AutoDoc AI

Pranav Prabhu
ppranav02@vt.edu

1 Problem statement and analysis

In today's fast-paced tech environment, the drive to meet tight deadlines often leads developers to view documentation as an afterthought. This results in either rushed or, worse, incomplete documentation, which later manifests as technical debt—an expensive and time-consuming issue for companies to address. New team members, or even returning developers, face the daunting task of deciphering code without proper guidance, slowing down integration and progress. The consequences of this are twofold: firstly, we see a significant drop in productivity as developers spend additional time trying to understand poorly documented code rather than writing new features or fixing bugs. Secondly, this lack of clarity creates an accessibility barrier, making it difficult for a wider pool of developers to contribute to the code, especially in open-source projects.

Here's where AutoDoc AI enters as a groundbreaking solution. By harnessing the power of artificial intelligence, AutoDoc AI is uniquely designed to address these challenges head-on. It enables the automatic generation of thorough and accurate documentation, thereby increasing the productivity of current team members and easing the onboarding of future ones. This project aims to address the challenge of generating comprehensive, accurate, and easily understandable documentation for both in-code comments and external documentation websites. The primary target audience includes software development teams, open-source contributors, and companies seeking to enhance their software maintenance and developer onboarding processes.

2 Use-Case scenarios

2.1 Compliance and Regulatory Requirements

This project aims to simplify the documentation process for compliance and regulatory requirements in software development. By automatically generating comprehensive documentation from the code base, it ensures that developers follow industry-specific regulations. This streamlines workflows reduces non-compliance risks, and facilitates audits, benefiting software development teams and regulatory compliance efforts alike.

2.2 Education and Training

This project provides a valuable tool for onboarding new developers and enhancing their understanding of code bases. By automatically generating comprehensive documentation directly from the code, it offers a clear and structured overview of the software architecture, design patterns, and best practices. This enables efficient knowledge transfer and facilitates learning for developers joining the team. Additionally, the generated documentation can serve as educational material for software development courses or workshops, helping students grasp complex concepts and practical implementation details more effectively.

2.3 Streamlining API Development

This project streamlines API development by automating the documentation process, ensuring that developers have comprehensive and up-to-date information about the APIs they are working with. By

extracting relevant details directly from the code base, such as endpoint routes, request parameters, response formats, and authentication requirements, it generates clear and accurate documentation for APIs. This documentation serves as a reference for developers, enabling them to understand the functionality of each API endpoint and how to interact with it effectively. By eliminating the need for manual documentation efforts, the project accelerates the API development process, reduces errors, and enhances collaboration among development teams.

2.4 Open Source Projects

This initiative provides a crucial solution by automating documentation processes. By extracting key details directly from the code base, it generates comprehensive documentation that is accessible to developers and contributors. This streamlines onboarding processes for new contributors, enhances collaboration within the community, and ensures that the project remains well-documented and easily understandable for all stakeholders. Additionally, by simplifying the documentation effort, it encourages broader participation and adoption of the open-source project, ultimately contributing to its success and sustainability.

2.5 Quality Assurance

This project revolutionizes documentation practices by automating the process. By extracting pertinent details from the codebase, it generates comprehensive documentation, ensuring that QA teams have a thorough understanding of the software's functionalities, interfaces, and dependencies. This facilitates more efficient testing processes, allowing QA professionals to focus on identifying and addressing potential issues rather than spending time deciphering complex code. Additionally, the automated documentation ensures consistency and accuracy, enabling QA teams to maintain high standards of quality throughout the software development lifecycle.

3 Literature Review

In my comprehensive review of the literature concerning automated code documentation, I encountered several pioneering studies that have shaped my understanding and approach to the development of the AutoDoc project. My exploration began with the research presented by Junaed Younus Khan and Gias Uddin, which utilized Codex, a GPT-3 based model, to automate the creation of code documentation. Their work, "Automatic Code Documentation Generation Using GPT-3," demonstrated significant improvements over previous state-of-the-art techniques, particularly in the ability of Codex to adapt to various programming languages with minimal examples. This study was instrumental in confirming my belief in the potential of language models to understand and document code effectively, guiding my decision to integrate similar models in AutoDoc.

Additionally, I delved into Tyler Thomas Procko and Steve Collins' research on "Automatic Code Documentation with Syntax Trees and GPT," which integrates abstract syntax trees (AST) with GPT models. Their approach highlighted the importance of combining structured code analysis with natural language generation capabilities to enhance the automation of documentation within the software development lifecycle (SDLC). Their insights into the challenges of maintaining documentation relevancy as code evolves informed my methodology, particularly in the design of my agents that analyze and document dynamic code changes seamlessly.

My literature review further extended to the work described in "CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing" by Ahmed Elnaggar et al. This study introduced us to CodeTrans, an encoder-decoder transformer model tailored for multiple software engineering tasks. The versatility of CodeTrans in handling tasks such as code documentation generation, source code summarization, and more, using various training strategies, provided a broad perspective on the capabilities of transformer models in software engineering. Their success in employing multi-task learning strategies and achieving state-of-the-art results across numerous tasks inspired us to consider similar approaches in training the models, ensuring robust performance across diverse documentation needs.

From these studies, I synthesized that the integration of deep learning techniques with traditional code analysis tools (like syntax trees) and the strategic training of models on specific tasks (as seen in CodeTrans) could significantly elevate the effectiveness of automated code documentation

tools. Moreover, the emphasis on readability and informativeness in documentation, as discussed in these papers, reinforced my commitment to producing not only accurate but also user-friendly documentation that adheres to best practices in software development.

Overall, my literature review underscored the transformative potential of AI in automating code documentation, which has historically been a tedious yet crucial part of software development. These insights have profoundly influenced the design and functionality of AutoDoc, ensuring that it not only automates documentation but also enriches the development process with timely, accurate, and useful documentation outputs.

4 AI Applications and Model

AutoDoc is a multi-agent system that can accurately scan a codebase, understand its structure and functionalities, and generate initial versions of both in-code documentation and web-based documentation. Large-language models (LLMs) will be the driving force for the agents, as their emergence has made it significantly easier to achieve these goals in a short period of time. From this, we can say my solution falls under three main applications of AI:

- **Agents:** AI agents are designed to reason, make informed decisions, and determine a sequence of actions to take. AutoDoc's agents will be driven by large-language models to further assist in the decision-making process.
- **Natural Language Processing (NLP):** NLP enables machines to **process** and understand human **natural language**, whether that be in the form of text or speech. Introducing large-language models will not only allow the agents to understand any natural language text queries thrown its way, but also programming languages.
- **Natural Language Generation (NLG):** Similarly to NLP, NLG enables machines to **generate** human **natural language**, whether that be in the form of text or speech. AutoDoc's LLM-powered agents should be able to convert the given query into a readable, written response, whether it be a new query for another agent or the actual code documentation itself.

With the emergence of large-language models, there has been concurrent development of technologies aimed for working with them. These technologies provide a range of tools and frameworks designed to enhance the capabilities and applications of LLMs across various domains. Here are the key technologies I used to bring AutoDoc to life:

- **LangChain:** LangChain is one of the more popular frameworks for developing LLM-driven applications. For my use case, it will allow me to interact with different large-language models, as well as create custom agents and tools.
- **LangGraph:** LangGraph is built on top of LangChain, providing the ability to coordinate multiple agents across multiple steps of action in a cyclic manner. Since I need multiple agents for parsing, editing function declarations, writing documentation, etc., this will be extremely useful. Also, LangGraph allows you to visualize the "graphs" you create.
- **Ollama:** While OpenAI's GPT models produce impressive results, the API can be costly. To take account of this, I decided to use Ollama, which is one of the best open-source LLMs that's up to par with GPT. Specifically, I used the LLaMa 3 model.
- **Tree-Sitter:** Tree-sitter is an open-source library designed to efficiently parse code or other structured data. It provides accurate syntax highlighting, code analysis, and other language-related tasks. This will be incorporated as a LangChain custom tool, making it possible for the LLM-driven agents to interpret programming languages.

Now, let's put it all together. Look at Figure 1, which is a LangGraph visualization that depicts one full iteration of the AutoDoc model.

The provided LangGraph visualization in Figure 1 encapsulates a cohesive workflow that demonstrates the sophisticated orchestration of AutoDoc's various components. The graph illustrates the cyclical nature of the process, beginning with the **_start_** node, symbolizing the initiation of the code analysis. This initiation leads to the **read_file** node, which denotes the agent responsible for ingesting the

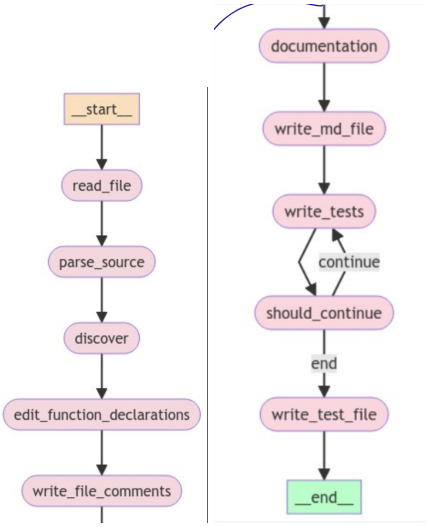

Figure 1: Graph visualization provided by LangGraph.

source files. Following this, the **parse_source** node indicates the utilization of a parsing agent, likely employing a tool such as Tree-Sitter, to dissect and understand the Swift programming language's syntax and structure.

The **discover** node that follows suggests a phase of identification and analysis, where the system likely maps out the functions, variables, and classes within the codebase. The transition from discovery to the **edit_function_declarations** node implies that the system is not merely passive in its documentation but actively refines code by editing function declarations for clarity or completeness. This step is crucial as it demonstrates the system's ability to not only interpret but also to enhance the codebase it documents.

Post-editing, the process moves towards documentation with the **write_file_comments** node, indicating the agent's role in generating in-line documentation that resides within the code itself, providing immediate context to developers as they interact with the codebase. Subsequently, the process advances to the **documentation node**, which represents a more expansive documentation effort, culminating in the creation of Markdown documents as described by the **write_md_file** node.

The graph then presents a decision-making node, **should_continue**, illustrating the system's dynamic capability to assess whether additional iterations or actions are necessary. If the system determines the need for further documentation, it proceeds to **write_tests**, suggesting an additional layer of automated test generation, a critical component of modern software development. The finality of the cycle is denoted by the end node, and the workflow completes with **write_test_file**, indicating the storage or deployment of generated test cases.

This visualization is a testament to the complex yet streamlined process AutoDoc employs to tackle the multifaceted challenges of code documentation. By leveraging LLMs within a well-coordinated multi-agent system, the AutoDoc project ensures a comprehensive documentation lifecycle that spans from initial parsing to the final generation of test documentation. Each node or agent is meticulously designed to fulfill a specific role within the broader system, culminating in a robust, self-sustaining model for continuous code analysis and documentation generation. The cyclical nature of the LangGraph visualization underscores the iterative approach of AutoDoc, emphasizing its capacity to refine and update documentation as the codebase evolves, thus maintaining the relevancy and accuracy of the documentation it produces.

5 Results and Demonstration

AutoDoc can currently understand and process three programming languages: (1) Python, (2) JavaScript, and (3) Swift. I specifically chose to build support for these languages first because they're easier to work with and are considered some of the most popular languages in the world of software

development. With this knowledge, AutoDoc can then generate in-code and test case documentation within a user's codebase. Furthermore, a general overview of an entire codebase can be generated and written inside a Markdown file.

As of now, the system is run from the command line.

5.1 Demonstration

The demonstration of AutoDoc involves a detailed walkthrough of its capabilities via command line interfaces. To illustrate, we run the AutoDoc system on a sample project to showcase how effectively it parses and documents the code. Here is a step-by-step guide on how it works:

1. First, the user pulls the necessary resources and initializes the AutoDoc environment using the commands detailed in the installation guide.
2. Next, the user inputs the command to process a given file. For example:

```
python autodoc.py path/to/your/file.py
```

3. AutoDoc then reads the file, parses the code, and applies its documentation algorithm to generate comments directly within the code and an overall codebase summary in a separate Markdown file.

5.2 Results

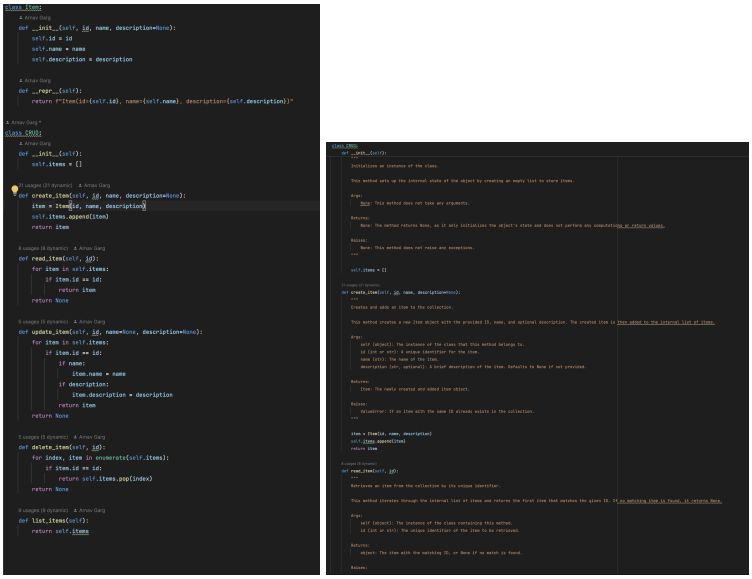

Figure 2: Original code with commented code documentation.

6 Code and Documentation

The AutoDoc project is structured to efficiently harness the capabilities of advanced language models, ensuring robust automated documentation and code analysis. At the heart of the project lies a modular architecture composed of several specialized agents, each designed to handle distinct aspects of the documentation and code analysis processes. These agents—such as the *ExtractEntrypointsAgent*, *DocumentationAgent*, and *WalkCallStackAgent*—are crafted to work in tandem, orchestrating a seamless flow of data and tasks. Each agent utilizes templates and large language models (LLMs) to perform specific functions, such as extracting API endpoints, generating documentation, and walking through call stacks to gather user function details.

The source code repository is structured to facilitate easy navigation and modification, with a clear separation of concerns evident in the organization of files and directories. Key components like the agents.py file define the core functionalities and interactions of different agents, leveraging the *langchain.llms* and *langchain.prompts* for integrating LLM capabilities. On the other hand, the autodoc.py file encapsulates the initialization and orchestration of the overall system, interfacing with technologies such as HuggingFaceInstructEmbeddings and Chroma for embedding and retrieval operations, respectively. Documentation is meticulously maintained alongside the code, with comprehensive comments and markdown files detailing usage, parameters, and architectural design, aiding in clarity and ease of use for both internal developers and external contributors. This thorough documentation strategy not only supports current project needs but also ensures scalability and adaptability to future enhancements and integrations.

7 Lessons learned

The development and implementation of the AutoDoc project provided several invaluable lessons that are instrumental for the future of automated documentation tools and AI-driven software development processes. One of the most critical insights gained is the importance of integrating advanced Natural Language Processing (NLP) and Natural Language Generation (NLG) techniques to handle the complexities of diverse programming languages and their nuances. Building AutoDoc revealed that while NLP can effectively understand the context and semantics within a codebase, NLG's capability to generate coherent, accurate, and human-like documentation is pivotal. This dual application not only streamlined the documentation process but also highlighted the challenge of maintaining linguistic consistency across different types of documentation, from in-code comments to comprehensive API guides.

Throughout the project, I encountered and overcame challenges related to the adaptability of AI models to understand less common or newly emerging programming constructs. This necessitated continuous training and refinement of the models, using a mix of supervised and unsupervised learning techniques to improve their understanding and output quality. The iterative testing and feedback loops with end-users, primarily developers and QA professionals, provided critical insights into user needs and expectations. This user-centric approach helped refine the AI's output to be more aligned with practical, real-world application, rather than just theoretical accuracy.

Moreover, the use of open-source tools such as Tree-Sitter and Ollama proved essential in reducing costs and fostering community engagement. Engaging with the open-source community not only enhanced the tool's capabilities through collaborative development but also ensured that AutoDoc remained accessible to a broader range of developers, including those in academic and non-profit sectors. However, relying on these tools also taught us the lesson of dependency management and the need for robust error handling to anticipate and mitigate potential disruptions arising from external updates or changes in these platforms. These lessons will guide future enhancements, ensuring AutoDoc evolves into a more resilient and user-focused tool that continues to meet the dynamic demands of the software development industry.

8 Future work

In my upcoming initiatives, I aim to enhance the project's reach by adding support for more programming languages, ensuring broader applicability across diverse software environments. I will prioritize refining the documentation generation process for each language, striving for clarity and precision tailored to individual syntaxes and conventions. Seamless integration with prevalent development environments and version control systems (VCS) is a key objective, facilitating effortless documentation updates within developers' toolsets. Additionally, user customization features will empower developers to tailor documentation templates to their project's unique needs. Real-time documentation updates synchronized with code changes, along with automated versioning and change logs, will maintain relevance and transparency across development teams. These efforts seek to advance the project's utility and accessibility for a wider community of developers.

References

[1] Khan, Junaed Younus, and Gias Gias Uddin. “Automatic Code Documentation Generation Using GPT-3.” Arxiv, 10 Oct. 2022, https://arxiv.org/pdf/2209.02235.pdf.

[2] Procko, Tyler Thomas, and Steve Collins. “Automatic Code Documentation with Syntax Trees and GPT: Alleviating Software Development’s Most Redundant Task.” ResearchGate, Jan. 2023, https://www.researchgate.net/publication/373911060_Automatic_Code_Documentation_with_Syntax_Trees_and_GPT_Alleviating_Software_DevelopmentâĂŹs_Most_Redundant_Task.

[3] Elnaggar, Ahmed, et al. “CodeTrans: Towards Cracking the Language of Silicon’s Code Through Self-Supervised Deep Learning and High Performance Computing.” Edited by Leslie Pack Kaelbling, Arxiv, Apr. 2021, https://arxiv.org/pdf/2104.02443v2.