AI Based Code Documentation Generation

Abhishek S. Bhosale[1], Aditya G. Mahajan[2], Kedar S. Pawar[3], Abhishek V. Ulagadde[4], Prof. Madhuri Mane[5]

[1,2,3,4] *Student, Dept. of Computer Engineering, Pune Institute of Computer Technology Pune, India.*

[5] *Professor, Dept. of Computer Engineering Pune Institute of Computer Technology Pune, India.*

***Abstract*—Automated documentation generation has become increasingly important in large-scale software development due to the growing complexity and size of modern codebases. Traditional documentation methods are labor-intensive, prone to inaccuracies, and often struggle to keep up with rapid development cycles. The integration of large language models (LLMs), such as GPT-4 and Codex, has introduced significant improvements in automated documentation, but challenges remain, particularly in handling large files and complex module dependencies. This survey paper reviews the current state of research on intelligent code chunking, dependency resolution, and multilanguage support in documentation generation tools. It also proposes a novel system that combines these techniques to generate context-aware documentation for large, multi-language codebases, ensuring accurate and comprehensive documentation generation. The survey examines the technological advancements, challenges, and potential applications of this approach in software engineering.**

***Index Terms*—LLM, Automation, Modern codebases, GPT 4,Context-aware documentation, Multi-language support, Intelli gent Code Chunking**

I. INTRODUCTION

In software development, documentation plays a critical role in ensuring code maintainability, team collaboration, and future extensibility. However, manual documentation is often tedious, time-consuming, and prone to human error. Auto mated documentation generation tools have been introduced to address these issues, but traditional systems still struggle to handle large, modular codebases and interdependent code files, especially when spanning multiple programming languages. Recent advancements in large language models (LLMs) have demonstrated potential in improving documentation generation by leveraging natural language processing (NLP) and machine learning techniques to understand code semantics. Despite these improvements, several challenges persist:

1) Handling Large Code Files: LLMs are constrained by token limits, making it difficult to generate documentation for large functions or classes.

2) Managing Module Imports and Dependencies: Many systems rely on internal and external module imports, and the documentation of the main code often depends on these modules. Generating documentation without resolving dependencies can lead to incomplete or fragmented outputs.

3) Multi-Language Codebases: Modern software projects are often written in multiple programming languages, requiring documentation tools to adapt to the syntax and structure of each language.

This survey paper presents a review of the current literature on these topics and introduces a novel system that addresses these challenges using intelligent code chunking, dependency resolution, and multi-language support.

II. LITERATURE REVIEW

A. Automated Software Documentation Generation

Automating the process of generating documentation for software systems has been a long-standing research topic in the field of software engineering. Early work in this area focused on creating simple code comments and basic summaries by parsing programming languages. [1]McBurney and McMillan (2014) introduced a technique for automatically generating descriptive comments for Java methods by analyzing method signatures, control flow, and naming conventions. Their work was an important step toward bridging the gap between code semantics and documentation but it was only limited to the specific languages and predefined templates. [2]Nazar, Hu, and Shihab (2016) explored the use of heuristic-based methods to generate documentation by mining software repositories. They emphasized the importance of linking code changes with documentation updates but recognized the limitations of static analysis tools in generating dynamic and context-aware documentation for large, modular systems.

B. Large Language Models for Code Understanding

With the emergence of large language models like GPT 3, GPT-4, and Codex, the field of code understanding and documentation generation has seen rapid advancements. LLMs have proven to be particularly effective in interpreting programming languages, generating human-like text, and producing detailed documentation. [3]Chen et al. (2021) introduced Codex, an LLM fine-tuned on large-scale code repositories, which demonstrated the ability to generate high-quality documentation and even assist in code completion tasks. Codex was specifically trained to understand both the syntax and semantics of various programming languages, making it a powerful tool for generating contextually relevant documentation across diverse languages. [4]Feng et al. (2020) developed CodeBERT, a model trained on paired code and natural language data to improve code understanding and documentation tasks. The authors showed that CodeBERT could accurately generate documentation for both small and large code snippets, though it still faced challenges when dealing with large functions or complex module dependencies.

C. Code Chunking for Large Codebases

As large language models have a limited context window (typically between 4,000 and 32,000 tokens), handling large codebases, where individual files or functions may exceed these limits, becomes a critical challenge. Recent literature emphasizes the importance of intelligent code chunking to split large code segments into manageable chunks while preserving context [5]Allamanis et al. (2018) proposed a method for automatically summarizing large code functions by splitting them into smaller, more digestible pieces and then using neural models to generate summaries. Their approach to hierarchical summarization highlighted the need for maintaining context across multiple chunks to ensure a coherent overall documentation. [6]Liu et al. (2020) developed techniques for using hierarchical summarization in combination with LLMs to provide high-level summaries of large codebases by generating lower-level summaries for each chunk of code and then aggregating them.

D. Dependency Resolution and Module Imports

[7]Beller, Bacchelli, and Zaidman (2014) analyzed the complexities introduced by dependencies in software systems and discussed how dependencies between modules, libraries, and frameworks make it difficult to maintain accurate and coherent documentation. They argued for tools that can automatically detect dependencies and process them in a correct sequence. [8]Zimmermann et al. (2004) proposed dependency tracking mechanisms that automatically map inter-module relationships within a project. They found that by analyzing module imports and external library usage, documentation tools could provide a more complete understanding of a system's architecture and behavior.

E. Multi-Language Support in Documentation Tools

Modern software projects often involve multiple programming languages, each with its own syntax, structure, and dependencies. [9]Rahman, Roy, and Adams (2019) highlighted the challenges of building documentation systems that support multiple languages and proposed the use of abstract syntax trees (ASTs) and regex-based parsers to handle language specific constructs. [10]Wang et al. (2021) discussed the complexities involved in developing language-agnostic LLM models that support documentation generation across languages like Python, C++, Java, and JavaScript. They showed that language-specific variations in syntax and control structures could be effectively handled by combining general NLP models with language specific parsers, a key design decision in the current project.

F. Attention Is All You Need

[11]Vaswani et al. (2017) introduced the Transformer architecture which revolutionized the field of natural language processing and has become a foundational model for large language models (LLMs) in various applications, including code summarization and documentation. Central to the Trans former is the attention mechanism, which enables the model to weigh the importance of different tokens in a sequence dynamically. This mechanism allows the Transformer to effectively handle long sequences of input, unlike traditional RNN based approaches that struggled with retaining information across lengthy spans. By capturing contextual dependencies within and across sentences, the attention mechanism makes Transformers highly suited for tasks like code summarization and documentation generation, where understanding relationships and dependencies within code snippets is essential. This architectural breakthrough paved the way for models that generate meaningful and coherent documentation from complex codebases.

G. CodexGLUE - A Benchmark Dataset for Code Understanding and Generation

[12]Lu et al. (2021) introduced CodexGLUE, a benchmark specifically designed to evaluate the effectiveness of models on code understanding and generation tasks. CodexGLUE consists of various datasets and evaluation tasks tailored for programming-related tasks such as code summarization, code completion, and bug detection. With the introduction of CodexGLUE, the research community gained a standardized means to assess model capabilities in understanding and generating code. This benchmark has driven advancements in model performance for code documentation, as it enables the fine-tuning of models on complex programming tasks and assists in generating high-quality, contextually relevant documentation. The dataset has encouraged the development of models that better understand code semantics, structure, and intent, making it a crucial addition to the literature on automated code documentation.

H. Code2Vec- Learning Distributed Representations of Code

[13]Chen et al. (2019) proposed the Code2Vec model, a neural network architecture designed to learn distributed vector representations of code snippets. Code2Vec captures the semantics of code by representing it in the form of "code embeddings," which allow for more accurate code summarization and automatic documentation generation. By encoding the structural paths within the abstract syntax tree (AST) of a code snippet, Code2Vec generates embeddings that encapsulate both the syntax and semantics of code elements. This capability enables models to provide concise yet informative summaries that enhance the quality of documentation generated for code snippets. Code2Vec's approach of embedding code semantics has influenced numerous advancements in models designed for code understanding and has established an essential method ology for creating documentation that is both contextually aware and semantically accurate. This model is an integral part of ongoing research efforts aimed at improving automatic documentation quality through embedding-based techniques.

III. PROPOSED SOLUTION

The intelligent documentation generation system proposed in this survey paper combines intelligent code chunking, module dependency resolution, and multi-language support to address the challenges identified in the literature. This system enables accurate and context-aware documentation generation for large and complex codebases by overcoming the limitations of LLM token windows and managing dependencies between files.

A. Code Chunking

The system splits large files into smaller, manageable chunks based on logical boundaries such as function definitions, loops, and methods. These chunks are processed individually by an LLM to generate documentation, and a hierarchical summarization approach is used to aggregate lower-level summaries into a high-level overview of the entire file. This ensures that even the largest code files are documented accurately.

B. Module Dependency Resolution

By analyzing import statements and module dependencies, the system determines the correct order in which to process and document files. Imported modules are documented first, ensuring that when the main code file is documented, it references already-generated documentation for its dependencies, leading to more accurate and comprehensive outputs.

C. Multi-Language Support

The system supports multiple programming languages, including Python, C++, Java, and JavaScript. By using language specific parsing tools such as ASTs, the system can adapt to the syntax and structure of different languages, ensuring that the generated documentation is both syntactically correct and contextually relevant.

IV. CHALLENGES AND LIMITATIONS

Despite its advantages, the proposed system faces several challenges:

1. LLMToken Limitations: While code chunking helps mitigate token window limits, there may still be edge cases where very large functions or deeply nested structures challenge the system's ability to preserve context.
2. Dependency Resolution Complexity: The system relies on accurate dependency resolution. Cyclic dependencies or external modules that are not properly linked may result in incomplete documentation.
3. Performance Overheads: Processing large codebases with complex dependencies and chunking

could introduce performance bottlenecks, especially when integrated into continuous deployment pipelines.

V. CONCLUSION

This survey paper reviews the current state of research in automated documentation generation and highlights key advancements in code chunking, module dependency resolution, and multi-language support. The proposed system addresses critical challenges in generating context-aware documentation for large and complex codebases, ensuring that the documentation accurately reflects the structure and relationships within the code. While there are still challenges to overcome, such as performance optimization and handling complex dependencies , the system provides a promising solution to the longstanding issue of automated documentation in modern software engineering.

REFERENCES

[1] Paul W. McBurney. 2015. Automatic documentation generation via source code summarization. In Proceedings of the 37th International Conference on Software Engineering- Volume 2 (ICSE '15). IEEE Press, 903–906.

[2] Nazar, M., Hu, A., and Shihab, E. (2016). Mining software repositories for automated documentation. Journal of Software Engineering.

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

[4] Feng, Z., et al. (2020). CodeBERT: A pre-trained model for program ming and natural languages. arXiv preprint arXiv:2002.08155.

[5] Allamanis, M., et al. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys.

[6] Liu, Y., et al. (2020). Hierarchical summarization for programming languages. Proceedings of the Conference on Empirical Methods in Natural Language Processing.

[7] Beller, M., Bacchelli, A., and Zaidman, A. (2014). On the impact of test suite metrics on automated documentation. International Conference on Software Testing, Verification and Validation.

[8] Zimmermann, T., et al. (2004). Mining version histories to guide software changes. Proceedings of the International Conference on Software Engineering.

[9] Rahman, F., Roy, C. K., and Adams, B. (2019). Multi-language software documentation generation. IEEE Transactions on Software Engineering.

[10] Wang, Y., et al. (2021). Language-agnostic code analysis with LLMs. Journal of Software Engineering.

[11] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. arXiv:1706.03762

[12] Lu, S., Chowdhury, M. F. M., Huang, P., White, M., & Tian, Y. (2021). CodexGLUE: A benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664.

[13] Chen, U., Liu, Y., & Song, Y. (2019). Code2Vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL), 1–29.

[14] Zhang, X., Hou, X., Qiao, X. et al. A review of automatic source code summarization. Empir Software Eng 29, 162 (2024). https://doi.org/10.1007/s10664-024-10553-6

[15] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 220, 1–13. https://doi.org/10.1145/3597503.3639183

[16] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 4998–5007, Online. Association for Computational Linguistics.

[17] N. J. Abid, N. Dragan, M. L. Collard and J. I. Maletic, "Using stereotypes in the automatic generation of natural language summaries for C++ methods," 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 2015, pp. 561 565, doi: 10.1109/ICSM.2015.7332514.