Ability of LLMs to generate documentation

albara Abumansoor, Ali-Algamdi, Abdullah Hani, Ahmad Al-Harbi, Fadi Al-Zahrani

February 2025

1 Abstract

2 Introduction

In this literature review, we investigate how Large Language Models (LLMs) can generate effective documentation for software development. We analyze five papers to determine their relevance to our study, focusing on the accuracy, usefulness, and maintainability of LLM-produced documentation.

3 Papers

3.1 Comparative Analysis of Large Language Models for Code Documentation Generation

3.1.1 idea

The idea of this paper is to do a comprehensive analysis for large language models (LLMs), previous approaches were comparing only one model or not considering the various levels of documentation, while this paper has considered the level of documentation and comparing between open verse closed sources (LLMs) [3].

3.1.2 method

Their methodology is using checklist-based system, depending on six measurements: accuracy, completeness, relevance, understandability, readability and time taken.a Also their prompts has some structured criteria which are: persona , context, task and format, additionally prompts used into 3 different styles: file level, function level, in-line level.[3].

3.1.3 datasets

They used a set of 14 python code snippets were selected from a list of well documented publicly available repositories.[3].

3.1.4 evaluation metric

used a chick-list for metrics and six attributes, first accuracy: which how correct the document describe the code,scaling from 1 to 3, 1 is least and 3 is most. second completeness: which completeness of the documentation indicates the extent, scaling from 0-5 for various documentation levels. third relevance: how the generated document is relevant to actual document, scale from 1 to 4. 4th understandability: how person who is reading the document is able to understand, scale from 1 to 4. 5th readability: how formatting document is scale from 0-5. 6th time taken: how much time taken the model to fully generate a document after prompts.[3].

3.1.5 results

Their result is that all the large language models (LLMs) has more performance over the original documentation including (ChatGPT, brad) except (StarChat), additionally file level documentation has lowest performance in all attributes except the time taken.[3].

3.1.6 reflection

although the idea of using Large language models (LLMs) is relevantly new, but it has a good impact for well-written documents due to many factors that we are discussed about. the study does not show how they response to prompts after execution.

3.2 Testing the Effect of Code Documentation on Large Language Model Code Understanding

3.2.1 idea

This paper represents an empirical investigation into the influence of the attributes of code and documentation on LLM performance. Also it shows that documents that are misleading may considerably jeopardize code understanding.[5]

3.2.2 method

The method applied within this study consists of assessing the capability of the generation of successful unit tests through LLMs under varying documentation conditions. and then classify the test results under 3 categories: Runtime Error, Failure, Success.[5]

3.2.3 datasets

The study exploits 164 human-solved practical code issues from HumanEval, the widely recognized benchmark to test LLMs on code comprehension.

3.2.4 evaluation metric

The (LLM) is prompted to generate unit tests for functions from the HumanEval dataset. The generated unit tests are executed and categorized into three possible outcomes: Runtime Error: The test crashes before completion. Failure: The test runs but does not pass. Success: The test runs and passes successfully.[5]

3.2.5 results

After comparing between Chat(gpt-3.5 and GPT-4) the results was GPT-3.5 generate significantly more runtime errors than GPT-4. The random comments scenario had the worst performance for both models (22.1% success for GPT-3.5, 68.1% for GPT-4), confirming that incorrect documentation negatively impacts LLM understanding. Replacing variable names with random strings had no significant effect on test success, except Replacing with animals names, it caused a small drop in success rates. The results confirm that GPT-4 generates more successful and meaningful test cases than GPT-3.5.[5]

3.2.6 reflection

LLMs perform best with accurate documentation, while misleading comments harm their effectiveness. Variable renaming has minimal impact, but GPT-4 consistently outperforms GPT-3.5 in generating successful unit tests.[5]

3.3 A Study on the Efects on Developer Productivity

3.3.1 idea

The study explores the merging of generative AI and large language models (LLMs) into software-documented processes and evaluates the resulting impact on developer productivity. The purpose is to utilize an LLM-based documentation system to automate the creation and retrieval of software documentation to minimize human manual effort with efficient work practices.[1]

3.3.2 method

Their methodology is a controlled experiment, followed by a survey, to further assess the effectiveness of the LLM-powered documentation system. Participants were required to generate and make use of documentation using both manual and LLM-assisted approaches. The key productivity metrics measured and compared between these two approaches were effectiveness, velocity, and quality.[1]

3.3.3 datasets

Their own dataset, which is derived from a controlled experiment and survey.[1]

3.3.4 evaluation metric

The key productivity metrics measured and compared between the two approaches were: Effectiveness, which looked at how relevant or accurate the documentation. Velocity, according to how long it would take to create or retrieve documentation using either manual or LLM-assisted methods. Quality, which was rated for understandability, completeness, and readability.[1]

3.3.5 results

The results show that the documentation system that uses LLM is a great productivity enhancer. It helps developers create and understand documentation more quickly and accurately than doing it manually.[1]

3.3.6 reflection

although the idea of using large language models (LLMs) is relevantly new, but it has a good impact to enhance the developer productivity in terms of effectiveness, velocity and quality. [1]

3.4 An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation

3.4.1 idea

This paper discusses RepoAgent, an open-source framework designed to generate and maintain repository-level code documentation using Large Language Models (LLMs). Traditional documentation methods require significant time and effort, often resulting in outdated or incomplete information. RepoAgent aims to automate this process by leveraging LLMs to generate high-quality documentation that not only describes code functionality but also provides practical usage guidance. By integrating with Git, the framework ensures that documentation remains synchronized with code updates, reducing the need for manual intervention. [4]

3.4.2 method

The paper outlines three key components of RepoAgent's workflow:

Global Structure Analysis – The framework begins by parsing the repository's source code, extracting meta-information about classes and functions using Abstract Syntax Tree (AST) analysis. It then constructs a project tree, which preserves hierarchical relationships within the code. Additionally, it employs the Jedi library to identify caller-callee relationships, allowing the LLM to generate documentation with a deeper understanding of code dependencies.

Documentation Genertion – RepoAgent utilizes a structured prompt template to guide the LLM in producing detailed documentation. The generated

output includes functionality descriptions, parameter explanations, code descriptions, usage notes, and example outputs. Once the documentation is generated, it is formatted into Markdown, making it more accessible and easier to navigate.[4]

Automated Documentation Updates – By integrating with Git hooks, RepoAgent automatically tracks code modifications and updates only the affected parts of the documentation. This feature ensures that documentation remains accurate and up to date without requiring manual revisions.

3.4.3 datasets and evaluation metric

The paper evaluates RepoAgent on nine Python repositories of varying sizes, ranging from less than 1,000 to over 10,000 lines of code. Two of these repositories Transformers and LlamaIndex were used in blind preference tests to compare human-authored documentation with documentation generated by RepoAgent.[4]

To assess performance, the paper employs multiple evaluation metrics:

Human Preference Testing – In two blind studies, RepoAgent's documentation was preferred over human-written documentation 70% of the time for Transformers and 91.33% for LlamaIndex. Reference Recall – The abilty of RepoAgent to correctly identify and document caller-callee relationships was tested against other code summarization methods. The results indicate that RepoAgent outperforms models that rely solely on isolated code snippets. Format Alignment – The generated documentation was assessed based on its adherence to a structured format. The results show that LLMs like GPT-4 performed well in maintaining structured documentation, while smaller models like LLaMA-2-7B exhibited inconsistencies. Parameter Identification Accuracy – The framework was tested for its ability to correctly extract and describe function parameters. RepoAgent, when powered by GPT-4-0125, demonstrated the highest accuracy among evaluated models.[4]

3.4.4 results

The paper highlights several key contributions of RepoAgent:

High Quality, Repository Level Documentation Unlike traditional approaches that summarize individual functions, RepoAgent generates context-aware documentation that considers the entire repository structure. Automated Synchronization with Code Changes By leveraging Git hooks, the system ensures that documentation remains up to date without requiring manual intervention. Enhanced Readability and Guidance The framework produces documentation that not only describes code but also provides examples, usage notes, and potential limitations, making it more informative for developers.[4]

3.4.5 reflection

The paper presents RepoAgent, a framework that automates repository-level documentation generation and maintenance using Large Language Models (LLMs).

By considering the global structure of a repository rather than isolated code snippets, RepoAgent generates detailed, structured, and context-aware documentation, surpassing human-authored documentation in clarity and usability in 70%–91.33% of cases. It integrates with Git workflows to ensure documentation remains up-to-date, significantly reducing manual effort. However, RepoAgent has several limitations, including language dependency (currently supporting only Python), LLM reliance (requiring high-quality models like GPT-4 for optimal results), and the need for human oversight to prevent inaccuracies or hallucinations. Additionally, the lack of standardized evaluation benchmarks makes it difficult to quantitatively compare its performance with other methods. To address these challenges, future research could explore multi-language support, interactive documentation querying, fine-tuning open-source models, and developing standardized evaluation metrics for automatic documentation generation. Despite its limitations, RepoAgent lays a strong foundation for LLM-powered software documentation, offering an effective solution for improving documentation quality and maintainability in large-scale codebases.

3.5 Automated API Docs Generator using Generative AI

3.5.1 idea

In this paper, we present APIDocBooster-an extract-then-abstract framework that combines the best of both worlds: extractive (i.e., producing faithful summaries with no restriction to length) and abstractive summarization (i.e., producing coherent and concise summaries). APIDocBooster consists of two major stages: 1) Context-aware Sentence Section Classification (CSSC) and 2) UPdate SUMmarization (UPSUM). CSSC classifies API-relevant information extracted from various sources into sections of API documentation. In UPSUM, extractive summaries describing the incoming data in their own words are followed by generation of abstractive summaries utilizing in-context learning, where the extractive summaries serve as guiding knowledge. [2]

3.5.2 method

API documentation summaries are generated by APIDocBooster, which classifies relevant sentences from Stack Overflow and YouTube by Context-aware Sentence Section Classification (CSSC). It then extracts key information using UPdate SUMmarization (UPSUM), which implements an EXTractive UPdate summarization algorithm (ExtUP). Finally, UPSUM produces abstractive summaries for every API documentation section. [2]

3.5.3 datasets

We collect relevant Stack Overflow posts and YouTube videos in the first phase and it consists of 4,344 sentences then the second phase comprises 48 extractive summaries, generated from the data gathered in the first phase. [2]

Source : https://github.com/autumn-city/APIDocBooster

3.5.4 evaluation metric

This experiment introduces three research questions. The first question concerns assessing how well CSSC classifies sentences into relevant sections. The second question concerns assessing the effectiveness of ExtUP with regard to updating extractive summarization. The final question examines the success of APIDocBooster for the augmenting of API documentation. [2]

3.5.5 results

CSSC provides good results in classifying sentences into intended sections, yielding high-weighted precision, and ExtUP showing significant progress in update extractive summarization, meaning it is well put to extract short and pertinent summaries from fresh content. This endorses the obvious enhancement of API documentation using APIDocBooster. The end-users have reported better clarity and usability. [2]

3.5.6 reflection

This article presents APIDocBooster: a framework that combines both extractive and abstractive summarizations for API documentation enhancement. CSSC for sentence classification and UPSUM for summarization have been able to achieve very high precision and satisfactory update results. Evaluation tests point towards enhanced clarity and usability, backed by end-user feedback. The method makes significant strides in automating the augmented generation of API documentation.

References

[1] Adam Alrefai and Mahmoud Alsadi. Large language models for documentation: A study on the effects on developer productivity, 2024.

[2] Prakhar Dhyani, Shubhang Nautiyal, Aditya Negi, Shikhar Dhyani, and Preeti Chaudhary. Automated api docs generator using generative ai. In *2024 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, pages 1–6. IEEE, 2024.

[3] 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*, pages 65–73, 2024.

[4] 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.

[5] William Macke and Michael Doyle. Testing the effect of code documentation on large language model code understanding. *arXiv preprint arXiv:2404.03114*, 2024.