Remember Your Trace: Memory-Guided Long-Horizon Agentic Framework for Consistent and Hierarchical Repository-Level Code Documentation

Suyoung Bae[1], Jaehoon Lee[1], Changkyu Choi[3,4] YunSeok Choi[2]*, Jee-Hyong Lee[1]*
[1] College of Computing and Informatics, Sungkyunkwan University, South Korea
[2] Department of Applied Artificial Intelligence, Sungkyunkwan University, South Korea
[3] Department of Informatics, University of Oslo, Norway
[4] Department of Game Development, University of Inland Norway, Norway

Abstract

Automated code documentation is essential for modern software development, providing the contextual grounding that both human developers and coding agents rely on to navigate large codebases. Existing repository-level approaches process components independently, causing redundant retrieval and conflicting descriptions across documents while producing outputs that lack hierarchical structure. Therefore, we propose **MemDocAgent**, a long-horizon agentic framework that generates documentation within a single, integrated context spanning the entire repository. It combines two components: (i) *Dependency-Aware Traversal Guiding* that predetermines a traversal order respecting dependency and granularity hierarchies; (ii) *Memory-Guided Agentic Interaction*, in which the agent interacts with RepoMemory, a shared memory accumulating prior work traces through read, write, and verify operations. Through an in-depth multi-criteria evaluation, MemDocAgent achieves the best performance over both open- and closed-source baselines and demonstrates practical applicability in real software development workflows. [1]

1 Introduction

Repository-level documentation requires understanding over an entire codebase to generate coherent natural-language descriptions that span individual function behaviors, dependency structures, and overall architectural patterns [1–3]. As software repositories scale in size and complexity, such documentation serves as a critical interface for both human developers and coding agents, providing the contextual grounding needed for code navigation and collaboration [4–6]. Manual writing at this scale is prohibitively expensive, motivating a new line of work on automating repository-level documentation [7–12].

Automating this task is inherently a long-horizon: each document depends not only on the local code being summarized, but also on information retrieved, interpreted, and written in earlier documentation steps. However, existing repository-level documentation systems, ranging from open-source frameworks [1–3] to closed-source products [13, 14], decompose a repository into components and document each component independently, *treating each as a self-contained task without shared state across the repository*. As a result, the same source files are repeatedly retrieved across iterations, with an average overlap of 50% across existing systems as shown in Figure 1. More critically, prior documentation outputs are not reused across iterations, leaving the system unable to detect dependency conflicts. This leads to contradictory descriptions of the same entity or architectural logic across

*Corresponding authors
[1] Code and Data are available at `https://github.com/bsy99615/MemDocAgent`.

Preprint.

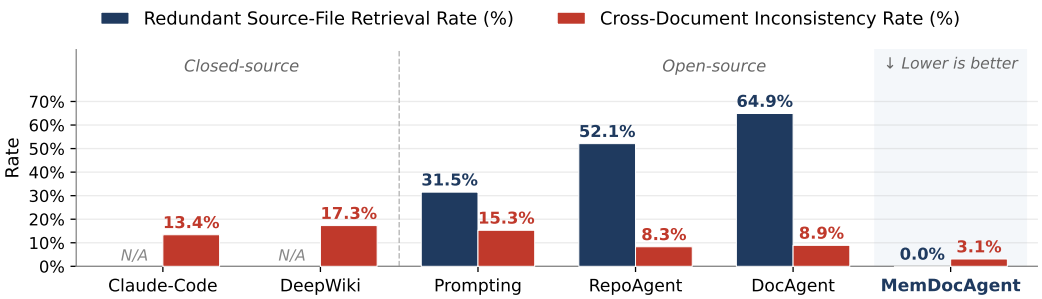


Figure 1: Limitations in existing documentation systems. While existing systems suffer from substantial repeated source-file retrieval and cross-document inconsistencies, MemDocAgent eliminates repeated retrieval and reduces inconsistency by 75.5%. *N/A* denotes closed-source systems whose internal pipelines are inaccessible, so repeated retrieval cannot be measured. The cross-document inconsistency rate is computed post hoc over the final document set by a separate LLM-judge (Claude Haiku 4.5), applied identically to all systems: it is the fraction of dependency-linked document pairs whose descriptions the judge deems contradictory.

documents, resulting in an average cross-document inconsistency rate of 13% across existing systems as shown in Figure 1.

Beyond these procedural limitations, existing methods *lack hierarchical structure in their documentation outputs*. High-quality repository documentation should provide a multi-level view of a codebase, connecting fine-grained component behavior, module-level responsibilities, and repository-wide architectural design into a coherent whole [15]. However, existing methods typically document only a narrow slice of this hierarchy: component-level approaches generate localized docstrings but overlook architectural context [1, 3], whereas repository-level approaches summarize global structure but omit fine-grained implementation details [2]. Consequently, the resulting documentation is either too local to explain system-level design or too coarse to support concrete code understanding. This hierarchical gap reinforces the need for a long-horizon documentation process that can accumulate, preserve, and connect information across many interdependent documentation steps.

Existing systems treat repository documentation as short-horizon, component-local tasks, whereas the task requires accumulating and reconciling repository-wide knowledge across many documentation steps. We therefore formulate repository-level documentation as a *long-horizon agentic problem* [16], where a single agent maintains persistent state, reuses prior retrievals and outputs, and builds a coherent hierarchical documentation of the codebase. To this end, we propose **MemDocAgent**, a long-horizon agentic framework in which a single agent reasons over the entire repository within one continuous trajectory, with persistent access to its accumulated retrievals and outputs. Along this trajectory, the agent produces hierarchical documentation spanning the component, module, and repository levels. This design is supported by two core components: *Dependency-Aware Traversal Guiding* and *Memory-Guided Agentic Interaction*.

Dependency-Aware Traversal Guiding predetermines a traversal order that respects the dependency relations and the granularity hierarchy, and provides it as guidance to the agent to ensure hierarchical structure and coverage completeness. In *Memory-Guided Agentic Interaction*, the agent interacts with RepoMemory, a shared memory that accumulates prior work traces such as retrieved components, intermediate reasoning, and generated documentation throughout the trajectory. Crucially, RepoMemory is not only a database that passively accumulates generated documentation, but also an active manager that governs the documentation process. Beyond maintaining hierarchical completeness, it serves as a consistency-enforcement mechanism, actively keeping interdependent documents consistent with one another.

To validate the effectiveness of MemDocAgent, we comprehensively evaluate it against existing baselines along four dimensions: *completeness*, *truthfulness*, *helpfulness*, and *information sufficiency*. Existing metrics capture coverage, accuracy, and usefulness, but overlook implementation-level sufficiency for practical code understanding. We therefore introduce information sufficiency as a quantitative metric that measures whether the generated documentation alone provides enough information to reproduce the original source code. Empirically, MemDocAgent achieves the best

performance across all four criteria, demonstrating that its documentation captures both the breadth and the depth required to support real software development workflows.

2 Related work

Long-horizon coding agents. Recent coding agents have advanced from solving isolated programming tasks to executing extended trajectories with numerous sequential tool calls and reasoning steps [17–27]. This long-horizon regime builds on agentic foundations such as ReAct [28], Reflexion [29], and multi-agent frameworks [30–32]. A key challenge is that working memory accumulates across turns and exceeds the context window, causing agents to lose track of prior objectives or propagate errors throughout the trajectory [33–36].

To handle long contexts, prior work has explored external memory systems that maintain and dynamically organize context outside the model [37–46], along with summarization and compression based techniques for context management [47–49]. However, these methods are designed for single output settings where memory supports a single final answer, and fall short when multiple sequentially generated outputs must remain mutually consistent. We instead reformulate repository documentation as generating mutually consistent documents over one cumulative trajectory, where memory must preserve cross-document consistency rather than support a single answer.

Repository-level code documentation. Early code documentation methods focused on function-level summarization, training models to produce short natural language descriptions for individual code units [50–54]. With the emergence of LLMs, this scope expanded to summarization at the class and project levels [55–61], and recent systems have begun to operate over entire repositories. RepoAgent [3] extracts dependency relations and generates component-level docstrings in topological order through an LLM pipeline, while DocAgent [1] adopts a multi-agent pipeline for the same component-level objective. CodeWiki [2], in contrast, hierarchically decomposes a repository and recursively synthesizes documentation via a multi-agent pipeline targeting repository-level abstractions.

While these methods successfully extend documentation generation to the repository scale, they treat each component as an independent unit, thereby failing to maintain cross-document consistency or dependency-aware reasoning across the long-horizon documentation trajectory. This per-component processing leads to redundant retrievals and inter-document contradictions that undermine the practical utility of the resulting documentation.

3 Preliminaries

Repository-level hierarchical documentation. Given a repository $\mathcal{R}$ with a hierarchical directory structure and multiple source files, our task is to produce a hierarchical documentation set $\mathcal{D}$ over three granularities: *components* (functions, methods, classes), *modules* (directories), and the *repository* itself. The corresponding documentation units are:

$$\mathcal{U} \;=\; \{u_i^{(c)}\}_{i=1}^{N_c} \;\cup\; \{u_j^{(m)}\}_{j=1}^{N_m} \;\cup\; \{u^{(r)}\}, \tag{1}$$

Where N_c and N_m are the numbers of components and modules, and the superscripts (c), (m), (r) indicate the granularity level, respectively. The system processes every $u \in \mathcal{U}$ to generate a corresponding document $d \in \mathcal{D}$, yielding $|\mathcal{D}| = N_c + N_m + 1$ documents in total.

Long-horizon reasoning trajectories. We model the documentation generation process as a long-horizon reasoning trajectory, formally defined as $\tau = (t_0, a_0, o_0, t_1, a_1, o_1, \ldots, t_T, a_T, o_T)$, where each turn i comprises a thought t_i, an action a_i, and an observation o_i following the ReAct pattern [28], and T denotes the total number of turns in the trajectory. In repository-level documentation generation, a single trajectory typically spans over a thousand turns as the agent processes hundreds of components, dozens of modules, and the repository itself.

4 MemDocAgent

MemDocAgent frames repository-level documentation as a cumulative long-horizon process, where a single agent processes every documentation unit within one continuous trajectory and reuses

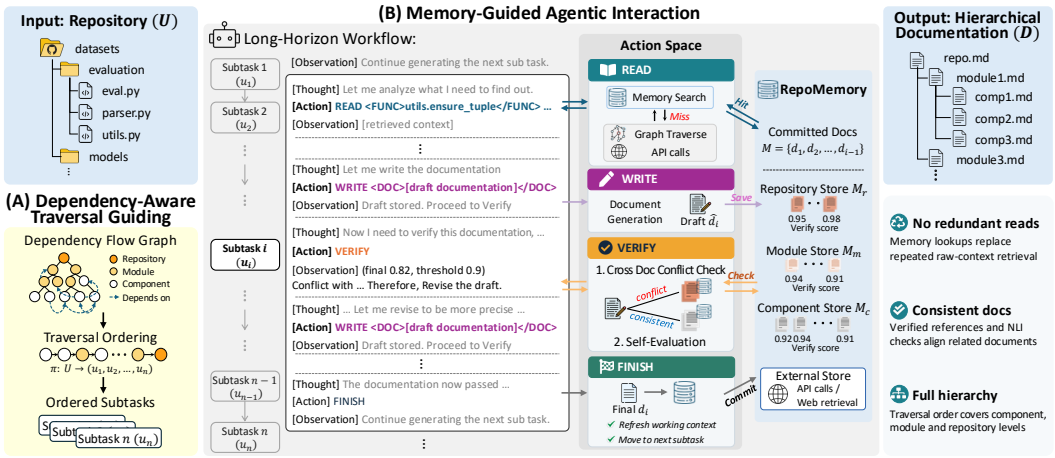


Figure 2: Overview of MemDocAgent. (A) *Dependency-aware traversal guiding* first computes a traversal order that respects both dependency relations and the granularity hierarchy. Following this order, (B) *memory-guided agentic interaction* treats each unit as a multi-turn sub-task, where the agent interacts with RepoMemory through READ, WRITE, and VERIFY to generate d_i and commits the verified document upon FINISH. The local context is refreshed before moving to the next sub-task, and this loop repeats until all units are documented, producing the final hierarchical documentation.

information accumulated along the way. This design is supported by two key components. First, *dependency-aware traversal* (§4.1) that orders the trajectory so that each unit is documented after its dependencies and child units, ensuring both context-grounded generation and repository coverage. Second, *memory-guided agentic interaction* (§4.2), where the agent interacts with a shared external memory to retrieve, reuse, and update information across sub-tasks. Figure 2 illustrates our framework.

4.1 Dependency-aware traversal guiding

To structurally guarantee the completeness of $\mathcal{D}$ for the target repository, we predetermine a traversal order $\pi = (u_1, \ldots, u_N)$ over $\mathcal{U}$ and injects it into the agent as sequential sub-tasks. We first construct a directed dependency graph $G = (V, E)$ over $\mathcal{R}$, where each node $v \in V$ corresponds to a documentation unit and a directed edge $(u, v) \in E$ indicates that u depends on v.

Specifically, we traverse the abstract syntax tree (AST) [62] of each file to collect code entities (functions, methods and classes) as nodes, and connect them with directed edges representing their relations (function calls, class inheritance, attribute access and imports). Each node stores its AST type and code context, and each edge stores the dependency relation as metadata.

Unlike DocAgent [1], which uses a topological order reflecting only component-level dependencies, π jointly satisfies two constraints: (1) component-level dependency order, where u_i precedes u_j whenever u_j depends on u_i; and (2) granularity-level hierarchical order, where each module is processed only after all of its components and sub-modules are documented, and the repository unit is processed only after all modules are complete. By jointly satisfying both, π ensures that the agent always processes a unit with all of its dependent and child units already documented. Since code dependencies may contain cycles, we collapse each strongly connected component into a single super-node using Tarjan's SCC algorithm [63] before topological ordering. Detailed graph construction and the traversal order algorithm are provided in Appendix A.1.

4.2 Memory-guided agentic interaction

To ensure both retrieval efficiency and cross-document consistency throughout the long-horizon trajectory, this section introduces a centralized shared memory, RepoMemory, and designs an interaction mechanism in which every action of the agent interacts with this memory.

Table 1: The structure of RepoMemory.

Memory store		Key	Value
Documentation store	component_store	component_id	path, document, claims, depends_on, source code, type, verification_score
	module_store	module_path	path, document, claims, child_units, verification_score
	repo_store	repo_name	path, document, claims, child_units, verification_score
External store	search_cache	query	search result

4.2.1 RepoMemory

RepoMemory consists of a *Documentation store* and an *External store* as shown in Table 1. The documentation store organizes generated documents by hierarchy level, storing component, module, and repository-level records with their identifiers, paths, documents, decomposed claims, verification scores, source code and type (function, method, class) for components, child units for module and repository units. The external store caches search results by query to avoid redundant retrievals during the trajectory.

4.2.2 Action space

Unlike prior repository-level documentation systems whose actions are executed independently, every action in MemDocAgent is grounded in interaction with RepoMemory: READ retrieves context from RepoMemory, WRITE produces a document using the retrieved context, VERIFY cross-validates the generation against documents already committed to RepoMemory, and FINISH commits the verified document to RepoMemory. At each turn, the agent observes the result of its previous action and autonomously selects the next action among these four operations to complete the current documentation sub-task. All instructions for each action are defined in the agent's system prompt (Appendix A.2 for the full system prompt and Appendix A.4 for trajectory examples).

READ: Memory-guided adaptive retrieval. Through READ, the agent adaptively decides whether additional context is needed for each sub-task u_i. If so, it calls the retrieval tool with two types of structured requests: *internal requests* that specify related component identifiers within the repository, and *external requests* that pose natural-language queries about external algorithms or libraries.

The retrieval tool first checks RepoMemory by key: component IDs for internal requests and query strings for external requests. If the key exists, the stored value is returned at $O(1)$ cost; otherwise, the tool retrieves the information from the codebase via the dependency graph $\mathcal{G}$ or from the external retrieval API, and stores it for later reuse. For module and repository-level sub-tasks, the retrieval tool additionally retrieves the documentation of all direct child units from the *Documentation Store*. Since the traversal order π guarantees that these children have already been documented, this step reuses existing documentation without additional codebase retrieval.

WRITE: Hierarchical document generation. When the agent thinks that sufficient context has been collected, it performs WRITE to generate a draft document $\hat{d}_i$. The generation is guided by a granularity-specific format, defined in the agent's system prompt, that specifies the structure and required content of the output for each granularity.

VERIFY: Self-evaluation and cross-document conflict verification. Repository-level documentation often describes the same functionality from multiple perspectives across files, so without verification these descriptions easily drift apart and degrade overall consistency. Therefore, VERIFY assesses each draft $\hat{d}_i$ along two axes. First, a self-evaluation scores factual consistency, completeness, and helpfulness according to the agent's system prompt instructions (See Table 4 in Appendix A.2), and second, a cross-conflict verification scores inconsistencies between $\hat{d}_i$ and its dependency-related documents, each on $[0, 1]$.

For the cross-conflict verification, inspired by NLI-based inconsistency detection for text summarization [64–66], we decompose $\hat{d}_i$ into atomic claims via an LLM and verify each claim with a local NLI model against a trusted reference set from RepoMemory. This reference set contains only the dependency-related records whose verification score exceeds a threshold, so a low-quality document never becomes a conflict-checking criterion and early errors cannot propagate. The resulting contradiction rate serves as the cross-conflict score. $\hat{d}_i$ passes VERIFY only if the average of two verified

scores exceeds a threshold. Otherwise, the detected conflicts are returned as a conflict report in the next turn's observation, prompting the agent to issue an additional READ or to re-WRITE the draft. If $\hat{d}_i$ still fails to pass after the agent has reached its predefined maximum number of revisions, we commit the highest-scoring draft to RepoMemory to preserve coverage. More details are provided in Appendix A.3.

FINISH: Memory commit and trajectory continuation. Once $\hat{d}_i$ passes VERIFY, the agent finalizes it as d_i, invokes FINISH to commit d_i to the corresponding granularity store, and refreshes its local context for the next sub-task. This loop repeats until all units in $\mathcal{U}$ are documented, yielding the final hierarchical documentation set $\mathcal{D}$.

5 Experimental setups

5.1 Evaluation metrics

We evaluate MemDocAgent along four criteria. Beyond *completeness*, *truthfulness*, and *helpfulness*, which are established quality indicators, we further propose *information sufficiency*. This criterion quantifies whether the documentation contains enough information to reproduce source code that integrates and executes coherently within the repository. We outline the evaluation metrics below; full details are in Appendix B.1.

Completeness. Completeness measures the extent to which the generated documentation adheres to standard structural conventions (*Section Presence*) and includes essential components (e.g., function, class) expected for a given code element (*Entity Coverage*). The final completeness score is computed as the average of the section presence and entity coverage scores.

Section Presence Score. For each documentation, we measure the proportion of required sections that are present, using rule-based pattern matching. This score is a sanity check that verifies whether MemDocAgent actually produces the intended hierarchical structure with all prescribed sections.

Entity Coverage Score. We extract the set of core entities from the target source code via AST analysis, and measure the proportion of these entities that are referenced in the documentation.

Truthfulness. Truthfulness measures whether the generated documentation faithfully reflects the actual code, covering both factual correctness (*Consistency*) and contextual relevance (*Relevance*). We adopt a fine-grained documentation evaluation framework that decomposes generated documentation and evaluates each segment using predefined evaluation criteria [67, 68] to compute these two scores. The final truthfulness score is computed as the product of the consistency and relevance scores.

Consistency Score. We measure the proportion of claims in the documentation that are factually supported by the source code, identified at fine-grained segment level.

Relevance Score. We measure the proportion of documentation content that pertains to the source code, filtering out off-topic or extraneous descriptions.

Helpfulness. We assess the semantic quality and practical utility of each documentation through an LLM-as-a-judge protocol following DocAgent [1]. The judge rates each required section on a 5-point Likert scale using a section-specific rubric, and the normalized section scores are averaged into a single helpfulness score.

Information Sufficiency. Surface-level documentation quality alone does not capture whether the documentation is actually usable in practice. Motivated by the real-world scenario where documentation serves as a context tool for coding agents, we introduce an execution-based metric that measures whether the documentation contains sufficient information to support practical code reconstruction. We operationalize this by measuring code regeneration performance: if a model can regenerate the original component using only the documentation and the function signature, the documentation is likely to preserve the behavior, dependencies, and implementation details needed for downstream use.

To evaluate information sufficiency, we collect 564 component-level test cases from DevEval [69]. The evaluation proceeds as follows: (1) remove each target function body while keeping its signature;

Table 2: Main comparison results. (a) Overall results on average completeness, truthfulness, and helpfulness and (b) the same metrics separated by documentation granularity. Within each backbone group, the best result is **bold** and the second-best underlined. * marks in section scores set to 1.0 by default, as their internal format instruction is inaccessible. "–" denotes non-applicable granularities.

(a) Overall results on average completeness, truthfulness, and helpfulness.

Backbone	Method	#Doc	Completeness			Truthfulness			Helpfulness
			Section Score	Coverage Score	Complete.	Consistency	Relevance	Truthful.	
Open-Source Baselines									
Qwen3-Coder	Prompting	3,323	0.838	0.837	0.838	0.809	0.803	0.649	0.458
	RepoAgent	3,650	0.659	0.967	0.813	0.910	0.918	0.835	0.391
	DocAgent	3,198	0.792	0.898	0.845	0.900	0.889	0.800	0.556
	CodeWiki	80	0.770	0.473	0.622	0.906	0.852	0.772	0.628
	MemDocAgent	3,323	**0.959**	**0.998**	**0.979**	**0.968**	**0.946**	**0.916**	**0.690**
GPT-5-mini	Prompting	3,323	0.790	0.473	0.632	0.883	0.840	0.741	0.617
	RepoAgent	3,650	0.555	**1.000**	0.778	0.898	0.947	0.850	0.449
	DocAgent	3,198	0.776	0.944	0.860	0.898	0.925	0.831	0.671
	CodeWiki	80	**0.975**	0.579	0.777	0.847	0.771	0.653	0.708
	MemDocAgent	3,323	0.916	**1.000**	**0.958**	**0.983**	**0.968**	**0.952**	**0.800**
Closed-Source Baselines									
Claude-Code (`/init`)		20	1.000*	0.488	0.744	0.845	0.869	0.734	0.696
DeepWiki		404	1.000*	0.840	0.920	0.815	0.935	0.762	0.755

(b) Results separated by documentation granularity (component-, module-, and repository-level).

Backbone	Method	Component-Level			Module-Level			Repository-Level		
		Complete.	Truthful.	Helpful.	Complete.	Truthful.	Helpful.	Complete.	Truthful.	Helpful.
Open-Source Baselines										
Qwen3-Coder	Prompting	0.886	0.694	0.577	0.870	0.666	0.313	0.758	0.587	0.483
	RepoAgent	0.813	0.835	0.391	–	–	–	–	–	–
	DocAgent	0.845	0.800	0.556	–	–	–	–	–	–
	CodeWiki	–	–	–	0.630	0.769	0.606	0.613	0.775	0.650
	MemDocAgent	**0.987**	**0.962**	**0.691**	**0.975**	**0.902**	**0.702**	**0.975**	**0.883**	**0.677**
GPT-5-mini	Prompting	0.631	0.761	0.627	0.626	0.766	0.626	0.640	0.696	0.598
	RepoAgent	0.778	0.850	0.449	–	–	–	–	–	–
	DocAgent	0.860	0.831	0.671	–	–	–	–	–	–
	CodeWiki	–	–	–	0.809	0.597	0.737	0.744	0.708	0.679
	MemDocAgent	**0.952**	**0.974**	**0.750**	**0.974**	**0.924**	**0.858**	**0.949**	**0.959**	**0.791**

(2) provide the signature and documentation context to a code-generation model; (3) regenerate the missing function body; and (4) reinsert the generated code into the repository and evaluate it with functional correctness (*Pass@k*) and structural similarity (*CodeBLEU*).

5.2 Baselines

We compare MemDocAgent against four open-source and two closed-source systems. Details are in the Appendix B.2.

Open-source baselines. RepoAgent [3], DocAgent [1], and CodeWiki [2] are agent-based systems for repository-level documentation generation. We additionally include *Prompting*, a baseline that prompts the LLM to document each target unit independently, with first-hop dependency context.

Closed-source baselines. DeepWiki [13] is a commercial system widely used in industry for automated repository-level documentation. We also include Claude-Code [14], treating the `README.md` produced by its `/init` command as the repository-level documentation.

5.3 Implementation details

To evaluate the effectiveness of MemDocAgent on repository-level documentation generation, we select 20 Python repositories from DevEval [69] via stratified sampling, applying three criteria: (i) balanced coverage across the 10 domains in DevEval, (ii) the presence of unit tests to enable information sufficiency evaluation, and (iii) substantial structural depth and dependency complexity at the repository level. Detailed statistics are provided in Appendix B.3.

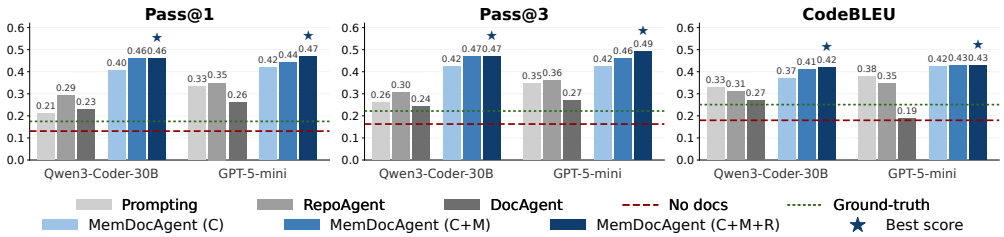


Figure 3: Information sufficiency comparison results. Pass@1, @3 and CodeBLEU when regenerating code from each method's documentation alone, averaged over two code generators, with the x-axis denoting the backbone used for each open-source baseline.

Our contribution is designing a novel pipeline for repository-level code documentation; the question we aim to answer is which pipeline design yields better documentation. To isolate this effect, we compare all baselines with the same backbone LLMs and hyperparameters and evaluate them under identical settings. We implement MemDocAgent with two backbone LLMs: Qwen3-Coder-30B-A3B (open-source from HuggingFace) and GPT-5-mini (closed-source from OpenAI). Each repository is documented with a maximum of 10 steps per sub-trajectory, up to 2 revisions per unit, and a verify threshold of 0.9. For evaluation, we use Claude Haiku 4.5 as the LLM judge for truthfulness and helpfulness, and GPT-4o-mini and Claude Haiku 4.5 as code generators for evaluating information sufficiency. To mitigate self-preference bias, we deliberately use evaluator models that differ from any backbone used by MemDocAgent or the baselines. More implementation details are in Appendix B.4.

6 Experimental results

6.1 Comparisons with baselines

Documentation quality comparisons. Table 2 presents a performance comparison between MemDocAgent and existing baselines. Note that the number of generated documents (#Doc) differs because baselines use different documentation granularities and targets. Table 2a shows that MemDocAgent achieves the best overall performance across both open-source backbones. With Qwen3-Coder, it improves completeness, truthfulness, and helpfulness to 0.979, 0.916, and 0.690, outperforming the strongest baselines by 15.9%, 9.7%, and 9.9%, respectively. Moreover, existing baselines exhibit clear granularity trade-offs. RepoAgent attains high completeness but lower helpfulness, whereas CodeWiki produces concise repository-level documents but misses fine-grained details. In contrast, MemDocAgent generates hierarchical documentation with shared memory, improving coverage, consistency, and utility over both open and closed-source systems.

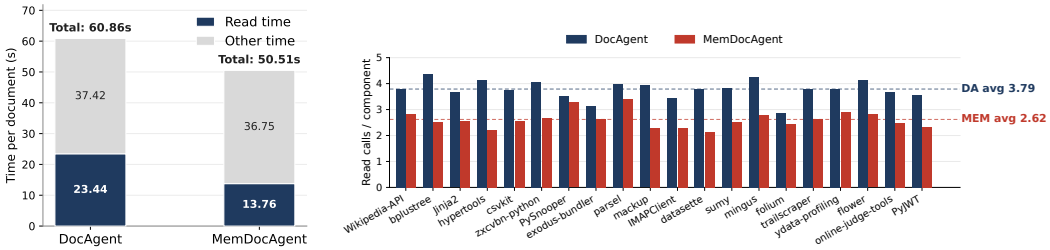
To ensure a fair comparison within each documentation granularity, we also report results broken down by granularity in Table 2b. The result shows that while existing baselines cover only parts of this hierarchy, MemDocAgent consistently maintains high completeness, truthfulness, and helpfulness across all levels. Per-repository results, further scalability and generalization analysis are provided in Appendix C.1, C.2, and C.3, respectively.

Information sufficiency comparisons. We measure whether documentation alone provides sufficient information to reproduce the original code by feeding the documentation as the only context to two generation models (GPT-4o-mini and Claude Haiku 4.5) and evaluating Pass@1, Pass@3, and CodeBLEU against ground-truth code. To examine whether progressively richer hierarchical context helps the model regenerate code that integrates coherently and executes correctly within the repository, we evaluate three MemDocAgent variants that progressively widen the supplied context: *(C)* the target component document only; *(C+M)* the component with its direct parent module document; and *(C+M+R)* the component with parent module and repository document. We additionally include two reference settings: *No docs* (signature only) and *Ground-truth* (signature with the human-written component docstring obtained from DevEval [69]).

Figure 3 shows the evaluation result (see Appendix C.4 for full result.). Across both backbones, every existing baseline surpasses the *No docs* setting, confirming that automatically generated documentation provides useful signal for code reproduction. Among them, MemDocAgent achieves the strongest

Table 3: Ablation study on key components of MemDocAgent. All ablations are conducted under the Qwen3-Coder backbone. Values in parentheses indicate the drop in percentage points relative to ours.

	Documentation Quality			Information Sufficiency		
Method	Completeness	Truthfulness	Helpfulness	Pass@1	Pass@3	CodeBLEU
MemDocAgent	**0.979**	**0.916**	**0.690**	**0.461**	**0.467**	**0.419**
w/o think	0.917 (−6.2)	0.777 (−13.9)	0.583 (−10.7)	0.312 (−14.9)	0.319 (−14.8)	0.225 (−19.4)
w/o memory system	0.903 (−7.6)	0.768 (−14.8)	0.593 (−9.7)	0.324 (−13.7)	0.333 (−13.4)	0.190 (−22.9)
w/o conflict verify	0.900 (−7.9)	0.781 (−13.5)	0.592 (−9.8)	0.282 (−17.9)	0.290 (−17.7)	0.233 (−18.6)



(a) Average read time across 20 repositories.

(b) Read calls per component across 20 repositories. Dashed lines mark the overall averages.

Figure 4: Efficiency comparison between DocAgent (previous SOTA) and MemDocAgent.

performance across all settings. Generated documentation can score higher than the *Ground-truth* reference because human-written docstrings are intended as short and concise summaries, whereas the baselines and our method produce richer descriptive documentation.

Compared among three MemDocAgent variants, performance increases monotonically as granularity levels are added, with *(C+M+R)* attaining the best score in every setting. That higher-level documents consistently help indicates that our framework produces cross-level consistent documentation whose repository-wide context and architectural information enable the model to regenerate code that executes correctly within the repository.

6.2 Ablation study

We conduct an ablation study to assess the contribution of each proposed component. In Table 3, we compare with three ablation settings. *w/o think* removes the agent's think step before each action, *w/o memory system* replaces RepoMemory with a stateless cache that is cleared after each documentation unit, and *w/o conflict verify* disables cross-document conflict verification. The result shows that removing any single component causes substantial drops across all metrics, confirming that every component is essential. The three components affect different aspects of documentation. Together, ReAct-style reasoning, RepoMemory, and conflict verification each contribute a complementary dimension of quality, and only their combination yields MemDocAgent's full hierarchical documentation capability. We also conduct further ablations: a comparison of memory architectures (Appendix C.5), an analysis of the effectiveness of traversal order (Appendix C.6), prompt-controlled comparisons (Appendix C.7), and an analysis of hyperparameters (Appendix C.8).

6.3 Efficiency analysis

As shown in Figure 4a, the effect of eliminating redundant source-file retrieval is consistently observed in retrieval cost. DocAgent performs on average 23.44 seconds, whereas MemDocAgent completes the same task in 13.76 seconds, reducing read time by 41%. Figure 4b confirms that this gain is not driven by a few favorable repositories, as MemDocAgent requires fewer read calls per component in all 20 repositories (3.79 → 2.62 on average). We attribute this consistent improvement to RepoMemory, which replaces repeated raw-context reads with lookups of verified dependency documents. Further time and cost analyses are reported in Appendix C.9. Together, these results show that MemDocAgent achieves the broadest documentation coverage while remaining more time and cost efficient than DocAgent.

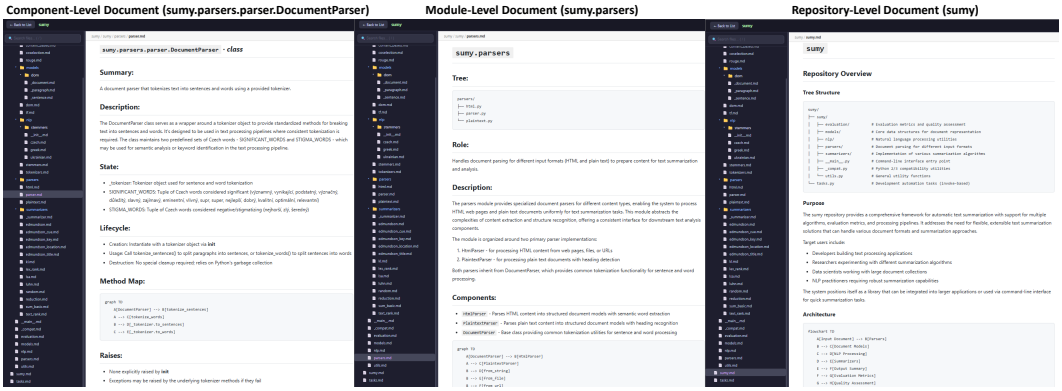


Figure 5: Example hierarchical documentation generated by MemDocAgent for the *Sumy* repository, illustrating the three granularity levels (left to right). Each captures granularity-appropriate content, jointly forming a coherent technical from implementation details to high-level architectural patterns.

6.4 Case study

As illustrated in Figure 5, MemDocAgent generates hierarchical documentation across component-, module-, and repository-levels, ensuring each granularity captures context-appropriate content that ranges from implementation details to high-level architectural patterns. Additional examples are provided in Appendix C.10.

6.5 Effectiveness of VERIFY attempts

Figure 6 shows how self-evaluation and cross-document conflict scores change over repeated VERIFY attempts. 63.8% of components pass on the first attempt with an average score of approximately 0.97, while the remaining 24.4% and 11.3% are resolved within two and three attempts, respectively, with documentation quality improving at each additional attempt. When a component receives a low score at the initial verification, MemDocAgent autonomously refines it, leading to higher self-evaluation and cross-document consistency scores in subsequent attempts and thereby ensuring accurate and consistent documentation. These results demonstrate that the VERIFY action effectively improves documentation quality through targeted re-verification, applied only to components with detected conflicts or self-evaluation failures.

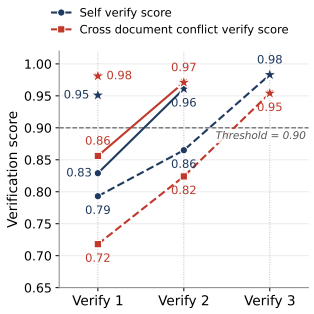


Figure 6: Documentation improvements across VERIFY attempts.

7 Conclusion

We introduced MemDocAgent, a long-horizon agentic framework for repository-level documentation in which a single agent reasons over the entire repository by combining *dependency-aware traversal guiding* and *memory-guided agentic interaction*. MemDocAgent preserves repository-wide context across hierarchical documentation steps, enabling consistent documentation from component-level implementation details to repository-level architecture. Across evaluations, MemDocAgent outperformed existing baselines, improving completeness, truthfulness, helpfulness, and information sufficiency. These results demonstrated the ability of our framework to produce documentation that is more useful for developers and coding agents in real software development workflows.

Limitation

Operating over long agentic trajectories introduces several considerations worth discussing. First, long-horizon agentic documentation task requires computational resources because both input context and generated outputs can grow large over the trajectory. To prevent context explosion and excessive

runtime, we impose a maximum trajectory length, which can cause rare failures where the agent terminates before completing a sub-task. Second, because the agent depends on the model's reasoning ability, it may occasionally enter a repeated loop without completing the sub-task. Although the overall impact is minimal, with only 0.08% of components terminating without meeting the verification threshold in our experiments, this limitation should still be considered (Failure analysis in Appendix A.3.4). Future work should explore more efficient trajectory control, adaptive stopping criteria, and stronger recovery mechanisms. Third, beyond the agent's runtime behavior, our evaluation itself has a scope limitation. We evaluate at the component level information sufficiency because DevEval's human-curated unit tests exist only at the function level, enabling objective, execution-based verification via Pass@k; this is a deliberate choice for metric rigor. We acknowledge it does not directly measure the standalone utility of higher-level documents, but our results offer indirect evidence: regeneration performance increases monotonically as context widens (Figure 3), showing that higher-level documents carry architectural information that materially aids regeneration.

AI Use Statement

We used generative AI tools in the following ways: (i) code development, including implementation of the proposed method, baselines, and experiment scripts; (ii) translation and text editing, for improving clarity and consistency of the manuscript; and (iii) figure and table preparation, including generating plotting scripts and formatting tables. All AI-generated code was reviewed and tested by the authors. All figures, tables, and reported numbers were checked against the raw experimental outputs, and all references were verified by the authors against their original sources. The AI tools did not serve as authors. The authors take full responsibility for the entire content of this paper, including all text, claims, figures, tables, references, and artifacts produced with the aid of generative AI.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (RS-2019-II190421, 15%; IITP-2026-RS-2024-00360227, 15%), National IT Industry Promotion Agency(NIPA) grant funded by the Korea government(MSIT) (No.RS-2026-25621569, 14%), National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2025-03522980, 14%; RS-2025-005 21391, 14%; RS-2025-25433088, 14%) and Research Council of Norway (RCN) through the MishMash Centre for AI and Creativity (357438, 14%).

References

[1] Dayu Yang, Antoine Simoulin, Xin Qian, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, and Grey Yang. DocAgent: A multi-agent system for automated code documentation generation. In Pushkar Mishra, Smaranda Muresan, and Tao Yu, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL) (Volume 3: System Demonstrations)*, pages 460–471. Association for Computational Linguistics, 2025.

[2] Anh Nguyen Hoang, Minh Le-Anh, Bach Le, and Nghi DQ Bui. Codewiki: Evaluating ai's ability to generate holistic documentation for large-scale codebases. *arXiv preprint arXiv:2510.24428*, 2025.

[3] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In Delia Irazu Hernandez Farias, Tom Hope, and Manling Li, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations*, pages 436–464. Association for Computational Linguistics, 2024.

[4] David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, 2010. URL `https://api.semanticscholar.org/CorpusID:38934599`.

[5] Golara Garousi, Vahid Garousi-Yusifoğlu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664–682, 2015. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2014.08.003. URL `https://www.sciencedirect.com/science/article/pii/S095058491400192X`.

[6] Lea Katalina Kivinen. Ai-driven chatbot as a support tool for developers during the onboarding process. 2023.

[7] David Lorge Parnas. Software engineering (extended abstract) an unconsummated marriage. *ACM SIGSOFT Software Engineering Notes*, 22(6):1–3, 1997.

[8] Andrew Forward and Timothy C Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, 2002.

[9] Sergio Cozzetti B De Souza, Nicolas Anquetil, and Káthia M De Oliveira. 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, 2005.

[10] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210, 2019. URL `https://api.semanticscholar.org/CorpusID:174800564`.

[11] Junji Zhi, Vahid Garousi-Yusifoğlu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 2015.

[12] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.

[13] Cognition AI. DeepWiki. `https://deepwiki.com/`, 2025.

[14] Anthropic. Claude Code. `https://www.anthropic.com/claude-code`, 2025.

[15] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. Evaluating usage and quality of technical software documentation: an empirical study. In *Proceedings of the 17th international conference on evaluation and assessment in software engineering*, pages 24–35, 2013.

[16] Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks. *arXiv preprint arXiv:2503.09572*, 2025.

[17] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[18] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

[19] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, 2024.

[20] Huy Nhat Phan, Tien N Nguyen, Phong X Nguyen, and Nghi DQ Bui. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*, 2024.

[21] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2188–2200. IEEE, 2025.

[22] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

[23] Haotian Luo, Huaisong Zhang, Xuelin Zhang, Haoyu Wang, Zeyu Qin, Wenjie Lu, Guozheng Ma, Haiying He, Yingsha Xie, Qiyang Zhou, et al. Ultrahorizon: Benchmarking agent capabilities in ultra long-horizon scenarios. *arXiv preprint arXiv:2509.21766*, 2025.

[24] Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, et al. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025.

[25] Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. *arXiv preprint arXiv:2512.22087*, 2025.

[26] Shuzheng Gao, Wenhao Zeng, Zhaojian Yu, Jianqiao Wangni, Chaozheng Wang, Kai Cai, Shilin He, and Michael R Lyu. Swe-mem: Learning adaptive memory management for long-horizon coding agents. *arXiv preprint arXiv:2606.28434*, 2026.

[27] Doyeon Kim, Suyoung Bae, Yumin Lee, and Jee-Hyong Lee. Aˆ 2agent: Action-aware reinforcement learning for repository-level code localization agents. *arXiv preprint arXiv:2608.29831*, 2026.

[28] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

[29] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

[30] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

[31] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 15174–15186, 2024.

[32] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

[33] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

[34] Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=3lm8lWYxiq.

[35] Guangya Wan, Mingyang Ling, Xiaoqi Ren, Rujun Han, Sheng Li, and Zizhao Zhang. Compass: Enhancing agent long-horizon reasoning with evolving context. *arXiv preprint arXiv:2510.08790*, 2025.

[36] Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

[37] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

[38] Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *CoRR*, abs/2310.08560, 2023. URL https://doi.org/10.48550/arXiv.2310.08560.

[39] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[40] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 19724–19731, 2024.

[41] Bernal J Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models. *Advances in neural information processing systems*, 37:59532–59569, 2024.

[42] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 2025.

[43] Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32779–32798, 2025.

[44] Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xiaodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, et al. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025.

[45] Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. Experepair: Dual-memory enhanced llm-based repository-level program repair. *Proceedings of the ACM on Software Engineering*, 3(FSE):3954–3976, 2026.

[46] Boshi Wang, Weijian Xu, Yunsheng Li, Xuemei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory. In *International Conference on Learning Representations*, volume 2026, pages 111266–111285, 2026.

[47] Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. Resum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

[48] Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

[49] Mo Li, LH Xu, Qitai Tan, Long Ma, Ting Cao, and Yunxin Liu. Sculptor: Empowering llms with cognitive agency via active context management. *arXiv preprint arXiv:2508.04664*, 2025.

[50] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *2013 21st International conference on program comprehension (ICPC)*, pages 23–32. IEEE, 2013.

[51] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1195. URL https://aclanthology.org/P16-1195/.

[52] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3931–3937, 2019.

[53] Yunseok Choi, Cheolwon Na, Hyojun Kim, and Jee-Hyong Lee. Readsum: retrieval-augmented adaptive transformer for source code summarization. *IEEE Access*, 11:51155–51165, 2023.

[54] Bibek Poudel, Adam Cook, Sekou Traore, and Shelah Ameli. Documint: Docstring generation for python using small language models. *arXiv preprint arXiv:2405.10243*, 2024.

[55] Junaed Younus Khan and Gias Uddin. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2022.

[56] Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey Nikolenko. ProConSuL: Project context for code summarization with LLMs. In Franck Dernoncourt, Daniel Preoţiuc-Pietro, and Anastasia Shimorina, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 866–880, Miami, Florida, US, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-industry.65. URL https://aclanthology.org/2024.emnlp-industry.65/.

[57] Vladimir Makharev and Vladimir Ivanov. Code summarization beyond function level. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 153–160. IEEE, 2025.

[58] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[59] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, pages 1–13, 2024.

[60] Demin Song, Honglin Guo, Yunhua Zhou, Shuhao Xing, Yudong Wang, Zifan Song, Wenwei Zhang, Qipeng Guo, Hang Yan, Xipeng Qiu, et al. Code needs comments: Enhancing code llms with comment augmentation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13640–13656, 2024.

[61] Liuwen Cao, Hongkui He, Hailin Huang, Jiexin Wang, and Yi Cai. Rethinking-based code summarization with chain of comments. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert, editors, *Proceedings of the 31st International Conference on Computational Linguistics*, pages 3043–3056, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.204/.

[62] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, 2019. doi: 10.1109/ICSE.2019.00086.

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

[64] Philippe Laban, Tobias Schnabel, Paul N Bennett, and Marti A Hearst. Summac: Re-visiting nli-based models for inconsistency detection in summarization. *Transactions of the Association for Computational Linguistics*, 10:163–177, 2022.

[65] Alessandro Scirè, Karim Ghonim, and Roberto Navigli. Fenice: Factuality evaluation of summarization based on natural language inference and claim extraction. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 14148–14161, 2024.

[66] Joonho Yang, Seunghyun Yoon, ByeongJeong Kim, and Hwanhee Lee. FIZZ: Factual inconsistency detection by zoom-in summary and zoom-out document. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 30–45, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.3. URL https://aclanthology.org/2024.emnlp-main.3/.

[67] Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 12076–12100, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.741. URL https://aclanthology.org/2023.emnlp-main.741/.

[68] Suyoung Bae, CheolWon Na, Jaehoon Lee, Yumin Lee, YunSeok Choi, and Jee-Hyong Lee. ReFEree: Reference-free and fine-grained method for evaluating factual consistency in real-world code summarization. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14352–14377, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.653. URL https://aclanthology.org/2026.acl-long.653/.

[69] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.214. URL https://aclanthology.org/2024.findings-acl.214/.

[70] Ram Rachum, Alex Hall, Iori Yanokura, et al. Pysnooper: Never use print for debugging again, jun 2019. URL https://github.com/cool-RR/PySnooper.

Appendix Contents

Appendix A: Additional details about MemDocAgent

- A.1 Algorithms of dependency-aware traversal guiding
- A.2 System prompts used in MemDocAgent
- A.3 Self-evaluation and cross-document conflict verification: detailed procedure
  - A.3.1 Self-evaluation score
  - A.3.2 NLI-based cross-document conflict evaluation score
  - A.3.3 Quantitative verification analysis
  - A.3.4 Analysis of verification-failure cases
- A.4 Agent trajectory examples on Pysnooper

Appendix B: Additional details about experimental setups

- B.1 Evaluations
- B.2 Baselines
- B.3 Datasets
- B.4 Implementation details

Appendix C: Additional experimental results

- C.1 Repository-wise experimental results
- C.2 Scalability analysis
- C.3 Cross-language generalization on large-scale repositories
- C.4 Details of information sufficiency
- C.5 Memory architecture ablation
- C.6 Traversal order ablation
- C.7 Prompt controlled comparisons
- C.8 Hyperparameter sensitivity
- C.9 Efficiency analysis
- C.10 More examples of generated documentation

A Additional details about MemDocAgent

A.1 Algorithms of dependency-aware traversal guiding

Algorithm 1 describes dependency graph construction, and Algorithm 2 presents the topological traversal used for hierarchical generation.

Algorithm 1 Building the unified dependency graph $\mathcal{G}$

Require: Components $\mathcal{C}$ with $\text{DEPENDS_ON}(c)$ and $\text{PATH}(c)$; repo id r
Ensure: Graph $\mathcal{G} : \mathcal{V} \to 2^{\mathcal{V}}$ over $\mathcal{V} = \mathcal{C} \cup \mathcal{M} \cup \{r\}$, where $u \to v$ means "u depends on / contains v"

 # Tier 1: component-level dependency edges
for $c \in \mathcal{C}$ **do**
  $\mathcal{G}[c] \leftarrow \text{DEPENDS_ON}(c)$ ▷ call, inheritance, attribute access, import
end for
 # Module hierarchy induced from file paths
Derive $\mathcal{M}$ from $\{\text{DIRNAME}(\text{PATH}(c)) : c \in \mathcal{C}\}$ and its prefix closure
$\text{CHILDREN}(m) \leftarrow \{\, c \in \mathcal{C} : \text{DIRNAME}(\text{PATH}(c)) = m \,\} \cup \{\, m' \in \mathcal{M} : \text{DIRNAME}(m') = m \,\}$
 # Tier 2: module-level containment edges
for $m \in \mathcal{M}$ **do**
  $\mathcal{G}[m] \leftarrow \text{CHILDREN}(m)$
end for
 # Tier 3: repository-level containment edges
$\mathcal{G}[r] \leftarrow \{\, v \in \mathcal{M} \cup \mathcal{C} : v$ has no parent module $\}$
return $\mathcal{G}$

Algorithm 2 Calculating dependency-aware traversal order π

Require: Unified graph $\mathcal{G}$ (Alg. 1)
Ensure: Order π such that (i) every component precedes its dependents, and (ii) every module follows its children, with r last

$\mathcal{G} \leftarrow \text{CONDENSESCC}(\mathcal{G})$ ▷ Tarjan [63]: collapse cycles into super-nodes
roots $\leftarrow \{v \in \mathcal{G} : v$ has no incoming edge$\}$
visited $\leftarrow \emptyset$; $\pi \leftarrow [\,]$
function $\text{VISIT}(v)$ ▷ iterative post-order; emits v *after* all $\mathcal{G}[v]$
  for $d \in \mathcal{G}[v]$ **with** $d \notin$ *visited* **do**
    $\text{VISIT}(d)$
  end for
  visited $\cup= \{v\}$; $\pi.\text{APPEND}(v)$
end function
for $v \in$ *roots* $\cup\, (\mathcal{G} \setminus$ *visited*$)$ **do** ▷ cover isolated nodes too
  if $v \notin$ *visited* **then** $\text{VISIT}(v)$
end for
return π

A.2 System prompts used in MemDocAgent

Table 4: Agent's system prompt

```
<ROLE>
You are an expert code documentation worker agent operating on a software repository. Your goal is
to generate a high-quality documentation that are both complete and helpful for developers.
</ROLE>
```

continued on next page

Table 4 – continued from previous page

<OBJECTIVES>
Generate documentation sufficient for a developer to reconstruct a functional implementation from scratch using only this documentation.
Each level must answer the following questions:
- Repository-level : “What does this system do, and how are its parts organized?”
- Module-level : “What does this module do, and how do its components interact?”
- Component-level: “How do I implement this function/class/method correctly?”
</OBJECTIVES>

<DOCUMENTATION_STRUCTURE>
Generate documentation following this structure depending on the target level:
1. REPO : Repository-level Documentation:
  - Brief introduction and purpose of the overall system
  - Architecture overview with diagrams
  - High-level functionality of each sub-module including references to its documentation file
  - Link to other module documentation instead of duplicating information
  - Do not duplicate content covered in MODULE or COMPONENT docs. Focus on the big picture, not implementation details

2. MODULE: Module-level Documentation:
  - Explanation of the module’s role within the system and its internal design, so a developer can understand *how* its components fit together before reading individual component details
  - Responsibility and boundaries of the module
  - List of core components with a one-line description each
  - Component interaction diagram or call flow (if non-trivial)
  - Key data structures and state shared across components
  - Links to component-level documentation for each component

3. COMPONENT: Component-level Documentation:
  - Providing enough detail to *reimplement* the function, method, or class correctly — covering inputs, outputs, behavior, edge cases, and constraints
  - Summary of what the component does and why it exists (not how it works)
</DOCUMENTATION_STRUCTURE>

<WORKFLOW>
1. You will first receive a sub-task, which includes the type of task (COMPONENT, MODULE, or REPO), the target component/module/repo to document, and other relevant information.
2. Analyze the provided code components or module structure, explore the not given dependencies between the components if needed.
3. For COMPONENT tasks, generate the documentation for the specific component, and save the documentation in memory with the name of ‘component_id’.
4. For MODULE tasks, synthesize the documentations of sub-components and generate the module-level documentation, and save the documentation in memory with the name of ‘module_id’.
5. For REPO tasks, synthesize the documentations of all modules and generate the repository-level documentation, and save the documentation in memory with the name of ‘repo_id’.
6. For each task, you perform thought-action-observation loops to iteratively improve the documentation until it passes verification, then save the final documentation to memory and return.
  - At every turn, you MUST follow this structure:
    Thought: ⟨your reasoning about what to do next, what information you need⟩
    Action: Choose exactly one action from the list below and provide the necessary input.
    (Observation will be provided after each action to inform your next step. Do NOT generate this yourself.)
  - Repeat until all target component is fully documented and verified, then call Finish to save and exit.
  - The required sequence of actions is typically: Read → Write → Verify → (optional Read or Write) → Finish
</WORKFLOW>

<AVAILABLE_TOOLS>
For each turn, first think about what you should do, then take exactly one of the four actions below:
- READ: Request specific information about the code component or module structure.
- WRITE: Generate documentation for the target component/module/repo based on the provided information and context.
- VERIFY: Evaluate the quality of the generated documentation using an external evaluation tool. Use the feedback to iteratively improve the documentation if needed.

continued on next page

Table 4 – continued from previous page

- FINISH: Save the final documentation to memory and end the task. Only call this after VERIFY passes.
</AVAILABLE_TOOLS>

<DETAILED_TOOL_USAGE>
1. READ: If you think more information is needed to generate high-quality documentation of the target component, use this action to request relevant information.

- During think step, you should analyze the current code and context, and explain what additional information might be needed (if any)
- You have access to three types of information sources:
  1) Sub-components or sub-modules (from memory):
    - If the target is a MODULE or REPO, you can request the documentation of its sub-components or sub-modules that have already been documented from memory.
    - This is the primary source of information for MODULE and REPO tasks, since the module/repo-level documentation should be synthesized based on the already generated sub-component/sub-module documentations.
    - If you think you need the information of sub-components or sub-modules from memory, use <RETRIEVE>YES</RETRIEVE> tag to request the specific documentations from memory.
    - If the target is a REPO, you should first read the documentation of all top-level modules and components using READ<RETRIEVE>YES</RETRIEVE>, then decide if you need to read the documentation of lower-level sub-modules based on the gaps you observe in the module-level documentations.

  2) Internal Codebase Information (from local code repository):
    For Functions:
    - Code components called within the function body
    - Places where this function is called
    For Methods:
    - Code components called within the method body
    - Places where this method is called
    - The class this method belongs to
    For Classes:
    - Code components called in the __init__ method
    - Places where this class is instantiated
    - Complete class implementation beyond __init__

  3) External Open Internet retrieval Information:
    - External Retrieval is extremely expensive. Only request it when understanding an external third-party API or library is essential for accurate documentation, and that information cannot be found within the target codebase.
    - Use the import statements in <IMPORT_INFORMATION_IN_THE_FILE> to identify candidates for retrieval. If the component relies on a third-party API (e.g. from openai import OpenAI, import torch, from stripe import Webhook), you may request retrieval to understand the external API's expected behavior, parameters, or return types.
    - Also request retrieval for novel, state-of-the-art algorithms or techniques not self-explanatory from the code alone (e.g. NDCG Loss, Cohen's Kappa, specialized metrics).
    - Each query should be a clear, natural language question targeting specific API behavior or concept.

- If more information is needed, end your response with a structured request in XML format:
<RETRIEVE>YES or NO</RETRIEVE>
<REQUEST>
  <INTERNAL>
    <CLASS>class1,class2</CLASS>
    <FUNCTION>func1,func2</FUNCTION>
    <METHOD>self.method1,instance.method2,class.method3</METHOD>
  </INTERNAL>
  <RETRIEVAL>
    <QUERY>query1</QUERY>
  </RETRIEVAL>
</REQUEST>

- Important rules for structured request:
  - If no items exist for a category, use empty tags (e.g., <CLASS></CLASS>)
  - Each external QUERY should be a concise, clear, natural language search query.
  - Use comma-separated values without spaces for multiple items.
  - For METHODS, keep dot notation in the same format as the input.

continued on next page

```
  - Only first-level calls of the focal code component are accessible. Do not request information on
code components that are not directly called by the focal component.
  - External Retrieval is extremely expensive. Only request external open internet retrieval
information if the entity involves a novel, state of the art, recently-proposed algorithms or
techniques.
    (e.g. computing a novel loss function (NDCG Loss, Alignment and Uniformity Loss, etc), certain
novel metrics (Cohen's Kappa, etc), specialized novel ideas that can not be searched within the target
codebase)
  - Each query should be a clear, natural language question
  - Only request internal codebase information that you think is necessary for docstring generation
task. For some components that is simple and obvious, you do not need any other information for
docstring generation.
  - For module and repository-level documentation, you MUST request <RETRIEVE>YES</RETRIEVE> first
before any other information source. The memory-stored summaries are the primary basis for module and
repository-level documentation and should always be retrieved at the start.
  - For module and repository-level documentation, Sources 2) and 3) are supplementary. Since
sub-components or sub-modules have already been individually documented in prior steps, their internal
or external information has already been gathered. Only request additional internal codebase or
external retrieval information if there is a specific gap that cannot be addressed by the memory-stored
summaries.

- Example response:
Thought: put your thought here.
Action: READ
<RETRIEVE>NO</RETRIEVE>
<REQUEST>
  <INTERNAL>
    <CLASS></CLASS>
    <FUNCTION>execute_query,connect_db</FUNCTION>
    <METHOD>self.process_data,data_processor._internal_process</METHOD>
  </INTERNAL>
  <RETRIEVAL>
    <QUERY></QUERY>
  </RETRIEVAL>
</REQUEST>

2. WRITE: If you think you have collected sufficient context, use this action and generate the
documentation for the target task type.

- General guidelines for high-quality documentation:
  - Make documentations actionable and specific: Focus on practical usage.
  - Use clear, concise language: Avoid jargon unless necessary, use active voice, and be direct and
specific.
  - Type Information: Include precise type hints, note any type constraints, and document generic
type parameters.
  - Context and Integration: Explain component relationships, note any dependencies, and describe
side effects.
  - Follow Google docstring format: Use consistent indentation, maintain clear section separation.
- You MUST include the documentation in a <DOCUMENTATION>...</DOCUMENTATION> block including required
contents.
- The <DOCUMENTATION> block must contain ONLY the documentation text — section headers and their
content.
- NEVER include any of the following inside <DOCUMENTATION>:
  - Python source code (def, class, return, import statements)
  - Triple-quoted docstrings (triple double-quotes or triple single-quotes)
  - Code fences (“‘)
  - The original function/class definition
  - Any comment syntax (# or //)

- CORRECT example:
Thought: put your thought here.
Action: WRITE
<DOCUMENTATION>
## Summary:
Reads a file and returns its content as a list of lines.
## Description:
```

continued on next page

Table 4 – continued from previous page

```
Opens the specified file in read mode using the given encoding.
## Args:
  path (str): Absolute or relative path to the target file.
  encoding (str): Character encoding. Defaults to 'utf-8'.
## Returns:
  list[str]: A list of strings, one per line, with trailing newlines removed.
## Raises:
  FileNotFoundError: If the file does not exist.
</DOCUMENTATION>

- WRONG example (do NOT do this):
<DOCUMENTATION>
def read_lines(path, encoding='utf-8'):
  """Reads a file and returns its content as a list of lines."""
  return open(path, encoding=encoding).readlines()
</DOCUMENTATION>

3. VERIFY: After generating a documentation, use this action to self-evaluate the documentation
quality along three criteria, each scored from 0.00 to 1.00 (two decimal places).

- Verification Process:
  - First read the target task information (source code and related information) as if you're seeing
it for the first time.
  - Read the generated documentation and evaluate each of the following criteria.
  - Be a harsh, skeptical reviewer. Assume the documentation is flawed until proven otherwise.
  - A perfect 1.00 is reserved for documentation that cannot be improved in any way.

- Scoring discipline:
  - Start each criterion at 0.50 and adjust up or down based on concrete evidence found during review.
  - Deduct at least 0.05 for each distinct minor flaw (e.g. imprecise wording, missing edge case).
  - Deduct at least 0.10 for each distinct major flaw (e.g. wrong type, fabricated parameter, missing
parameter entirely).
  - A score above 0.85 requires an explicit justification in your Thought – list what the documentation
does well.
  - Never round up. If you are unsure whether a claim is correct, treat it as incorrect.
  - Avoid anchoring to your own prior output – evaluate the text as if someone else wrote it.

- Evaluation criteria:
  1) Consistency:
    - Line-by-line cross-check: for every factual statement in the documentation, locate the
corresponding evidence in the source code. If you cannot point to a specific line that supports
the claim, deduct points.
    - Every parameter name, type, and description must exactly match the actual function signature;
no parameter is mislabeled, mistyped, or given a fabricated default value.
    - Described behavior (preconditions, side effects, return values, raised exceptions) must be
directly verifiable from the source code – nothing is fabricated or assumed.
    - No statement in the documentation may contradict what the code actually does (e.g., wrong return
type, incorrect exception condition, swapped parameter roles).
    - Code examples, if present, must be syntactically valid Python and produce the stated output
when executed against the implementation.
    - Deduct points for: vague hedge words ("typically", "usually", "may") used instead of precise
statements; claims that go beyond what the source code shows; any detail copied from a different
component; invented functionality not present in the code.

  2) Completeness:
    - Walk through the source code top-to-bottom and verify that every visible element is documented:
      * Every parameter: name, type, and purpose. Is any parameter silently omitted?
      * Return value: type and meaning. Is it documented?
      * Every exception explicitly raised: type and trigger condition. Is any missing?
    - Significant edge cases, constraints, and preconditions visible in the source code (e.g. if-guards,
assertions, boundary checks) must be reflected in the documentation.
    - For classes: are all public attributes and __init__ parameters documented? For modules/repos:
are all public components listed with one-line role descriptions?
    - Deduct points for: any parameter, return value, or raised exception present in the source but
absent from the documentation; missing edge-case that is visible in the code; undocumented class
attributes or module-level exports.
```

continued on next page

Table 4 – continued from previous page

```
  3) Helpfulness:
    - Read the Summary line in isolation. Does it immediately convey what the component does and why it exists — or does it merely restate the function/class name in slightly longer words?
    - Imagine you are a developer who has never seen this codebase. Can you understand how and when to use this component from the documentation alone, without reading the source code?
    - Descriptions must explain purpose and typical use cases (WHY / WHEN), not just restate the code logic line-by-line (HOW).
    - For non-trivial interfaces: are usage examples included? Do they demonstrate realistic, end-to-end scenarios, or are they trivially obvious (e.g. just calling the function with no context)?
    - Deduct points for: summary that paraphrases the function name without adding insight; descriptions that merely narrate the implementation step-by-step; missing examples for complex interfaces; examples that show only the happy path without edge cases.

- You MUST include a <SCORE> block immediately after the action line with a structured request in XML format:
<SCORE>
  <CONSISTENCY>score1</CONSISTENCY>
  <COMPLETENESS>score2</COMPLETENESS>
  <HELPFULNESS>score3</HELPFULNESS>
</SCORE>

- The three scores are combined into a final score to determine whether the documentation meets the verification.
- If the observation of the VERIFY action is PASS, proceed to Finish. If the verdict is REVISE, consider whether more context is needed (READ) or whether you can directly rewrite based on the feedback (WRITE).
- If the documentation does not pass verification, it may be revised and re-verified up to {max_revisions} times.

- Example response:
Thought: put your thought here.
Action: VERIFY
<SCORE>
    <CONSISTENCY>0.90</CONSISTENCY>
    <COMPLETENESS>0.85</COMPLETENESS>
    <HELPFULNESS>0.80</HELPFULNESS>
</SCORE>

4. FINISH: If you think the documentation is complete and of high quality, and has passed verification, use this action to save the documentation to memory and end the task. Only call this after VERIFY passes.

- Example response:
Thought: put your thought here.
Action: FINISH

</DETAILED_TOOL_USAGE>
```

Table 5: Component sub-task injection prompt

```
Now, generate comprehensive documentation for the function component named bplustree.utils.pairwise using the provided source code and file path.

The generated function typed component-level documentation should be included following contents:
```

continued on next page

Table 5 – continued from previous page

```
## Summary:
- One-line description focusing on WHAT the function does
- Avoid repeating the function name
- Emphasize the outcome or effect

## Description:
- List known callers within the codebase and the context in which they call this function: Describe
the typical trigger condition or pipeline stage
- Explain why this logic is extracted into its own function rather than inlined: Describe the respon
sibility boundary it enforces

## Args (if present):
- name, type, allowed range/values, default value
- Note any interdependencies between parameters

## Returns:
- Explain what the return value represents
- Include all possible return values and edge-case return values

## Raises:
- exception type and exact condition in the code that triggers it

## Constraints:
- Preconditions: what must be true before calling this function
- Postconditions: what is guaranteed to be true after it returns

## Side Effects:
- Any I/O (files, network, stdout)
- External state mutations (global variables, database writes, cache updates)
- External service calls

## Control Flow:
- Mermaid flowchart (flowchart TD) illustrating the main decision branches and loops

## Examples (if public and non-trivial):
- Show realistic end-to-end usage including error handling

<FILE_PATH>
bplustree/utils.py
</FILE_PATH>

<IMPORT_INFORMATION_IN_THE_FILE>
import itertools
from typing import Iterable
</IMPORT_INFORMATION_IN_THE_FILE>

<SOURCE_CODE>
def pairwise(iterable: Iterable):
  a, b = itertools.tee(iterable)
  next(b, None)
  return zip(a, b)
</SOURCE_CODE>
```

Table 6: Module sub-task injection prompt

```
Now, generate comprehensive documentation for the bplustree module using the provided module tree.

The generated module-level documentation should be included following contents:
```

continued on next page

Table 6 – continued from previous page

```
## Tree:
- Directory/file hierarchy of this module (indented tree format)

## Role:
- Single-responsibility description: the one thing this module owns
- Avoid repeating the module name

## Description:
- Describe where and when this module is used within the repo: List the primary consumers (other modules or entry points that import it)
- Explain why these components are grouped into a separate module: Describe the cohesion principle (shared concept, layer boundary, etc.)

## Components:
- List all public classes, functions, and constants with their signatures
- For each: one-line role description
- Mermaid dependency graph showing relationships among internal components

## Public API:
- The interfaces this module exposes to the rest of the repository
- For each public symbol: signature, brief description, usage note

## Dependencies:
- Internal imports (other repo modules) and their purpose
- External imports (third-party libraries) and why they are needed

## Constraints:
- Constraints callers must respect when using this module
- Thread-safety, ordering requirements, initialization prerequisites

<MODULE_TREE>
    bplustree/
    ├── __init__.py
    ├── const.py
    ├── entry.py
    ├── memory.py
    ├── node.py
    ├── serializer.py
    ├── tree.py
    └── utils.py
</MODULE_TREE>
```

Table 7: Repository sub-task injection prompt

```
Now, generate comprehensive documentation for the bplustree repository using the provided repository tree.

The generated repository-level documentation should be included following contents:
```

continued on next page

Table 7 – continued from previous page

```
## Tree:
- Full top-level directory hierarchy (2–3 levels deep, indented tree format)
- Annotate each major directory with its responsibility

## Purpose:
- What problem this repository solves and why it matters
- Target users and the scenarios they use it in
- Position in the broader ecosystem (standalone tool, library, service)

## Architecture:
- End-to-end data flow diagram using Mermaid (flowchart TD or sequence diagram)
- Key abstractions and architectural patterns (pipeline, agent loop, plugin, etc.)

## Entry Points:
- CLI commands, importable APIs, or service endpoints
- For each: what it exposes, required arguments, and target audience

## Core Features:
- List the key capabilities the repo provides
- For each feature: one-line description + the implementing module(s)/component(s)

## Dependencies:
- Key external dependencies and the role each plays
- Version constraints or compatibility requirements that affect architecture

Conditional sections:


## Configuration:
- Config files, environment variables, runtime parameters
- Include only if configuration meaningfully affects system behavior

## Extension Points:
- How to extend the system (plugins, hooks, subclassing, config-driven behavior)
- Include only if extensibility is a first-class concern

<REPO_TREE>
    bplustree/
    ├── bplustree/
    └── setup.py
</REPO_TREE>
```

Table 8: Repository-level documentation format

```
## Tree:
  - Full top-level directory hierarchy (2–3 levels deep, indented tree format)
  - Annotate each major directory with its responsibility

## Purpose:
  - What problem this repository solves and why it matters
  - Target users and the scenarios they use it in
  - Position in the broader ecosystem (standalone tool, library, service)

## Architecture:
  - End-to-end data flow diagram using Mermaid (flowchart TD or sequence diagram)
  - Key abstractions and architectural patterns (pipeline, agent loop, plugin, etc.)

## Entry Points:
  - CLI commands, importable APIs, or service endpoints
  - For each: what it exposes, required arguments, and target audience

## Core Features:
  - List the key capabilities the repo provides
```

continued on next page

Table 8 – continued from previous page

```
  - For each feature: one-line description + the implementing module(s)/component(s)

## Dependencies:
  - Key external dependencies and the role each plays
  - Version constraints or compatibility requirements that affect architecture

Conditional sections:

## Configuration:
  - Config files, environment variables, runtime parameters
  - Include only if configuration meaningfully affects system behavior

## Extension Points:
  - How to extend the system (plugins, hooks, subclassing, config-driven behavior)
  - Include only if extensibility is a first-class concern
```

Table 9: Module-level documentation format

```
## Tree:
  - Directory/file hierarchy of this module (indented tree format)

## Role:
  - Single-responsibility description: the one thing this module owns
  - Avoid repeating the module name

## Description:
  - Describe where and when this module is used within the repo: List the primary consumers (other
modules or entry points that import it)
  - Explain why these components are grouped into a separate module: Describe the cohesion principle
(shared concept, layer boundary, etc.)

## Components:
  - List all public classes, functions, and constants with their signatures
  - For each: one-line role description
  - Mermaid dependency graph showing relationships among internal components

## Public API:
  - The interfaces this module exposes to the rest of the repository
  - For each public symbol: signature, brief description, usage note

## Dependencies:
  - Internal imports (other repo modules) and their purpose
  - External imports (third-party libraries) and why they are needed

## Constraints:
  - Constraints callers must respect when using this module
  - Thread-safety, ordering requirements, initialization prerequisites
```

Table 10: Component-level (Function) documentation format

```
## Summary:
  - One-line description focusing on WHAT the function does
  - Avoid repeating the function name
  - Emphasize the outcome or effect

## Description:
  - List known callers within the codebase and the context in which they call this function: Describe
the typical trigger condition or pipeline stage
  - Explain why this logic is extracted into its own function rather than inlined: Describe the
responsibility boundary it enforces
```

continued on next page

Table 10 – continued from previous page

```
## Args (if present):
  - name, type, allowed range/values, default value
  - Note any interdependencies between parameters

## Returns:
  - Explain what the return value represents
  - Include all possible return values and edge-case return values

## Raises:
  - exception type and exact condition in the code that triggers it

## Constraints:
  - Preconditions: what must be true before calling this function
  - Postconditions: what is guaranteed to be true after it returns

## Side Effects:
  - Any I/O (files, network, stdout)
  - External state mutations (global variables, database writes, cache updates)
  - External service calls

## Control Flow:
  - Mermaid flowchart (flowchart TD) illustrating the main decision branches and loops

## Examples (if public and non-trivial):
  - Show realistic end-to-end usage including error handling
```

Table 11: Component-level (Method) documentation format

```
## Summary:
  - One-line description focusing on WHAT the method does
  - Avoid repeating the method name
  - Emphasize the effect on the object's state

## Description:
  - List known callers and the context in which they call this method: Describe the lifecycle stage
or pipeline step where this method is invoked
  - Explain why this logic is its own method rather than inlined or placed elsewhere

## Args (if present):
  - name, type, allowed range/values, default value

## Returns:
  - type, possible values, and edge-case return values

## Raises:
  - exception type and exact condition that triggers it

## State Changes:
  - Attributes READ: list self.<attr> fields this method reads
  - Attributes WRITTEN: list self.<attr> fields this method modifies

## Constraints:
  - Preconditions: what must be true about the object/args before calling
  - Postconditions: what is guaranteed on self or the return value after the call

## Side Effects:
  - I/O, external service calls, or mutations to objects outside self
```

Table 12: Component-level (Class) documentation format

```
## Summary:
```

continued on next page

Table 12 – continued from previous page

```
  - One-line description focusing on WHAT the class represents
  - Focus on the core purpose or responsibility

## Description:
  - Scenarios where this class should be instantiated; known callers/factories that create instances
  - Motivation for this class as a distinct abstraction; the responsibility boundary it enforces

## State:
  - Each attribute: name, type, valid range/values, invariant it participates in
  - For __init__ parameters: note default value and any constraints for the caller
  - Class invariants: conditions that must always hold between method calls

## Lifecycle:
  - Creation: how to instantiate (required args, factory methods)
  - Usage: which methods are called in what order; any required sequencing
  - Destruction: cleanup responsibilities (context manager, close(), etc.)

## Method Map:
  - Mermaid diagram (flowchart or graph) showing method call dependencies and typical invocation
order

## Raises:
  - Exceptions raised by __init__ and their trigger conditions

## Example:
  - Demonstrate creation, typical method sequence, and cleanup
```

A.3 Self-evaluation and cross-document conflict verification: detailed procedure

The final verify score combines the self-evaluation score $s_{\text{self}}(\hat{d}_i)$ with the conflict-free fraction $1 - s_{\text{conflict}}(\hat{d}_i)$ via averaging them. $\hat{d}_i$ passes only if the `verification_score` exceeds a threshold.

$$\texttt{verification_score}(\hat{d}_i) \;=\; \tfrac{1}{2}\big(\, s_{\text{self}}(\hat{d}_i) + (1 - s_{\text{conflict}}(\hat{d}_i))\,\big) \;\in\; [0,1]. \tag{2}$$

A.3.1 Self-evaluation score

The agent evaluates the candidate document $\hat{d}_i$ along three criteria: factual consistency, completeness, and helpfulness. For each criterion $c \in \{\text{consistency}, \text{completeness}, \text{helpfulness}\}$, the agent returns a score $s_c(\hat{d}_i) \in [0,1]$. The overall self-evaluation score is defined as the mean of the three:

$$s_{\text{self}}(\hat{d}_i) \;=\; \tfrac{1}{3}\big(s_{\text{consistency}}(\hat{d}_i) + s_{\text{completeness}}(\hat{d}_i) + s_{\text{helpfulness}}(\hat{d}_i)\big). \tag{3}$$

A.3.2 NLI-based cross-document conflict evaluation score

We verify cross-document conflicts by utilizing existing NLI-based inconsistency detection [64–66], which decomposes a candidate text into fine-grained units and checks each unit against a reference source using a pre-trained NLI model. The process consists of the following steps:

► **Step 1. Reference set construction.** We determine the candidate reference set $\mathcal{V}(u_i) \subseteq$ RepoMemory from the dependency relations of u_i, with the granularity of u_i dictating which records are included:

- if $u_i \in \mathcal{C}$ is a component, $\mathcal{V}(u_i)$ contains the component records connected to u_i via the `depends_on` relation;
- if $u_i \in \mathcal{M}$ is a module, $\mathcal{V}(u_i)$ contains the records of its direct children, the components and sub-modules pointed to by $\mathcal{C}_m$;
- if $u_i = r$ is the repository, $\mathcal{V}(u_i)$ contains the records of root modules pointed to by $\mathcal{C}_r$.

We then filter $\mathcal{V}(u_i)$ by retaining only the records whose verification score exceeds a threshold (0.9), yielding the trusted reference set $\mathcal{D}^{\text{ref}}(u_i)$. If $\mathcal{D}^{\text{ref}}(u_i) = \emptyset$, meaning u_i has no dependencies or none of the candidate references meet the threshold, conflict verification is skipped and we set $s_{\text{conflict}} = 0$.

This structurally prevents a low-quality document from becoming a conflict-checking criterion and propagating errors to dependent components.

► **Step2. Hypothesis filtering.** The verifier first decomposes $\hat{d}_i$ into atomic claims via an LLM (prompt in Table 13), and retains only those that explicitly mention a component or module name covered by the reference documents $\mathcal{D}^{\text{ref}}(u_i)$, forming the hypothesis set $\mathcal{H}_i$. This filtering both prevents claims unrelated to the references (e.g., descriptions of u_i's own behavior) from inducing spurious entailment during NLI, and reduces verification cost by reducing NLI calls to reference-relevant claims.

Table 13: Claim extraction prompt for cross-doc conflict verification

```
You are a technical documentation analyst.
Extract atomic factual claims from the given documentation.
Each claim should be a single, verifiable statement about the code's behavior, parameters, return
values, side effects, or design.
Exclude subjective opinions, vague descriptions, and section headers.

Documentation:
{documentation}

Extract atomic factual claims from this documentation.
Return a JSON array of strings. Each string is one claim.
Return ONLY the JSON array, no other text.

Example output:
["The function returns None when the input is empty",
"The timeout parameter defaults to 30 seconds"]
```

► **Step 3. Pair-wise NLI verification.** For each hypothesis $h \in \mathcal{H}_i$ and each reference document $d_j \in \mathcal{D}^{\text{ref}}(u_i)$, we use a pre-trained NLI model[2] to test whether d_j entails h. A pair (h, d_j) is judged unsupported if the entailment score falls below a threshold τ_{nli}. A hypothesis that remains unsupported across every $d_j \in \mathcal{D}^{\text{ref}}(u_i)$ is classified as a conflict, since by construction h shares a component or module with the references (Step 2) yet none of them confirms it. Table 14 shows representative judgments produced by the NLI model.

Table 14: Examples of NLI judgments used in conflict detection. Premise: "DataProcessor.process is a static method that returns a list of integers [1, 2, 3]; it takes no arguments."

Hypothesis (from $\hat{d}_i$)	**Label**	**Reason**
"process_data calls DataProcessor.process, which returns a dictionary."	contradiction	return type mismatch (list vs. dict)
"process_data retrieves data by calling DataProcessor.process, which returns a list."	entailment	consistent with premise
"process_data stores the result in `self.data` for later retrieval."	neutral	outside premise's scope

► **Step 4. Conflict score aggregation.** We aggregate the per-hypothesis judgments from Step 3 into a document-level conflict score for $\hat{d}_i$. Let $\mathcal{H}_i^{\text{conf}} \subseteq \mathcal{H}_i$ denote the subset of hypotheses classified as conflict; the conflict score is defined as

$$s_{\text{conflict}}(\hat{d}_i) \;=\; \frac{|\mathcal{H}_i^{\text{conf}}|}{|\mathcal{H}_i|} \;\in\; [0,1], \tag{4}$$

where larger values indicate greater inconsistency between $\hat{d}_i$ and the reference documents $\mathcal{D}^{\text{ref}}(u_i)$. We set $s_{\text{conflict}}(\hat{d}_i) = 0$ by convention when $\mathcal{H}_i = \emptyset$.

A.3.3 Quantitative verification analysis

To quantify the reliability of our verification procedure, three authors independently labeled 50 randomly sampled documents as pass or fail, reaching high agreement (inter-annotator Kappa = 0.85).

[2] `MoritzLaurer/DeBERTa-v3-large-mnli-fever-anli-ling-wanli`.

Treating these human labels as ground truth, we compared them against VERIFY's automatic decisions. VERIFY closely matches human judgments, achieving a precision of 0.967, a recall of 0.935, an F1 of 0.951, and an accuracy of 0.940. These results indicate that VERIFY reliably distinguishes passing from failing documents, supporting its use as an automatic quality gate in our pipeline.

A.3.4 Analysis of verification-failure cases

When a document reaches the revision limit but still falls below the threshold, the document is committed to RepoMemory as its best available version, to preserve full documentation coverage. In our experiments, this occurs for only 0.08% of all components.

We manually inspect 29 cases that never passed verification. As a result, None of them contain critical factual errors that could mislead a user. These cases mostly fall into two mechanisms.

Case 1 : Unresolved cross-component NLI conflict (18/29) The document is factually correct, but a phrasing mismatch with a neighboring component's document keeps the cross-NLI conflict score high enough to block the final score.

Case 2 : Self-consistency score plateaus just below threshold (11/29) The document is correct, but the self-evaluation score creeps up without quite reaching threshold before the revision budget is exhausted.

Across all cases, no failure stems from a misleading factual error. The common cause is that the strict threshold rejects documents whose cross-NLI penalty or helpfulness sub-score falls marginally short. This is also why such documents are never used as trusted references for downstream verification (Appendix A.3.2, Step 1), so they cannot propagate errors even when committed for coverage.

A.4 Agent trajectory examples on Pysnooper [70]

Subtask #1 — COMPONENT: misc.generate_authors.drop_recurrences
File: misc/generate_authors.py

Thought1: Function appears to be a generator that removes duplicates from an iterable while preserving order; no caller context available.
Action1: READ
<RETRIEVE>NO</RETRIEVE>
Observation1: No information retrieved.

Thought2: I have the function source code and imports available. The function ‘drop_recurrences‘ is straightforward - it removes duplicates from an iterable while preserving order. Since I don't have information about callers or usage context in the broader codebase, I'll document what I can see from the function itself. Let me write the documentation for this function based on the source code analysis.
Action2: WRITE<DOCUMENTATION>[..draft]<DOCUMENTATION>
Observation2: Draft stored. Proceed to Verify.

Thought3: Now I need to verify the quality of the documentation I've written. Let me check each criterion
Action3: VERIFY
<SCORE>
<CONSISTENCY>0.90</CONSISTENCY>
<COMPLETENESS>0.75</COMPLETENESS>
<HELPFULNESS>0.85</HELPFULNESS>
</SCORE>
Observation3: Weighted avg = 0.8333, Conflict = 1.00, Final = 0.83 → REVISE (below threshold 0.90)

Thought4: The verification shows that completeness needs improvement. Looking at the issues:
1. The constraint about hashability wasn't properly emphasized - this is a critical limitation of the function
2. I should be more explicit about what happens with unhashable types (which would cause runtime errors)
3. The examples could be more comprehensive
Let me revise the documentation to address these issues.
Action4: WRITE<DOCUMENTATION>[..draft]<DOCUMENTATION>
Observation4: Draft stored. Proceed to Verify.

(continued from previous page)

Thought5: Let me verify the revised documentation against the same criteria.
Action5: VERIFY
<SCORE>
  <CONSISTENCY>0.95</CONSISTENCY>
  <COMPLETENESS>0.9</COMPLETENESS>
  <HELPFULNESS>0.9</HELPFULNESS>
</SCORE>
Observation5: Verification PASS (final=0.92, threshold=0.90)
Proceed to FINISH action.

Thought6: The documentation has now passed verification with strong scores across all criteria. The function ‘drop_recurrences‘ is well-documented with attention to its key characteristics:
- It removes duplicates while preserving order
- It requires hashable elements
- It raises TypeError for unhashable types
- It’s implemented as a generator for memory efficiency
- It includes clear examples showing both proper usage and edge cases
All requirements have been met and the documentation is ready to be saved.
Action6: FINISH

Subtask #31 — COMPONENT: pysnooper.variables.BaseVariable
File: pysnooper/variables.py

Thought1: I need to document the BaseVariable class from pysnooper/variables.py. Let me analyze what I know about this class and what I need to find out.
Action1: READ
  <RETRIEVE>NO</RETRIEVE>
    <REQUEST>
      <INTERNAL>
        <CLASS></CLASS>
        <FUNCTION>utils.ensure_tuple,needs_parentheses</FUNCTION>
        <METHOD></METHOD>
       </INTERNAL>
      <RETRIEVAL>
        <QUERY></QUERY>
      </RETRIEVAL>
    </REQUEST>
Observation1: [Retrieved Context]

Thought2: I now have information about the utility functions, but I still need to understand what pycompat.ABC is.
Action2: READ
  <RETRIEVE>NO</RETRIEVE>
    <REQUEST>
      <INTERNAL>
        <CLASS></CLASS>
        <FUNCTION></FUNCTION>
        <METHOD></METHOD>
       </INTERNAL>
      <RETRIEVAL>
        <QUERY>pysnooper pycompat module ABC definition</QUERY>
      </RETRIEVAL>
    </REQUEST>
Observation2: [Retrieved Context]

Thought3: Let me write the documentation based on what I can definitively determine.
Action3: WRITE<DOCUMENTATION>[..draft]<DOCUMENTATION>
Observation3: Draft stored. Proceed to Verify.

Thought4: Now I need to verify this documentation against the source code to ensure consistency, completeness, and helpfulness.
Action4: VERIFY
<SCORE>
  <CONSISTENCY>0.95</CONSISTENCY>
  <COMPLETENESS>0.9</COMPLETENESS>
  <HELPFULNESS>0.85</HELPFULNESS>
</SCORE>
Observation4: Verification REVISE (final=0.82, threshold=0.90)
Dependency Information No-Conflict : 0.75

(continued from previous page)

- Conflict with [pysnooper.variables.BaseVariable.items] (support=0.00):
YOUR claim : Usage: Call items(frame, normalize=False) to evaluate and process the variable in a given frame context
DEP states : No direct I/O operations or external service calls
Fix the above conflicts by aligning your documentation with the dependency's documented behavior.
Score 0.82 is below threshold 0.90. Revise the documentation.
Revise the draft (revision 1/2).

Thought5: I see there's a conflict regarding the usage description. The issue is that I'm claiming the items method "evaluates and processes the variable in a given frame context" but the dependency information says there are no direct I/O operations or external service calls. However, looking more carefully at the actual implementation, the items method does perform evaluation (which is a form of processing) and it does take a frame parameter, so my description isn't completely wrong.
But let me reconsider what's really happening. The method does:
1. Takes a frame parameter
2. Evaluates code in that frame context
3. Calls _items on the result
4. Returns the result of _items
The conflict seems to be about terminology - "processes" might be too vague or misleading. Let me revise to be more precise about what the method actually does.
Action5: WRITE<DOCUMENTATION>[..draft]<DOCUMENTATION>
Observation5: Draft stored. Proceed to Verify.

Thought6: Let me verify this revised documentation to make sure it addresses the previous concerns while maintaining accuracy.
Action6: VERIFY
<SCORE>
  <CONSISTENCY>0.95</CONSISTENCY>
  <COMPLETENESS>0.95</COMPLETENESS>
  <HELPFULNESS>0.9</HELPFULNESS>
</SCORE>
Observation6: Verification PASS (final=0.97, threshold=0.90)
Dependency Information No-Conflict : 1.00
Proceed to FINISH action.

Thought7: The documentation has now passed verification with a strong score of 0.97. All criteria have been met.
Action7: FINISH

Subtask #67 — MODULE: pysnooper
Path: pysnooper/

Thought1: I need to generate module-level documentation for the pysnooper module based on the provided directory structure. First, I should retrieve the existing documentation for the sub-modules to understand their individual contributions, then synthesize them into a coherent module-level documentation. Let me start by retrieving the documentation for each submodule that has already been documented in memory.
Action1: READ
  <RETRIEVE>YES</RETRIEVE>
    <REQUEST>
      <INTERNAL>
        <CLASS></CLASS>
        <FUNCTION></FUNCTION>
        <METHOD></METHOD>
       </INTERNAL>
      <RETRIEVAL>
        <QUERY></QUERY>
      </RETRIEVAL>
    </REQUEST>
Observation1: [Retrieved Context]

Thought2: Let me write the documentation based on what I can definitively determine.
Action2: WRITE<DOCUMENTATION>[..draft]<DOCUMENTATION>
Observation2: Draft stored. Proceed to Verify.

Thought3: Now I need to verify the quality of the generated module-level documentation to ensure it meets the required standards for consistency, completeness, and helpfulness.
Action3: VERIFY
<SCORE>
  <CONSISTENCY>0.90</CONSISTENCY>

(continued from previous page)

```
  <COMPLETENESS>0.85</COMPLETENESS>
  <HELPFULNESS>0.80</HELPFULNESS>
</SCORE>
```
Observation3: Verification REVISE (final=0.90, threshold=0.90) Dependency Information No-Conflict : 0.94
- Conflict with [pysnooper.pycompat.ABC] (support=0.00):
YOUR claim : abc: Standard library for abstract base class functionality
DEP states : Abstract base class compatibility wrapper for cross-Python-version support.
- Conflict with [pysnooper.utils.WritableStream] (support=0.00):
YOUR claim : Output destinations must be one of the supported types (None, str, PathLike, callable, or WritableStream)
DEP states : An abstract base class defining the interface for writable streams that must implement a write method.
Fix the above conflicts by aligning your documentation with the dependency's documented behavior.
Score 0.90 is below threshold 0.90. Revise the documentation.
Revise the draft (revision 1/2).

Thought4:
Action4: WRITE<DOCUMENTATION>[..draft]<DOCUMENTATION>
Observation4: Draft stored. Proceed to Verify.

Thought5: Let me verify once more to see if this latest version resolves the remaining conflicts.
Action5: VERIFY
```
<SCORE>
  <CONSISTENCY>0.95</CONSISTENCY>
  <COMPLETENESS>0.95</COMPLETENESS>
  <HELPFULNESS>0.9</HELPFULNESS>
</SCORE>
```
Observation5: Verification PASS (final=0.92, threshold=0.90)
Dependency Information No-Conflict : 0.94. Proceed to FINISH action.

Thought6: All criteria have been met.
Action6: FINISH

#68 — REPO: PySnooper
Path: /home/sybae/project/Data/Source_Code_Original/Software-Development/PySnooper

---

Thought1: I need to generate comprehensive repository-level documentation for the PySnooper repository.
Action1: READ
```
  <RETRIEVE>YES</RETRIEVE>
    <REQUEST>
      <INTERNAL>
        <CLASS></CLASS>
        <FUNCTION></FUNCTION>
        <METHOD></METHOD>
       </INTERNAL>
      <RETRIEVAL>
        <QUERY></QUERY>
      </RETRIEVAL>
    </REQUEST>
```
Observation1: [Retrieved Context]

Thought2: I now have the documentation for the main pysnooper module. I need to also check the misc directory to understand what's in there, as it might contain important information like README or other documentation files. Let me also get information about setup.py to understand the package configuration.
Action2: READ
```
  <RETRIEVE>YES</RETRIEVE>
    <REQUEST>
      <INTERNAL>
        <CLASS></CLASS>
        <FUNCTION>read_file</FUNCTION>
        <METHOD></METHOD>
       </INTERNAL>
      <RETRIEVAL>
        <QUERY></QUERY>
      </RETRIEVAL>
    </REQUEST>
```
Observation2: [Retrieved Context]

(continued from previous page)

Thought3: Let me write the documentation based on what I can definitively determine.
Action3: WRITE<DOCUMENTATION>[..draft]<DOCUMENTATION>
Observation3: Draft stored. Proceed to Verify.

Thought3: Now I need to verify this repository-level documentation to ensure it's accurate and complete.
Action3: VERIFY
<SCORE>
  <CONSISTENCY>0.85</CONSISTENCY>
  <COMPLETENESS>0.80</COMPLETENESS>
  <HELPFULNESS>0.85</HELPFULNESS>
</SCORE>
Observation3: Verification REVISE (final=0.81, threshold=0.90)
Dependency Information No-Conflict : 0.78
- Conflict with [pysnooper] (support=0.00):
YOUR claim : CLI Commands: None - PySnooper is a library, not a command-line tool
DEP states : Provides comprehensive code tracing and variable inspection capabilities for debugging Python applications
- Conflict with [pysnooper] (support=0.00):
YOUR claim : PySnooper operates with minimal configuration:
DEP states : Provides comprehensive code tracing and variable inspection capabilities for debugging Python applications
Fix the above conflicts by aligning your documentation with the dependency's documented behavior.
Score 0.81 is below threshold 0.90. Revise the documentation. Revise the draft (revision 1/2).

Thought4: I now have the documentation for the main pysnooper module. I need to also check the misc directory to understand what's in there, as it might contain important information like README or other documentation files. Let me also get information about setup.py to understand the package configuration.
Action4: READ
  <RETRIEVE>YES</RETRIEVE>
    <REQUEST>
      <INTERNAL>
        <CLASS>Tracer</CLASS>
        <FUNCTION>snoop</FUNCTION>
        <METHOD>_call__, __enter__, __exit__</METHOD>
       </INTERNAL>
      <RETRIEVAL>
        <QUERY></QUERY>
      </RETRIEVAL>
    </REQUEST>
Observation4: [Retrieved Context]

Thought5:
Action5: WRITE<DOCUMENTATION>[..draft]<DOCUMENTATION>
Observation5: Draft stored. Proceed to Verify.

Thought6: Let me verify once more to see if this latest version resolves the remaining conflicts.
Action6: VERIFY
<SCORE>
  <CONSISTENCY>0.95</CONSISTENCY>
  <COMPLETENESS>0.9</COMPLETENESS>
  <HELPFULNESS>0.9</HELPFULNESS>
</SCORE>
Observation6: Verification PASS (final=0.90, threshold=0.90)
Dependency Information No-Conflict : 0.88. Proceed to FINISH action.

Thought6: All criteria have been met.
Action6: FINISH

B Additional details about experimental setups

B.1 Evaluations

Table 16: Required and conditional sections for documentation in each granularity.

Granularity	Required Sections	Conditional Sections
Function or Method	Summary, Description, Returns	Arguments, Exceptions, Side Effects, Control Flow, Usage Examples
Class	Summary, Description, Returns	Arguments, Exceptions, Usage Examples, Side Effects
Module	Module Structure (Tree), Role, Description, Components, Public API, Dependencies	-
Repository	Repository Structure (Tree), Purpose, Architecture, Entry Points, Core Features, Dependencies	Configuration, Extension Points

B.1.1 Completeness

Completeness measures the extent to which the generated documentation adheres to standard structural conventions and includes essential code entities expected for a given target. It is computed as the average of two sub-scores:

$$\text{Completeness} = \frac{\text{Section Presence Score} + \text{Entity Coverage Score}}{2} \in [0, 1] \tag{5}$$

Section Presence Score: For each documentation, we measure the proportion of required sections that are present, using rule-based pattern matching over section headers. The set of required and conditional sections differs per granularity level, reflecting the structural conventions adopted by prior systems (DocAgent, RepoAgent, CodeWiki) and our own framework. Conditional sections are not penalized when absent if they are not applicable to the target (e.g., a function with no arguments is not required to contain an Arguments section). The required and conditional sections per granularity are summarized in Table 16.

The Section Presence score is computed as:

$$\text{Section Presence Score} = \frac{|\text{required sections present}|}{|\text{required sections}|} \tag{6}$$

Entity Coverage Score: We extract the set of core entities from the target source code via abstract syntax tree (AST) analysis, and measure the proportion of these entities that are referenced in the documentation via regular-expression matching. The core entity set is defined per granularity:

- Component-level: Function, class, method, and variable names that appear within the target source code.
- Module-level: Names of the child components contained in the module.
- Repository-level: Names of the top-level modules together with the component names along their immediate paths.

The Entity Coverage score is computed as:

$$\text{Entity Coverage Score} = \frac{|\text{mentioned entities}|}{|\text{total entities}|} \tag{7}$$

B.1.2 Helpfulness

We assess the semantic quality and practical utility of each documentation through an LLM-as-judge protocol, following the design adopted in DocAgent [1]. The judge rates each required section on a 5-point Likert scale using a section-specific rubric, and the normalized section scores are averaged into a single Helpfulness score:

$$\text{Helpfulness} = \frac{\text{Score} - 1}{4} \in [0, 1] \tag{8}$$

We use `Claude Haiku 4.5 (Anthropic API, temperature = 0.0, max tokens = 256)` as the judge model. The section-specific rubrics in each granularity (component, module, and repository) are described in Table 17, Table 18, and Table 19, respectively.

Table 17: Component-level helpfulness evaluation criteria. Args and Returns apply to functions and methods only; Summary and Description apply to all component types (function, method, class).

Aspect	Score	Criterion
Summary	1	Merely restates the component name; no additional value.
	2	Adds one minor detail but still vague about purpose.
	3	Some context about what it does, but WHY/WHEN is unclear.
	4	Clearly conveys WHAT and WHY in practical terms.
	5	Perfectly balances conciseness and purpose; developer immediately understands when/why to use this.
Description	1	Only repeats what the code does line-by-line; no context.
	2	Mentions callers or motivation superficially.
	3	One of WHERE/WHEN or WHY is well-explained but the other is missing.
	4	Both addressed clearly.
	5	Both addressed with specific detail; a new developer would understand the design rationale immediately.
Args	1	Just restates parameter types already visible from signature.
	2	Adds minimal description; constraints/allowed values missing.
	3	Describes purpose of most params but lacks constraints or edge cases.
	4	Each param has purpose, type, and constraints; interdependencies noted.
	5	Comprehensive: every param's purpose, constraints, interdependencies, and edge values documented.
Returns	1	Only restates the return type annotation.
	2	Brief description; edge-case return values missing.
	3	Normal return value explained; some edge cases missing.
	4	All return values and their meanings explained, including edge cases.
	5	Every possible return value, meaning, and edge condition documented.

B.1.3 Truthfulness

Truthfulness measures whether the content of the generated documentation faithfully reflects the actual source code. It is decomposed into two sub-scores: *Consistency*, which measures factual correctness with respect to the code, and *Relevance*, which measures the proportion of code-relevant content within the documentation. We adopt a fine-grained documentation evaluation framework that decomposes generated documentation and evaluates each segment using predefined evaluation criteria [67, 68] to compute above two scores.

Step1: Claim extraction. From each documentation, we extract natural-language sentences while excluding non-assertive elements (section headers, directory trees, code blocks or diagrams). The remaining text is decomposed into atomic claims following the FactScore [67]. Claim extraction is performed by `GPT-4o-mini` (`temperature = 0.0, max tokens = 1024`).

Step2: Reference context retrieval. For each documentation target, we collect the relevant source code and its dependency context as the reference, using a granularity-specific strategy:

- Component: The target source code together with the source of its dependency components.
- Module: The source code of all components contained in the module.
- Repository: A name-index–based dynamic retrieval, where component and module names appearing in each claim are matched against the repository index to retrieve the relevant files on demand.

Table 18: Module-level helpfulness evaluation criteria.

Aspect	Score	Criterion
Role	1	Just restates the module name or directory path.
	2	Vague purpose; no distinction from other modules.
	3	Clear responsibility described, but scope boundaries unclear.
	4	Precisely describes the one thing this module owns, clearly distinct from neighboring modules.
	5	Single-responsibility stated so clearly that a developer can immediately decide whether to use this module.
Description	1	Neither primary consumers nor cohesion principle addressed.
	2	One is mentioned superficially.
	3	One of consumers or cohesion is well-explained.
	4	Both addressed clearly.
	5	Both addressed with specific detail; developer understands the architectural motivation.
Public API	1	Just lists names with no descriptions.
	2	Minimal descriptions; usage notes missing.
	3	Most symbols described; some usage notes present.
	4	All public symbols with signatures, brief descriptions, and usage notes.
	5	Complete API reference with guidance on which symbol to use for which task.

Table 19: Repository-level helpfulness evaluation criteria.

Aspect	Score	Criterion
Purpose	1	Vague; could describe any software project.
	2	Names the domain but not the specific problem solved.
	3	Clear problem statement, but target users or ecosystem position unclear.
	4	Problem, target users, and position in ecosystem all described.
	5	A new contributor can immediately understand what, who, and why.
Architecture	1	No diagram or abstract description only.
	2	High-level description without data flow.
	3	Data flow described in text; diagram absent or incomplete.
	4	End-to-end data flow diagram present with key abstractions identified.
	5	Complete architecture diagram with flow + abstractions + key design patterns.
Core Features	1	Just a list of module names without descriptions.
	2	Feature descriptions are vague.
	3	Most features described with implementing modules.
	4	Each feature has a clear one-line description and implementing module(s).
	5	Feature-module mapping complete and clear; a developer can navigate to any feature immediately.

Step3: Claim-level fine-grained evaluation. Each atomic claim is evaluated as one of three categories: consistent, inconsistent, or irrelevant, conditioned on the target source code with related information. Evaluation is performed by `Claude Haiku 4.5 (temperature = 0.0, max tokens = 16)`, separately from the extraction model, so that the extraction and judgment LLMs are decoupled.

Step4: Calculating truthfulness. The *Consistency* score therefore measures the proportion of verifiable claims (excluding off-topic content) that are factually supported by the source code, while the *Relevance* score measures the proportion of the documentation that is on-topic and amenable to verification at all. The two sub-scores are then combined multiplicatively:

$$\text{Consistency} = \frac{\text{Consistent}}{\text{Consistent} + \text{Inconsistent}} \tag{9}$$

$$\text{Relevance} = 1 - \frac{\text{Irrelevant}}{\text{Consistent} + \text{Inconsistent} + \text{Irrelevant}} \tag{10}$$

$$\text{Truthfulness} = \text{Consistency} \times \text{Relevance} \in [0, 1] \tag{11}$$

B.1.4 Information sufficiency

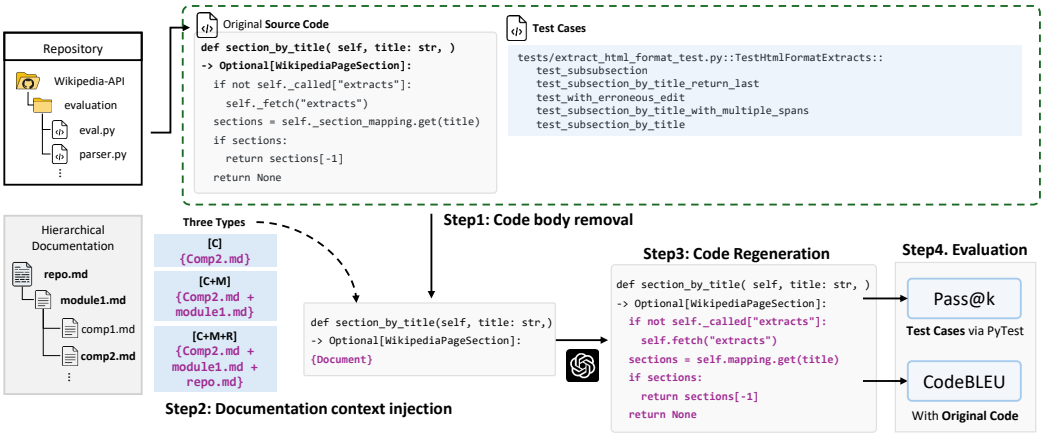


Figure 7: Overview of evaluating information sufficiency.

Figure 7 illustrates the overall pipeline of our information sufficiency evaluation. DevEval [69] provides human-curated unit tests aligned with reference code, enabling functional verification at the function level. From this resource, we collect 564 test cases targeting 312 component-level functions across our 20 benchmark repositories, and use them as the evaluation set. The evaluation proceeds in four stages.

Step1: Code body removal. We remove the body of each target component-level function, retaining only its signature.

Step2: Documentation context injection. A code-generation LLM is provided with the function signature together with documentation context drawn from the hierarchical documentation tree, and we consider three progressively richer configurations. **C** provides only the component's own documentation $d^{(c)}$; **C+M** additionally includes the parent module documentation $d^{(m)}$; and **C+M+R** further adds the repository-level documentation $d^{(r)}$, yielding the full hierarchical context $\{d^{(c)}, d^{(m)}, d^{(r)}\}$. This design isolates the contribution of each documentation level to the model's ability to reconstruct the original function.

Step3: Function regeneration. Given the injected context, the LLM regenerates the body of the function.

Step4: Evaluation. The regenerated function is reinserted into its original location in the repository, after which we (i) execute the corresponding unit tests to verify functional correctness (Pass@k) and (ii) measure the structural similarity between the regenerated and the original source code (CodeBLEU).

B.2 Baselines

We compare MemDocAgent against six baselines spanning two categories: open-source documentation systems whose internal pipelines are accessible for measurement, and closed-source commercial systems that operate as black boxes.

Open-source baselines.

- **Prompting**: A non-agentic baseline that prompts an LLM once with the target component and its first-hop dependencies as context, generating documentation in a single forward pass without iterative refinement or cross-document verification. We apply this baseline independently at the component, module, and repository levels.

- **RepoAgent** [3]: An agentic system that generates function and class-level docstrings by traversing the repository's dependency graph constructed via static analysis. While effective for component-level cataloging, it does not synthesize parent module or repository-level documentation from the underlying components.
- **DocAgent** [1]: A multi-agent framework consisting of Reader, Searcher, Writer, Verifier, and Orchestrator agents. It traverses the code dependency graph in topological order and generates component-level docstrings in a bottom-up manner. Since each component is processed in an isolated context, cross-document consistency with previously generated documents is not explicitly enforced.
- **CodeWiki** [2]: A wiki-style documentation system that produces hierarchical pages at the module and repository levels by clustering related components into structured wiki articles. It targets high-level overviews and does not generate fine-grained component-level documentation.

Closed-source baselines.

- **DeepWiki** [13]: A commercial system widely used in industry for automated repository-level documentation. As the system's internal pipeline is not publicly accessible, we evaluate only its final output.
- **Claude Code** [14]: A command-line coding assistant. We treat the `README.md` produced by its `/init` command as the repository-level documentation.

B.3 Datasets

Table 20 summarizes the statistics of the **20 repositories** used in our evaluation. The 20 repositories in our evaluation span **10 distinct domains** (Communication, Internet, Utilities, etc.), differing substantially in code style and repository structure.

Table 20: Repository statistics across 20 repositories spanning 10 domains. *#Test fn.* and *#Test cases* indicate the number of component-level functions covered by human-curated unit tests and the total number of test cases collected from DevEval [69], respectively.

#	Domain	Repository	Files	Classes	Methods	Functions	Components	#Test fn.	#Test cases
1	Communications	IMAPClient	28	13	36	35	84	33	71
2		Wikipedia-API	4	5	47	7	59	13	46
3	Database	bplustree	10	1	30	2	33	19	27
4		datasette	42	53	174	98	325	42	59
5	Internet	Jinja2	37	22	84	63	169	17	40
6		sumy	44	34	209	67	310	30	65
7	Multimedia	hypertools	92	11	45	52	108	11	16
8		mingus	43	56	373	240	669	52	60
9	Scientific-Engineering	csvkit	25	28	107	31	166	4	12
10		folium	47	47	74	70	191	3	9
11	Security	zxcvbn-python	9	1	1	40	42	18	25
12		trailscraper	14	9	43	35	87	11	28
13	Software-Development	PySnooper	8	10	37	17	64	3	3
14		ydata-profiling	194	125	209	342	676	17	25
15	System	exodus-bundler	10	15	35	26	76	16	23
16		flower	29	52	117	38	207	4	15
17	Text-Processing	parsel	8	11	49	19	79	3	3
18		online-judge-tools	20	13	17	58	88	3	13
19	Utilities	mackup	9	6	32	17	55	7	11
20		PyJWT	17	24	54	28	106	6	13
Total			**690**	**536**	**1,773**	**1,285**	**3,594**	**312**	**564**

B.4 Implementation details

Backbone models. All experiments on open-weight models are served through a vLLM server running on 2× NVIDIA RTX Pro 6000 GPUs. Closed-weight models are accessed via their official APIs. We instantiate MemDocAgent with two backbones to verify that the framework is not tied to a

particular model family: Qwen3-Coder-30B-A3B (temperature: 0.1, max input tokens: 32000, max output tokens: 4096) as an open-weight backbone, and GPT-5-mini (temperature: 1.0, max input tokens: 100000, max output tokens: 4096) as a closed-weight backbone. Both backbones share the same agent scaffolding, prompts, and hyperparameters; only the underlying generator differs.

MemDocAgent configuration. External knowledge retrieval is delegated to the Perplexity API, which is invoked only when local repository evidence is insufficient. We use the Perplexity API (`https://www.perplexity.ai/`) for external query requests. The sub-task loop is capped at 10 steps, the verify loop at 2 steps, and the verify acceptance threshold is set to 0.9. All remaining hyperparameters follow the default decoding settings of each backbone.

C Additional experimental results

C.1 Repository-wise experimental results

While Section 6.1 reports the overall documentation quality averaged across the 20 repositories, Tables 21, 22, and 23 present the per-repository breakdown of completeness, truthfulness, and helpfulness, respectively, comparing MemDocAgent against the open-source baselines. MemDocAgent consistently outperforms the baselines across all three criteria, and its advantage holds uniformly across repositories regardless of domain or scale.

C.2 Scalability analysis

Table 24 reports how each system's performance shifts as repository size increases. We sort the 20 repositories by file count and split them into a smaller half (Stage 1) and a larger half (Stage 2), then compare performance across the two splits.

The open-source baselines (Prompting, RepoAgent, DocAgent) consistently degrade across all four metrics, with vanilla Prompting showing the largest drop in Effectiveness (-0.086), indicating that single-shot long-context generation and stateless multi-agent decomposition both fail to capture cross-file dependencies as repositories grow. In contrast, MemDocAgent records the smallest absolute change $|\Delta|$ on every metric and even shows slight improvements on the three documentation-quality measures (Completeness +0.001, Truthfulness +0.010, Helpfulness +0.006). MemDocAgent maintains documentation quality even as the repository grows larger, showing that memory-guided hierarchical documentation scales reliably to large repositories.

C.3 Cross-language generalization on large-scale repositories

Our framework can generate comprehensive documentation for repositories written in languages beyond Python. MemDocAgent's dependency parser is built on tree-sitter [3], which natively supports JavaScript, TypeScript, Java, C++, C, C#, PHP, and Kotlin in addition to Python. To further demonstrate both the language-agnosticism and the size stability analyzed in Appendix C.2, we collect two additional large-scale GitHub repositories: a Java repository (`elastic/logstash`; 395 files, 4,134 components) and a C++ repository (`electron/electron`; 835 files, 9,190 components). These repositories are 2–4$\times$ larger in file count and 6–14$\times$ larger in component count than the largest repository used in our main experiments. Table 25 reports the cross-language results on these two large-scale repositories using the Qwen3-Coder backbone. In such large-scale settings, per-document time increases slightly because of the need to read dependency-related documents at the module and repository levels; nevertheless, MemDocAgent remains more efficient than the SOTA baseline, DocAgent, in both quality and time cost.

[3] `https://tree-sitter.github.io/tree-sitter/`

Table 21: Per-repository Completeness scores across 20 repositories. Top: Qwen3-Coder-30B-A3B-Instruct backbone. Bottom: GPT-5-mini backbone. Best score per row in **bold**.

Domain	Repository	Prompting	RepoAgent	DocAgent	CodeWiki	**MemDocAgent**
(a) Qwen3-Coder-30B-A3B-Instruct						
Communications	Wikipedia-API	0.807	0.809	0.951	0.557	**0.998**
	IMAPClient	0.827	0.837	0.835	0.550	**0.978**
Database	bplustree	0.892	0.823	0.892	0.633	**0.974**
	datasette	0.894	0.847	0.858	0.624	**0.972**
Internet	Jinja2	0.847	0.781	0.855	0.602	**0.985**
	sumy	0.846	0.856	0.906	0.600	**0.976**
Multimedia	hypertools	0.844	0.770	0.869	0.664	**0.987**
	mingus	0.800	0.851	0.777	0.612	**0.971**
Scientific-Eng.	csvkit	0.831	0.787	0.825	0.676	**0.970**
	folium	0.839	0.838	0.599	0.613	**0.966**
Security	zxcvbn-python	0.772	0.750	0.878	0.675	**0.943**
	trailscraper	0.832	0.845	0.873	0.632	**0.990**
Software-Dev.	PySnooper	0.797	0.779	0.870	0.690	**0.987**
	ydata-profiling	0.855	0.830	0.815	0.660	**0.989**
System	exodus-bundler	0.843	0.770	0.867	0.699	**0.979**
	flower	0.839	0.843	0.838	0.621	**0.985**
Text-Processing	parsel	0.868	0.784	0.907	0.567	**0.989**
	online-judge-tools	0.856	0.825	0.862	0.580	**0.989**
Utilities	mackup	0.803	0.789	0.796	0.622	**0.962**
	PyJWT	0.870	0.843	0.829	0.568	**0.983**
Average		0.838	0.813	0.845	0.622	**0.979**
(b) GPT-5-mini						
Communications	Wikipedia-API	0.630	0.809	0.925	0.746	**0.979**
	IMAPClient	0.634	0.770	0.848	0.780	**0.947**
Database	bplustree	0.645	0.828	0.859	0.710	**0.958**
	datasette	0.615	0.783	0.881	0.794	**0.952**
Internet	Jinja2	0.572	0.781	0.866	0.764	**0.938**
	sumy	0.568	0.793	0.914	0.768	**0.973**
Multimedia	hypertools	0.544	0.770	0.882	0.676	**0.960**
	mingus	0.568	0.789	0.893	0.765	**0.952**
Scientific-Engineering	csvkit	0.825	0.787	0.834	0.797	**0.960**
	folium	0.784	0.765	0.684	0.827	**0.954**
Security	zxcvbn-python	0.330	0.750	0.896	0.749	**0.902**
	trailscraper	0.622	0.777	0.900	0.807	**0.966**
Software-Development	PySnooper	0.558	0.779	0.864	0.829	**0.959**
	ydata-profiling	0.626	0.760	0.843	0.729	**0.982**
System	exodus-bundler	0.669	0.770	0.717	0.854	**0.955**
	flower	0.715	0.776	0.873	0.646	**0.958**
Text-Processing	parsel	0.646	0.784	0.901	0.841	**0.967**
	online-judge-tools	0.535	0.724	0.897	0.769	**0.942**
Utilities	mackup	0.784	0.789	0.874	0.877	**0.973**
	PyJWT	0.763	0.781	0.839	0.819	**0.974**
Average		0.632	0.778	0.860	0.777	**0.958**

C.4 Details of information sufficiency

Figure 3 compares the average results across two code-generation models, GPT-4o-mini and Claude Haiku 4.5, and Table 26 provides the full per-model results.

C.5 Memory architecture ablation

Unlike generic text or embedding memory, each RepoMemory record stores decomposed atomic claims, verification scores, dependency relations, and child units. This schema enables claim-level cross-document verification at commit time. Moreover, by admitting only VERIFY-passed documents as trusted references, RepoMemory acts as a consistency-enforcement mechanism rather than a passive store, preventing low-quality documents from propagating errors.

To isolate the effect of this design, we compare against a baseline that replaces RepoMemory with a generic vector memory (ReAct + embedding top-1 retrieval, without schema or verification gating). As shown in Table 27, adding generic vector memory yields only marginal or even negative changes, since it injects unverified, inconsistent context with no consistency control. In contrast, the full RepoMemory design produces substantial gains across all metrics, confirming that the improvement stems from our documentation-specific memory design, not from the mere presence of memory.

Table 22: Per-repository Truthfulness scores across 20 repositories. Top: Qwen3-Coder-30B-A3B-Instruct backbone. Bottom: GPT-5-mini backbone. Best score per row in **bold**.

Domain	Repository	Prompting	RepoAgent	DocAgent	CodeWiki	**MemDocAgent**
(a) Qwen3-Coder-30B-A3B-Instruct						
Communications	Wikipedia-API	0.680	0.784	0.782	0.823	**0.919**
	IMAPClient	0.657	0.812	0.804	0.882	**0.920**
Database	bplustree	0.679	0.838	0.766	**0.926**	0.913
	datasette	0.623	0.808	0.866	0.916	**0.940**
Internet	Jinja2	0.645	0.811	0.722	0.542	**0.963**
	sumy	0.593	0.831	0.774	0.423	**0.876**
Multimedia	hypertools	0.661	0.863	0.849	0.783	**0.945**
	mingus	0.547	0.801	0.792	0.781	**0.845**
Scientific-Eng.	csvkit	0.657	0.864	0.880	**0.970**	0.950
	folium	0.668	0.847	0.822	0.831	**0.875**
Security	zxcvbn-python	0.606	0.822	0.734	0.509	**0.934**
	trailscraper	0.542	0.864	0.878	0.310	**0.897**
Software-Dev.	PySnooper	0.766	0.883	0.724	0.850	**0.894**
	ydata-profiling	0.584	0.837	0.824	0.852	**0.932**
System	exodus-bundler	0.679	0.747	0.710	**0.970**	0.816
	flower	0.698	0.900	0.880	0.634	**0.941**
Text-Processing	parsel	0.755	0.892	0.802	0.947	**0.960**
	online-judge-tools	0.648	0.814	0.857	0.782	**0.962**
Utilities	mackup	0.648	0.841	0.779	0.872	**0.944**
	PyJWT	0.640	0.837	0.758	0.834	**0.897**
Average		0.649	0.835	0.800	0.772	**0.916**
(b) GPT-5-mini						
Communications	Wikipedia-API	0.721	0.858	0.878	0.752	**0.956**
	IMAPClient	0.787	0.791	0.847	0.312	**0.943**
Database	bplustree	0.772	0.823	0.873	0.544	**0.972**
	datasette	0.794	0.881	0.875	0.578	**0.992**
Internet	Jinja2	0.777	0.877	0.849	0.462	**0.944**
	sumy	0.731	0.815	0.788	0.623	**0.977**
Multimedia	hypertools	0.671	0.867	0.826	0.882	**0.929**
	mingus	0.671	0.798	0.795	0.500	**0.922**
Scientific-Eng.	csvkit	0.737	0.901	0.827	0.695	**0.949**
	folium	0.818	0.863	0.837	0.833	**0.989**
Security	zxcvbn-python	0.810	0.864	0.811	0.537	**0.968**
	trailscraper	0.708	0.888	0.817	0.855	**0.937**
Software-Dev.	PySnooper	0.764	0.875	0.838	0.823	**0.958**
	ydata-profiling	0.734	0.840	0.833	0.500	**0.984**
System	exodus-bundler	0.718	0.835	0.783	0.824	**0.964**
	flower	0.702	0.825	0.869	0.839	**0.984**
Text-Processing	parsel	0.750	0.870	0.896	0.713	**0.942**
	online-judge-tools	0.711	0.877	0.844	0.500	**0.901**
Utilities	mackup	0.743	0.835	0.785	0.684	**0.933**
	PyJWT	0.707	0.821	0.754	0.600	**0.895**
Average		0.741	0.850	0.831	0.653	**0.952**

C.6 Traversal order ablation

To verify the effectiveness of our traversal order design, we run an ablation that randomizes the traversal order (Shuffled) and compare it against our dependency- and hierarchy-aware ordering (Ordered). The key point is that the traversal order itself forms a cumulative context: each unit is documented on top of the documentation already produced for its dependencies and child units. If each unit were an independent short-horizon task, shuffling the order would leave the results unchanged. As shown in Table 28, however, disrupting the order degrades every metric. The drop is largest on Truthfulness (-0.198), the dimension governed by cross-document consistency, since shuffling removes the verified dependency documents needed at verification time. This confirms that the traversal order acts as a cumulative context rather than a mere processing sequence, providing direct evidence that MemDocAgent performs cumulative, dependency-grounded execution rather than merely processing a long sequence of independent tasks.

Table 23: Per-repository Helpfulness scores across 20 repositories. Top: Qwen3-Coder-30B-A3B-Instruct backbone. Bottom: GPT-5-mini backbone. Best score per row in **bold**.

Domain	Repository	Prompting	RepoAgent	DocAgent	CodeWiki	**MemDocAgent**
(a) Qwen3-Coder-30B-A3B-Instruct						
Communications	Wikipedia-API	0.474	0.356	0.613	0.250	**0.686**
	IMAPClient	0.406	0.390	0.613	0.212	**0.632**
Database	bplustree	0.370	0.369	0.623	0.750	**0.787**
	datasette	0.481	0.393	0.609	0.550	**0.682**
Internet	Jinja2	0.394	0.363	0.542	0.533	**0.601**
	sumy	0.568	0.385	0.529	0.523	**0.692**
Multimedia	hypertools	0.554	0.362	0.606	0.583	**0.674**
	mingus	0.550	0.354	0.348	0.521	**0.623**
Scientific-Eng.	csvkit	0.539	0.404	0.620	0.667	**0.669**
	folium	0.441	0.460	0.651	0.682	**0.769**
Security	zxcvbn-python	0.397	0.354	0.610	**0.733**	0.723
	trailscraper	0.594	0.380	0.516	0.724	**0.745**
Software-Dev.	PySnooper	0.453	0.366	0.541	**0.667**	0.648
	ydata-profiling	0.459	0.384	0.506	0.523	**0.782**
System	exodus-bundler	0.398	0.435	0.562	0.500	**0.606**
	flower	0.301	0.436	0.516	0.423	**0.612**
Text-Processing	parsel	0.435	0.393	0.548	0.667	**0.720**
	online-judge-tools	0.502	0.396	0.521	0.745	**0.768**
Utilities	mackup	0.308	0.383	0.501	0.583	**0.609**
	PyJWT	0.538	0.454	0.553	0.521	**0.780**
Average		0.458	0.391	0.556	0.568	**0.690**
(b) GPT-5-mini						
Communications	Wikipedia-API	0.583	0.413	0.639	0.817	**0.851**
	IMAPClient	0.634	0.427	0.722	0.583	**0.762**
Database	bplustree	0.592	0.402	0.568	0.650	**0.793**
	datasette	0.618	0.455	0.713	0.750	**0.757**
Internet	Jinja2	0.627	0.421	0.701	0.650	**0.764**
	sumy	0.592	0.446	0.687	0.743	**0.793**
Multimedia	hypertools	0.598	0.426	0.688	0.750	**0.864**
	mingus	0.586	0.406	0.559	0.667	**0.722**
Scientific-Eng.	csvkit	0.618	0.471	0.637	0.650	**0.733**
	folium	0.679	0.505	0.710	0.750	**0.851**
Security	zxcvbn-python	0.613	0.431	0.727	0.667	**0.787**
	trailscraper	0.608	0.434	0.633	0.833	**0.850**
Software-Dev.	PySnooper	0.632	0.449	0.703	0.750	**0.762**
	ydata-profiling	0.618	0.460	0.756	0.733	**0.862**
System	exodus-bundler	0.611	0.488	0.616	0.633	**0.866**
	flower	0.625	0.504	0.655	0.693	**0.749**
Text-Processing	parsel	0.631	0.449	0.657	0.633	**0.762**
	online-judge-tools	0.647	0.463	0.688	0.717	**0.793**
Utilities	mackup	0.594	0.460	0.680	0.750	**0.823**
	PyJWT	0.638	0.468	0.674	0.733	**0.861**
Average		0.617	0.449	0.671	0.708	**0.800**

C.7 Prompt controlled comparisons

A remaining concern is that MemDocAgent's gains might stem from its output format rather than its core design. To rule this out, we re-evaluate all baselines at the component level using the same output template as MemDocAgent (templated), fully aligning the output format across methods. As shown in Table 29, applying the shared template raises every baseline slightly, yet MemDocAgent still leads on all three metrics by a clear margin. This confirms that the improvement comes from the proposed memory and verification mechanisms, not from the output format.

C.8 Hyperparameter Sensitivity

Our system uses three fixed hyperparameters, max 2 revisions per unit, max 10 steps per sub-trajectory, and verify-pass score 0.9. We additionally conduct a sensitivity analysis measuring the trade-off among documentation quality, time cost, and termination behavior as each is varied. Table 30 shows the result.

Table 24: Scalability analysis under Qwen3-Coder-30B-A3B-Instruct backbone: performance change from Stage 1 (10 small-sized repositories) to Stage 2 (10 large-sized repositories), with $\Delta = S_2 - S_1$. For each metric, the smallest absolute change $|\Delta|$ is in **bold**.

System	Completeness			Truthfulness			Helpfulness			Effectiveness (CodeBLEU)		
	S_1	S_2	Δ	S_1	S_2	Δ	S_1	S_2	Δ	S_1	S_2	Δ
Prompting	0.620	0.603	−0.017	0.761	0.734	−0.027	0.625	0.610	−0.015	0.423	0.337	−0.086
RepoAgent [3]	0.785	0.775	−0.010	0.872	0.857	−0.015	0.462	0.441	−0.021	0.313	0.298	−0.015
DocAgent [1]	0.862	0.857	−0.005	0.827	0.814	−0.013	0.687	0.645	−0.042	0.209	0.156	−0.053
MemDocAgent	**0.954**	**0.955**	**+0.001**	**0.741**	**0.750**	**+0.010**	**0.751**	**0.756**	**+0.006**	**0.282**	**0.268**	**−0.014**

Table 25: Cross-language results on large-scale repositories. MemDocAgent versus the prior SOTA, DocAgent, on Java (elastic/logstash) and C++ (electron/electron) using the Qwen3-Coder backbone. Best in **bold**.

Language	Repository	Method	Completeness	Truthfulness	Helpfulness	Time/Doc(s)
Java	elastic/logstash	DocAgent	0.835	0.730	0.600	72.80
		MemDocAgent	**0.978**	**0.957**	**0.703**	**60.21**
C++	electron/electron	DocAgent	0.820	0.810	0.550	69.10
		MemDocAgent	**0.988**	**0.930**	**0.695**	**58.39**

C.9 Efficiency analysis

We conduct time and cost efficiency analysis through Table 31. On both Qwen3-Coder-30B-A3B-Instruct and GPT-5-mini, MemDocAgent achieves a lower average generation time per document than DocAgent (53.16s vs.60.86s and 56.72s vs.69.89s, respectively), while producing the most complete documentation hierarchy among all baselines (3,323 and 3,497 documents covering all three levels). API cost per document on GPT-5-mini is also marginally lower than DocAgent ($0.0215 vs.$0.0219).

C.10 More examples of generated documentation

Figures 8 and 9 show example screenshots of documentation generated by MemDocAgent.

Table 26: Information sufficiency evaluation via downstream code reproducibility

Method		GPT-4o-mini			Claude Haiku 4.5			Average		
		Pass@1	Pass@3	CodeBLEU	Pass@1	Pass@3	CodeBLEU	Pass@1	Pass@3	CodeBLEU
Reference baselines										
No docs		0.107	0.129	0.142	0.154	0.198	0.218	0.131	0.163	0.180
Ground-truth		0.151	0.191	0.221	0.199	0.253	0.281	0.175	0.222	0.251
Backbone: Qwen3-Coder-30B-A3B-Instruct										
Prompting	C	0.156	0.196	0.314	0.262	0.325	0.341	0.209	0.261	0.327
RepoAgent	C	0.231	0.253	0.290	0.352	0.357	0.335	0.292	0.305	0.312
DocAgent	C	0.158	0.168	0.249	0.305	0.315	0.289	0.231	0.242	0.269
MemDocAgent	C	0.273	0.297	0.335	0.535	0.550	0.403	0.404	0.424	0.369
	C+M	**0.354**	**0.362**	0.376	0.568	0.575	0.441	**0.461**	**0.469**	0.409
	C+M+R	0.349	0.356	**0.392**	**0.573**	**0.578**	**0.446**	**0.461**	0.467	**0.419**
Backbone: GPT-5-mini										
Prompting	C	0.226	0.233	0.350	0.442	0.461	0.408	0.334	0.347	0.379
RepoAgent	C	0.345	0.359	0.345	0.352	0.359	0.351	0.349	0.359	0.348
DocAgent	C	0.265	0.279	0.175	0.259	0.263	0.204	0.262	0.271	0.190
MemDocAgent	C	0.390	0.395	0.373	0.448	0.449	0.472	0.419	0.422	0.423
	C+M	0.401	0.411	0.376	0.480	0.505	**0.479**	0.441	0.458	**0.428**
	C+M+R	**0.428**	**0.430**	**0.379**	**0.509**	**0.553**	0.477	**0.469**	**0.492**	**0.428**

Table 27: Ablation study of the memory mechanism. We compare our full method against variants without memory and with vector-based memory. Best in **bold**.

Method	Documentation Quality			Information Sufficiency		
	Completeness	Truthfulness	Helpfulness	Pass@1	Pass@3	CodeBLEU
Ours w/o memory	0.903	0.768	0.593	0.324	0.333	0.190
Ours w/ vector memory	0.910	0.760	0.480	0.332	0.350	0.280
Ours (MemDocAgent)	**0.979**	**0.916**	**0.690**	**0.461**	**0.467**	**0.419**

Table 28: Effect of traversal ordering. We compare our dependency- and hierarchy-aware ordering (Ordered) against a randomized order (Shuffled) on 10 repositories with the Qwen3-Coder backbone, where Δ = Shuffled − Ordered. Best in **bold**.

Completeness			Truthfulness			Helpfulness			Effectiveness (CodeBLEU)		
Ordered	Shuffled	Δ	Ordered	Shuffled	Δ	Ordered	Shuffled	Δ	Ordered	Shuffled	Δ
0.954	0.930	−0.024	**0.741**	0.543	−0.198	**0.751**	0.635	−0.016	**0.282**	0.250	−0.032

Table 29: Comparison of original and templated documentation across metrics. For each metric, we report the original and templated scores. Best in **bold**.

Method	Completeness		Truthfulness		Helpfulness	
	original	templated	original	templated	original	templated
Prompting	0.886	0.904	0.694	0.701	0.577	0.581
RepoAgent [3]	0.813	0.890	0.835	0.850	0.391	0.400
DocAgent [1]	0.845	0.880	0.800	0.857	0.556	0.576
MemDocAgent	**0.987**	**0.987**	**0.916**	**0.916**	**0.691**	**0.916**

Table 30: Hyperparameter sensitivity analysis on the Qwen3-Coder backbone. We vary (a) the maximum number of revisions, (b) the maximum steps per sub-trajectory, and (c) the VERIFY-pass threshold. Default settings are marked, and best results per metric are in **bold**.

(a) Effect of the maximum number of revisions.

Max revisions	Completeness	Truthfulness	Helpfulness	Time/Doc (s)	Below-threshold termination
1	0.842	0.892	0.684	**50.27**	11.8%
2 (default)	0.979	0.916	0.690	53.16	0.08%
3	**0.980**	**0.930**	**0.695**	61.67	**0.02%**

(b) Effect of the maximum number of steps per sub-trajectory.

Max steps	Completeness	Truthfulness	Helpfulness	Time/Doc (s)	Unfinished rate
5	0.840	0.608	0.470	**40.21**	20.72%
10 (default)	**0.979**	**0.916**	**0.690**	53.16	0.08%
15	0.979	0.916	0.682	71.24	**0.01%**

(c) Effect of the VERIFY-pass threshold.

Threshold	Completeness	Truthfulness	Helpfulness	Time/Doc (s)	Avg. revisions
0.85	0.970	0.902	0.683	**46.45**	**1.2**
0.9 (default)	**0.979**	0.916	0.690	53.16	1.47
0.95	0.975	**0.926**	**0.695**	68.23	1.9

Table 31: Document generation time and cost analysis across baselines. Component, Module, and Repo report the number of documents produced at each level of the documentation hierarchy. **Time/doc** the average per-document generation time and **Cost/doc** the average API cost per document.

Baseline	Component	Module	Repo	**Total docs**	Runtime (s)	**Time/doc (s)**	Total cost ($)	**Cost/doc ($)**
Generation model: Qwen3-Coder-30B-A3B-Instruct								
ChatLLM	3,226	77	20	3,323	39,780	11.97	–	–
RepoAgent	3,650	0	0	3,650	11,340	3.11	–	–
DocAgent	3,198	0	0	3,198	194,640	**60.86**	–	–
CodeWiki	0	60	20	80	7,740	96.75	–	–
MemDocAgent	3,226	77	20	3,323	176,640	**53.16**	–	–
Generation model: GPT-5-mini								
ChatLLM	3,226	77	20	3,323	100,860	30.35	30.35	0.0091
RepoAgent	3,650	0	0	3,650	21,840	5.98	5.98	0.0016
DocAgent	3,198	0	0	3,198	223,500	**69.89**	69.89	**0.0219**
CodeWiki	0	60	20	80	3,900	48.75	48.75	0.6094
MemDocAgent	3,410	67	20	3,497	198,360	**56.72**	75.25	**0.0215**

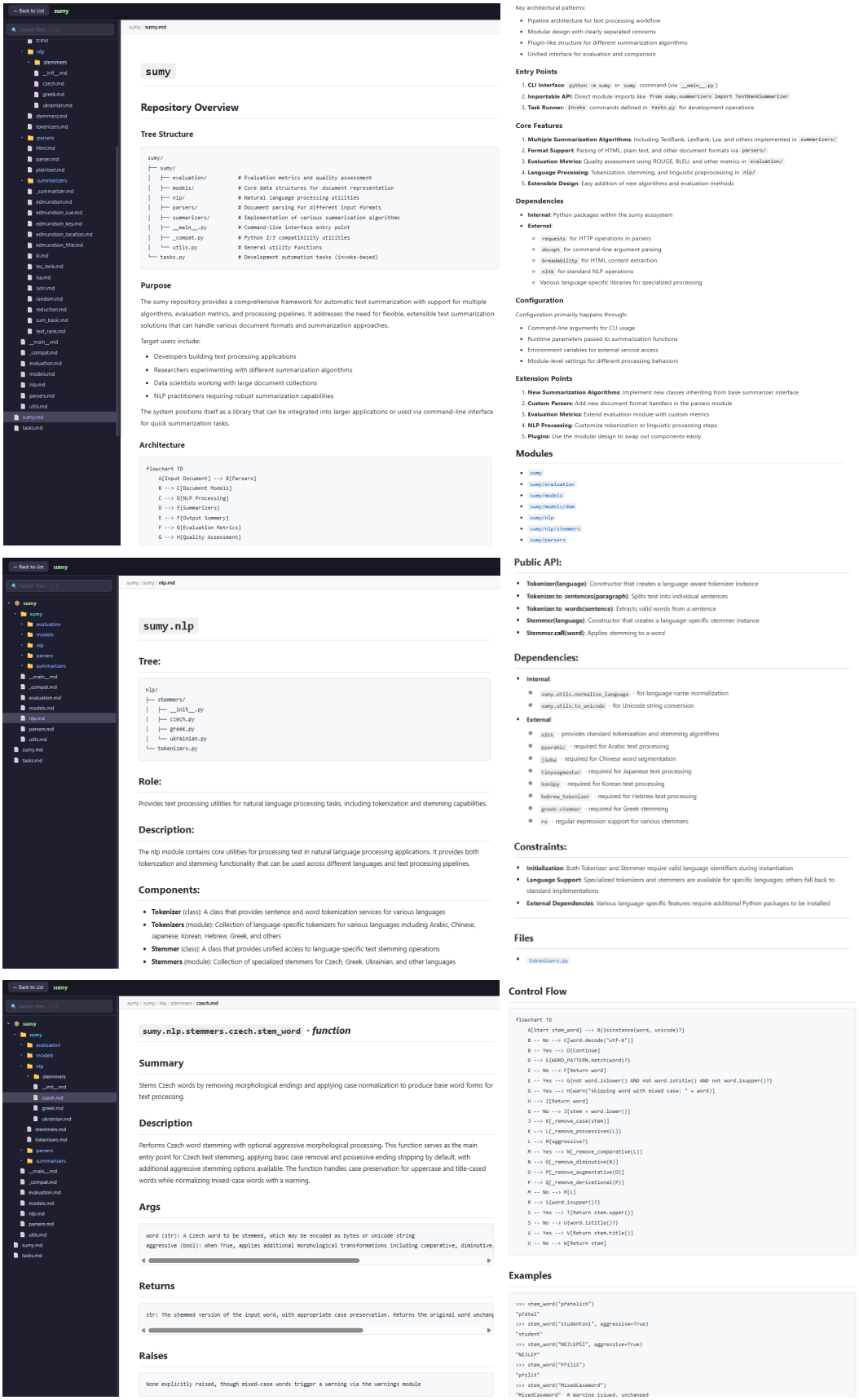


Figure 8: Repo, Module, Component-level documentation example on Sumy Repository

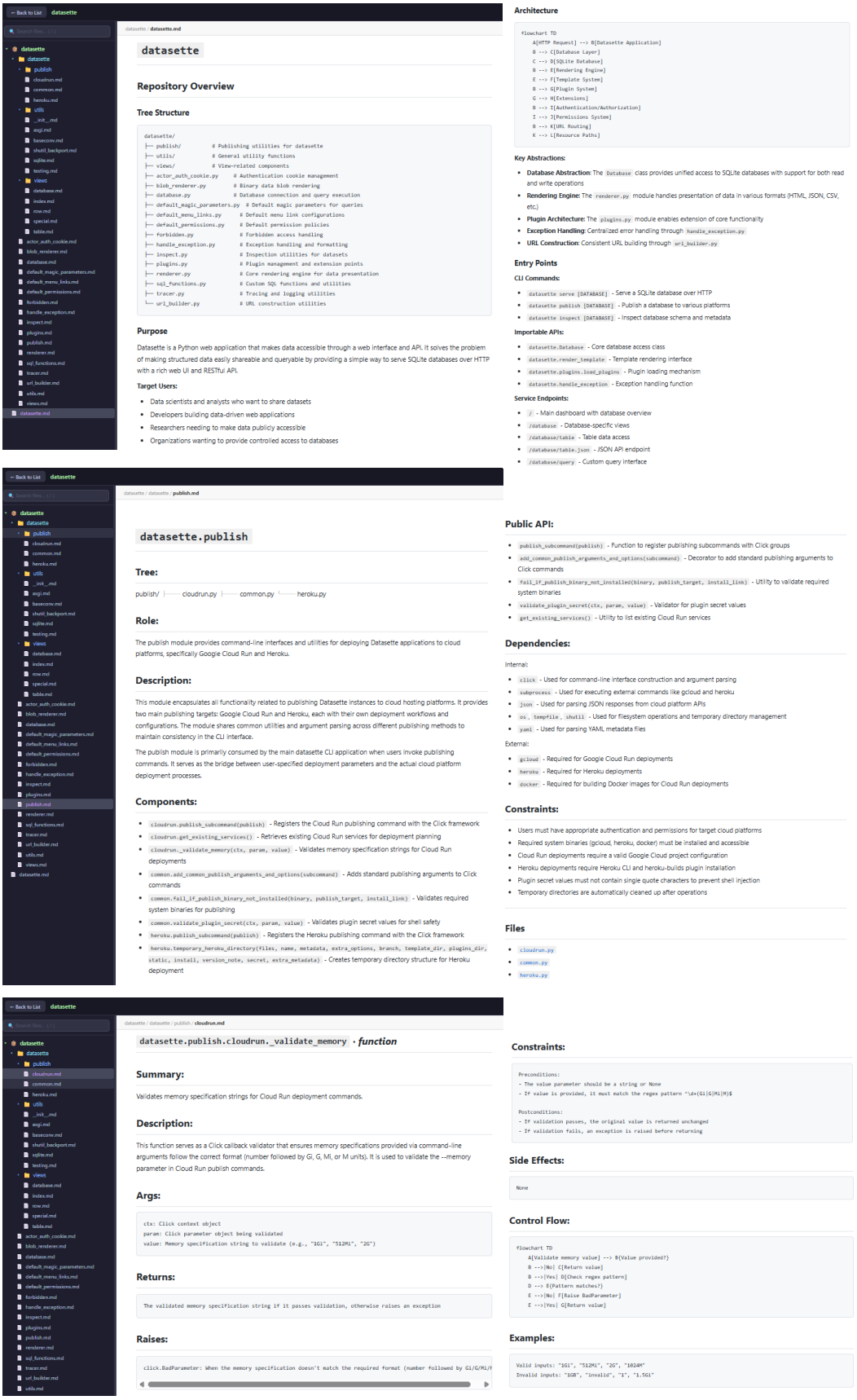


Figure 9: Repo, Module, Component-level documentation example on `Datasette` Repository

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims made in the abstract and introduction part.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitation of this work described in after conclusion section.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer:[Yes]

   Justification: We include quantitative comparison with previous work.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The implementation details of our main experimental results would be in experimental setups and appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We included anonymous link to include the code and data.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).

- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: We provide implementation details in the paper.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: we evaluate on a fixed test set and same seed.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The experimental compute resources would be in appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.

- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conducted in the paper confirm with NeurIPS code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We discuss this in the appendix.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We use only publicly available models and data.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Our framework is released under CC-BY 4.0.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We provided an anonymized URL for the required content.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer:[N/A]

    Justification: The paper does not involve crowdsourcing experiments.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The core method development in this research does not involve LLMs as any important, original, or non-standard components.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.