designed to give a fair and straightforward evaluation of the documentation. In order to make the evaluations more objective and reduce biases, we created a checklists for some metrics (namely completeness, relevance and readability) and the evaluation on those metrics is done according to the checklists. The evaluation parameters and their corresponding metrics are listed down below:

(1) **Accuracy:**
Accuracy is the measure of how correctly the documentation describes the code. This metric checks if the statements generated in the documentation is factually true or not, given the source code as the context. The accuracy is rated on a scale of 1-3, with 1 being the least accurate and 3 being perfectly accurate.

(2) **Completeness:**
Completeness of the documentation indicates the extent to which all the (important) parts of the code have been covered or not. Completeness is rated using a 5 point checklist on a scale of 0-5, on the basis of the number of checkpoints the documentation satisfied. Following are the checklist items for the various documentation levels:

- **For file level:**
  - Brief description of the file/script's purpose is included
  - List of dependencies is provided
  - Instructions or notes on how to use the script are included
  - List of main functions or classes defined in the file (if applicable)
  - Any special notes or considerations relevant to the script (e.g., context of use, limitations)
- **For function level:**
  - The description of the function's purpose is clear
  - All parameters are described with type and purpose
  - Return values are explained (if any)
  - Exceptions or errors the function may raise are documented.(if any)
  - Any additional notes or warnings are included
- **For inline level:**
  - Complex code blocks have accompanying comments
  - Non-obvious algorithmic decisions are explained (if any)
  - Workarounds or technical debt is noted (if any)
  - Any assumptions in the code are stated
  - Deprecated methods or upcoming changes are flagged. (depends on LLM)

(3) **Relevance:** Relevance shows how relevant the generated documentation is to the actual code, essentially it's ability to stick to the subject matter of the code and not go off topic. Relevance is different from accuracy in terms of focusing on the alignment of documentation with the subject matter of the code. Relevance is rated on a scale of **1** to **4** as explained below:
1 - No relevant information; completely off-topic or non-existent.
2 - Somewhat relevant; mixes essential and non-essential information.

3 - Mostly relevant; includes most of the key details necessary for understanding the function.
4 - Fully relevant; every piece of information helps understanding the function.

(4) **Understandability:** Understandability indicates how well a person reading the documentation is able to understand what the piece of code means/ how to use the code. Rating understandability is very much dependent on the level of experience and skill a user has in software development. Hence, there is a scope of understandability being a little subjective as an evaluation parameter, and is evaluated on the scale of **1** to **4**, where each value means the following:
1 - Completely unintelligible; impossible to understand.
2 - Somewhat understandable; could benefit from clearer language or examples.
3 - Mostly understandable; fairly easy to grasp but could be improved.
4 - Fully understandable; extremely clear and easy to understand, even for someone unfamiliar with the topic.

(5) **Readability:** Readability rates the formatting of the documentation. A well formatted documentation makes it convenient for the reader to go through the documentation. Readability is rated using a **5**-point checklist on a scale of **0-5**, on the basis of the number of checkpoints the documentation satisfies. The checkpoints are given below:
- **File level:** Header comment has clear delineation (e.g., lines or asterisks). (Documentation is clearly separated from the code), Consistent indentation is used for nested information, Lines are properly aligned (horizontal scrolling is absent), Is metadata like author, date, etc. present, Presence of spacing between Sections.
- **Function level:** Is the docstring clearly distinguishable from the code, Is there consistent formatting in the docstring (e.g., for parameters, returns), Any examples within the docstring are properly indented, Line breaks are used to separate different sections within the docstring, Is the docstring structured logically (e.g., description, parameters, returns, examples)?
- **Inline level:** Are comments placed close to the code they describe or do comments follow the same indentation as the code they are describing, Comments are brief and do not exceed the length of the code line, Consistent style is used for single-line and multi-line comments, Is there adequate spacing around comments for clarity, Are comments placed logically in the code for easy association with the relevant code?

(6) **Time Taken:** This is a simple measure of the time taken by the model to fully generate the response after the prompt has been entered.

3.4 Prompts

The prompts used as inputs to the LLMs were carefully constructed using the concepts of prompt engineering to generate the best possible responses from the large language models. The same prompt was used across models for the same level of documentation to maintain homogeneity and fairness while comparing the performances of models. The prompts were generally structured in a way such that they included the following:

- **Persona:** The LLM was assigned a persona
- **Context:** A context was given to the LLM under which the documentation is to be generated
- **Task:** The LLM was given the task using clear action words.
- **Format:** The LLM was specified the format in which the documentation was to be generated.

The prompts used for each level of documentation were slightly different, and are given below:

File level: *As a code documentation assistant, you are responsible for documenting at the file/script level of the given code snippet. When provided a file level code, your approach involves adding a header comment at the top of the file. This comment should be the documentation for the code and include all relevant information needed to understand or use the script. Code is as follows: <insert code>*

Function level: *As a code documentation assistant, you are programmed to document at the function level of the given code snippet. Your approach involves placing comments directly under the def statement of the function. The output should be the entire code along with the documentation of the function written. Code is as follows: <insert code>*

Inline level: *As a code documentation assistant, you are assigned to document at the in-line level of the given code snippet. When in-line comments are needed, you insert comments within the code itself. The output should be the entire code, along with the documentation you've added. Code is as follows: <insert code>*

3.5 LLMs Used

In our comparative analysis of code documentation generation capabilities among various large language models (LLMs), both open-source/source-available and closed-source models were utilized. The LLMs included in our study are GPT-3.5, GPT-4, Bard, LLama2, and Starchat.

The closed source proprietary LLMs used are the following:

- **GPT-3.5 [16]:** Developed by OpenAI, GPT-3.5 is an advanced iteration of the Generative Pre-trained Transformer models, boasting around 175 billion parameters. It is one of the most popular LLMs used as of right now.
- **GPT-4 [27]:** Also from OpenAI, GPT-4 is a more sophisticated and larger model than its predecessor, with improved performance in nuanced language understanding and generation. It is built on 1.76 trillion parameters and is also able to handle visual inputs along with textual ones.
- **Bard [25]:** Bard is a closed-source LLM developed by Google and is built on 137 billion parameters. It is one of the major players in the LLM space after OpenAIs chatGPT.

The open source/source available LLMs used are the following:

- **LLama2 [33]:** LLama2 is a source-available LLM developed by Meta. It has different model sizes with 7, 13 and 70 billion parameters. The one used for this study has 70 billion parameters.
- **Starchat [23]:** Starchat 2 is a fine tuned version of the StarCoder code completion focused model and is one of the more

commonly used code focused open source LLM. It is built on around 15 billion parameters.

3.6 Data Analysis

A total of 14 code snippets (including snippets of inline level, function level and file level) were collected from various code bases as mentioned in subsection 3.2. These code snippets were then used to generate documentation from each of the 5 LLMs (GPT-3.5, GPT-4, Bard, LLama 2, StarChat) using the prompts corresponding to the documentation level as mention in subsection 3.4.

All the generated documentations (14 documentations * 5 LLMs = 70 documentations) along with the original documentations of the 14 code snippets (14 documentations) were then evaluated by 2 of the authors of the paper. The evaluators have a good knowledge about the practice of source code documentations and have themselves created several documentations for various projects.

To avoid potential effects of the evaluator's biases on the evaluation of the datasets, percentage of disagreement was used for checking inter-rater reliability [19]. Both the evaluators rated 6 code snippets, i.e. 6 snippets * 6 documentations each = 36 documentations, on 5 metrics each, namely accuracy, completeness, relevance, understandability and readability (time taken was not used as it was a temporal measure and can not be effected by biases). This makes it a total of 180 evaluations (36 documentations * 5 metrics = 180 evaluations). Out of these 180 evaluations, there were disagreements on only 21 of them, which makes the disagreement score to be 11.67 percent which is a fairly low disagreement score. Disagreement score is calculated as disagreement score = (disagreements/total evaluation)*100.

The low disagreement score is partly due to the checklist system used to evaluate the documentations on different metrics. It is also because of the uniformly high quality documentations generated by most of the LLMs most of the time. Most disagreements (13 out of 21) were present in the "understandability" metric due to the subjectivity of the metric.

The evaluators rated the each documentation on the 6 mentioned metrics (namely accuracy, completeness, relevance, understandability, readability and time taken) in accordance with the checklist system and rating systems as described in section 3.3 [1].

The evaluated data was then visualized and analysed python libraries like pandas, numpy, matplotlib and seaborn. The code for the data analysis can be accessed here [1]. The findings and the visualizations are reported in the section 4.

4 Evaluation

We conducted a thorough analysis of 14 Python code snippets (as mentioned in Section 3.2), each accompanied with its respective documentation (generated by humans) for reference and comparison. We then ran various forms of analyses over the dataset, following are the results and observations:

4.1 Overall Analysis

We compute each metric (Accuracy, Completeness, Relevance, Understandability, Readability and Time Taken) for all the datasets and generated an output from each LLM. We then compute the metric by considering all the pairs of dataset and LLM we used for evaluation. For example, for computing the average accuracy in Table 1, we compute the average of accuracy achieved across all the pairs of dataset and LLM.

Table 1 presents a summary of the descriptive statistics like mean, standard deviation, range and distribution of values etc. for various parameters, including Accuracy, Completeness, Relevance, Understandability, Readability, and Time Taken.

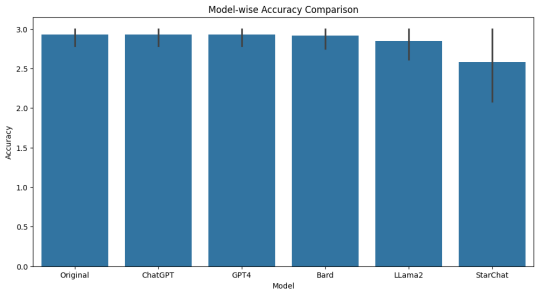

Figure 1: Model-wise accuracy comparison

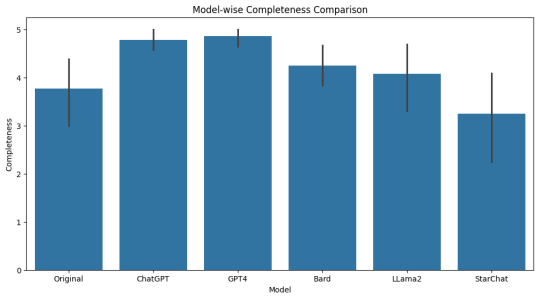

Figure 2: Model-wise completeness comparison

4.2 Comparison across different LLMs

To enhance the clarity of our findings, we utilized bar graph visualizations to compare mean ratings across different language models for each parameter.

In figure 1, we can see that there is a significant drop in average accuracy in StarChat, meanwhile the rest of the LLMs have almost on par accuracy as compared to the original documentation.

In figure 2, we can see that GPT-3.5 and GPT significantly outperform even the original documentation in terms of completeness and thoroughness of the code documentation, meanwhile Bard and LLama 2 have somewhat the same, and slightly better and StarChar has slightly worse performance in terms of completeness than the original documentation.

In figure 3, LLama 2 and StarChat have a slightly worse performance as compared to GPT-3.5 and 4 and Bard, which are on par with the original documentation.

From figure 4, we can see that the documentation generated by GPT-3, GPT-4 and Bard have better understandability as compared to the original documentation. LLama 2 and StarChat have a somewhat similar performance as the original documentation.

Metric	Accuracy	Completeness	Relevance	Understandability	Readability	Time Taken
mean	2.860	4.192	3.833	3.551	4.525	16.977
std	0.445	1.206	0.567	0.766	0.935	17.994
min	0.000	0.000	0.000	0.000	0.000	0.000
25%	3.000	4.000	4.000	3.000	4.000	0.000
50%	3.000	5.000	4.000	4.000	5.000	11.000
75%	3.000	5.000	4.000	4.000	5.000	30.250
max	3.000	5.000	4.000	4.000	5.000	75.000

Table 1: Accuracy, Completeness, Relevance, Understandability, Readability and Time Taken averaged across all LLMs

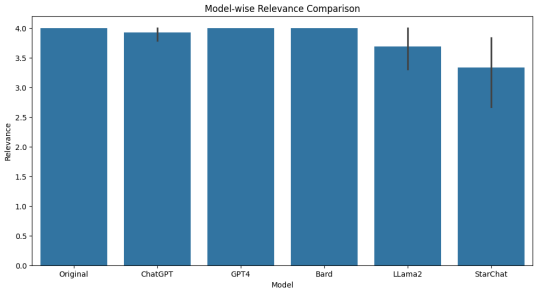

Figure 3: Model-wise relevance comparison

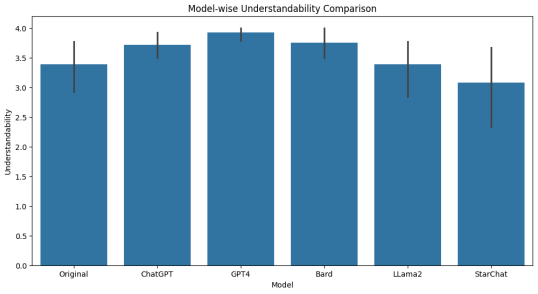

Figure 4: Model-wise understandability comparison

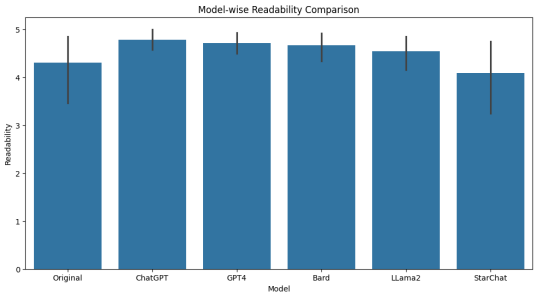

Figure 5: Model-wise readability comparison

From figure 5, we can see that all the models have high Readability, and most of them outperform the original documentation by small margins.

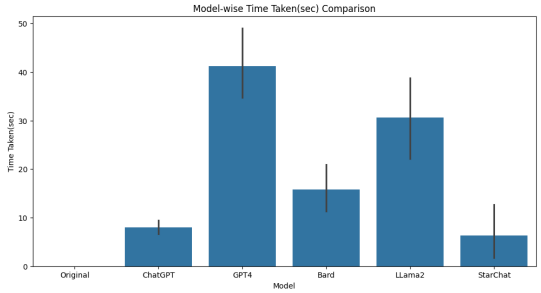

Figure 6: Model-wise Time taken comparison

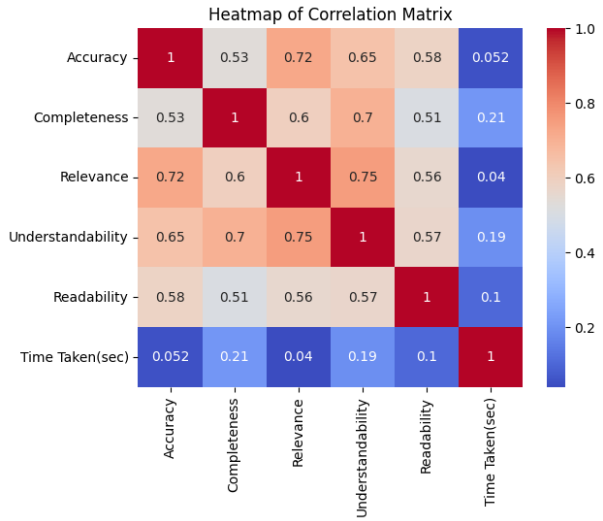

Figure 7: Metrics correlation heatmap

From figure 6, we can say that the time taken by GPT-4 to generate the documentation is very high as compared to the other models. GPT-3.5 and StarChat generate the documentation in the shortest amount of time. Bard and LLama 2 lie somewhere in between. This is the only criteria observed where GPT-4 has performed the worst by a large margin.

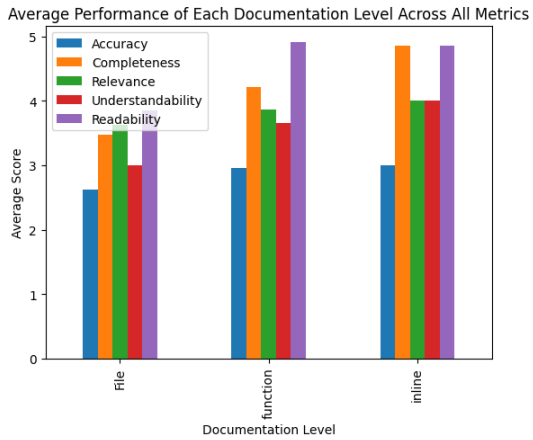

Figure 8: Documentation level-wise metrics comparison

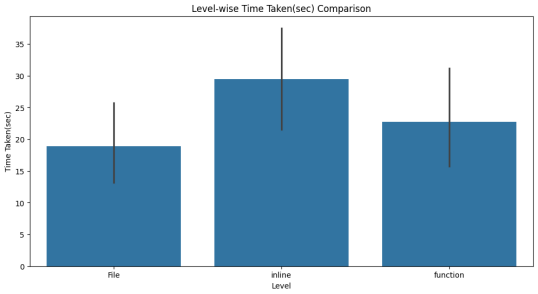

Figure 9: Documentation level-wise time taken comparison

4.3 Correlation between different metrics

As we can see from the correlation heatmap in figure 7, there is a somewhat strong correlation between Accuracy, Understandability and Relevance (each about 0.7-0.75). Other than this, there is not strong correlation between the other parameters. It is also interesting to note that the time taken by each LLM is completely independent of all the other parameters.

4.4 Comparison across different documentation levels

After looking at and comparing the performances of each LLM across different metrics, we will now look at the general performance of all the LLMs across different documentation levels, namely inline level documentation, function level documentation and file level documentation.

From figure 8, we can see that performance of the LLMs while generating file level documentation has a consistently worse performance as compared to the performance while generating inline and function level documentation in all parameters.

From figure 9, it can be seen that even though file level documentation generated is rated less, it still takes the least amount of time and documentation is generated faster than the other 2 levels of documentation.

4.5 Statistical Analysis

For statistical analysis, ANOVA (Analysis of Variance) [31] is employed to assess the significance of differences among group means in multi-group comparisons. This method is particularly essential when evaluating multiple models across various parameters, such as performance metrics. ANOVA utilizes F-statistics and p-values to determine whether observed variances across models are statistically significant or occur by chance, thereby providing a rigorous and technical approach to model evaluation and comparison. The alpha level or significance level used for this analysis is 0.05.

Criterion	F-statistic	P-value
Accuracy	1.1803	0.3271
Completeness	3.9874	0.0030
Relevance	3.1599	0.0123
Understandability	2.2788	0.05562
Readability	1.0665	0.3861
Time Taken	29.5365	1.0497×10^{-16}

Table 2: Statistical Analysis Results

- **Accuracy:**
  The F-statistic is relatively low, and the p-value is higher than the common alpha level of 0.05. This suggests that there is not enough statistical evidence to conclude that the model significantly affects accuracy.
- **Completeness:**
  The F-statistic is higher, and the p-value is below 0.05. This indicates a statistically significant effect of the model on completeness.
- **Relevance:**
  A higher F-statistic and a p-value below 0.05 suggest a significant effect of the model on relevance.
- **Understandability:**
  The F-statistic indicates some effect, but the p-value is slightly above 0.05. This suggests that the model's impact on understandability is borderline significant; it's close to being significant but doesn't quite reach the conventional threshold.
- **Readability:**
  Both the F-statistic and the p-value indicate that the model does not have a significant effect on readability.
- **Time Taken:**
  The very high F-statistic combined with an extremely low p-value (practically zero) strongly suggests that the model has a significant effect on the time taken.

5 Discussion

Our comparative analysis of code documentation generated by various Large Language Models (LLMs) yielded several significant findings. Firstly, with the exception of Starchat, all LLMs demonstrated either equivalent or superior performance compared to the original documentation, highlighting their potential for automating documentation tasks. Secondly, closed-source models like GPT-3.5, GPT-4, and Bard consistently outperformed their open-source counterparts, Llama2 and Starchat, across a majority of evaluation parameters. Notably, GPT-4 exhibited the longest time taken

for generation, followed by Llama2 and Bard, while ChatGPT and Starchat had comparable times. Lastly, file-level documentation consistently displayed poorer performance across all parameters (except for time taken) when compared to inline and function-level documentation.

Following are some implications, recommendations, and suggestions for the software development process in accordance with the findings of our paper:

(1) **Performance Discrepancies between LLMs:** The varying performance of LLMs underscores the importance of carefully selecting the model based on specific project requirements and evaluation criteria. Software developers should consider conducting pilot tests with multiple LLMs to determine which model best suits their documentation needs.
(2) **Closed-Source vs. Open-Source Models:** The superiority of closed-source models suggests potential benefits in investing in proprietary LLMs for documentation tasks.
(3) **Time Taken for generation considerations:** The variation in generation time among LLMs highlights the importance of balancing performance with efficiency in documentation tasks. Developers should weigh the trade-offs between documentation quality and generation time when selecting LLMs for their projects. Research efforts should focus on developing strategies to optimize generation time without compromising documentation quality, enhancing the overall efficiency of LLM based documentation workflows.
(4) **File-Level Documentation Challenges:** The comparatively inferior performance of file-level documentation emphasizes the need for alternative approaches to improve its effectiveness. Future research could explore innovative methods for enhancing the quality and utility of file-level documentation, such as incorporating contextual information or utilizing specialized LLM architectures fine-tuned for larger code structures.

6 Conclusion

Based on our comparative analysis, some very interesting findings have emerged. Except for Starchat, all Large Language Models (LLMs) demonstrate either equivalent or superior performance when compared to the original documentation, with Starchat consistently yielding suboptimal results. Secondly, closed-source models such as GPT-3.5, GPT-4, and Bard consistently outperform their open-source counterparts, Llama2 and Starchat, across a majority of evaluation parameters. Notably, in terms of generation time, GPT-4 exhibits the longest duration, followed by Llama2 and Bard, while ChatGPT and Starchat have comparable generation times. Lastly, file-level documentation displays significantly poorer performance across all parameters (except for time taken) as compared to inline and function-level documentation.

References

[1] 2024. GitHub - EmperorRP/Data-Evaluations—Comparative-Analysis-of-LLMs-for-Code-Documentation-Generation — github.com. https://github.com/EmperorRP/Data-Evaluations---Comparative-Analysis-of-LLMs-for-Code-Documentation-Generation. [Accessed 30-03-2024].

[2] [n.d.]. bitcoin/contrib/message-capture/message-capture-parser.py — github.com. https://github.com/bitcoin/bitcoin/blob/e25af11225d9d94ecf7068bf7a9a359268786fbe/contrib/message-capture/message-capture-parser.py.

[3] [n.d.]. bitcoin/contrib/verify-binaries/verify.py — github.com. https://github.com/bitcoin/bitcoin/blob/e25af11225d9d94ecf7068bf7a9a359268786fbe/contrib/verify-binaries/verify.py.

[4] [n.d.]. GitHub - sqlalchemy/sqlalchemy: The Database Toolkit for Python — github.com. https://github.com/sqlalchemy/sqlalchemy/tree/main.

[5] [n.d.]. reddit/r2/r2/lib/inventory.py — github.com. https://github.com/reddit-archive/reddit/blob/753b17407e9a9dca09558526805922de24133d53/r2/r2/lib/inventory.py.

[6] [n.d.]. scikit-learn/benchmarks/bench_glmnet.py — github.com. https://github.com/scikit-learn/scikit-learn/blob/1495f69242646d239d89a5713982946b8ffcf9d9/benchmarks/bench_glmnet.py.

[7] [n.d.]. scikit-learn/sklearn/cluster/_kmeans.py — github.com. https://github.com/scikit-learn/scikit-learn/blob/92c9b1866/sklearn/cluster/_kmeans.py.

[8] [n.d.]. sqlalchemy/examples/association/basic_association.py — github.com. https://github.com/sqlalchemy/sqlalchemy/blob/8503dc2e948908199cd8ba4e6b1d1ddcf92f4020/examples/association/basic_association.py.

[9] [n.d.]. sqlalchemy/examples/association/dict_of_sets_with_default.py — github.com. https://github.com/sqlalchemy/sqlalchemy/blob/8503dc2e948908199cd8ba4e6b1d1ddcf92f4020/examples/association/dict_of_sets_with_default.py.

[10] [n.d.]. sqlalchemy/examples/performance/bulk_inserts.py — github.com. https://github.com/sqlalchemy/sqlalchemy/blob/8503dc2e948908199cd8ba4e6b1d1ddcf92f4020/examples/performance/bulk_inserts.py.

[11] [n.d.]. sqlalchemy/test/sql/test_functions.py at main · sqlalchemy/sqlalchemy — github.com. https://github.com/sqlalchemy/sqlalchemy/blob/main/test/sql/test_functions.py.

[12] [n.d.]. tensorflow/tensorflow/python/ops/clip_ops.py — github.com. https://github.com/tensorflow/tensorflow/blob/v2.13.0/tensorflow/python/ops/clip_ops.py.

[13] [n.d.]. tensorflow/tensorflow/python/util/compat.py at v2.13.0 · tensorflow/tensorflow — github.com. https://github.com/tensorflow/tensorflow/blob/v2.13.0/tensorflow/python/util/compat.py.

[14] Toufique Ahmed and Premkumar Devanbu. 2023. Few-shot training LLMs for project-specific code-summarization. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22). Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3551349.3559555

[15] Meysam Alizadeh, Maël Kubli, Zeynab Samei, Shirin Dehghani, Juan Diego Bermeo, Maria Korobeynikova, and Fabrizio Gilardi. 2023. Open-Source Large Language Models Outperform Crowd Workers and Approach ChatGPT in Text-Annotation Tasks. https://doi.org/10.48550/arXiv.2307.02179 arXiv:2307.02179 [cs].

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. https://doi.org/10.48550/arXiv.2005.14165 arXiv:2005.14165 [cs] version: 4.

[17] Denise Che. [n. d.]. Automatic Documentation Generation from Source Code. ([n. d.]).

[18] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2023. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. arXiv:2304.11384 [cs.SE]

[19] Natasa Gisev, J. Simon Bell, and Timothy F. Chen. 2013. Interrater agreement and interrater reliability: Key concepts, approaches, and applications. Research in Social and Administrative Pharmacy 9, 3 (2013), 330–338. https://doi.org/10.1016/j.sapharm.2012.04.004

[20] Junaed Younus Khan and Gias Uddin. 2022. Automatic Code Documentation Generation Using GPT-3. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ACM, Rochester MI USA, 1–6. https://doi.org/10.1145/3551349.3559548

[21] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. In Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (<conf-loc>, <city>Turku</city>, <country>Finland</country>, </conf-loc>) (ITiCSE 2023). Association for Computing Machinery, New York, NY, USA, 124–130. https://doi.org/10.1145/3587102.3588785

[22] T.C. Lethbridge, J. Singer, and A. Forward. 2003. How software engineers use documentation: the state of the practice. IEEE Software 20, 6 (Nov. 2003), 35–39. https://doi.org/10.1109/MS.2003.1241364

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu,

Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! https://doi.org/10.48550/arXiv.2305.06161 arXiv:2305.06161 [cs].

[24] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. 2022. Generating Diverse Code Explanations using the GPT-3 Large Language Model. In Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2 (ICER '22, Vol. 2). Association for Computing Machinery, New York, NY, USA, 37–39. https://doi.org/10.1145/3501709.3544280

[25] James Manyika and Sissie Hsiao. [n. d.]. An overview of Bard: an early experiment with generative AI. ([n. d.]).

[26] Felipe Meneses. [n. d.]. Documentation Is All You Need. ([n. d.]).

[27] OpenAI. 2023. GPT-4 Technical Report. https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774 [cs].

[28] Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. 2022. A Review on Source Code Documentation. ACM Transactions on Intelligent Systems and Technology 13, 5 (June 2022), 84:1–84:44. https://doi.org/10.1145/3519312

[29] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In Proceedings of the 28th International Conference on Intelligent User Interfaces (Sydney, NSW, Australia) (IUI '23). Association for Computing Machinery, New York, NY, USA, 491–514. https://doi.org/10.1145/3581641.3584037

[30] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. 2015. To document or not to document? An exploratory study on developers' motivation to document code. In Advanced Information Systems Engineering Workshops: CAiSE 2015 International Workshops, Stockholm, Sweden, June 8-9, 2015, Proceedings 27. Springer, 100–106.

[31] Lars St»hle and Svante Wold. 1989. Analysis of variance (ANOVA). Chemometrics and Intelligent Laboratory Systems 6, 4 (Nov. 1989), 259–272. https://doi.org/10.1016/0169-7439(89)80095-4

[32] Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman Nath. 2023. HotGPT: How to Make Software Documentation More Useful with a Large Language Model?. In Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23). Association for Computing Machinery, New York, NY, USA, 87–93. https://doi.org/10.1145/3593856.3595910

[33] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. https://doi.org/10.48550/arXiv.2307.09288 arXiv:2307.09288 [cs].

[34] Tingwei Zhu, Zhong Li, Minxue Pan, Chaoxuan Shi, Tian Zhang, Yu Pei, and Xuandong Li. 2023. Deep is Better? An Empirical Comparison of Information Retrieval and Deep Learning Approaches to Code Summarization. ACM Trans. Softw. Eng. Methodol. (nov 2023). https://doi.org/10.1145/3631975 Just Accepted.