DocAgent: A Multi-Agent System for Automated Code Documentation Generation

Dayu Yang* **Antoine Simoulin†** **Xin Qian†** **Xiaoyi Liu†** **Yuwei Cao†** **Zhaopu Teng†** **Grey Yang**
Meta AI
{dayuyang,antoinesimoulin,xinqian,xiaoyiliu,yuweicao,zhaoputeng,glyang}@meta.com

Abstract

High-quality code documentation is crucial for software development especially in the era of AI. However, generating it automatically using Large Language Models (LLMs) remains challenging, as existing approaches often produce incomplete, unhelpful, or factually incorrect outputs. We introduce DocAgent, a novel multi-agent collaborative system using topological code processing for incremental context building. Specialized agents (Reader, Searcher, Writer, Verifier, Orchestrator) then collaboratively generate documentation. We also propose a multi-faceted evaluation framework assessing Completeness, Helpfulness, and Truthfulness. Comprehensive experiments show DocAgent significantly outperforms baselines consistently. Our ablation study confirms the vital role of the topological processing order. DocAgent offers a robust approach for reliable code documentation generation in complex and proprietary repositories. Our code[1] and video[2] are publicly available.

1 Introduction

High-quality code documentation is essential for effective software development (De Souza et al., 2005; Garousi et al., 2015; Chen and Huang, 2009), and has become increasingly important as AI models depend on accurate docstrings[3] for code comprehension tasks (Zhou et al., 2022; Yang et al., 2024; Anthropic, 2025). However, creating and maintaining documentation is labor-intensive and prone to errors (McBurney et al., 2017; Parnas, 2010). Even top-starred open-source repositories on GitHub often exhibit low docstring coverage and quality,[4] leading to documentation that frequently lags behind code changes (Aghajani et al., 2019; Robillard, 2009; Uddin et al., 2021).

While LLM-based solutions—such as Fill-in-the-Middle (FIM) predictors (Roziere et al., 2023; GitHub, 2024) and chat agents (Meta, 2025; OpenAI, 2022)—offer automation, extensive studies (Dvivedi et al., 2024; Zhang et al., 2024; Zan et al., 2022; Zheng et al., 2024), along with our empirical analyses (§4), reveal three recurring limitations. First, these approaches often omit essential information (e.g., parameter or return-value descriptions). Second, they typically offer minimal context or rationale, limiting the usefulness of the generated documentation. Third, they sometimes hallucinate non-existent components, especially in large or proprietary repositories, undermining factual correctness (Zan et al., 2022; Ma et al., 2024; Abedu et al., 2024).

We identify three primary challenges that drive these shortcomings. **(1) Context Identification and Retrieval:** Large, complex repositories make it non-trivial to pinpoint which files, dependencies, or external references are genuinely relevant for a given component. **(2)Navigating Complex Dependencies:** Codebases often exhibit dependency chains that exceed typical LLM context limits, requiring strategic context management. **(3)Robust and Scalable Evaluation:** Existing evaluation metrics like BLEU or ROUGE(Roy et al., 2021; Guelman et al., 2024) incompletely capture the multi-faceted goals of documentation, while human evaluation, though more reliable, is expensive and subjective(Luo et al., 2024).

To tackle these challenges, we introduce **DocAgent**, a multi-agent system that processes code in a topologically sorted order and leverages specialized agents (**Reader**, **Searcher**, **Writer**, **Verifier**, **Orchestrator**) to collaboratively generate documentation. This mimics human workflows and manages context effectively. We also propose an automatic and robust multi-faceted evaluation

*Corresponding Author.
†Equal contribution.
[1] https://github.com/facebookresearch/DocAgent
[2] https://youtu.be/e9IjObGe9_I
[3] We use "code documentation" and "docstring" interchangeably throughout the paper.
[4] See Appendix C for more details.

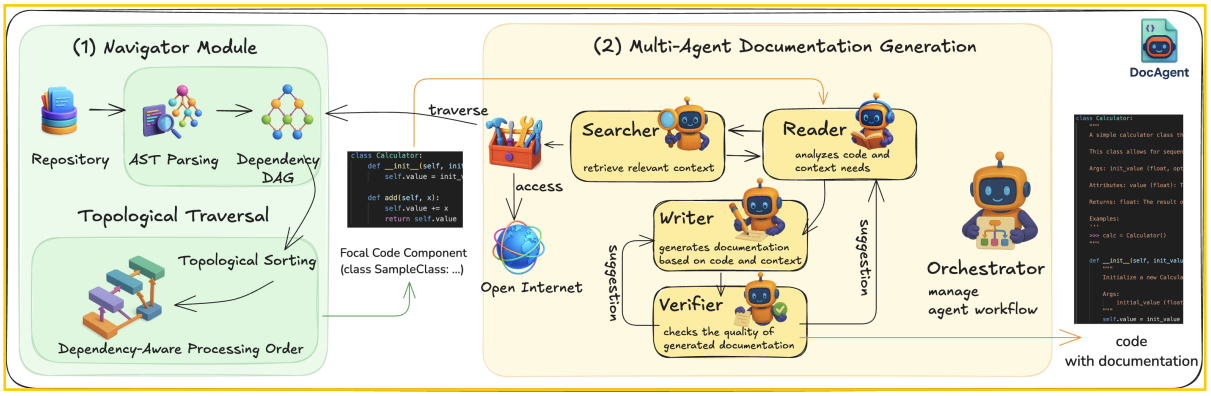

Figure 1: Architecture of DocAgent: (1) The Navigator Module uses AST parsing for a Dependency DAG and topological traversal. (2) The Multi-Agent framework uses specialized agents (Reader, Searcher, Writer, Verifier) with tools for context-aware documentation generation.

framework assessing **Completeness**, **Helpfulness**, and **Truthfulness** via deterministic checks and LLM-as-judge. Our main contributions are: 1) DocAgent, A multi-agent, topologically structured system for context-aware documentation generation. 2) A robust evaluation framework measuring completeness, helpfulness, and factual consistency of code documentation. 3) Comprehensive experiments on diverse repositories show DocAgent consistently outperforms state-of-the-art baselines.

2 Methodology

DocAgent operates in two stages to handle complex dependencies and ensure context relevance. First, the *Navigator* determines an optimal, dependency-aware processing order (§2.1). Second, a *Multi-Agent System* incrementally generates documentation, leveraging specialized agents for code analysis, information retrieval, drafting, and verification (§2.2). Figure 1 illustrates this architecture.

2.1 Navigator: Dependency-Aware Order

Generating accurate documentation often requires understanding its dependencies. However, naively including the full context of all direct and transitive dependencies can easily exceed context window limit especially in large, complex repositories. To address this, the *Navigator* module establishes a processing order that ensures components are documented only after their dependencies have been processed, thereby enabling incremental context building.

Dependency Graph Construction. DocAgent first performs static analysis on the entire target repository. It parses the Abstract Syntax Trees (ASTs) of source files to identify code components (functions, methods, classes) and their interdependencies. These dependencies include function/method calls, class inheritance, attribute access, and module imports. These components and relationships are used to construct a directed graph where nodes represent code components and a directed edge from A to B signifies that A depends on B ($A \rightarrow B$). To enable topological sorting, cycles within the graph are detected using Tarjan's algorithm (Tarjan, 1972) and condensed into a single super node. This results in a Directed Acyclic Graph (DAG) representing the repository's dependency structure.

The process begins with static analysis of the entire target repository. Abstract Syntax Trees (ASTs) are parsed for all source files to identify core code components (e.g., functions, methods, classes) and their interdependencies. These dependencies encompass function/method calls, class inheritance relationships, attribute accesses, and module imports. Based on this analysis, a directed graph is constructed where nodes represent code components and a directed edge from component A to component B ($A \rightarrow B$) signifies that A depends on B (i.e., B must be understood to fully understand A)[5].

Topological Traversal for Hierarchical Generation. Using the DAG, the Navigator performs a topological sort to determine the documentation generation order. The traversal adheres to the "Dependencies First" principles: A component is processed only after all components it directly depends on have been documented[6]. This topological ordering ensures that, by the time the multi-agent system generates documentation for a given component,

[5] Cycles within the graph are detected using Tarjan's algorithm (Tarjan, 1972) and condensed into a single node.

[6] Methods are documented before their enclosing class.

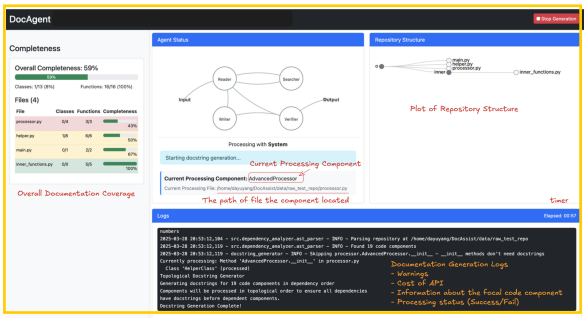

Figure 2: Screenshot of DocAgent live code documentation generation page.

all of its dependencies have already been described. Therefore, each code documentation only needs the information of its one-hop dependencies, eliminating the need to pull in an ever-growing chain of background information.

2.2 Multi-Agent Documentation Generation

Following Navigator's order, the multi-agent system generates documentation for each component using four specialized agents coordinated by an Orchestrator. Input is the focal component's source code including newly generated documentation.

Reader. The Reader agent initiates the process by analyzing the focal component's code. Its primary goal is to determine the information required to generate a comprehensive and helpful code documentation. It assesses the component's complexity, visibility (public/private), and implementation details to decide: *If additional context is needed:* Simple, self-contained components might not require external information. *What context is needed:* This involves identifying specific internal dependencies (functions/classes it uses), usage contexts (where the component is called, revealing its purpose), or external concepts (algorithms, libraries, domain knowledge) referenced implicitly or explicitly.

The agent outputs structured XML requests for two types of information requests (1) internal information about related code components, and (2) external knowledge for specialized algorithms or techniques.

The internal information request consists with the dependency and the reference. Dependency means the focal component calls other components defined in the repository, where reader will determinate if a dependent is needed or not to provide necessary context information.

Reference means the focal component is called in somewhere in the code repository, showing how it can be used in the real-world application and therefore reveal the purpose of the focal code component. This is particularly important for public functions or APIs exposed to the users of the repository.

External requests target information not directly present or inferable from the codebase itself, such as domain-specific knowledge or third-party library functionalities (see Appendix B).

Searcher. The Searcher agent is responsible for fulfilling the Reader's information requests using specialized tools: *Internal Code Analysis Tool:* This tool leverages static analysis capabilities to navigate the codebase. It can retrieve the source code and existing documentation of specified internal components, identify call sites for the focal component, trace dependencies using the precomputed graph or on-the-fly analysis, and extract relevant structural information (e.g., class hierarchies, method signatures). *External Knowledge Retrieval Tool:* This tool interfaces with external knowledge sources via a generic retrieval API . It formulates queries based on the Reader's requests for external concepts and processes the results to extract pertinent explanations, definitions, or descriptions.

The Searcher consolidates the retrieved internal code information and external knowledge into a structured format, which serves as the context for the subsequent agents.

Like two human agents collaborate on a project and talk with each other, after Searcher send the retrieved information back to the reader, reader read the updated context and the focal code component, and see if the context is adequate for generating the documenation. If reader still feel the retrieved context is still not adequate, reader can further send information request to the searcher. So the information request, and new information can be sent back and forth between reader and searcher, until adequate information is retrieved.

Writer. The Writer agent receives the focal component's code and the structured context compiled by the Searcher. Its task is to generate the code documentation. The generation process is guided by prompts that specify the desired structure and content based on the component type: *Functions/Methods:* Typically require a summary, extended description, parameter descriptions (Args), return value description (Returns), raised exceptions (Raises), and potentially usage examples (especially for public-facing components). *Classes:* Typically require a summary, extended description,

initialization examples, constructor parameter descriptions (Args), and public attribute descriptions (Attributes).

The Writer synthesizes information from both the code and the provided context to produce a draft code documentation adhering to these requirements.

Verifier. The Verifier take the context, code component, and generated code documentation from the writer as inputs, evaluates the quality of code documentation against predefined criteria: information value, detail level, and completeness. Upon evaluation, the Verifier either approves the documentation or provides specific improvement suggestions through structured feedback.

Verifier can talk to writer if the issue can be address without additional context information, for example: format issue, which can be easily address by asking writer to rewrite.

If the issue is relevant to lack of information, and additional context is needed, veirfier can also provide suggestion to reader, and additional information will be gathered through another Reader-Searcher cycle.

Orchestrator. An Orchestrator manages the agent workflow through an iterative process. The cycle begins with the Reader analyzing the focal component and requesting necessary context. The Searcher gathers this information, after which the Writer generates a docstring. The Verifier then evaluates the docstring quality, either approving it or returning it for revision. This process continues until a satisfactory code documentaion is generated or a maximum iteration limit is reached.

Adaptive Context Management: To handle potentially large contexts retrieved by the Searcher, especially for complex components, the Orchestrator implements an adaptive context truncation mechanism. It monitors the total token count of the context provided to the Writer. If the context exceeds a configurable threshold (based on the underlying LLM's limits), the Orchestrator applies a targeted truncation strategy. It identifies the largest sections within the structured context (e.g., external knowledge snippets, specific dependency details) and selectively removes content from the end of these sections to reduce the token count while preserving the overall structure. This ensures that the context remains within operational limits, balancing contextual richness with model constraints.

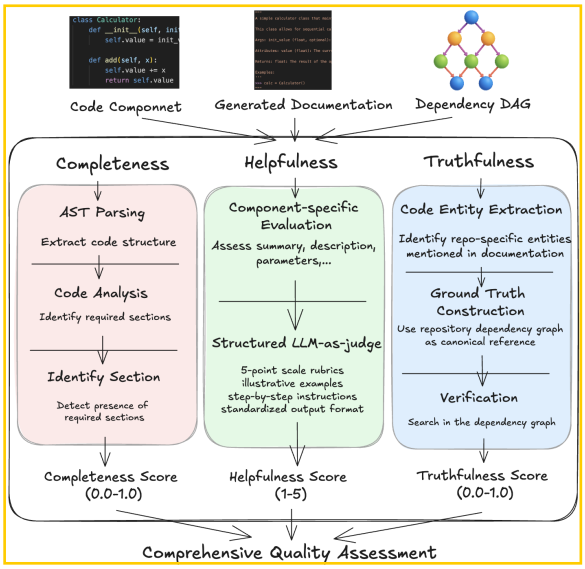

Figure 3: Multi-facet Evaluation Framework of code documentation, assessing quality along three dimensions: (1) Completeness measures structural adherence to documentation conventions; (2) Helpfulness evaluates practical utility; and (3) Truthfulness verifies factual accuracy.

3 Evaluation Framework

Evaluating the quality of automatically generated code documentation is challenging. Traditional metrics commonly used in natural language generation, such as BLEU or ROUGE cannot be used because of lack of gold references (Roy et al., 2021; Guelman et al., 2024). Simple heuristics like documentation length are insufficient indicators of actual utility. While human evaluation provides the most accurate assessment (Luo et al., 2024), it is inherently subjective, expensive, and difficult to scale, rendering it impractical for large-scale experiments or continuous integration scenarios.

To overcome these limitations, we propose a comprehensive and scalable evaluation framework designed to systematically assess documentation quality along three crucial dimensions: *Completeness*, *Helpfulness*, and *Truthfulness*. This multi-faceted approach combines deterministic structural checks, LLM-based qualitative assessments, and fact-checking against the codebase itself, providing a holistic view of the generated documentation's value. Our methodology is informed by established software engineering best practices for documentation and addresses the specific shortcomings observed in existing LLM-based generation systems.

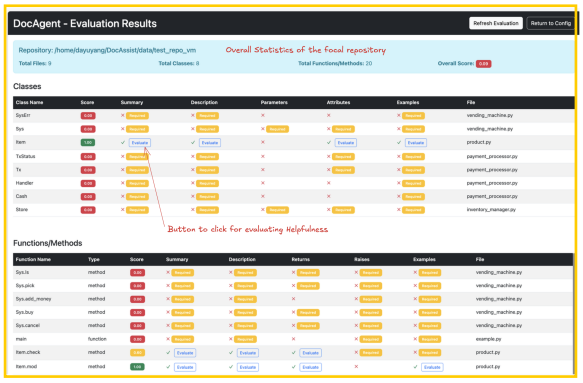

Figure 4: Screenshot of DocAgent Live Evaluation Framework

3.1 Completeness

Completeness measures the extent to which the generated documentation adheres to standard structural conventions and includes essential components expected for a given code element (e.g., function, class). High-quality code documentation typically includes not only a summary but also descriptions of parameters, return values, raised exceptions, and potentially usage examples, dynamically depending on the element's signature, body and visibility.

To quantify completeness, we employ an automated checker based on Abstract Syntax Tree (AST) analysis and regular expressions. The process involves: **AST Parsing:** Identifying code components (classes, functions, methods) and extracting their generated docstrings. **Code Analysis:** Analyzing the code signature and body (e.g., presence of parameters, return statements, `raise` statements) and visibility (public/private) to determine the *required* documentation sections dynamically. For instance, a function without parameters does not require an "Args" section, while a public class method might benefit more from an "Example" section than a private helper function. **Section Identification:** Detecting the presence of standard sections (e.g., Summary, Description, Args, Returns, Raises, Examples, Attributes for classes) within the docstring using predefined patterns and structural cues. **Scoring:** Calculating a completeness score for each docstring as the proportion of required sections that are present. This yields a normalized score between 0.0 and 1.0.

This deterministic approach provides an objective measure of structural adherence, indicating whether the documentation meets basic formal requirements.

3.2 Helpfulness

Helpfulness assesses the semantic quality and practical utility of the documentation content. A helpful docstring goes beyond merely restating code elements; it elucidates the *purpose*, *usage context*, *design rationale*, and potential *constraints* of the code. Key aspects include: **Clarity and Conciseness:** Is the summary informative yet brief? **Descriptive Depth:** Does the extended description provide sufficient context, explain the 'why' behind the code, or mention relevant scenarios or edge cases? **Parameter/Attribute Utility:** Are descriptions for inputs and attributes meaningful, specifying expected types, value ranges, or constraints, rather than just echoing names? **Guidance:** Does the documentation effectively guide a developer on *when* and *how* to use the component?

Assessing these qualitative aspects automatically is challenging. Inspired by recent work on evaluating complex generation tasks (Wang et al., 2024; Zhuge et al., 2024), we utilize an LLM-as-judge approach, carefully structured to enhance robustness and consistency. To mitigate potential biases and variability associated with LLM judgments, we implement a sophisticated framework: **Component-Specific Evaluation:** We decompose the evaluation by assessing distinct parts of the docstring separately (e.g., summary, main description, parameter descriptions) using tailored prompts for each. **Structured Prompt Engineering:** Each prompt includes: 1) *Explicit Scoring Rubrics:* Detailed criteria for a 5-point Likert scale (1=Poor to 5=Excellent), defining expectations for each score level regarding clarity, depth, and utility. 2) *Illustrative Examples:* Concrete examples of good and bad documentation snippets corresponding to different score levels, grounding the evaluation criteria. 3) *Step-by-Step Instructions:* Guiding the LLM to analyze the code, compare the docstring against the rubric, consider the code's context, and justify its rating. 4) *Standardized Output Format:* Requiring the LLM to provide structured output, including detailed reasoning, specific suggestions for improvement (if applicable), and the final numerical score. This facilitates analysis and consistency checking.

This structured approach allows for scalable assessment of semantic quality, moving beyond surface-level checks to gauge the documentation's actual value to a developer.

3.3 Truthfulness

A critical dimension of documentation quality is its factual accuracy, or *Truthfulness*. Documentation, especially when generated by LLMs unfamiliar with a specific private codebase, can suffer from "hallucinations"—confidently referencing non-existent methods, parameters, or classes, or misrepresenting relationships between components. Such inaccuracies severely undermine trust and can mislead developers.

We evaluate Truthfulness by verifying whether entities mentioned in the generated documentation actually exist within the target repository and are referenced correctly. Our pipeline comprises three stages: **Code Entity Extraction:** An LLM is prompted to identify mentions of repository-specific code components (classes, functions, methods, attributes) within the generated docstring. The prompt specifically instructs the model to distinguish these from standard language keywords, built-in types (e.g., `list`, `dict`), and common external library components, focusing on internal references. **Ground Truth Construction:** We leverage the dependency graph constructed by the Navigator module 2.1. This graph serves as the ground truth, containing a canonical representation of all code components and their locations within the repository. **Verification:** Each extracted entity mention is cross-referenced against the dependency graph.

We quantify Truthfulness using the **Existence Ratio**: the proportion of unique repository-specific entities mentioned in the documentation that correspond to actual entities in the codebase.Existence Ratio $= \frac{|\text{Verified Entities}|}{|\text{Extracted Entities}|}$

A high ratio indicates that the documentation is well-grounded in the actual code structure, minimizing the risk of hallucinated references.

Together, these three dimensions—Completeness, Helpfulness, and Truthfulness—provide a robust and nuanced framework for evaluating automatic code documentation systems, enabling quantitative comparisons and deeper insights into their strengths and weaknesses.

4 Experiment

4.1 Baselines

We compare DocAgent against two representative baseline systems commonly used for code documentation generation: **FIM** (Fill-in-the-middle): Simulates inline code completion tools that predict documentation based on surrounding code. We use CodeLlama-13B (Roziere et al., 2023), an open model trained with FIM tasks (Bavarian et al., 2022). Abbreviated as **FIM-CL**. **Chat**: Represents generating documentation by providing the code snippet directly to a chat-based LLM. We test two leading models: GPT-4o mini [7](OpenAI, 2022) and CodeLlama-34B-instruct(Roziere et al., 2023). Abbreviated as **Chat-GPT** and **Chat-CL**, respectively.

4.2 Experiment Setup

Data. We select a representative subset of Python repositories to ensure diversity in size, complexity, and domain. The dataset comprises modules, functions, methods, and classes with varying degrees of dependency density (details in Appendix D).

Systems. We evaluate two variants of our proposed system, differing only in the backbone LLM used by the agents: **DA-GPT**: DocAgent utilizing GPT-4o mini. **DA-CL**: DocAgent utilizing CodeLlama-34B-instruct[8].

Statistical Significance. All claims of statistical significance are based on paired t-tests with a significance threshold of $p < 0.05$[9]

4.3 Experiment Results

We evaluate the systems using the framework proposed in Section 3, focusing on Completeness, Helpfulness, and Truthfulness.

4.3.1 Completeness

System	Overall	Function	Method	Class
DA-GPT	0.934†	0.945†	0.935†	**0.914**†
DA-CL	**0.953**†‡	**0.985**†‡	**0.982**†‡	0.816†‡
Chat-GPT	0.815	0.828	0.823	0.773
Chat-CL	0.724	0.726	0.744	0.667
FIM-CL	0.314	0.291	0.345	0.277

Table 1: Average Completeness Scores. †: Significantly better than corresponding Chat baseline. ‡: Significantly better than FIM baseline.

As shown in Table 1, both DocAgent variants significantly outperform their respective Chat counterparts. DocAgent (CodeLlama-34B) achieves an

[7] 2024-07-18 version

[8] The choice of backbone LLM is orthogonal to the DocAgent framework itself. We use GPT-4o-2024-08-06 universally for running evaluation for more robust results.

[9] Due to space limitations, we are unable to include the full prompts and detailed experimental setup in the paper. However, all configurations are available in our project's public release repository.

overall score of 0.953, representing a substantial improvement of 0.229 points over Chat. Similarly, DocAgent (GPT-4o mini) scores 0.934 overall, significantly higher than Chat at 0.815. These improvements are statistically significant across all component types. FIM performs poorly, achieving an overall completeness score of only 0.314. This highlights the effectiveness of DocAgent's structured, context-aware generation process compared to simply prompting an LLM with the code in isolation.

4.3.2 Helpfulness

As shown in Table 2, DocAgent (GPT-4o mini) achieves the highest overall helpfulness score, significantly outperforming the corresponding Chat baseline. demonstrating its ability to generate clearer and more informative content by leveraging retrieved context.

System	Overall	Summary	Description	Parameters
DA-GPT	**3.88**†	**4.32**†	**3.60**†	**2.71**
DA-CL	2.35‡	2.36†‡	2.43‡	2.00
Chat-GPT	2.95	3.56	2.42	2.20
Chat-CL	2.16	2.04	2.37	1.80
FIM-CL	1.51	1.30	2.45	1.50

Table 2: Average Helpfulness Scores. †: Significantly better than corresponding Chat. ‡: Significantly better than FIM.

DocAgent (CodeLlama-34B) also shows an improvement over its Chat counterpart, producing significantly more helpful summaries. Furthermore, DocAgent (CodeLlama-34B) also significantly outperforms FIM. Across aspects, generating helpful parameter descriptions appears most challenging. DocAgent (GPT-4o mini) achieves the highest score even here, suggesting its structured approach aids in this difficult task, although room for improvement remains.

4.3.3 Truthfulness

The results in Table 3 demonstrate the superior factual accuracy of documentation generated by DocAgent. DocAgent (GPT-4o mini) achieves the highest Existence Ratio at 95.74%, indicating that the vast majority of its references to internal code components are correct. DocAgent (CodeLlama-34B) also performs strongly with a ratio of 88.17%.

This contrasts sharply with the baselines. The Chat approaches exhibit significantly lower truthfulness, with Chat (GPT-4o mini) at 61.10% and Chat (CodeLlama-34B) at 68.03%. This suggests that simply providing the code snippet to a chat model often leads to inaccurate assumptions or hallucinations about the surrounding codebase context. FIM performs worst, with an Existence Ratio of only 45.04%, implying that nearly half of its references to repository entities might be incorrect. This low score highlights a significant risk of misleading developers when using FIM for documentation.

System	Verified	Extracted	Existence Ratio (%)
DA-GPT	265	305	**95.74%**
DA-CL	354	600	88.17%
Chat-GPT	366	347	61.10%
Chat-CL	366	488	68.03%
FIM-CL	338	131	45.04%

Table 3: Truthfulness Analysis: Existence Ratio (%). Higher is better. Extracted = extracted entities; Verifed = verified entities in §3.3.

4.4 Ablation Study

To isolate the contribution of the dependency-aware processing order determined by the Navigator module (§ 2.1), we conducted an ablation study. We created variants of DocAgent (DA-Rand-GPT, DA-Rand-CL) that process components in a random order[10].

4.4.1 Impact on Helpfulness

System	Overall	Summary	Description	Parameters
DA-GPT	**3.88**†	**4.32**†	**3.60**	**2.71**
DA-Rand-GPT	3.44(-0.44)	3.62(-0.70)	3.30(-0.30)	2.20(-0.51)
DA-CL	**2.35**†	**2.36**†	**2.43**	**2.00**
DA-Rand-CL	2.18(-0.17)	1.88(-0.48)	2.42(-0.10)	2.00(0.00)

Table 4: Ablation: Average Helpfulness Scores. † If DocAgent significantly better than its Random variant.

The results in Table 4 demonstrate the benefit of the Navigator's topological sorting in improving Helpfulness. For both underlying LLMs, the full DocAgent achieved significantly higher overall helpfulness scores compared to its random-order counterpart. With GPT-4o mini, the full DocAgent scored 3.69 overall, significantly higher than DocAgent-Random's 3.44. The improvement was particularly pronounced in summary generation. Similarly, with CodeLlama-34B, the full DocAgent scored 2.39 overall, significantly outperforming DocAgent-Random's 2.18. Again, the summary scores showed a significant difference.

[10]Completeness was omitted from the ablation study because it depends on the code's structure, not the Navigator's processing order.

4.4.2 Impact on Truthfulness

We also evaluated the impact of removing the hierachical generation order on the factual accuracy (Truthfulness). Without the Navigator, the Searcher can still retrieve dependent code components. However, since the 'Dependencies First' principle is not followed, these components are less likely to have already generated documentation available for context.

System	Verified	Extracted	Existence Ratio (%)
DA-GPT	187	224	**94.64%**
DA-Rand-GPT	164(-23)	166(-58)	86.75(-7.89)%
DA-CL	190	343	**87.76%**
DA-Rand-CL	188(-2)	360(+17)	83.06(-4.70)%

Table 5: Ablation: Truthfulness Analysis (Existence Ratio %). Use 50 randomly sampled code components from full data to evaluate.

Table 5 demonstrates that the topological sort also improves truthfulness. Both full DocAgent variants achieve higher Existence Ratios than their random-order counterparts. Existence ratio of DocAgent (GPT-4o-mini) drops from 94.64% to 86.75% without the sort, and the ratio of DocAgent (Codellama-34B) drops from 87.76% to 83.06%.

Collectively, the ablation results confirm that the Navigator's dependency-aware topological ordering is a crucial component of DocAgent, significantly contributing to both the helpfulness and factual accuracy of the generated documentation by enabling effective incremental context management.

5 Conclusion

We addressed the challenge of automatically generating high-quality code documentation, a task where existing LLM-based methods often struggle with incompleteness, lack of helpfulness, and factual inaccuracies. We introduced **DocAgent**, a novel tool-integrated, multi-agent system that leverages a dependency-aware topological processing order determined by a **Navigator** module. This allows specialized agents (Reader, Searcher, Writer, Verifier, Orchestrator) to collaboratively generate documentation by incrementally building context from dependencies. We also proposed a robust and scalable evaluation framework assessing **Completeness**, **Helpfulness**, and **Truthfulness**. Our experiments on diverse Python repositories demonstrate that DocAgent significantly outperforms FIM and Chat baselines consistently, producing more complete, helpful, and factually accurate documentation. An ablation study confirmed the critical contribution of the topological processing order to both helpfulness and truthfulness. DocAgent represents a promising step towards reliable and useful automated code documentation generation for complex and proprietary software.

6 Ethics and Limitations

DocAgent, while advancing automated code documentation, has inherent limitations and ethical considerations. Technically, processing extremely large codebases may still challenge LLM context limits despite topological sorting and context management. Relying solely on static analysis restricts understanding of dynamic behavior, and the current Python focus requires effort for adaptation to other languages.

Ethically, the primary concern is factual accuracy; generated documentation, though improved, may still contain hallucinations or inaccuracies, potentially misleading developers. The underlying LLMs may propagate biases from their training data into the documentation. Over-reliance on such tools could potentially hinder developers' deep code comprehension skills. Applying DocAgent to proprietary code necessitates careful handling, especially regarding external queries, to avoid inadvertently leaking sensitive information. Finally, the computational resources required for LLM-driven multi-agent systems represent a notable cost and environmental consideration. Future work should address these limitations, focusing on robustness, bias mitigation, and deeper evaluation, while emphasizing that human oversight remains crucial in practical deployment.

References

Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. 2024. Llm-based chatbots for mining software repositories: Challenges and opportunities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 201–210.

Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE.

Wasi U Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *ACL*.

Anthropic. 2025. Model context length increases with the new context protocol. Accessed: 2025-03-27.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Jie-Cherng Chen and Sun-Jen Huang. 2009. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Qian Chen, Binyuan Tang, Yankai Zhang, Binhua Wang, Zhifang Zhang, and Qun Zhang. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2305.03047*.

Cheng-Han Chiang and Hung-yi Lee. 2023. Can large language models be an alternative to human evaluations? In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*.

Colin B Clement, Andrew Terrell, Hanlin Mao, Joshua Dillon, Sameer Singh, and Dan Alistarh. 2020. Pymt5: Multi-mode translation of natural language and python code with transformers. In *EMNLP*.

Sergio Cozzetti B De Souza, Nicolas Anquetil, and Káthia M de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75.

Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pages 65–73.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, and Daxin Jiang. 2020. Codebert: A pre-trained model for programming and natural languages. In *EMNLP*.

Golara Garousi, Vahid Garousi-Yusifoğlu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. 2015. Usage and usefulness of technical software documentation: An industrial case study. *Information and software technology*, 57:664–682.

GitHub. 2024. How github copilot is getting better at understanding your code. Accessed: 2025-03-27.

Liron Guelman, Alon Lavie, and Eran Yahav. 2024. Using large language models to document code: A first quantitative and qualitative assessment. *arXiv preprint arXiv:2403.04264*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Nan Duan, Ming Zhou, and Daxin Jiang. 2021. Graphcodebert: Pre-training code representations with data flow. In *ICLR*.

Seungone Kim, Soobin Kim, Alice Oh, and Gunhee Han. 2023. Prometheus: Inducing fine-grained evaluation capability in language models. *arXiv preprint arXiv:2310.08491*.

Michael Krumdick, Jason Wei, Xinyang Chen, Shangbin Du, Shu Xu, Dale Schuurmans, and Ed H Chi. 2025. No free labels: Limitations of llm-as-a-judge without human grounding. *arXiv preprint arXiv:2503.05061*.

Yukyung Lee, Wonjoon Cho, and Jinhyuk Kim. 2024. Checkeval: A reliable llm-as-a-judge framework for evaluating text generation using checklists. *arXiv preprint arXiv:2403.18771*.

Raymond Li, Lewis Tunstall, Patrick von Platen, Jungtaek Kim, Teven Le Scao, Thomas Wolf, and Alexander M. Rush. 2023a. Starcoder: May the source be with you! *Preprint*, arXiv:2305.06161.

Xiang Li, Qinyuan Zhu, Yelong Cheng, Weizhu Xu, and Xi Liu. 2023b. Camel: Communicative agents for "mind" exploration. *arXiv preprint arXiv:2303.17760*.

Minqian Liu, Cheng Feng, Qing Lyu, Wenhao Zeng, Chao Zheng, Ruidan Zhang, and Steven C H Lin. 2023a. X-eval: Generalizable multi-aspect text evaluation via augmented instruction tuning. *arXiv preprint arXiv:2311.08788*.

Yang Liu, Yao Fu, Yujie Xie, Xinyi Chen, Bo Pang, Chenyan Qian, Teng Ma, and Dragomir Radev. 2023b. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, and 1 others. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? *arXiv preprint arXiv:2406.01422*.

Paul W McBurney, Siyuan Jiang, Marouane Kessentini, Nicholas A Kraft, Ameer Armaly, Mohamed Wiem Mkaouer, and Collin McMillan. 2017. Towards prioritizing documentation effort. *IEEE Transactions on Software Engineering*, 44(9):897–913.

Meta. 2025. Meta ai. https://ai.meta.com/meta-ai/. Accessed: 2025-03-27.

OpenAI. 2022. Introducing chatgpt. Accessed: 2025-03-27.

David Lorge Parnas. 2010. Precise documentation: The key to better software. In *The future of software engineering*, pages 125–148. Springer.

Yuzhang Qian, Zian Zhang, Liang Pan, Peng Wang, Shouyi Liu, Wayne Xin Zhao, and Ji-Rong Wen. 2023. Chatdev: Revolutionizing software development with ai-collaborative agents. *arXiv preprint arXiv:2307.07924*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741.

Martin P Robillard. 2009. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34.

Rahul Roy, Saikat Chakraborty, Baishakhi Ray, and Miryung Kim. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1344–1356.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Noah Shinn, Margaret Labash, and Stefano Ermon. 2023. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*.

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160.

Gias Uddin, Foutse Khomh, and Chanchal K Roy. 2021. Automatic api usage scenario documentation from technical q&a sites. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–45.

Yidong Wang, Qi Guo, Wenjin Yao, Hongbo Zhang, Xin Zhang, Zhen Wu, Meishan Zhang, Xinyu Dai, Qingsong Wen, Wei Ye, and 1 others. 2024. Autosurvey: Large language models can automatically write surveys. *Advances in Neural Information Processing Systems*, 37:115119–115145.

Yue Wang, Shuo Ren, Daya Lu, Duyu Tang, Nan Duan, Ming Zhou, and Daxin Jiang. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*.

Ziniu Wu, Cheng Liu, Jindong Zhang, Xinyun Li, Yewen Wang, Jimmy Xin, Lianmin Zhang, Eric Xing, Yuxin Lu, and Percy Liang. 2023. Autogen: Enabling next-generation multi-agent communication with language models. *arXiv preprint arXiv:2309.07864*.

Dayu Yang, Tianyang Liu, Daoan Zhang, and 1 others. 2025. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. *arXiv preprint arXiv:2502.19411*.

Guang Yang, Yu Zhou, Wei Cheng, Xiangyu Zhang, Xiang Chen, Terry Yue Zhuo, Ke Liu, Xin Zhou, David Lo, and Taolue Chen. 2024. Less is more: Docstring compression in code generation. *arXiv preprint arXiv:2410.22793*.

Shinn Yao, Jeffrey Zhao, Dian Yu, Kang Chen, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Yaqing Zan, Mingyu Ding, Bill Yuchen Lin, and Xiang Ren. 2022. When language model meets private library. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics.

Kaiyu Zhang, Yifei Wang, Yue Yu, Yujie Li, Zihan Lin, Dongxu Zhang, Yichi Zhou, Yifei Xu, Ang Chen, Weiyi Zhang, and 1 others. 2024. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *arXiv preprint arXiv:2401.10650*.

Shiyue Zhang, Binyi Li, Jason Wei, Aditi Raghunathan Vyas, and Percy Liang. 2023a. Themis: A flexible and interpretable nlg evaluation model. *arXiv preprint arXiv:2309.12082*.

Xiaoqing Zhang, Zhirui Wang, Lichao Yang, Wei Zhang, and Yong Zhang. 2023b. Mapcoder: Map-reduce-style code generation with multi-agent collaboration. *arXiv preprint arXiv:2307.15808*.

Lianmin Zheng, Shangbin Du, Yuhui Lin, Yukuo Shao, Zi Lin, Zhen Liu, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*.

Zihan Zheng, Jiayi Zheng, Weiyan Liu, Yizhong Wang, Chen Liu, Xiang Lorraine Li, Mu Li, Wenhao Zhang, Diyi Huang, and Xiang Ren. 2024. How well do llms generate code for different application domains? *arXiv preprint arXiv:2401.13727*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv: 2207.05987*.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, and 1 others. 2024. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*.

A Related Work

LLM Agent Recent advancements in LLM agents have enabled automating complex code-related tasks (Yang et al., 2025). Single-agent frameworks like ReAct (Yao et al., 2022) and Reflexion (Shinn et al., 2023) integrate action-reasoning and self-reflection. Multi-agent systems (CAMEL (Li et al., 2023b), AutoGen (Wu et al., 2023)) extend these ideas with role-specialized LLMs and structured communication to handle more complex problems. In software development, MapCoder (Zhang et al., 2023b), RGD (Chen et al., 2023), and ChatDev (Qian et al., 2023) leverage specialized agents for many downstream tasks, achieving state-of-the-art code generation. These insights on multi-agent coordination and workflow structuring underpin our DocAgent framework, which adopts a topologically-aware, tool-integrated multi-agent design.

Code Summarization Pre-trained encoders such as CodeBERT(Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021) introduced bimodal and structure-aware learning, while encoder-decoder models PLBART (Ahmad et al., 2021) and CodeT5 (Wang et al., 2021) unified code generation and summarization. PyMT5 (Clement et al., 2020) extended T5 for Python docstring translation with multi-mode support. Recently, LLMs (OpenAI Codex (Chen et al., 2021), StarCoder (Li et al., 2023a), CodeLlama (Roziere et al., 2023)) have demonstrated strong zero-shot summarization. However, they often lack repository-level context, dependency awareness, and collaboration—limitations our multi-agent, context-aware DocAgent aims to overcome.

B Why External Information is needed

The external open-internet information request is necessary for writing documentation for some novel, newly-proposed ideas, like novel evaluation method, algorithm, model structure, loss functions. For example, DPO (Rafailov et al., 2023) is a reinforcement learning algorithm proposed in 2023. Codellama has the knowledge cutoff in Sep 2022. So when using codellama for documentation generation, without accessing mathematical intuition and description of DPO from the open internet, codellama will not possible to write helpful documentation that describe the intuition behind the implementation of DPO.

C Scarcity of Code Documentation

We analyzed 164 top-starred Python repositories (created after January 1, 2025), encompassing 13,314 files and 115,943 documentable nodes (functions, classes, and methods). Of these nodes, only 27.28% contained any documentation, with 66.46% of repositories exhibiting less than 30% coverage (Figure 5). Furthermore, 62.25% of repositories averaged 30 words or fewer per documentation block (Figure 6), while only 3.98% exceeded an average of 100 words, illustrating the widespread brevity and overall scarcity of code documentation.

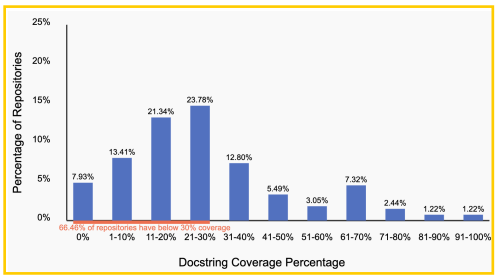

Figure 5: Distribution of repositories by code documentation coverage.

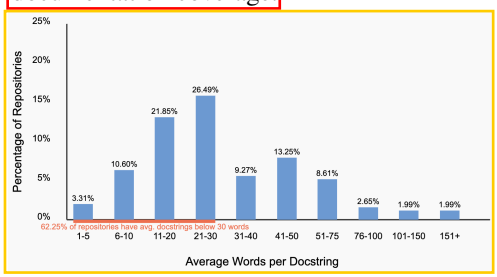

Figure 6: Distribution of repositories by average words per documentation.

D Data

We gathered 164 top-stared Python repositories from GitHub, each created after January 1, 2025, having more than 50 stars, and exceeding 50 KB in size. From this corpus, we selected 9 repositories reflecting the overall distribution in terms of lines of code and topological complexity. Figure 7 shows the selected repositories (red points) overlaid on the broader distribution. Eventually, we collected 366 code components (120 functions, 178 methods, and 68 classes) for evaluation, with a separate subset of 50 distinct code components (randomly sampled from the full set) used specifically for our truthfulness ablation study.

E Robust LLM-as-judge

Assessing the qualitative aspects of Helpfulness automatically is inherently challenging due to subjec-

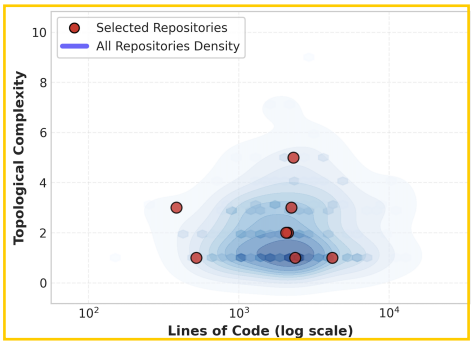

Figure 7: Distribution of repositories used for docstring generation evaluation.

tivity. We employ an LLM-as-judge approach, but incorporate rigorous mechanisms inspired by existing work to enhance reliability and consistency, mitigating known issues like positional bias or variability (Wang et al., 2024; Zhuge et al., 2024): **Decomposed Evaluation:** Instead of a single holistic judgment, the LLM evaluates distinct parts of the docstring (e.g., summary, parameter descriptions, overall description) separately, using tailored prompts for each part (Liu et al., 2023a; Lee et al., 2024). **Structured Prompting:** Each prompt provides the LLM with:

- *Explicit Rubrics:* Detailed criteria defining expectations for different levels on a 5-point Likert scale (1=Poor to 5=Excellent) concerning clarity, detail, and utility specific to the docstring part being evaluated (Kim et al., 2023; Zhang et al., 2023a).
- *Illustrative Examples:* Few-shot examples demonstrating good and bad documentation snippets corresponding to different score levels, grounding the rubric criteria (Zheng et al., 2023; Chiang and Lee, 2023).
- *Chain-of-Thought Instructions:* Guiding the LLM to first analyze the code, then compare the corresponding docstring section against the rubric, justify its rating step-by-step, and identify specific strengths or weaknesses (Liu et al., 2023b; Zheng et al., 2023).
- *Standardized Output Format:* Requiring the LLM to output its rating along with detailed justifications in a structured format (e.g., JSON), facilitating aggregation and analysis while ensuring the LLM adheres to the evaluation protocol (Liu et al., 2023b; Lee et al., 2024; Krumdick et al., 2025).

This structured LLM-as-judge approach aims to provide a scalable yet nuanced assessment of the documentation's practical value to developers.

F More System Screenshots

Figure 8 shows the configuration page before initiating the code documentation generation process. The page mainly consists of three parts: the target repository path, LLM configuration, and flow control (for the orchestrator).

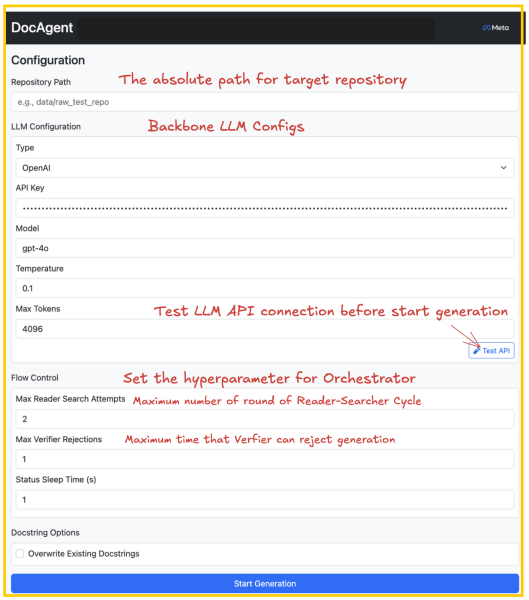

Figure 8: Screenshot of the configuration page.

Figure 9 displays the window that appears after clicking the "Evaluate" button. Since using an LLM as a judge is costly (consuming approximately 500 tokens per evaluation), this feature is optional in the web UI. Only when the user clicks the "Evaluate" button will the evaluation be triggered, after which the button changes to display the generated score.

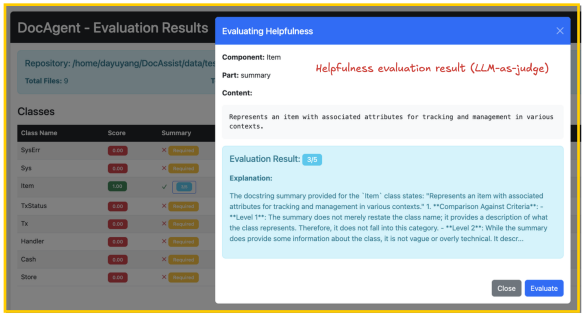

Figure 9: Screenshot of the helpfulness evaluation window.