Automated DocString Generation with Large Language Models

Malte Tenkamp and Christian Zwiessler
(Student IDs: 7431298, 7431669)

University of Cologne, 50923 Cologne, Germany
{mtenkamp,czwiessl}@smail.uni-koeln.de
https://github.com/czwiessler/sq-llm-docstring-gen-kt

Abstract. In software development, adequate comments are essential to ensure the readability and maintainability of source code. However, writing such documentation is often neglected as it requires time and incurs costs. By leveraging Large Language Models (LLMs), missing docstrings can be automatically generated, and existing ones can be reviewed for accuracy and improved if necessary. This report focuses on the development of an end-to-end system that processes Python files as input, generates missing docstrings, and reviews and edits existing ones. It examines which LLM models are currently market leaders, their suitability for this use case, and how they can be integrated. Additionally, previous approaches before the emergence of modern LLMs, such as GPT-3.5 and its successors, are analyzed. Furthermore, potential evaluation methods for assessing the quality of the generated docstrings are discussed.

1 Introduction

In software development, docstrings are recognized as essential for improving code comprehension and maintainability. They act as a bridge between code and its intended functionality, allowing developers to better understand, modify, and extend systems. However, the creation and upkeep of high-quality docstrings are often neglected, primarily due to the significant time and effort required. This neglect frequently results in poorly documented codebases, leading to increased costs and delays during debugging, scaling, or integrating new team members.
Historically, developers primarily relied on manual documentation or basic rule-based tools, which were often tedious and prone to inconsistency.
With advancements in LLMs, new opportunities for automated docstring generation have emerged. These models have shown the capability to analyze and summarize code effectively, producing documentation that often matches or even surpasses manually written counterparts in quality and relevance. Building upon this potential, a system for automated docstring generation has been developed, enabling Python source code to be enhanced with precise and contextually appropriate documentation.
This report focuses on the methodology, architecture, results, and limitations of this development and implementation process. The implemented system is

designed to accept Python files as input and returns the scripts with detailed docstrings for classes, methods, and functions. By addressing the challenges associated with manual documentation, this work highlights how LLMs-based solutions can contribute to improving software development practices. Further, this report also comprises a theoretical approach toward the automated evaluation of docstrings, including how it is done currently, what the limitations are, and how the proposed approach could tackle these limitations.

2 History and State of the Art of Automated Source Code Annotation

2.1 Pre-LLM Approaches

To gain a comprehensive overview of the field of automated docstring generation, a first step is to examine the evolution it has undergone in the past decades. Since the annotation of Python docstrings are just one specific materialization of the broader topic of automated annotation generation, the following backward-directed literature analysis also comprises non-docstring forms of code annotation and programming languages other than Python.

Early Rule-Based Systems
The origins of automatic code annotation trace back to manual and semi-automated tools like Doxygen, introduced in the late 1990s [4]. These systems required developers to write specially formatted comments within the source code, which the tools would then extract and format into structured documentation. While these tools standardized documentation formats and made it easier to maintain, the burden of writing the actual content still lay with the developers. Their functionality was largely static, relying on predefined templates and syntax, with no inherent capability to generate new content from the code's logic or structure.

Static Analysis and Rule-Based Approaches
A shift towards static, rule-based approaches emerged in the early 2010s, focusing on reducing the manual workload. One example is the Rulebook Generator (RbG) introduced in 2016 by Moser and Pichler [10], a system designed for scientific software that consists of static code analysis to extract documentation directly from annotated *C++* source code. Developers would add structured comments using specific tags (e.g., @symbol, @substitute, @suppress), allowing RbG to identify relevant code segments for documentation. The tool could analyze mathematical formulas, control structures, and generate visual elements like function plots and flow diagrams. This marked a significant step towards context-aware documentation, but the approach still relied heavily on initial developer annotations and faced challenges in adoption due to workflow disruptions. Furthermore, the context the corresponding code was surrounded by was not yet taken into account during the translation process.

Incorporating Context and Structural Information
The focus on generating documentation that explains not only what code does but also why it exists became prominent in 2014 with McBurney and McMillan's paper [9]. Their approach combined call graph analysis, PageRank ranking, and linguistic models like the Software Word Usage Model (SWUM) to generate context-rich, natural language summaries of Java methods. Instead of summarizing a method's internal logic, their technique examined how methods interacted with each other within the software system. By identifying the most important methods and extracting keywords through SWUM, their system could produce summaries that explained a method's role in the broader codebase. The final natural language summaries were generated using the SimpleNLG library. This approach represented a significant step toward contextual and structured code summarization.

Neural Networks and Attention
A novel approach came with the application of Neural Machine Translation (NMT) techniques to source code annotation tasks. In 2017, Barone and Sennrich [1] treated docstring generation as a translation problem, where the source language is some programming language and the target language is natural language. They provided an extensive parallel corpus of over 150,000 Python functions and their corresponding docstrings as a basis for model training. For their approach, they employed a Sequence-to-Sequence (Seq2Seq) architecture with a bidirectional Recurrent Neural Network (RNN) and the then newly developed attention mechanism [14]. The encoder processed code tokens, while the decoder generated corresponding docstrings word-by-word, focusing on the most relevant parts of the code using attention. This method marked the first substantial shift towards end-to-end automated documentation using deep learning and overcoming methods of limited complexity such as static rules and manual annotations.

Long Short-Term Memory-based Approaches
Building on neural network-based methods, Yao et al. (2019) [16] introduced a novel approach combining Reinforcement Learning (RL) with bi-directional LSTMs for documentation generation. Their technique, known as Advantage Actor-Critic (A2C), indirectly optimized code annotations by focusing on improving code retrievability. The model was trained to generate annotations that enhanced the performance of code search tasks, rather than merely mimicking human-written docstrings. The encoder-decoder structure with attention mechanisms processed code and generated natural language annotations, while the RL component rewarded the system for generating annotations that led to higher retrieval accuracy.

Since the late 1990s, the evolution of automatic code annotation has progressed from static rule-based systems to sophisticated machine learning and reinforcement learning models. Over time, new and better possibilities of including code

context and minimizing documentation workload for software engineers were explored. In recent years, transformer and LLM-based approaches became overwhelmingly dominant in this field. In the rest of this section, it will be examined how this development came about and what technologies are used exactly.

2.2 State of the Art: Transformer and LLM-based Methods

The current state of automated source code documentation is characterized by the dominance of deep learning and transformer-based architectures, with a strong shift towards leveraging LLMs for more comprehensive documentation generation.

Domination of Transformer-based Architectures
According to Zhu et al.(2024) [18], transformer-based models have emerged as top performers, particularly in function-level and class-level documentation tasks. These models excel at handling the structural complexity of code, effectively capturing method signatures and contextual cues. Exposing the paradigm change apparent in this field, the researchers also found that transformers significantly outperform RNN-based models (e.g., ast-attendgru, code2seq) due to their superior ability to manage long-range dependencies and complex code structures, as evidenced by higher ROUGE-L and METEOR scores. Closed-source LLMs like GPT-3.5 and GPT-4 have further raised the bar, consistently outperforming human-written documentation in metrics such as accuracy, completeness, and readability [2]. However, these models still encounter challenges in module-level documentation, where the need to synthesize larger contexts becomes more demanding.

Other Technologies are almost fully Displaced
Rai et al. (2022) [12] emphasize the stark decline of traditional template-based and IR-based approaches as described in subsection 2.1 in favor of deep learning techniques, which have proven more effective in generating high-quality, context-aware documentation. The research also notes a predominance of function-level documentation due to its relative simplicity compared to class-level summaries, which demand more nuanced understanding of code relationships and design patterns. To cite a contrasting example to the ongoing technological shift, Venkatkrishna et al. (2023) [15] explored a modular approach, using task-specific pre-trained models to address different components of the annotation process, such as parameter descriptions and datatype identification. While their study explicitly excludes LLMs and is thus out of the scope of this paper, it highlights the potential of combining specialized models for a holistic documentation solution beyond transformer-based architectures.

3 Method

In order to achieve the best possible result given the context of the problem description, we followed the principles of the Peffers Design Science Research

(DSR) methodology [6] to guide the development of the docstring generation tool. The DSR methodology emphasizes an iterative approach to solving design problems through various steps such as problem identification, defining of solution objectives, and design and development. Accordingly, as the problem to be approached is defined in the given problem statement, our process began with deriving development objectives (requirements). We then iteratively built and refined the tool, evaluating its effectiveness at each stage. By informing our design with literature findings, we ensured that our design choices were grounded in current research and practice. The iterative process design enabled us to explore the topic and directly implement learnings made during the journey, ultimately resulting in a more effective tool for Python developers.

3.1 Requirements Engineering

To develop a robust system for automated docstring generation using LLMs, it was essential to establish a clear set of requirements. These requirements form the foundation for our implementation and serve as a benchmark for evaluating the system's effectiveness. The process of defining these requirements involved analyzing the problem statement, considering state-of-the-art approaches, and incorporating insights from related literature.

Functional requirements specify the core capabilities that the system must support to ensure effective docstring generation. Based on our initial analysis, we identified the following key functional requirements:

- The system must accept a Python script as input and process its class and function definitions.
- It should generate docstrings in *Google Style* format that satisfy the principle requirements of completeness and structure.
- The generated docstrings should be inserted directly into the original code while maintaining the script's executability.
- The system must support structured and nested annotations, ensuring that class-level and function-level docstrings are properly associated.
- The system will generate only class- and function-level docstrings, explicitly omitting module-level docstrings. Module-level docstrings often require contextual information about the module's purpose, domain, and metadata such as authorship and licensing, which are not directly derivable from the code or script itself.

Non-functional requirements address broader system constraints that influence performance, usability, and reliability. These requirements are equally crucial for ensuring the system's usability and efficiency:

- The integrity of the original code must not be compromised. The insertion of docstrings should not alter the logic or functionality of the script. Important syntax elements, including indentations and spacing, decorators, or import statements must be retained exactly as-is.

- Context-awareness is also a desirable capability of the tool. A context-aware documentation includes not only what a function takes as input, how it processes that input and what it returns, but also which position these steps have within a broader context of the class or script. To this end, other contents of the corresponding the script that each function is contained in should be considered.
- Computational costs, particularly those associated with API usage and model execution, should be minimized to not compromise system scalability.

These requirements acted as a basis to the architectural decisions and implementation strategy. By structuring the system around these specifications, it was ensured that the approach addressed key challenges in automated code annotation. Furthermore, these requirements provided a baseline against which we could measure improvements and evaluate system performance.

3.2 First Approach: Initial Implementation and Design Decisions

The initial version of the docstring generator was designed as a proof of concept to explore the feasibility of using LLMs for automated code documentation. The primary objective was to develop a pipeline capable of analyzing source code, extracting relevant structural elements, tasking the API with the generation of annotations and integrating generated docstrings without human intervention.

Technology Choices
The prototype was developed in Python, leveraging its extensive ecosystem for Natural Language Processing (NLP) and static code analysis. Further, the expected low computational intensity of the extraction and prompting components of the tool gave no reason to resort to a more efficient programming language such as *C++*. The decision to integrate an external API rather than training an in-house model was driven by the need for rapid iteration and high-quality results without significant computational overhead. Another consideration was that in a dynamically evolving field like NLP, it seemed wiser to not tie the tool to one specific language model. Rather, providing a model-indifferent architecture that can easily be adapted to using other models and thus continue to capitalize on the ongoing development seemed to embody a much more foresighted approach. OpenAI's API was chosen for its advanced language modeling capabilities, particularly in understanding and generating Python code documentation. As discussed in section 2.2, GPT-3.5 and GPT-4 were found to be the most effective in generating structured and contextually appropriate docstrings.

Architectural Considerations
The system architecture was designed to transform raw source code into annotated documentation while maintaining structural integrity. The following core principles influenced the initial development:

- **Modularity**: Each step of the processing pipeline was developed as an independent function, enabling flexibility in testing and iterative improvements.
- **Structural Preservation:** The prompt was designed to instruct the model to retain the original indentation and formatting so that the generated annotations would seamlessly integrate into the source code.
- **Automation**: The process aimed to minimize human intervention by designing a pipeline that could automatically analyze, clean, annotate, and reintegrate code.

System Components and Processing Flow
The initial implementation consisted of the following key components, each handling a specific phase of the annotation process:

1. **Docstring Generation:** OpenAI's GPT model was used to generate docstrings based on the whole script as input. By not removing old docstrings beforehand, this architecture ensured that the LLMs could revise existing docstrings and replace or complete if deemed necessary. Also, providing the whole script would ensure context-aware docstring generation. As a return, the model was instructed to provide an annotated code skeleton, including only class and function headers and retaining indentation structure.
2. **Original Code Cleaning:** The second step involved processing the original source code to remove existing legacy docstrings, ensuring that outdated annotations did not interfere with the revised or new documentation when being placed in their designated lines in the script.
3. **Annotation Mapping and Reintegration:** The generated docstrings were structured into an annotation dictionary, where method and class headers served as keys. This dictionary was then mapped back onto the cleaned source code, effectively reinserting the generated documentation at the correct locations.
4. **Script Saving and Return:** Finally, the annotated script was saved to the designated location and the pipeline terminated.

The system followed a structured pipeline, ensuring that each phase was independent yet seamlessly integrated into the overall workflow. The process is illustrated in Figure 1. Please note that for better readability of the diagram, rather trivial steps such as the user starting the pipeline were omitted and function names were adapted.

3.3 Learnings

Implementation Hurdles
The first implementation highlighted several critical challenges, particularly regarding the limitations of LLMs in generating structured and consistent code documentation. Key learnings include:

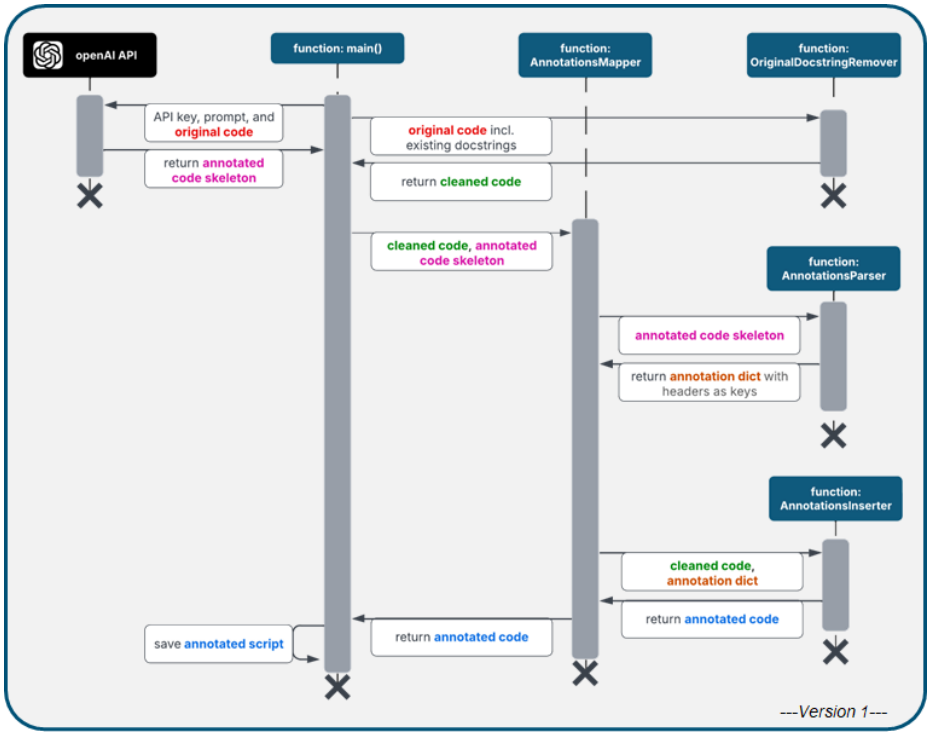

Fig. 1. Sequence Diagram of the First Implementation

- **Structure Aversion**: The model frequently struggled with keeping docstring formatting uniform, often resulting in incorrect indentations and missing sections.
- **Handling Large Inputs**: Increasing input sizes resulted in a decline in output quality, requiring strategies to manage complexity and improve coherence.
- **Ensuring Comprehensive Coverage**: Certain code elements were inconsistently annotated or omitted, necessitating better prompt engineering and validation mechanisms.
- **Optimizing Prompt Design**: Small variations in prompts significantly impacted the results, emphasizing the importance of refining input instructions for more predictable output.

Based on these findings, several takeaways were identified:

- **Keep It Simple**: Enforce a minimal but clear structure to ensure consistency in model-generated documentation.
- **Chunk & Review**: Split code into smaller, manageable chunks to maintain coherence and reduce complexity.

- **Step-by-Step Annotation**: Introduce a structured step-by-step annotation process to ensure comprehensive coverage of all code elements.
- **Cautious Prompt Design**: Carefully refine prompt phrasing and structure to achieve more predictable and reliable outputs. Pay attention to keywords and judge from an LLMs's perspective how these might be responded to.

Corresponding Literature Findings
The learnings we formulated closely align with recent research findings in the field of LLM-based code processing. Notable contributions from the literature include:

- **Dvivedi et al. (2024)** – Highlight the necessity of separately extracting function and class docstrings to ensure precise and appropriate annotations and to adhere to the construct-dependent structure requirements that are different for classes and functions.[2].
- **Khan et al. (2022)** – Demonstrate the importance of prompt engineering and the effectiveness of one-shot prompting techniques, reinforcing the benefits of refined prompt design [7].
- **Zhao et al. (2023)** – Emphasize the role of abstraction in structuring class and function documentation effectively, supporting the structured approach adopted in Version 2 [17].

Despite its limitations, the initial implementation of the first version provided valuable insights into the strengths and weaknesses of automated code annotation using LLMss. These extracted learnings form the foundation for refining the workflow to enhance reliability and consistency in quality for the next iteration.

3.4 Refined Approach: Addressing Problems and Optimizing Workflow

The refined system builds upon the foundational insights gained from the first approach, incorporating improvements to address structural inconsistencies, prompt sensitivity, and coverage gaps. Its architecture follows a sequential approach, ensuring a structured and logical flow of data processing. The core objective remains the extraction of definitions from source code, their annotation, and their seamless reintegration into the codebase.

Sequential Processing Workflow
The system follows a structured, step-by-step workflow to extract, process, and enhance source code with high-quality documentation. Each component plays a specific role in this pipeline, ensuring that the transformation from raw code to an annotated script is logically consistent. The following sections describe each processing step in detail.

The workflow begins with the **DefinitionsExtractor**, which scans the original source code to identify key structural elements. These include class definitions

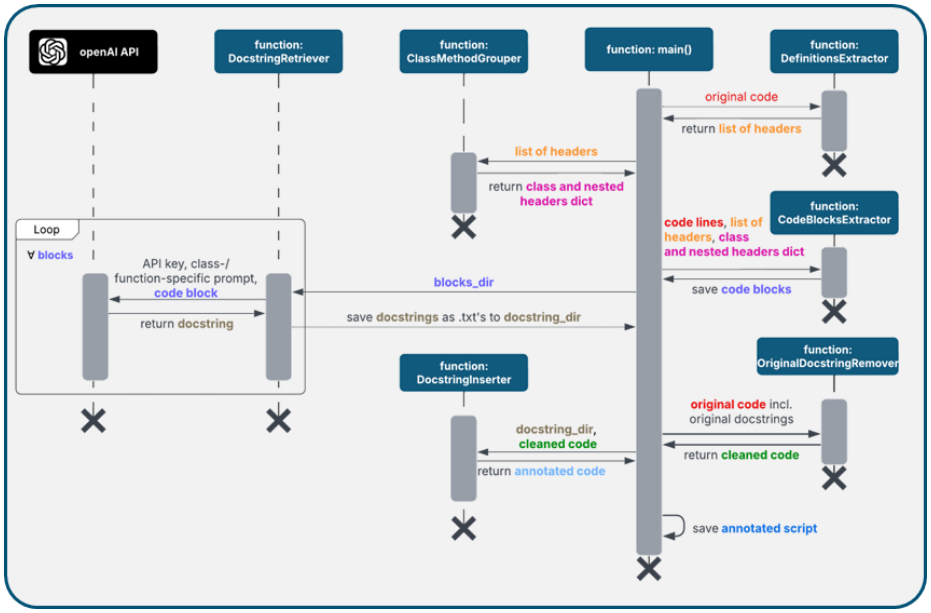

Fig. 2. Sequence Diagram of refined Docstring Generator

and method signatures, which serve as the foundation for further processing. This foundational step ensures that all relevant sections of the code are identified and categorized for further processing.

Once the headers are identified, they are passed to the **ClassMethodGrouper**, which organizes them into a structured format. If a method belongs to a class, it is nested under that class. The output of this stage is a dictionary where related components are grouped together. This step is critical for maintaining logical consistency, ensuring that methods and their parent classes remain correctly linked.

After structuring the headers, the **CodeBlocksExtractor** retrieves the corresponding code blocks for each class and method. This step ensures that all necessary context is preserved while extracting relevant portions of code. The extraction follows a structured approach:

- **Standalone methods:** Only the method signature and its implementation are extracted to ensure a focused analysis.
- **Class-level methods:** The class header is retained along with all contained method headers to maintain structural integrity.
- **Nested methods within a class:** The entire class structure is preserved to ensure a logical grouping of related components.

With the extracted code blocks in place, the **DocstringRetriever** processes them by querying **OpenAI's API**. If the generated response does not meet the required quality standards, a conditional resubmission mechanism is triggered, allowing the request to be retried up to three times. A carefully designed prompt, as shown in Figures 3 and 4, along with strict formatting rules and predefined example comments, ensures that the generated documentation remains consistent, well-structured, and tailored to each method's functionality. Additionally, special handling mechanisms adapt the docstrings based on the method type, enhancing clarity and usability.

To maintain accessibility and traceability, the generated annotations are temporarily stored as `.txt` files in a designated directory. This structured storage ensures that each annotation is correctly mapped to its corresponding class or method.

Before integrating the new annotations, the **OriginalDocstringRemover** processes the original source code by eliminating existing comments and outdated documentation. This step is crucial for preventing conflicts between old and newly generated docstrings, ensuring that only accurate and high-quality annotations remain in the final output.

In the final step, the **DocstringInserter** reintegrates the newly generated annotations into the source code. Each docstring from the *docstring_ dir* is mapped under its correspondent class or function header and finally, the annotated script is saved to the designated location.

Summary
This structured, sequential workflow ensures that each stage builds upon the previous one, creating a coherent and maintainable annotation process. By systematically organizing headers, extracting relevant code blocks, generating high-quality docstrings block-by-block, and reintegrating them in a structured manner, the system provides a reliable and scalable solution for automated code documentation. The combination of structured extraction and AI-powered annotation significantly enhances code readability and maintainability. Additionally, the modular nature of the workflow allows for seamless adaptation to evolving requirements, ensuring long-term flexibility and making it a robust, future-proof approach to documentation processing.

4 Results and Discussion

In this section, it is examined which of the initially derived requirements could be fulfilled to which extent, how the iterative refinements played into this performance outcome, and which limitations need to be considered.

Special Docstring Instructions for Methods

Generate a comprehensive Python docstring for the following function. The docstring should include:

1. A clear and concise description of the purpose.
2. A detailed explanation of caller method's parameters with their data types, excluding "self" and callee methods.
3. A description of the return value(s) with their data types.
4. Information about possible exceptions (errors) the function may raise.

Formatting rules:

- Use exactly 1 tab (4 spaces) for indentation.
- Ensure that 'Args', 'Returns', and 'Raises' are NEVER indented.

Example for a function style:

```
def add(a: int, b: int) -> int:
    \"\"\"
    Adds two integers and returns the result.

    Args:
        a (int): The first integer.
        b (int): The second integer.

    Returns:
        int: The sum of 'a' and 'b'.
    \"\"\"
```

IMPORTANT: - Do not include the function definition or Python at the beginning. - Just give me the docstring and do not use triple quotes or "' or "' in the response.

Fig. 3. One Version of the Docstring Prompt Instructions for Methods

4.1 Materialization of Learnings

Addressing Structure Aversion
A major issue in the first approach was the model's tendency to produce inconsistent docstring formatting. To mitigate this, Version 2 introduced a structured prompt design tailored to different documentation types. Instead of using a generic prompt, context-specific instructions were incorporated, ensuring that function and class docstrings adhered to predefined formatting conventions. The structured annotation process also helped maintain logical consistency across different code components.

Handling Large Inputs More Effectively
The decline in output quality with increasing input size was another notable limitation. This issue was tackled by changing the input strategy to provide only the specific method or class requiring documentation along with its immediate context. Instead of processing large code segments, the improved approach sup-

Special Docstring Instructions for Classes

Generate a comprehensive Python docstring for the following class. The docstring should include:

1. A clear and concise description of the purpose.
2. A detailed explanation of all parameters with their data types.
3. Information about possible exceptions (errors) the class may raise.

Additional details:

- Provide a general description of the class purpose.
- Include a description of the class attributes (if any) with their data types.
- For each method in the class, include a brief description (do not generate full docstrings for methods).

Formatting rules:

- Use exactly 1 tab (4 spaces) for indentation.
- Do not indent sections like 'Attributes' or 'Methods' etc.

Example for a class style:

```
class Calculator:
    \"\"\"
    A simple calculator class for basic arithmetic
    operations.

    Attributes:
        precision (int): The number of decimal places
        for results.

    Methods:
        add(a: float, b: float) -> float:
            Adds two numbers and returns the result.
        subtract(a: float, b: float) -> float:
            Subtracts the second number from the first.
    \"\"\"
```

IMPORTANT:

- Do not include the class definition or Python at the beginning.
- Just give me the docstring and do not use triple quotes or "' or "' in the response.

Fig. 4. One Version of the Docstring Prompt Instructions for Classes

plied the LLM with the method or class to be documented and, where applicable, included only the associated class and/or method headers.

Ensuring Comprehensive Coverage

The inconsistent annotation of certain code elements pointed to the need for a more systematic validation approach. To address this, the advanced approach implemented an iterative review process, allowing for conditional resubmission of outputs when errors or omissions were detected. Furthermore, automated verification steps, such as syntax validation and template-based consistency checks, helped refine the generated docstrings before integration into the codebase.

Refining Prompt Design for Predictable Results

Another critical finding from the first version was the sensitivity of output qual-

ity to minor prompt variations. In the enhanced version, a more cautious and differentiated prompt design was introduced as seen in Figures 3 and 4. Instead of relying on a single prompt format, prompts were dynamically adjusted based on the type of code element being documented. This included specifying expected output structures, integrating contextual metadata, and providing explicit formatting constraints. As a result, the polished version produced significantly more predictable and reliable documentation outputs compared to its predecessor.

Summary of Findings
The transition from the first approach to the final solution demonstrated that systematic refinements in prompt design, input structuring, and verification mechanisms can substantially improve the effectiveness of LLM-based code documentation. By addressing the core limitations identified in the first implementation, more structured, consistent, and scalable documentation generation was achieved.

4.2 Requirements Fulfillment

The implementation of these learnings led to an almost complete fulfillment of the defined functional and non-functional requirements.

Functional Requirements
The system accepts a Python script as input and processes its class and function definitions. This requirement is met by providing a terminal-accessible interface where users can specify input and output file paths. The tool parses the script structure and extracts code blocks for docstring generation.
Docstrings are generated in Google Style format with the specific style being adaptable via the prompt, making it flexible for different documentation standards. This ensures that generated documentation adheres to widely accepted formatting conventions.
The tool directly inserts the generated docstrings into the original script while preserving its executability. The returned script includes revised or newly added docstrings without affecting syntax correctness or functionality.

Non-Functional Requirements
The integrity of the original code is systematically assured, as the executable code itself is in no step altered by the language model. The tool ensures that all syntax elements, including indentation, decorators, and import statements, remain intact. This safeguard prevents unintended alterations that could disrupt script execution.
Context-awareness is partially fulfilled. Instead of analyzing the entire script for contextual understanding, the tool considers only close-by class and function headers. The effectiveness of this approach depends on the informativeness of these headers. While this method provides basic contextual understanding, it does not capture broader script-level dependencies.

Computational costs are controlled to ensure system scalability. The tool operates with an estimated cost of a maximum of $0.17 per thousand lines of code, depending on the chosen model. Users can opt for smaller models, such as GPT-3.5 Turbo, to reduce costs while accepting minor performance trade-offs. This cost-conscious design supports practical deployment without excessive resource consumption.

4.3 Limitations and Potential Approaches

The tool exhibits several limitations that impact its effectiveness in generating automated docstrings for Python scripts. These limitations primarily concern code quality, model performance, context-awareness, and compatibility with future Python versions.

Poor code quality presents a fundamental challenge to accurate docstring generation. Non-adherence to naming conventions, unclear function structures, and errors can lead to inaccurate or uninformative docstrings. Since the LLM is not tasked to search for errors and correct them, it expects correctly structured and functional code. As a result, it may struggle to generate meaningful documentation for poorly written scripts. Addressing this limitation is hard and can be considered out of scope for automated annotation tasks.

Contingent model performance naturally constrains the performance of GPT-based docstring generation. If the model fails to produce satisfactory results despite reinforcing key prompt elements after each of the at max. three resubmissions, the corresponding block is omitted from annotation. This safeguard prevents infinite loops but introduces the risk of missing documentation for certain code segments without alerting the user. A potential mitigation could involve implementing a user interface that lists all processed blocks, indicating whether a docstring was present before, if it was modified, and whether the tool ultimately omitted it. Such a feature would allow users to identify unannotated sections and manually intervene if necessary.

Context-awareness, as outlined in the requirements fulfillment section 4.2, is not fully achieved due to the limited context provided for each function or class. The tool relies only on close-by headers rather than a holistic script analysis, which can result in incomplete or misleading docstrings. One approach to improving context integration would involve multiple iterative passes over the script, where preliminary docstrings generated in the first round are used as contextual input for subsequent iterations. However, this method would significantly increase token consumption and computational costs, making it a less practical solution for large-scale projects.

Lastly, the tool's **reliance on current Python syntax** could become a limitation if future versions introduce fundamental changes. The *definitionsExtractor* and other components of the tool are dependent on established syntax patterns and keywords. A significant deviation in Python's structure could render the tool unable to accurately process scripts. While this remains a hypothetical concern, maintaining adaptability through regular tests would be necessary to ensure continued functionality across future Python releases.

5 Outlook: Toward a Theoretical Approach to Docstring Evaluation

Evaluation of automatically generated code annotations or summarizations can be seen as a rather immature topic. Over the past few years, quite some effort has been made by researchers to come up with an encompassive and precise evaluation method. However, the multi-faceted nature of the problem poses some problems. In this theoretical approach, the major shortcomings of often employed evaluation methods and scores are identified and auspicious approaches to tackle these are presented. On this basis, a structured and differentiated approach toward the heterogeneous annotation evaluation task is proposed to effectively cast the findings into one.

Shortcomings of Currently Employed Evaluation Methods
A variety of metrics are currently employed to evaluate automatically generated docstrings, broadly categorizable into qualitative self-defined metrics and quantitative statistical scores. Rai et al. (2022) [12] observe that qualitative, individually-defined metrics such as *completeness*, *accuracy*, and *relevance* in content evaluation, alongside *understandability* and *readability* for language evaluation, are widely used. For instance, Dvivedi et al. (2024) [2] utilize such metrics to assess generated code annotations in their paper about LLMs for code documentation generation. On the other hand, quantitative statistical scores originating from the machine translation domain, such as *BLEU*, *METEOR*, and *ROUGE* are also commonly employed for evaluation purposes. Zhu et al. (2024) [18] highlight the performance of Transformer-based models against RNN-based models using these metrics, showing significant performance gaps particularly in *ROUGE-L* and *METEOR*.
Despite their widespread usage, these evaluation scores entail notable limitations. The self-defined metrics, while offering more meaningful insights into the quality of docstrings, are inherently subjective and challenging to retrieve automatically. Their reliance on human judgment for consistency and accuracy complicates their integration into large-scale automated evaluation pipelines. Conversely, domain-foreign scores like *BLEU*, *ROUGE*, and *METEOR*, although easily measurable, exhibit essential shortcomings in the context of code documentation. Hu et al. (2022) [5] point out that these metrics do not align well with human judgment when evaluating code annotations. While *METEOR* demonstrates a stronger correlation with human evaluations compared to *BLEU*, Roy et al. (2021) [13] argue that it is nonetheless unreliable for approximating human judgment consistently.
Furthermore, these static scores struggle to capture the semantic dimension of code annotations effectively. The nuances and intricacies of code semantics often extend beyond the capabilities of metrics designed for textual similarity. Haque et al. (2022) [3] further emphasize that all metrics based on similarity scales between generated annotations and human-made reference annotations are compromised by the potential low quality and actuality issues of original annotations. Many

open-source codebases, frequently used as datasets for training and evaluation of documentation generation models, contain such compromised and deprecated annotations, thereby indirectly affecting the reliability of these metrics.

Evaluation Method Design Proposal

To overcome these challenges and create an automatable and robust assessment framework, it is proposed to split the task into a language evaluation task and a content evaluation task. Crucially, both evaluation methods should be designed to operate independently of potentially flawed original documentation.

Language evaluation should focus on assessing the readability and clarity of the annotations. Poudel et al. (2024) [11] suggest using language-only evaluation methods that do not depend on reference annotations. Metrics such as "Clarity," measured by the *Flesch–Kincaid readability score*, and "Conciseness," which assesses the information volume without unnecessary verbosity, provide objective, automatable measures of language appropriateness. The *Flesch–Kincaid readability score* quantifies how easy the text is to read based on sentence length and word complexity, making it suitable for ensuring that docstrings remain accessible to a broad audience. *conciseness* on the other hand, evaluates whether the annotation delivers essential information succinctly, avoiding redundancy while maintaining informativeness.

Content evaluation, on the other hand, should measure how well the semantic content of the generated annotation aligns with the actual code. The SIDE (Summary alIgnment to coDe sEmantics) score proposed by Mastropaolo et al. (2024) [8] offers a promising approach. This metric leverages contrastive learning to evaluate the alignment of code summaries with the underlying code semantics, independently of original developer-written comments. SIDE is built on MPNet, a pre-trained Transformer model that enhances language understanding by combining masked and permuted language modeling. By leveraging MPNet's ability to capture complex semantic relationships, SIDE provides a nuanced assessment of how accurately a summary reflects the code's functionality. This deep integration of advanced pre-training techniques makes SIDE a robust and innovative tool for content evaluation.

Potential Integration in Docstring Generation Tool

Integrating this evaluation framework into our existing docstring generation tool can significantly enhance the quality and reliability of the produced annotations. As illustrated in the tool's architecture, each code block from the blocks_dir is individually passed to the openAI API with a class- or function-specific prompt to generate the corresponding docstring. Currently, the tool performs only basic checks for style or structural flaws and may initiate a second or third attempt if the output repetitively falls short of quality standards. This step can be substantially improved by incorporating the proposed language and content evaluation methods. Specifically, if the *Flesch–Kincaid readability score* of a generated docstring falls below or exceeds predefined thresholds, the code block can be resubmitted to the API with a refined prompt instructing adjustments in readability.

Furthermore, if feasible, integrating a lightweight representation of SIDE's underlying MPNet model, pre-trained specifically for Python code, would allow for real-time content quality assurance. By setting a content quality threshold, code blocks failing to meet the standard could trigger additional API calls to refine the semantic accuracy of the docstrings. This eventually triple-layered evaluation could ensure that structural integrity, readability and semantic accuracy of the generated annotations are consistently maintained at a high standard. However, this approach would obviously entail higher generation costs in the form of generation time and API expenses, potentially making it an option for highest quality standards only.

In conclusion, while current evaluation methods for automatically generated docstrings exhibit significant shortcomings, emerging approaches offer promising solutions. By separating the evaluation into distinct language and content components and employing metrics that operate independently of flawed reference annotations, a more comprehensive and reliable evaluation framework can be established. This theoretical approach aims to pave the way for more precise, automatable, and meaningful evaluation of automatically generated code documentation in the future.

6 Conclusion

This project explored the use of Large Language Models (LLMs) for automated docstring generation in Python code. By analyzing source code, generating structured docstrings, and integrating them directly into scripts, the system aimed to improve documentation quality with minimal developer effort. Through iterative development, key issues such as inconsistent formatting, limited context awareness, and input size constraints were addressed, leading to a more reliable tool.

The final implementation met most functional and non-functional requirements, ensuring correct docstring formatting, structural preservation, and computational efficiency. However, challenges remain. The system struggles with annotating poor quality code, and its context awareness of each docstring is limited to several function and class headers rather than the broader script. Improving contextual integration without significantly increasing computational costs is a key area for future development.

In addition to the docstring generation tool, this project also explored methods for evaluating the quality of automatically generated docstrings. Domain-foreign metrics like BLEU and METEOR, while widely used, do not fully capture the accuracy or usefulness of documentation. Therefore, a structured evaluation approach combining readability scores and semantic alignment techniques is proposed as a potential solution to this challenge. While this evaluation approach is not integrated into the current system, it offers a foundation for future research aimed at improving docstring assessment.

Overall, while the tool demonstrates the feasibility of using LLMs for code documentation, further improvements in accuracy, efficiency, and evaluation methods are needed to enhance its practical application. The exploration of docstring evaluation methods provides a complementary perspective, highlighting the importance of measuring and ensuring documentation quality in future iterations of automated documentation tools.

References

1. Barone, A.V.M., Sennrich, R.: A parallel corpus of python functions and documentation strings for automated code documentation and code generation (2017), https://arxiv.org/abs/1707.02275
2. Dvivedi, S.S., Vijay, V., Pujari, S.L.R., Lodh, S., Kumar, D.: A comparative analysis of large language models for code documentation generation (2024), https://arxiv.org/abs/2312.10349
3. Haque, S., Eberhart, Z., Bansal, A., McMillan, C.: Semantic similarity metrics for evaluating source code summarization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. p. 36–47. ICPC '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524610.3527909, https://doi.org/10.1145/3524610.3527909
4. van Heesch, D.: Doxygen: Source code documentation generator tool (2008), http://www.stack.nl/ dimitri/doxygen/
5. Hu, X., Chen, Q., Wang, H., Xia, X., Lo, D., Zimmermann, T.: Correlating automated and human evaluation of code documentation generation quality. ACM Trans. Softw. Eng. Methodol. **31**(4) (Jul 2022). https://doi.org/10.1145/3502853, https://doi.org/10.1145/3502853
6. Ken Peffers, Tuure Tuunanen, M.A.R., Chatterjee, S.: A design science research methodology for information systems research. Journal of Management Information Systems **24**(3), 45–77 (2007). https://doi.org/10.2753/MIS0742-1222240302, https://doi.org/10.2753/MIS0742-1222240302
7. Khan, J.Y., Uddin, G.: Automatic code documentation generation using gpt-3 (2022), https://arxiv.org/abs/2209.02235
8. Mastropaolo, A., Ciniselli, M., Di Penta, M., Bavota, G.: Evaluating code summarization techniques: A new metric and an empirical characterization. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639174, https://doi.org/10.1145/3597503.3639174
9. McBurney, P.W., McMillan, C.: Automatic documentation generation via source code summarization of method context. In: Proceedings of the 22nd International Conference on Program Comprehension. p. 279–290. ICPC 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2597008.2597149, https://doi.org/10.1145/2597008.2597149
10. Moser, M., Pichler, J.: Documentation generation from annotated source code of scientific software: position paper. In: Proceedings of the International Workshop on Software Engineering for Science. p. 12–15. SE4Science '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2897676.2897679, https://doi.org/10.1145/2897676.2897679
11. Poudel, B., Cook, A., Traore, S., Ameli, S.: Documint: Docstring generation for python using small language models (2024), https://arxiv.org/abs/2405.10243
12. Rai, S., Belwal, R.C., Gupta, A.: A review on source code documentation. ACM Trans. Intell. Syst. Technol. **13**(5) (Jun 2022). https://doi.org/10.1145/3519312, https://doi.org/10.1145/3519312

13. Roy, D., Fakhoury, S., Arnaoudova, V.: Reassessing automatic evaluation metrics for code summarization tasks. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1105–1116. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468264.3468588, https://doi.org/10.1145/3468264.3468588
14. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need (2023), https://arxiv.org/abs/1706.03762
15. Venkatkrishna, V., Nagabushanam, D.S., Simon, E.I.O., Vidoni, M.: Docgen: Generating detailed parameter docstrings in python (2023), https://arxiv.org/abs/2311.06453
16. Yao, Z., Peddamail, J.R., Sun, H.: Coacor: Code annotation for code retrieval with reinforcement learning. In: The World Wide Web Conference. p. 2203–2214. WWW '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3308558.3313632, https://doi.org/10.1145/3308558.3313632
17. Zhao, J., Song, Y., Wang, J., Harris, I.G.: Gap-gen: Guided automatic python code generation (2023), https://arxiv.org/abs/2201.08810
18. Zhu, T., Li, Z., Pan, M., Shi, C., Zhang, T., Pei, Y., Li, X.: Deep is better? an empirical comparison of information retrieval and deep learning approaches to code summarization. ACM Trans. Softw. Eng. Methodol. **33**(3) (Mar 2024). https://doi.org/10.1145/3631975, https://doi.org/10.1145/3631975