Is Reasoning All You Need? An Empirical Study of Retrieval and Structured Reasoning for Python Class-Level Docstring Generation

Anonymous ACL submission

Abstract

Automated docstring generation is a high-fidelity software engineering task where factual correctness, interface coverage, and efficiency are critical. Recent work increasingly applies generation-time reasoning strategies—such as Chain-of-Thought (CoT), Tree-of-Thought (ToT), and Graph-of-Thought (GoT)—yet their benefits relative to retrieval-based grounding for code documentation remain unclear. We present a controlled empirical evaluation of 12 Python class-level docstring generation strategies, combining three architectural families—Plain LLM, Retrieval-Augmented Generation (RAG), and Iterative Critique RAG—with four reasoning modes: Base, CoT, ToT, and GoT. Strategies are evaluated using lexical and semantic metrics (ROUGE, BLEU, BERTScore), code-specific coverage measures, an LLM-as-a-Judge faithfulness metric, and latency-based cost analysis.

Our results reveal three consistent findings. First, retrieval is the primary driver of factual reliability: a simple RAG baseline achieves 73% faithfulness, compared to 50% for Plain LLMs and 57% for Iterative Critique RAG (+46% relative improvement). Second, reasoning-heavy strategies show diminishing returns: ToT and GoT increase latency by up to 15× without statistically significant gains in faithfulness or semantic similarity ($\leq$ +2-3% BERTScore). Third, SimpleRAG offers the best cost–quality trade-off, generating high-quality docstrings in 6 seconds versus 90 seconds for Tree-of-Thought pipelines. Overall, our findings indicate that for Python class-level docstring generation—a static, context-bound task—efficient retrieval-based grounding is more effective than generation-time reasoning. We release a reproducible, cost-aware benchmark to support principled evaluation of automated documentation systems.

1 Introduction

Large Language Models (LLMs) have enabled substantial progress in software engineering tasks such as code generation, explanation, and summarization. To improve reliability, recent work increasingly augments LLMs with external context, explicit reasoning, and self-correction mechanisms. Among these approaches, Retrieval-Augmented Generation (RAG) has become a standard technique for grounding model outputs in external documentation rather than relying solely on parametric knowledge.

Automatic docstring generation is a particularly important application of LLMs. Unlike open-ended summarization, docstring generation is a high-fidelity task in which generated descriptions must accurately reflect code behavior, parameters, and return values. Hallucinated or incorrect docstrings can mislead developers and propagate errors into downstream tooling, making this task a stringent benchmark for evaluating factual grounding in LLM-based systems.

While modern LLMs can produce fluent docstrings without external context, standalone generation frequently yields generic or incorrect descriptions, especially for unfamiliar or domain-specific code (Zhang et al., 2025b). Prior work therefore adopts RAG-based approaches that ground generation in style guides, API documentation, and coding conventions, demonstrating improved factual correctness through retrieval-based grounding. Figure 5 illustrates the qualitative difference between docstrings generated with and without external context.

Concurrently, recent research emphasizes generation-time reasoning strategies—such as Chain-of-Thought (CoT), Tree-of-Thought (ToT), Graph-of-Thought (GoT), and agentic self-correction—as general solutions to hallucination. These techniques have demonstrated strong per-

formance on tasks requiring multi-step reasoning and planning and are increasingly being applied to code documentation pipelines. This trend raises a critical but under-examined question for both researchers and practitioners: **whether the substantial computational and latency overhead introduced by explicit reasoning is justified for documentation tasks that are structurally simple, context-bound, and frequently deployed in latency-sensitive developer tooling**.

Despite the widespread adoption of both RAG and reasoning-based prompting, their interaction in structured, context-bound documentation tasks remains poorly understood. Existing studies on RAG for code documentation (Zhang et al., 2025c) focus on limited retrieval variants, while reasoning-centric work largely targets open-ended reasoning tasks rather than extractive documentation.

In this work, we present a systematic empirical benchmark of retrieval and reasoning strategies for Python class-level docstring generation. We evaluate twelve configurations formed by combining three architectural families—Plain LLMs, standard RAG, and iterative self-correction—with four reasoning modalities: Base generation, CoT, ToT, and GoT. This factorial design enables controlled isolation of the effects of retrieval, explicit reasoning, and self-verification. We conduct experiments on a curated dataset of Python class docstrings and evaluate performance using lexical and semantic metrics: ROUGE (Grusky, 2023), BLEU (Papineni et al., 2002), and BERTScore (Zhang* et al., 2020); code-specific coverage measures (parameter, return, and exception coverage); and a reference-free LLM-as-a-Judge (Bavaresco et al., 2025) faithfulness metric to detect hallucinations and unsupported claims. We additionally report inference latency to analyze cost–quality trade-offs.

Our results show that retrieval-based grounding is the dominant driver of reliable docstring generation: a lightweight RAG baseline achieves a Faithfulness score of 73%, substantially outperforming Plain LLMs (50%) and Self-Correction RAG (57%), corresponding to a 46% relative improvement over non-retrieval generation. In contrast, reasoning-heavy strategies such as Tree-of-Thought and Graph-of-Thought exhibit diminishing returns, increasing inference latency by up to 15× without statistically significant gains in faithfulness or semantic accuracy ($\leq$ +2-3% BERTScore). From an efficiency perspective, SimpleRAG offers the most favorable quality–cost trade-off, generating high-quality docstrings in approximately 6 seconds per sample compared to roughly 90 seconds for Tree-of-Thought pipelines—a 93% reduction in latency. Collectively, these findings demonstrate that for code documentation, an inherently static and context-bound task, efficient retrieval-based grounding is substantially more effective than generation-time reasoning.

2 Related Work

2.1 Automated Code Documentation

Automated code documentation, including summaries and docstrings, has been widely studied to improve software maintainability and developer productivity. Neural approaches to code summarization demonstrate that models trained jointly on code and natural language can generate fluent descriptions of program behavior. For example, Liu et al. (Liu et al., 2021) propose a retrieval-augmented summarization mechanism that integrates code semantics with dense representations, reflecting the evolution of retrieval-based techniques beyond simple prompt tuning. However, recent studies show that even strong code-specialized models frequently hallucinate undocumented parameters or behaviors when relying solely on parametric knowledge (Dainese et al., 2024; Sundaram et al., 2025). These findings motivate architectures that explicitly ground generation in external context.

2.2 Retrieval-Augmented Generation for Code Tasks

Retrieval-Augmented Generation (RAG) improves factual grounding by injecting external, non-parametric information into LLM prompts at inference time. RAG has been studied in both general NLP summarization and code-related tasks. For instance, (Zhang et al., 2025a) introduces the Graph of Records, which enriches RAG using historical LLM responses for long-context summarization, suggesting that richer context representations can benefit global documentation tasks. Several extensions to basic RAG—such as corrective retrieval, query reformulation, and source fusion—have been proposed. However, evaluations often focus on single architectural variants or rely primarily on surface-level text similarity metrics, leaving open questions about reliability, efficiency, and faithfulness in structured documentation tasks.

2.3 Advanced Reasoning Strategies for LLMs

Beyond retrieval, explicit generation-time reasoning strategies have been introduced to improve LLM performance on complex tasks. Chain-of-Thought (CoT) prompting (Wei et al., 2022) encourages step-by-step reasoning, while Tree-of-Thought (ToT) (Yao et al., 2023) and Graph-of-Thought (GoT) (Besta et al., 2024) explore and aggregate multiple reasoning paths. These methods show strong gains on tasks requiring multi-step inference or planning. However, their effectiveness for context-bound, extraction-oriented tasks—such as code summarization and documentation—remains unclear. When relevant context is already supplied through retrieval, additional reasoning may introduce speculative inference rather than improve factual grounding.

2.4 Evaluation of Code Summarization and LLM Outputs

Standard metrics such as BLEU and ROUGE are widely used to evaluate generated summaries and documentation but often correlate poorly with human judgments. We discuss this line of work in two parts: first, the limitations of surface-level automatic metrics for code documentation, and second, recent advances in LLM-based evaluation frameworks designed to better capture factual grounding and faithfulness. Recent work therefore emphasizes efficiency and cost alongside correctness. ShortenDoc (Yang et al., 2025), for example, demonstrates that compressing docstring-related prompts can significantly reduce inference costs without degrading performance. To better capture documentation quality, recent studies adopt code-specific metrics such as parameter coverage, return coverage, and faithfulness scores that extend beyond token overlap. In parallel, the LLM-as-a-Judge paradigm has emerged as an alternative evaluation approach. Studies such as (Li et al., 2025a) show that LLM judges can assist in ranking and evaluating retrieval outputs, motivating their use for detecting subtle grounding errors missed by traditional metrics.

2.5 Research Gap

While prior work has advanced retrieval-augmented generation, reasoning strategies, and evaluation methods, integrated comparisons of retrieval and reasoning for high-fidelity documentation remain limited. In particular, it is unclear whether reasoning-enhanced pipelines improve faithfulness over retrieval-only approaches when factual grounding and efficiency are critical. Moreover, most reasoning studies emphasize accuracy in isolation and do not evaluate hallucination behavior under retrieved context. Our study addresses this gap by systematically benchmarking retrieval and reasoning configurations for Python docstring generation across quality, faithfulness, and cost dimensions.

3 Methodology

3.1 Experimental Design

We adopt a factorial experimental design that crosses three architectural families with four generation-time reasoning modes, yielding twelve docstring generation strategies. This design enables controlled analysis of (i) retrieval versus no retrieval, (ii) the marginal utility of explicit reasoning, and (iii) the cost–benefit trade-offs introduced by iterative self-correction, while holding all other variables constant. All strategies are implemented within a unified, configuration-driven framework to ensure fair comparison and reproducibility. 1 shows the architecture diagram.

3.2 Task Definition

The target task is Python class-level docstring generation. Compared to function-level summarization, class docstrings require synthesizing information across methods, attributes, and inheritance structures, making the task particularly sensitive to hallucination and omission. Each model receives the raw source code of a class and generates a PEP-style docstring describing its purpose, parameters, attributes, return behavior, and potential exceptions. This setting reflects realistic documentation workflows and provides a stringent benchmark for evaluating factual grounding and interface completeness.

3.3 Architectural Families

We evaluate three architectural paradigms that represent increasing system complexity.

Plain LLM serves as a no-retrieval baseline, relying solely on the model's parametric knowledge and the input source code. This configuration establishes a lower bound for documentation faithfulness in the absence of external grounding.

Simple RAG follows the standard retrieval-augmented generation paradigm. A dense retriever selects the top-k relevant context chunks from an

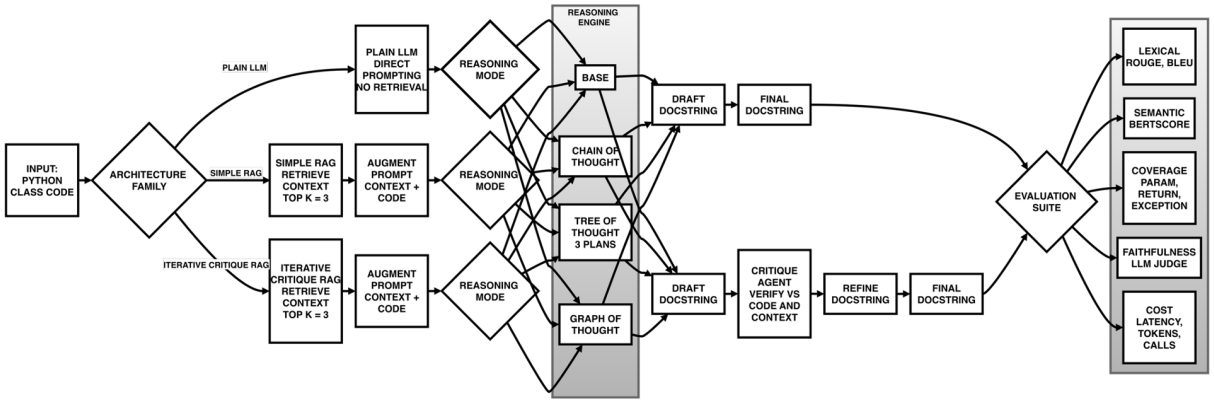

Figure 1: Experimental framework for Python docstring generation. Three architectural families combined with four reasoning modes yield twelve strategies, all evaluated using shared quality, coverage, faithfulness, and cost metrics.

indexed corpus and concatenates them with the source code prior to generation. This architecture isolates the effect of contextual grounding without introducing additional reasoning or control mechanisms.

Iterative Critique RAG applies a test-time generate–critique–refine loop in which the model revises an initial docstring based on self-identified omissions or unsupported claims. This heuristic self-correction approach, distinct from trained reflection-token frameworks such as SELF-RAG, incurs additional inference costs and is evaluated to determine whether lightweight self-review improves reliability beyond retrieval alone.

3.4 Reasoning Modes

Across all architectural families, we apply four prompting strategies (Li et al., 2025b) that vary the structure of generation-time reasoning.

The Base setting uses direct zero-shot (Izacard et al., 2022) prompting without explicit reasoning instructions. Chain-of-Thought (CoT) encourages step-by-step reasoning over parameters and behaviors. Tree-of-Thought (ToT) generates multiple candidate decomposition plans and synthesizes the most complete one. Graph-of-Thought (GoT) produces independent reasoning components (e.g., parameter analysis, return behavior, exception handling) and aggregates them into a final docstring. This axis enables systematic analysis of the benefits and computational costs of increasingly structured reasoning.

3.5 Evaluation Metrics

We employ a multi-dimensional evaluation framework covering documentation quality, completeness, faithfulness, and efficiency. Lexical similarity is measured using ROUGE-1 and BLEU, while semantic alignment is assessed with BERTScore. Documentation completeness is evaluated using Parameter, Return, and Exception Coverage, which quantify whether relevant interface elements are explicitly documented.

To detect hallucinations not captured by similarity or coverage metrics, we introduce a reference-free Faithfulness Score using an independent LLM-as-a-Judge (Elmitwalli et al., 2025). The judge evaluates whether statements in the generated docstring are supported by the source code and, when applicable, retrieved context, producing scores from 0.0 (unsupported) to 1.0 (fully grounded). This score serves as our primary reliability metric. We additionally record inference latency, model call counts, and token usage to quantify computational cost and analyze quality–efficiency trade-offs.

3.6 Reproducibility

All experiments are driven by centralized configuration files that fix retrieval parameters, prompt templates, model settings, and evaluation criteria across strategies. This ensures that observed differences arise solely from architectural and reasoning variations rather than uncontrolled experimental factors.

4 Experimental Setup

4.1 Dataset Construction

We constructed a blind docstring generation benchmark from the rag-docstring dataset, consisting of 250 Python classes collected from diverse public GitHub repositories. For each class, we extract two aligned artifacts: (i) the raw class code and (ii) its original docstring. To enforce a blind gen-

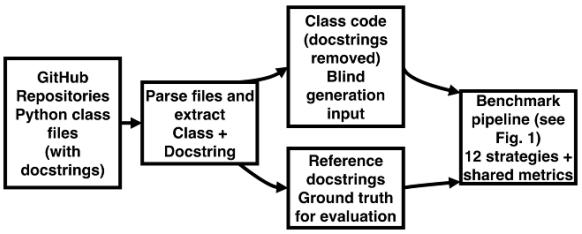

Figure 2: Dataset construction pipeline. Python classes are collected from GitHub, parsed to separate code and docstrings, and stripped of docstrings to create a blind generation setting; the extracted docstrings are used as reference annotations in the benchmarking framework (Figure 2).

eration setting, the docstring is removed from the input code prior to generation, while the extracted docstring is retained as the reference annotation. The resulting (code-without-docstring, reference-docstring) pairs form the evaluation dataset for all experiments. Figure 2 illustrates the dataset construction pipeline, and all samples are evaluated using the unified benchmarking framework shown in Figure 1.

4.2 Models and Hardware

All strategies are evaluated using a fixed set of models and hardware resources to avoid confounding effects.

- Generator LLM: Meta-Llama-3-8B-Instruct, quantized and served via Ollama, selected for its strong instruction-following capability and practical deployment footprint.
- Judge LLM: DeepSeek-Coder-V2, used exclusively as an independent evaluator for the LLM-as-a-Judge faithfulness metric.
- The embedding model all-MiniLM-L6-v2 (384 dimensions) was chosen to reflect realistic developer-tooling constraints rather than large, compute-heavy encoders.

All experiments were executed on a single NVIDIA A100 GPU, ensuring stable throughput and consistent latency measurements across all strategies. Reporting results on a high-performance accelerator allows observed cost differences to be attributed to architectural and reasoning choices rather than hardware limitations.

4.3 Retrieval Index and Parameters

To isolate the effects of reasoning and self-correction, retrieval parameters are fixed across all retrieval-based strategies (Lewis et al., 2020). We use dense semantic retrieval with cosine similarity over embedded text chunks and retrieve the top-$k = 3$ documents. Preliminary ablations showed that larger values of k introduced irrelevant context and consistently reduced faithfulness; we, therefore, adopt $k = 3$ as a conservative setting. Documents are segmented into 512-token chunks with a 50-token overlap to balance semantic coherence and retrieval granularity.

4.4 Execution Protocol and Logging

All twelve strategies are evaluated on the same set of 250 classes. Metrics are computed per sample and aggregated across the dataset. Alongside documentation quality, coverage, and faithfulness, we record end-to-end latency, token usage, and model-call counts for each strategy, enabling explicit analysis of quality–efficiency trade-offs. All prompts, hyperparameters, and evaluation settings are controlled via centralized configuration files to ensure reproducibility and consistent comparisons.

5 Results

5.1 Core Text Quality Metrics

We first examine standard text quality metrics, including lexical overlap (ROUGE-1, BLEU) and semantic similarity (BERTScore). Aggregate results are summarized in Table 1. Across architectural families, SimpleRAG (Base) achieves the highest semantic alignment, with a BERTScore of 0.598, outperforming both the Plain LLM baseline (0.595) and Self-Correction RAG (0.569). Paired Wilcoxon signed-rank tests confirm that the improvement of SimpleRAG over Plain LLM is statistically significant ($p < 0.05$). While reasoning-based prompting strategies occasionally yield marginal improvements in lexical overlap, these gains do not translate into consistent semantic improvements. Across all families, differences between Base prompting and reasoning-augmented variants (CoT, ToT, GoT) are not statistically significant with respect to BERTScore. These results suggest that increasing reasoning depth does not systematically improve semantic quality for docstring generation.

5.2 Faithfulness and Hallucination

Faithfulness, measured using an independent LLM-as-a-Judge, reveals the most pronounced differences between strategies. Quantitative scores are

Setting	Strategy	ROUGE-1	BLEU	BERTScore	Readability
No RAG	Plain	0.190	0.010	0.595	47.40
	CoT	0.191	0.007	0.602	38.14
	GoT	0.212	0.011	0.581	44.53
	ToT	0.210	0.012	0.581	49.56
RAG	**SimpleRAG**	0.200	0.006	**0.598**	39.95
	CoT	0.176	0.009	0.574	32.93
	GoT	0.205	0.008	0.579	42.86
	ToT	0.205	0.012	0.586	44.72
RAG + Reasoning	IterCrit RAG	0.203	0.009	0.569	50.68
	CoT	0.204	0.008	0.562	48.87
	GoT	0.221	0.006	0.585	50.70
	ToT	0.195	0.005	0.560	47.30

Table 1: Core text quality metrics averaged over 250 classes. Improvements in lexical and semantic similarity from reasoning strategies are inconsistent and do not correlate with faithfulness.

reported in Table 2.

Plain LLM strategies consistently cluster around a faithfulness score of 0.50, regardless of reasoning mode, indicating a persistent hallucination in the absence of grounding. In contrast, SimpleRAG (Base) achieves the highest faithfulness score (0.731), representing a 46% relative improvement over the Plain LLM baseline. This improvement is statistically significant under paired testing ($p < 0.001$).

Reasoning-augmented RAG variants do not surpass the base retrieval strategy. Although ToT-RAG and GoT-RAG achieve moderate faithfulness scores, they remain significantly below SimpleRAG ($p < 0.05$). These findings indicate that additional reasoning steps can introduce unsupported inferences rather than improve factual grounding for this task.

5.3 Documentation Coverage and Its Interaction with Faithfulness

We next analyze how comprehensively each strategy documents the code interface. Parameter, return, and exception coverage metrics are reported in Table 2, while their relationship with faithfulness is illustrated in Figure 3. SimpleRAG achieves the best balance between documentation completeness and correctness, maintaining high parameter, return, and exception coverage while also attaining the highest faithfulness. In contrast, Self-Correction RAG shows lower coverage with only moderate faithfulness, indicating that iterative critique can suppress valid interface information alongside hallucinations. Reasoning-heavy strategies occasionally increase coverage, but at a substantially higher computational cost and without corresponding gains in faithfulness. Crucially, higher coverage does not imply higher faithfulness. Encouraging exhaustive documentation can prompt models to speculate beyond the available evidence. Paired statistical tests confirm that SimpleRAG significantly outperforms Self-Correction and reasoning-enhanced variants in both coverage and faithfulness ($p < 0.05$).

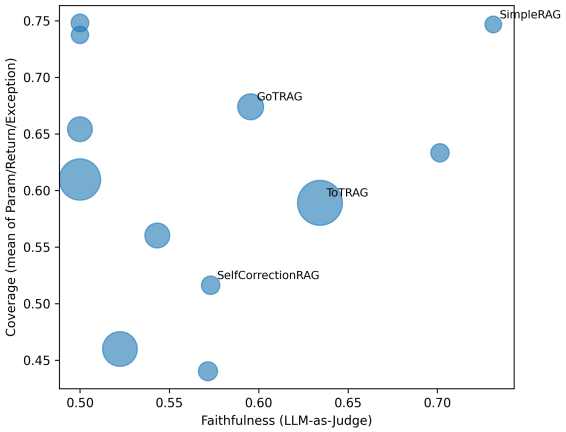

Figure 3: Relationship between documentation coverage and faithfulness. Bubble size indicates average execution time.

5.4 Computational Cost and Efficiency

We analyze computational efficiency using execution time and its components, reported in Table 3, with the overall quality–cost trade-off shown in Figure 4. All experiments were executed on a single NVIDIA A100 GPU under identical conditions. SimpleRAG (Base) achieves a mean latency of approximately 5.9 seconds per sample, making it suitable for interactive developer workflows. In contrast, ToT-RAG incurs

Setting	Strategy	ParamCov	ReturnCov	ExceptionCov	Faithfulness
No RAG	Plain	0.528	0.925	0.791	0.501
	CoT	0.563	0.866	0.784	0.500
	GoT	0.463	0.821	0.679	0.504
	ToT	0.379	0.746	0.703	0.502
RAG	**SimpleRAG**	0.455	**0.940**	**0.845**	**0.731**
	CoT	0.489	0.746	0.664	0.702
	GoT	0.428	0.896	0.698	0.596
	ToT	0.471	0.657	0.639	0.634
RAG + Reasoning	IterCrit RAG	0.287	0.612	0.649	0.573
	CoT	0.274	0.508	0.539	0.572
	GoT	0.408	0.642	0.631	0.543
	ToT	0.267	0.552	0.560	0.522

Table 2: Documentation coverage and faithfulness scores. SimpleRAG (Base) achieves the highest faithfulness despite not maximizing coverage, indicating that reasoning-based verbosity often introduces unsupported content.

Setting	Strategy	Latency (s)	API Calls	Retrieval (s)	Generation (s)
No RAG	Plain	7.44	1.0	0.00	7.44
	CoT	7.05	1.0	0.00	7.05
	GoT	21.93	1.0	0.00	21.93
	ToT	74.49	1.0	0.00	74.49
RAG	**SimpleRAG**	**5.90**	2.0	0.05	5.84
	CoT	8.69	2.0	0.05	8.64
	GoT	24.74	2.0	0.06	24.68
	ToT	89.52	2.0	0.28	89.23
RAG + Reasoning	IterCrit RAG	8.82	2.66	0.15	6.61
	CoT	9.89	2.75	0.20	6.66
	GoT	22.21	2.75	0.15	12.12
	ToT	50.28	2.69	0.29	12.05

Table 3: Computational cost comparison measured on a single NVIDIA A100 GPU. Reasoning-heavy strategies incur substantial latency without commensurate gains in faithfulness.

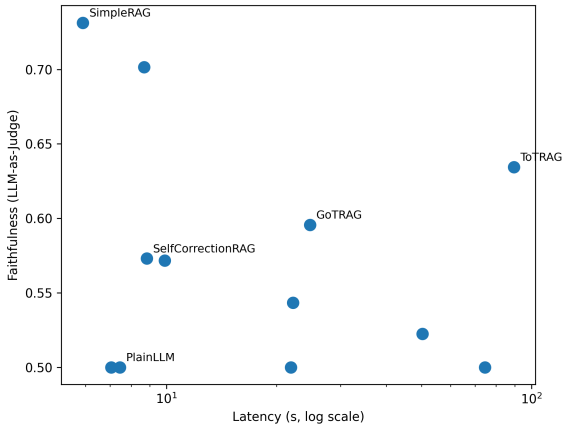

Figure 4: Trade-off between faithfulness and execution latency for all strategies. Latency is shown on a logarithmic scale.

a mean latency of approximately 89.5 seconds, representing a 15× increase over SimpleRAG. This difference is highly statistically significant ($p < 0.001$). Importantly, the increased computational cost does not yield corresponding improvements in faithfulness or semantic quality. Figure 4 shows that SimpleRAG consistently occupies the optimal region of the faithfulness–latency space, achieving the highest correctness at the lowest cost.

Overall, retrieval-based grounding consistently dominates generation-time reasoning in producing faithful docstrings. While self-correction improves over Plain LLMs, it remains unstable and inferior to SimpleRAG. Across all metrics, statistical tests confirm that SimpleRAG (Base) lies on the optimal faithfulness–cost Pareto frontier, achieving the best balance between reliability and efficiency.

6 Discussion

6.1 Why Reasoning Underperforms for Docstring Generation

Our results show that explicit generation-time reasoning strategies—Chain-of-Thought, Tree-of-Thought, and Graph-of-Thought—do not improve faithfulness for retrieval-augmented docstring generation and often degrade it. This outcome is best explained by the nature of the task. Docstring generation is a descriptive, context-bound problem: the objective is to accurately restate information explicitly present in the code and supporting documentation. Unlike tasks that require planning or inference, documentation benefits from strict adherence to observable structure. Reasoning-based prompting expands the model's deliberation space, encouraging speculative inferences about behavior or semantics that are not supported by the input. Without external verification, these inferences frequently lead to unsupported statements—a failure mode we term hallucination by inference. Empirically, reasoning-heavy strategies substantially increase computational cost while yielding equal or lower faithfulness than simpler alternatives.

6.2 Retrieval as a Constraint on Generation

In contrast, retrieval-based grounding acts as an effective constraint on generation. By injecting concrete external context, SimpleRAG narrows the output space to statements supported by retrieved evidence, reducing hallucination without additional control mechanisms. Even lightweight retrieval configurations achieve large gains in faithfulness, indicating that retrieval actively regularizes generation rather than merely supplementing parametric knowledge.

6.3 Implications for Agentic and Self-Correction Frameworks

The performance of Self-Correction RAG highlights important limitations of agentic approaches for documentation tasks. Although iterative critique improves upon Plain LLM baselines, it remains consistently inferior to SimpleRAG and exhibits high variance. Because the critic and generator share the same incomplete context and inductive biases, self-correction often reinforces earlier assumptions rather than correcting them, aligning with prior findings that effective agentic correction requires external verification signals. Overall, our results indicate that for structured, context-bound tasks such as code documentation, grounding mechanisms are more reliable than internal deliberation or self-reflection.

7 Conclusion

This work highlights the importance of task-aware architectural design in large language model–based software engineering systems, showing that increasingly complex reasoning mechanisms do not universally improve reliability. We present a systematic empirical evaluation of retrieval, reasoning, and self-correction strategies for automated Python docstring generation across twelve configurations and multiple quality, coverage, faithfulness, and cost metrics.

Our results demonstrate that retrieval-based grounding is the primary driver of reliable documentation, while generation-time reasoning yields diminishing—and, in several cases, negative—returns. A lightweight Retrieval-Augmented Generation (RAG) baseline consistently outperforms both standalone LLMs and reasoning-heavy pipelines in faithfulness while incurring substantially lower computational costs. In contrast, explicit reasoning strategies—including Chain-of-Thought, Tree-of-Thought, and Graph-of-Thought—significantly increase inference latency without improving factual correctness and often introduce additional unsupported content. We characterize this failure mode as hallucination by inference, where reasoning encourages speculative abstraction that is misaligned with the extractive, context-bound nature of documentation tasks.

These findings have direct implications for the design of automated documentation systems. Rather than indiscriminately applying complex agentic pipelines, retrieval-first architectures offer a more effective, efficient, and sustainable solution for developer-facing tools, where latency and reliability are critical constraints. Future work will extend this analysis to additional programming languages, documentation formats, and codebase-aware retrieval settings, and explore adaptive systems that invoke reasoning only when ambiguity is detected. We hope this study encourages more principled evaluation of when reasoning is necessary—and when efficient grounding alone is sufficient—in practical software engineering applications.

Limitations

These results should not be interpreted as evidence against the usefulness of reasoning strategies in tasks that genuinely require planning, execution, or multi-step inference. Our findings are specific to Python class-level docstring generation, a static and context-bound documentation task where correctness depends primarily on faithful restatement rather than inference. Several additional limitations remain. First, our evaluation focuses on a single programming language and documentation style, and results may not directly generalize to other languages or documentation formats. Second, although the dataset spans diverse GitHub repositories, its size is modest, and larger-scale studies could further validate the observed trends. Third, faithfulness is assessed using an LLM-as-a-Judge, which—while effective for relative comparison—may reflect biases of the evaluator model. Finally, we do not include human evaluation, which could provide complementary insights into developer-perceived usefulness and clarity.

Ethics Statement

We acknowledge two primary ethical considerations in this work. First, AI-assisted tools were used during writing and code development solely to improve clarity and presentation. All experimental design, analysis, and conclusions were developed and verified by the human authors, who take full responsibility for the content.

Second, all experiments were conducted using publicly available, open-source code and datasets under their respective licenses. No private, sensitive, or personally identifiable data were used, and data ownership and licensing constraints were respected.

More broadly, our findings support the responsible and sustainable use of large language models. By showing that lightweight retrieval-based grounding achieves higher reliability than reasoning-heavy pipelines while reducing inference costs by up to 15×, this work promotes efficient deployment and aligns with the principles of Green AI and responsible model usage.

References

Anna Bavaresco, Raffaella Bernardi, Leonardo Bertolazzi, Desmond Elliott, Raquel Fernández, Albert Gatt, Esam Ghaleb, Mario Giulianelli, Michael Hanna, Alexander Koller, Andre Martins, Philipp Mondorf, Vera Neplenbroek, Sandro Pezzelle, Barbara Plank, David Schlangen, Alessandro Suglia, Aditya K Surikuchi, Ece Takmaz, and Alberto Testoni. 2025. LLMs instead of human judges? a large scale empirical study across 20 NLP evaluation tasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 238–255, Vienna, Austria. Association for Computational Linguistics.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michał Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of thoughts: solving elaborate problems with large language models. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press.

Nicola Dainese, Alexander Ilin, and Pekka Marttinen. 2024. Can docstring reformulation with an llm improve code generation? In *Conference of the European Chapter of the Association for Computational Linguistics*.

Sherif Elmitwalli, John Mehegan, Sophie Braznell, and Allen Gallagher. 2025. Scalable evaluation framework for retrieval augmented generation in tobacco research using large language models. *Scientific Reports*, 15.

Max Grusky. 2023. Rogue scores. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1914–1934, Toronto, Canada. Association for Computational Linguistics.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022. Atlas: Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv: 2208.03299*.

Patrick Lewis, Ethan Perez, Aleksandara Piktus, F. Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, M. Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Neural Information Processing Systems*.

Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, Kai Shu, Lu Cheng, and Huan Liu. 2025a. From generation to judgment: Opportunities and challenges of LLM-as-a-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 2757–2791, Suzhou, China. Association for Computational Linguistics.

Feiyang Li, Peng Fang, Zhan Shi, Arijit Khan, Fang Wang, Weihao Wang, Zhangxin-hw, and Cui Yongjian. 2025b. CoT-RAG: Integrating chain of thought and retrieval-augmented generation to enhance reasoning in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 3119–3171, Suzhou, China. Association for Computational Linguistics.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid GNN. In *International Conference on Learning Representations*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Gireesh Sundaram, Balaji Venktesh V, and Sundharakumar K B. 2025. Docstringeval: Evaluating the effectiveness of language models for code explanation through docstring generation. In *2025 International Conference on Emerging Technologies in Computing and Communication (ETCC)*, pages 1–7.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Guang Yang, Yu Zhou, Wei Cheng, Xiangyu Zhang, Xiang Chen, Terry Yue Zhuo, Ke Liu, Xin Zhou, David Lo, and Taolue Chen. 2025. Less is more: Docstring compression in code generation. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Haozhen Zhang, Tao Feng, and Jiaxuan You. 2025a. Graph of records: Boosting retrieval augmented generation for long-context summarization with graphs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23780–23799, Vienna, Austria. Association for Computational Linguistics.

Tianyi Zhang*, Varsha Kishore*, Felix Wu*, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

Yuanliang Zhang, Yifan Xie, Shanshan Li, Ke Liu, Chong Wang, Zhouyang Jia, Xiangbing Huang, Jie Song, Chaopeng Luo, Zhizheng Zheng, Rulin Xu, Yitong Liu, Si Zheng, and Xiangke Liao. 2025b. Unseen horizons: Unveiling the real capability of llm code generation beyond the familiar. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 604–615. IEEE Press.

Yuxin Zhang, Yuxia Zhang, Zeyu Sun, Yanjie Jiang, and Hui Liu. 2025c. Laura: Enhancing code review generation with context-enriched retrieval-augmented llm. *Preprint*, arXiv:2512.01356.

A Appendix

A.1 Output of generated docstrings:

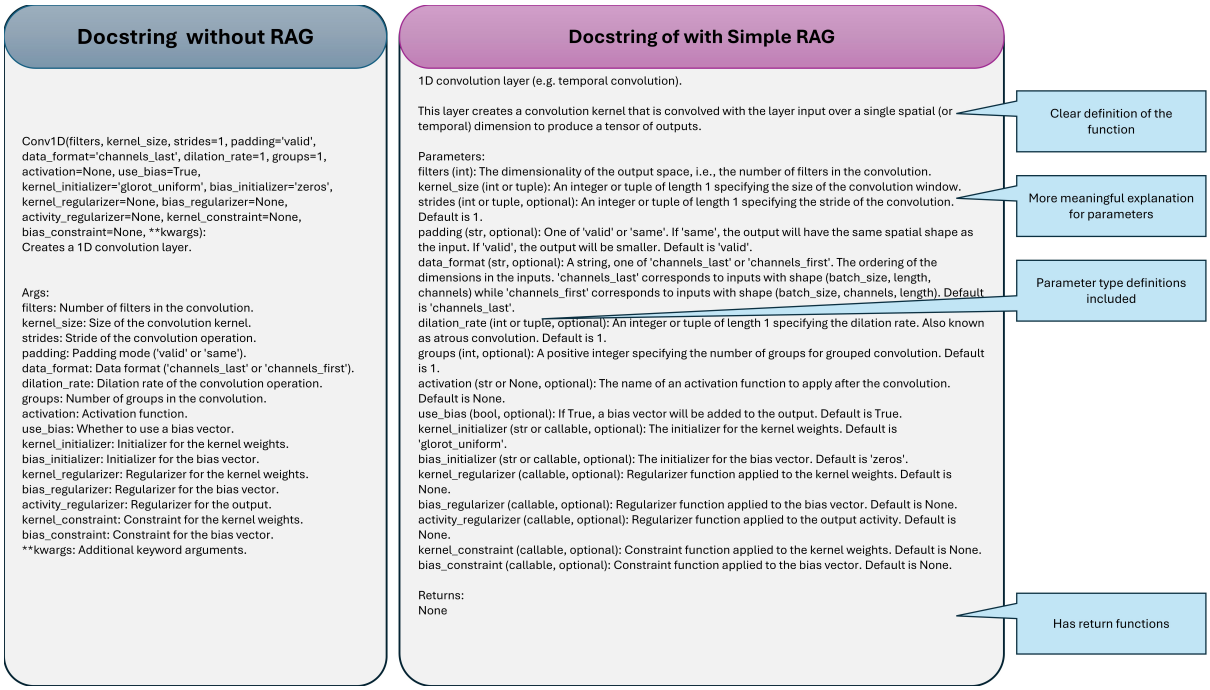

Figure 5: This image shows two docstrings generated. The one on the left shows the docstring generated without RAG and one on the right shows with RAG.

A.2 System Prompts:

```
You are an expert Python programmer and documentation specialist with deep knowledge of PEP 257 standards and Google-style docstring formatting.

**Your Role:**
- Generate comprehensive, accurate docstrings following PEP 257 standards
- Use Google-style docstring format with proper sections
- Include type hints and detailed parameter descriptions
- Document exceptions and return values thoroughly

**Context Usage:**
- Use retrieved context only when directly relevant to the code
- Incorporate context insights to enhance accuracy
- Avoid including irrelevant information from context

**Output Requirements:**
- Return ONLY the docstring content starting with triple quotes
- Do NOT include the original Python code
- Follow proper indentation and formatting
```

Figure 6: System Prompt

```
Generate a comprehensive Python docstring
following PEP 257 standards and Google-style
formatting.

**CRITICAL REQUIREMENT:**
- Return ONLY the docstring content (the text
between triple quotes)
- Do NOT include the original Python code
- Do NOT include the function/class definition

**Context Integration:**
- Use any relevant context provided earlier to
enhance accuracy
- Incorporate context insights into parameter
descriptions

**Code to document:**
```python
{code}
```

Figure 7: Final Generation Prompt
```

```
Generate a comprehensive search query to find
relevant documentation for creating Python
docstrings.

**Code Analysis:**
Analyze the provided Python code to extract:
- Function/class name and purpose
- Parameter names and types
- Return type and value

**Search Strategy:**
Create a query that will retrieve:
- PEP 257 documentation and examples
- Similar function documentation patterns

**Code to analyze:**
{code}

**Generate a focused search query for this
code.**
```

Figure 8: Retrieval Context Query Generation Prompt

A.3 Reasoning Prompts:

```
Generate a Python docstring for the code
below.

**Instructions:**
1. **Think Step-by-Step**: First, analyze
the code logic, parameters, returns, and
exceptions. Explain your reasoning clearly.
2. **Generate Docstring**: Based on your
analysis, generate the final docstring.

**Format:**
[REASONING]
<Your step-by-step analysis here>
[/REASONING]

[DOCSTRING]
<Your final docstring here (triple-quoted)>
[/DOCSTRING]

**Code:**
{code}
```

Figure 9: Chain of Thought Prompt

```
Decompose the task of generating a docstring
for the following Python class into multiple
alternative plans.

    • Generate three distinct docstring drafts
      focusing on different aspects of the
      class.

    • Evaluate which draft best covers purpose,
      parameters, returns, and exceptions.

    • Synthesize the best elements into a final
      PEP-style docstring.

      Avoid unsupported assumptions or
      hallucinated details.


**Code:**
{code}
```

Figure 10: Tree of Thought Prompt

```
Independently analyze the following aspects of
the Python class:

    • Class purpose

    • Parameters and attributes

    • Return behavior

    • Exceptions

      Combine these analyses into a single
      coherent PEP-style docstring. Ensure all
      statements are directly supported by the
      code or retrieved context.


**Code:**
{code}
```

Figure 11: Graph of Thought Prompt

```
Evaluate the quality of the generated Python docstring
against the original code using comprehensive criteria.

**Evaluation Framework:**
1. **Completeness**: Documents all
parameters/returns/exceptions
2. **Accuracy**: Descriptions match implementation
3. **Formatting**: PEP 257 compliance
4. **Clarity**: Professional language

**Scoring:**
- EXCELLENT (7-8 points)
- GOOD (5-6 points)
- NEEDS_IMPROVEMENT (0-4 points)

**Code to evaluate:**
{code}

**Generated docstring:**
{initial_docstring}

**Assessment (EXCELLENT/GOOD/NEEDS_IMPROVEMENT):**
```

Figure 12: Iterative Critique RAG – Critique Prompt

```
You are a strict technical judge evaluating Python
docstrings.

Context from Knowledge Base:
{retrieved_context}

Generated Docstring:
{generated_docstring}

Evaluation Rubric:
1. Hallucination Check: Does the docstring claim
parameters/returns not present in the code or context?
2. Contradiction Check: Does it contradict the provided
context logic?
3. Support Check: Is the description supported by the
context?

Task:
Assign a Faithfulness Score from 0.0 to 1.0.
- 1.0: Fully supported by context/code, no hallucinations.
- 0.5: Partially supported, some generic descriptions.
- 0.0: Major hallucinations or contradictions.

Return ONLY the numeric score in this format: Score:
<number>
```

Figure 13: LLM-as-a-Judge Prompt

A.4 Evaluation Framework:

To ensure a holistic assessment of the generated docstrings, we employed a comprehensive suite of evaluation metrics categorized into two main areas: code evaluation metrics and documentation coverage metrics.

A.4.1 Code evaluation metrics

These metrics assess the fundamental quality of the generated natural language text. We used 5 key metrics: ROUGE-1 score, BLEU score, BERT score, ease of reading, and conciseness. For measuring accuracy, the main metric that we use is the BERT score. The BERT score is calculated as follows:

$$BERT = 2 \times \frac{P_{BERT} \times R_{BERT}}{P_{BERT} + R_{BERT}}$$

Where P_{BERT} is the Precision BERT score, and R_{BERT} is the Recall BERT score.

To calculate the ease of reading, we use the Flesch reading ease metric, which is calculated based on the ratio of total words to total syllables, as mentioned below:

$$Clarity = \frac{206.825 - 1.5\frac{Total_w}{Total_s} - 84.6\frac{Total_{sy}}{Total_w}}{100}$$

where w are words, s are sentences, and sy is the generated syllabus.

Conciseness measures how precise the docstring is.

$$Conciseness = \frac{Compress_{Generated\ text}}{Compress_{Ground\ truth}}$$

The ideal conciseness score will be around 1, which indicates that the compressed text of the ground truth and the generated text are of the same length.

A.4.2 Documentation Coverage Metrics

These metrics evaluate how well the docstring documents the functional interface of the code.

Parameter Coverage measures the proportion of function and method parameters that are mentioned in the generated docstring vs. the actual code.

$$Parameter\ coverage = \frac{D_{Parameters}}{C_{Parameters}}$$

where $D_{Parameters}$ are the generated docstring parameters, and $C_{Parameters}$ are the code docstring parameters.
Return coverage is a binary score that checks whether a return statement in the code is documented.

$$\text{Return coverage} = \begin{cases} 1 & \text{if code has docstring and no return statement} \\ 1 & \text{if code and docstring both have a return statement} \\ 0 & \text{if code has a return statement but the docstring does not, or vice versa} \end{cases} \quad (1)$$

Faithfulness scores represent the relationship between the tokens in the generated docstrings and the retrieved context.

$$Faithfulness\ score = \frac{D_{Tokens} \cap RC_{Tokens}}{D_{Tokens}}$$

where D_{Tokens} represents tokens generated by the docstring, and RC_{Tokens} represents the retrieved context.

Exception Score measures the proportion of explicitly raised exceptions in the code that are mentioned in the docstring.

$$Exception\ coverage = \frac{D_{Exception}}{C_{Exceptions}}$$

where $D_{Exceptions}$ stands for the number of exceptions raised in the generated docstring, and $C_{Exceptions}$ is the number of exceptions in the code.