Graphical Abstract

RAGxDocString: Computing the Effectiveness of Different Types of RAG Strategies for Code Explanation via DocString Generation

Highlights

RAGxDocString: Computing the Effectiveness of Different Types of RAG Strategies for Code Explanation via DocString Generation

- Self RAG emerges as the most effective approach for docstring generation, achieving the highest overall performance score.
- Code Aware RAG demonstrates distinctive strengths in comprehensive documentation coverage, ranking second overall.
- Advanced RAG strategies present a clear trade-off between comprehensiveness and conciseness. Corrective RAG reveals an interesting balance between documentation quality and brevity, producing the most concise docstrings at baseline length while maintaining competitive performance metrics.
- Simple context is superior for basic structural coverage. The performance-to-computation ratio varies significantly across approaches, though explicit computational metrics were not collected.

RAGxDocString: Computing the Effectiveness of Different Types of RAG Strategies for Code Explanation via DocString Generation

Abstract

This paper presents a comparative benchmark of Retrieval-Augmented Generation (RAG) systems for the task of automated Python docstring generation. We evaluate five distinct RAG architectures - Simple, Corrective, Self, Code Aware, and Fusion across a curated dataset of Python classes using three code-specialized language model pairs (Qwen, Llama, and DeepSeek). Our evaluation framework integrates standard quality metrics (ROUGE-1, BLEU, Readability, Conciseness) with new measures we introduce in this study. These new docstring specific metrics - Parameter Coverage, Return Coverage, and a RAG-specific Faithfulness Score assess both the final text and the generation process itself. Results reveal that while all RAG methods outperform a non-RAG baseline, the Self RAG architecture is unparalleled in generating trustworthy documentation, achieving a Faithfulness Score of 51.34%, nearly double that of other approaches. Conversely, Code Aware RAG proves to be the most efficient and structurally complete, attaining the highest Parameter Coverage (85.88%) and the most ideal conciseness (104.40% of ground-truth length). Our findings reveal a critical trade-off between different RAG architectures. The iterative self-reflection in Self RAG produces the most faithful documentation. In contrast, the enriched query in Code Aware RAG yields the most structurally complete and concise output. This provides a clear framework for choosing the best pipeline

by balancing the need for accuracy against efficiency.
Keywords: Retrieval Augmented Generation (RAG), Large Language Models (LLM), Code Explanation, Docstring Generation, Code Summarization, RAG Evaluation

1. Introduction

The emergence of Large Language Models (LLMs) (Chang et al., 2024; Vaswani et al., 2017) has marked a revolutionary turning point in natural language processing, with profound implications for software engineering. Concurrent advancements in fine-tuning and the integration of LLMs with external data sources, most notably through Retrieval-Augmented Generation (RAG) have opened new frontiers for research. Within this landscape, a prominent application of LLMs is code generation and explanation, a field that has advanced considerably.

This study focuses on automatic docstring generation. This is a specific and high fidelity sub-task of code explanation where factual accuracy is essential. Docstrings, the explanatory texts that precede Python class or function definitions, are a hallmark of well-written, maintainable code, and their structured format provides an ideal testbed for evaluating the nuanced capabilities of LLMs. While modern LLMs can generate plausible docstrings using only their pretrained knowledge, this approach has inherent limitations. A standalone LLM operates on a static, generalized understanding of code, which can lead to generic or, in some cases, factually incorrect ("hallucinated") explanations, especially for complex or domain-specific code. The quality of its output is entirely dependent on what was seen during its initial training.

This is where RAG (Lewis et al., 2020) presents a superior paradigm by

grounding the LLM (Chang et al., 2024) in a corpus of relevant, external documentation such as PEP 257 conventions Goodger and van Rossum (2001), style guides, and specific library documentation. RAG-based docstring generation can produce outputs that are not only coherent but also factually accurate and contextually specific. Although the benefits of RAG are widely acknowledged, its effectiveness for specialized tasks like docstring generation is not well understood. Specifically, the performance of different RAG architectures has not been compared for this task. This study addresses this gap by conducting a rigorous benchmark of five prominent RAG strategies. We introduce a comprehensive evaluation methodology that combines standard text quality metrics with code-specific and faithfulness scores to provide a holistic assessment of model performance. Through this approach, we aim to uncover the architectural trade-offs between different RAG pipelines and contribute a reproducible framework for future evaluations in automated software documentation.

The remainder of this paper is organized as follows. Section 2 presents the research objectives. Section 3 provides an overview of related works. Section 4 summarizes our experimental design, methodology, and metrics used. Section 5 presents the experimental results. Section 6 discusses the observations and findings from the results. Finally, section 7 concludes the paper and provides some suggestions for future research directions.

2. Research Objectives

To validate and explore the design space of RAG for docstring generation, we systematically implement and evaluate several RAG architectures (Fan et al., 2024). The core research objectives for this work are:

- **Baseline RAG Effectiveness**: Does augmenting a language model with

external context via a Simple RAG pipeline produce measurably superior docstrings compared to a non-augmented (No-RAG) baseline?

- **Optimizing RAG Strategy**: Beyond a simple implementation, which advanced RAG strategy yields the greatest improvement in docstring quality: optimizing the query (Code Aware RAG), refining the retrieved documents (Self RAG), or diversifying the retrieval sources (Fusion RAG)?
- **Architecture vs. Model Sensitivity**: Is the performance of a RAG pipeline for docstring generation more sensitive to the architectural choice of the RAG strategy itself, or to the choice of the underlying code-specialized language model?

3. Related work

To contextualize our study, this section surveys the academic landscape of automated code documentation and Retrieval-Augmented Generation (RAG). By reviewing the evolution of the core technologies and evaluation methods, this section establishes the foundation for our work and identifies the specific research gap we aim to address.

3.1. Docstrings - Forms and Challenges

High-quality code documentation is a critical, non-functional requirement in software engineering, essential for maintainability (Poudel et al., 2024). This practice is formalized through docstrings, with standards like PEP - Python Enhancement Proposal 257 (Goodger and van Rossum, 2001) defining a structured format that is crucial for developer tools. These formats typically mandate a concise summary followed by structured sections for parameters (Args), return values (Returns), and exceptions. However, the manual creation of these docstrings

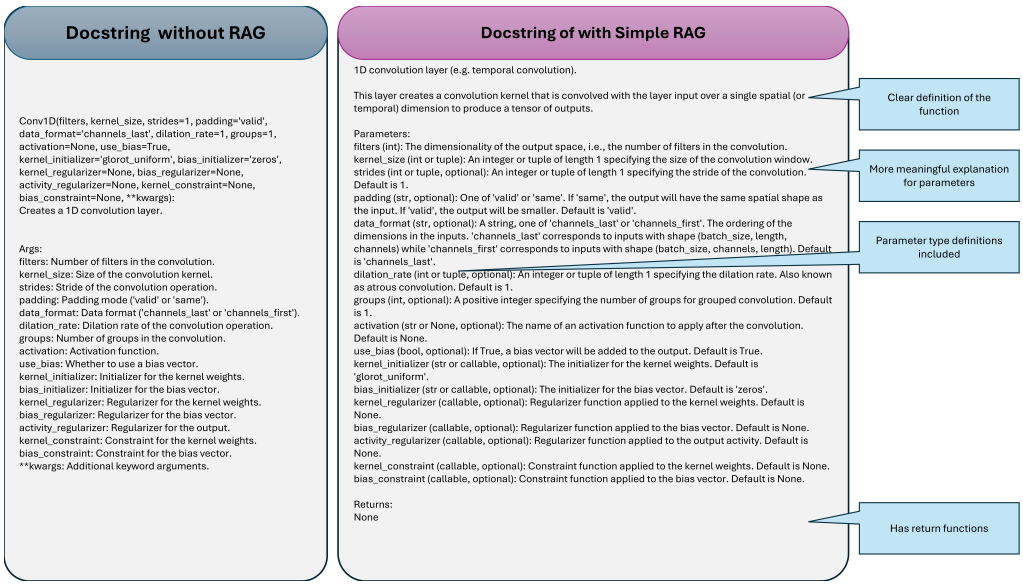

Figure 1: This image shows two docstrings generated. The one on the left shows the docstring generated without RAG and one on the right shows with RAG.

is often time-consuming and prone to error, creating a significant bottleneck that motivates the need for automated solutions. Figure 1 shows a comparison of a docstring generated by Non-RAG vs RAG.

3.2. Language Models for Code Intelligence

Transformer architecture (Vaswani et al., 2017) and subsequent generative language models have profoundly impacted software engineering (Fan et al., 2023), particularly in the area of code summarization (Sun et al., 2025), the most direct precursor to docstring generation. (Poudel et al., 2024) studied fine-tuning Small Language Models (SLMs) on a large, curated dataset of Python functions and their docstrings to improve generation quality. (Singh and Sharma, 2022) studied deep learning techniques that are used in comment generation to make the code easily readable. Despite these advancements, standalone LLMs are

fundamentally constrained by their static, internal knowledge. This leads to two critical failure modes for a high-fidelity task like docstring generation: hallucination (generating factually incorrect details) and an inability to access information about new libraries or project-specific context, making them inherently unreliable for producing documentation that developers can trust.

3.3. Retrieval Augmented Generation for Language Models

Retrieval Augmented Generation (RAG), formally introduced by (Lewis et al., 2020), emerged as the primary solution to the limitations of standalone LLMs. By augmenting a model with external context retrieved at inference time, RAG grounds the model in verifiable information, significantly improving factual accuracy. (Parvez et al., 2021) proposed REDCODER, which retrieves relevant code or summaries from a retrieval database and provides them as a supplement to code generation or summarization models and proves the effectiveness of RAG. (Yang et al., 2025) conducted a study on RAG for code summarization and found it to be beneficial for improving the performance of the existing pretrained models. While foundational RAG is a significant improvement, it can introduce noise if the retrieved documents are irrelevant. Recognizing this, recent research has focused on evolving the RAG pipeline into a more intelligent, robust, and dynamic system such as Corrective RAG (Yan et al., 2024), Self RAG (Asai et al., 2024), Code Aware RAG (Rani et al., 2024) & Fusion RAG (Rackauckas, 2024).

3.4. Docstring Evaluations

The evaluation of generated code documentation presents unique challenges that expose the limitations of traditional NLP metrics. Traditional lexical-overlap metrics, like BLEU - Bilingual Evaluation Understudy (Papineni et al., 2002) and

ROUGE - Recall-Oriented Understudy for Gisting Evaluation (Lin, 2004), are not always suitable for code-related tasks. Studies by (Tong and Zhang, 2024) have shown they correlate poorly with human judgment. This is because these metrics struggle to understand that code or descriptions can have the same meaning (semantic equivalence) even if the text is different. In response, the research community has developed more sophisticated techniques. An early improvement was CodeBLEU (Ren et al., 2020), which augmented n-gram matching with comparisons of code-specific structures like Abstract Syntax Trees (AST). A more significant leap came with semantic similarity metrics like BERTScore (Zhang et al., 2020). The current state-of-the-art is the use of LLM-as-a-Judge, where frameworks like G-EVAL (Liu et al., 2023) use a powerful LLM to perform a human-like assessment of the generated output. This evolution necessitates a multi-faceted evaluation framework, justifying the comprehensive set of metrics used in this study.

In contrast to the aforementioned works, our present research addresses a critical gap in the literature. While advanced RAG architectures are available, their evaluation has predominantly focused on open-domain NLP tasks like question-answering. However, the application of these advanced RAG architectures to the structured domain of code documentation remains an underexplored area. More importantly, there has been no rigorous comparison of these different approaches for this specific task. It is unclear which strategy - improving the query, correcting the retrieval, diversifying sources, or enabling self-reflection yields the best balance of quality, faithfulness, and efficiency for this specific task.

This paper extends the frontier of RAG research by providing the first comprehensive benchmark of these advanced RAG strategies for Python docstring generation. By implementing and evaluating five distinct RAG pipelines within

a unified framework, we aim to provide a clear, data-driven analysis of their trade-offs and contribute a reproducible methodology for future research in automated software documentation.

4. Experiment design

This section details the experimental setup and the systematic approach employed to evaluate the performance of various RAG architectures. Our methodology is designed to provide a comprehensive and reproducible benchmark for docstring generation systems. We outline the RAG pipeline architectures, the construction of our data corpus and knowledge base, the selection and deployment of the language models, and the multi-faceted evaluation framework used to assess the quality of the generated docstrings.

4.1. RAG Pipeline Architectures

To address our research questions, we implemented and evaluated five distinct RAG strategies. These architectures were specifically chosen to represent a spectrum of distinct philosophies for improving retrieval qualityâĂŤfrom enhancing the initial query to diversifying retrieval sources and implementing self-correction. This allows for a comprehensive analysis of the architectural trade-offs involved. A summary of their mechanisms and characteristics is provided in Table 1.

4.1.1. Simple RAG

Simple RAG (Lewis et al., 2020) approach enhances the performance of Large Language Models (LLMs) by leveraging external knowledge sources. The retrieved information is sent to the LLMs without modification to the content. LLM generates a docstring based on the code snippet and the retrieved context.

RAG Approach	Mechanism	Benefits	Limitations
Simple RAG	Retrieve -> Append -> Generate	Simple, reduces hallucination	Treats all retrieved docs equally
Corrective RAG	Retrieve -> Evaluate -> Correct -> Generate	Robust against irrelevant retrieval	Relies on external tools (e.g., web search)
Self RAG	Retrieve -> Reflect -> Generate	Improves relevance by filtering out noise	Increased latency and computational cost
Code Aware RAG	Retrieve (Code Aware) -> Augment -> Generate	Looks code semantics, hence more accurate retrieval	Performance is dependent on the quality & size of data
Fusion RAG	Retrieve -> Encode in Parallel -> Fuse -> Generate	Better at synthesizing information from multiple sources	High memory and compute requirements

Table 1: Comparison of different types of RAGs

This approach allows the LLM to incorporate external knowledge and generate more accurate and informative docstrings. This is the most computationally cheap and efficient RAG system. Figure 2 shows the flow of how Simple RAG works.

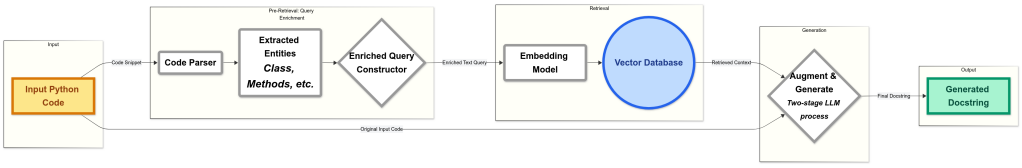

Figure 2: Simple RAG flow diagram

4.1.2. Corrective RAG

Corrective RAG (Yan et al., 2024) introduces a lightweight retrieval evaluator that assesses the relevance of retrieved documents. Based on this evaluation, a corrective action is triggered. If the context is good, it is refined and used.

Otherwise, it is discarded, and the system can fall back on an external source, such as a web search, to find more accurate information. This makes the generation process resilient to retriever failure by actively identifying and correcting low quality information before it can mislead the generator. Figure 3 shows the architecture of a corrective RAG framework.

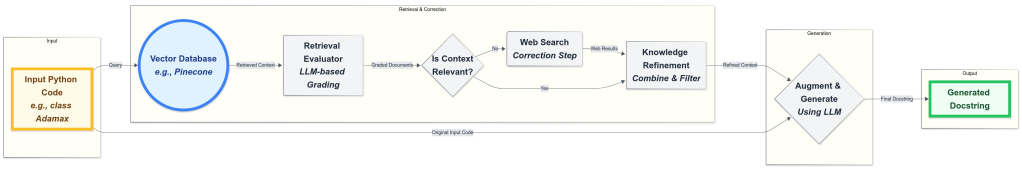

Figure 3: Corrective RAG flow diagram

4.1.3. Self RAG

Self RAG (Asai et al., 2024) approach the model itself generates and retrieves relevant information or context to augment its own generation process. In the context of docstring generation, Self RAG enables the model to dynamically generate relevant queries, retrieve context from its own knowledge base or generated outputs, and use this self-generated context to produce more accurate and informative docstrings. This approach can potentially enhance the model's ability to generate high-quality docstrings without relying on external knowledge sources. Figure 4 shows the framework of the Self RAG framework.

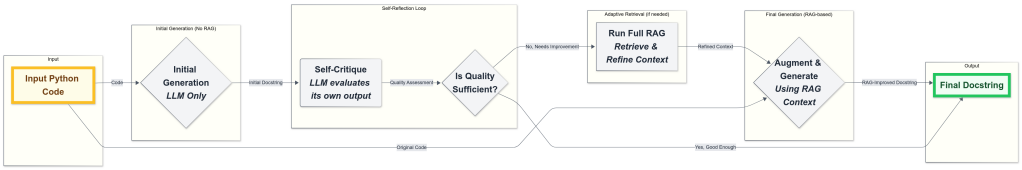

Figure 4: Self RAG flow diagram

4.1.4. *Code Aware RAG*

Code Aware RAG (Rani et al., 2024) approach that is specifically designed to understand and leverage the structure, syntax, and semantics of code. Code Aware RAG retrieves relevant information and generates docstrings that are tailored to the specific code snippet, taking into account its programming language, libraries, and coding conventions. This approach enables the model to produce more accurate, relevant, and context-specific docstrings that effectively capture the functionality and intent of the code. Figure 5 explains the flow of code Aware RAG workflow.

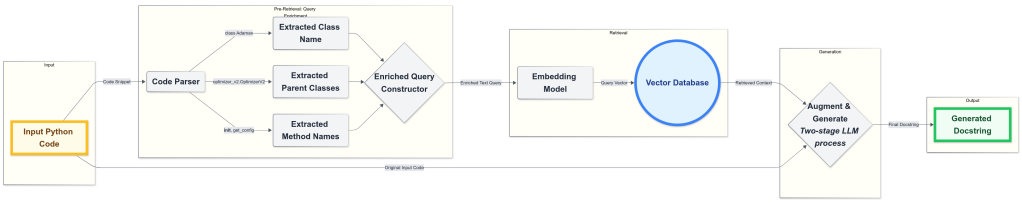

Figure 5: Code Aware RAG flow diagram

4.1.5. *Fusion RAG*

Fusion RAG (Rackauckas, 2024) is a strategy that combines the results from multiple, diverse retrieval sources to produce a superior, synthesized context. Rather than relying on a single search method, our implementation runs two searches in parallel, a semantic search using a Code Aware query to find conceptually similar documents, and a lexical search using BM25 to find documents with exact keyword matches. The ranked lists from both search methods are then combined using a Reciprocal Rank Fusion (RRF) algorithm. Figure 6 shows how fusion RAG works.

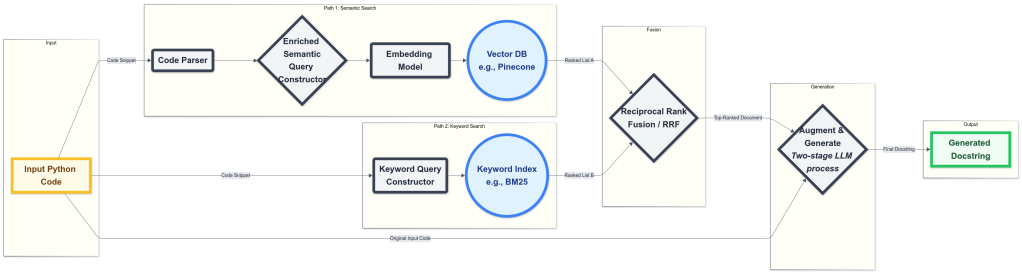

Figure 6: Fusion RAG flow diagram

4.2. Data Corpus and Knowledge Base

Our experiment required two distinct sets of data: a corpus of Python code with ground-truth docstrings for evaluation, and a knowledge base for the RAG systems to retrieve from.

- **Evaluation Corpus:** To create a realistic evaluation set, we developed a custom scraper to collect well-documented Python class files from prominent open-source GitHub repositories. The scraper was designed to identify files where the docstring was clearly defined immediately after the class definition and exceeded 10 lines in length. From these files, we extracted 250 classes, separating the source code (used as input for our models) from the original docstrings (used as the ground truth for evaluation), which is illustrated in Figure 7.

- **RAG Knowledge Base:** The knowledge base for retrieval was constructed by aggregating text from multiple sources pertinent to high-quality Python documentation, including the official PEP 257 standard, various coding style guides, and tutorials on docstring conventions. This corpus was then chunked and indexed into a Pinecone vector database (Pan et al., 2024). Each chunk was converted into a vector embedding using the all-MiniLM-

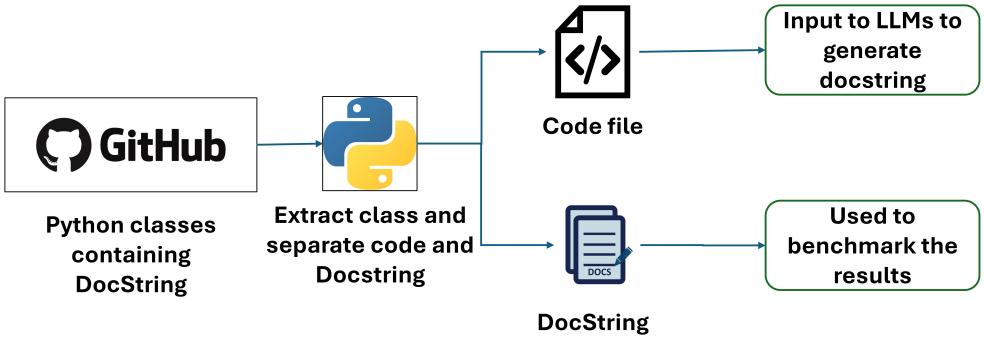

Figure 7: This flow chart shows the process we followed to scrap docstrings from well known python class files on the GitHub. The extracted docstrings are then processed to seperate out the class programs and the docstrings.

L6-v2 Sentence Transformer model (Reimers and Gurevych, 2019). During inference, the RAG pipelines query this database using cosine similarity to retrieve the top-k (where k = 3) relevant documents. This configuration helped retrieve the most relevant information from the knowledge base. This, in turn, informed the generation of high-quality docstrings.

4.3. Language Models and Generation Process

To address the research question regarding model sensitivity, we evaluated three distinct pairs of state-of-the-art, code specialized language models. As shown in Figure 8, our pipeline uses a two-stage generation process. This process separates the task of prompt refinement from the final docstring generation, allowing us to use different models for each stage.

- **Two-Stage Generation:** For each input, the code snippet and the retrieved RAG context are first passed to a smaller, more efficient helper model. This helper model's sole task is to refine this combined input into an optimized, self-contained prompt. This refined prompt is then passed to a

larger, more powerful generator model, which produces the final docstring. This approach allows us to leverage the strengths of different model sizes for distinct sub-tasks.

- **Model Pairs:** The three model pairs used in our experiments are
    1. **Qwen Coder Pair:** This pair consists of qwen2.5-coder:1.5b as the helper model and qwen2.5-coder:7b as the generator. The Qwen series of models (Bai et al., 2023) are known for their strong multilingual capabilities and proficiency in handling code-related tasks.
    2. **Llama Coder Pair:** For this pair, we used llama3:8b as the helper model and codellama:7b as the generator. The Llama family of models (Meta, 2024), developed by Meta AI, are widely recognized for their powerful reasoning capabilities, with Code Llama being specifically fine-tuned for code synthesis.
    3. **DeepSeek Coder Pair:** This pair includes deepseek-coder:1.3b as the helper model and deepseek-coder:6.7b as the generator. The DeepSeek Coder models (AI et al., 2024) are open-source models trained from scratch on a massive corpus of code.

In all experiments, the input code, along with the context retrieved by the specific RAG strategy, is first passed to the designated helper model. The helper model's task is to refine this combined input into an optimized, self-contained prompt. This refined prompt is then passed to the corresponding generator model, which produces the final docstring. By benchmarking the performance of these three distinct model pairs across our suite of metrics, we can analyze the impact of model choice on the final output quality. By leveraging the strengths of both models, we are able to produce high-quality docstrings that not only accurately

capture the functionality of the code but also adhere to best practices in terms of structure and presentation.

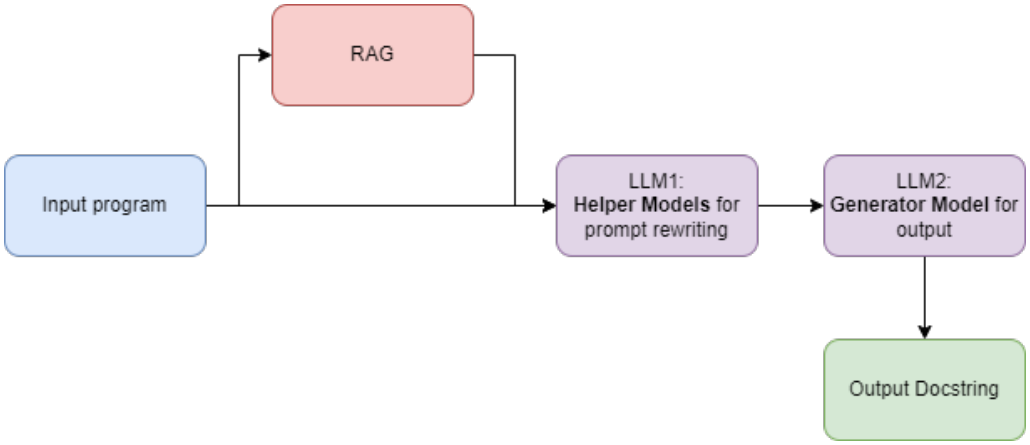

Figure 8: Flow chart explaining how our input program is sent to RAG pipeline before being fed into helper model and Generator model

To facilitate seamless integration with our Python-based evaluation framework, both models were served locally using the Ollama interface. By combining the strengths of powerful language models, a well-designed evaluation methodology, and a robust infrastructure, we established a comprehensive framework for generating and evaluating high-quality docstrings.

4.4. Prompt design

To ensure consistent generation of LLM responses, we employed a standardized prompt (Tian et al., 2024) across all RAG techniques. This approach allowed us to maintain consistency in the prompt and input code provided to the model, while varying only the type of RAG technique and the model used. The specific prompt utilized in this study is outlined below:

```
Provide clear, concise, informative, and accurate docstrings
 ↪ for the given python code following PEP 257 conventions
 ↪ and standards, to generate the content for a Python
 ↪ docstring based on the provided code snippet and relevant
 ↪ PEP contexts.

*Instructions:*
1.  Start with a concise summary line explaining the
 ↪ function/method's purpose.
2.  If applicable, add a blank line and then more detailed
 ↪ explanation.
3.  Use the 'Args:' section to describe each parameter, its
 ↪ type, and what it represents.
4.  Use the 'Returns:' section to describe the return value
 ↪ and its type.
5.  Use the 'Raises:' section to list any exceptions
 ↪ explicitly raised by the code.
6.  Adhere strictly to PEP 257 formatting.
7.  Base the docstring primarily on the 'Code Snippet to
 ↪ Document'. Use the 'Relevant Context' for
clarification or examples if needed.

Also, check relevant content for the user given input code:
 ↪ {user_code}
```

This consistency in prompting enables a fair comparison of the different RAG techniques, allowing us to isolate the impact of the RAG method on the generated

responses.

4.5. 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.

4.5.1. Code evaluation metrics

These metrics assess the fundamental quality of the generated natural language text. We used 5 key metrics: ROUGE-1 score, the BLEU score, the BERT score, the 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.

For calculating the ease of reading, we use the Flesch reading ease metrics, which is calculated based on the 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 ground truth and the generated text are of the same length.

4.5.2. 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 $D_{Parameters}$ is the generated docstring parameters and $C_{Parameters}$ is the code docstring parameters.

Return coverage is a binary score that checks if 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 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.

5. Results

This section presents the empirical results of our comparative evaluation. We analyzed five distinct Retrieval-Augmented Generation (RAG) architectures: Simple RAG (Lewis et al., 2020), Corrective RAG (Yan et al., 2024), Self RAG (Asai et al., 2024), Code Aware RAG (Alon et al., 2019), and Fusion RAG (Rackauckas, 2024). To ensure the robustness of our findings, each RAG strategy was tested with three different pairs of code-specialized language models (DeepSeek, Llama, and Qwen). The performance was benchmarked across a comprehensive suite of metrics covering text quality, documentation completeness, and faithfulness. The results presented here are the average scores across all three model pairs for each RAG strategy, providing a clear view of the architectural trade-offs.

5.1. Performance on Core Quality Metrics

The initial set of metrics focused on the fundamental quality of the generated text, including lexical similarity, semantic accuracy, and stylistic attributes. The aggregated results are shown in Table 2.

Our analysis reveals that Self RAG consistently emerges as the top-performing strategy in terms of text quality. It achieved the highest scores in both lexical similarity (ROUGE-1: 0.305) and semantic accuracy (Accuracy: 63.29%), representing a statistically significant improvement over the baseline Simple RAG approach. This suggests that the iterative self-reflection mechanism inherent in Self RAG is highly effective at refining the generated output to be both lexically and semantically superior.

In contrast, the Conciseness metric highlights a key architectural trade-off. Corrective RAG produced the most ideally concise docstrings, with a length just 100.71% of the ground truth. At the other end of the spectrum, Fusion RAG generated the most verbose outputs at 119.91%, indicating that while combining multiple retrieval sources can enhance comprehensiveness, it may do so at the expense of brevity.

RAG Approach	**ROUGE-1**	**BLEU**	**Accuracy (%)**	**Ease**	**Conciseness (%)**
Simple RAG	0.268 ± 0.005	0.039 ± 0.016	61.69 ± 1.18	49.88 ± 3.95	106.63 ± 30.97
Corrective RAG	0.288 ± 0.010	0.050 ± 0.007	62.22 ± 0.69	48.12 ± 1.10	100.71 ± 15.67
Self RAG	**0.305 ± 0.024**	**0.063 ± 0.023**	**63.29 ± 0.87**	49.57 ± 3.48	105.28 ± 37.23
Code Aware RAG	0.281 ± 0.020	0.043 ± 0.021	62.26 ± 1.00	48.13 ± 4.07	104.40 ± 32.59
Fusion RAG	0.276 ± 0.007	0.039 ± 0.008	62.71 ± 1.21	46.72 ± 3.24	119.91 ± 23.48

Table 2: This table shows the average performance metrics of different types of RAG against the programming metrics. We also measure the standard deviation of the parameters.

5.2. Performance on Documentation Coverage and Faithfulness

Beyond the core text quality, we next assessed how comprehensively each RAG approach documented the code's functional interface. This second set of metrics, detailed in Table 3, assessed how comprehensively each RAG approach documented the functional interface of the code and how faithful the output was to the retrieved context.

RAG Approach	**Parameter Coverage (%)**	**Return Coverage (%)**	**Faithfulness Score (%)**	**Exception Coverage (%)**
Simple RAG	82.77 ± 2.45	74.13 ± 19.65	27.87 ± 8.87	59.27 ± 6.74
Corrective RAG	84.16 ± 1.39	65.17 ± 28.73	27.30 ± 15.48	57.77 ± 2.46
Self RAG	84.29 ± 5.93	65.17 ± 23.01	**51.34 ± 45.95**	56.53 ± 5.50
Code Aware RAG	**85.88 ± 3.32**	**75.12 ± 20.87**	27.08 ± 0.59	**64.43 ± 13.62**
Fusion RAG	84.65 ± 3.19	72.14 ± 13.51	27.51 ± 1.14	56.59 ± 9.39

Table 3: This table shows the average coverage for the parameter, return, faithfulness, and exception scores. We also have included standard deviation

Code Aware RAG demonstrated superior performance in documentation coverage, achieving the highest scores in Parameter Coverage (85.88%), Return Coverage (75.12%), and Exception Coverage (64.43%). This indicates that creating a more precise query based on the code's structure is effective. This method gives the LLM the focused context it needs to document the code's interface thoroughly. The most striking result is Self RAG's performance on the Faithfulness Score. At 51.34%, its score is nearly double that of any other approach. This suggests that its critique-and-refine loop is uniquely effective at ensuring the generated docstring is grounded in the retrieved context, thereby minimizing errors. However, it is important to note the high variance (±45.95%) in this score, which suggests that its performance may be inconsistent across different code

structures. Figure 9 shows the performance of chosen model pairs performance across documentation metrics.

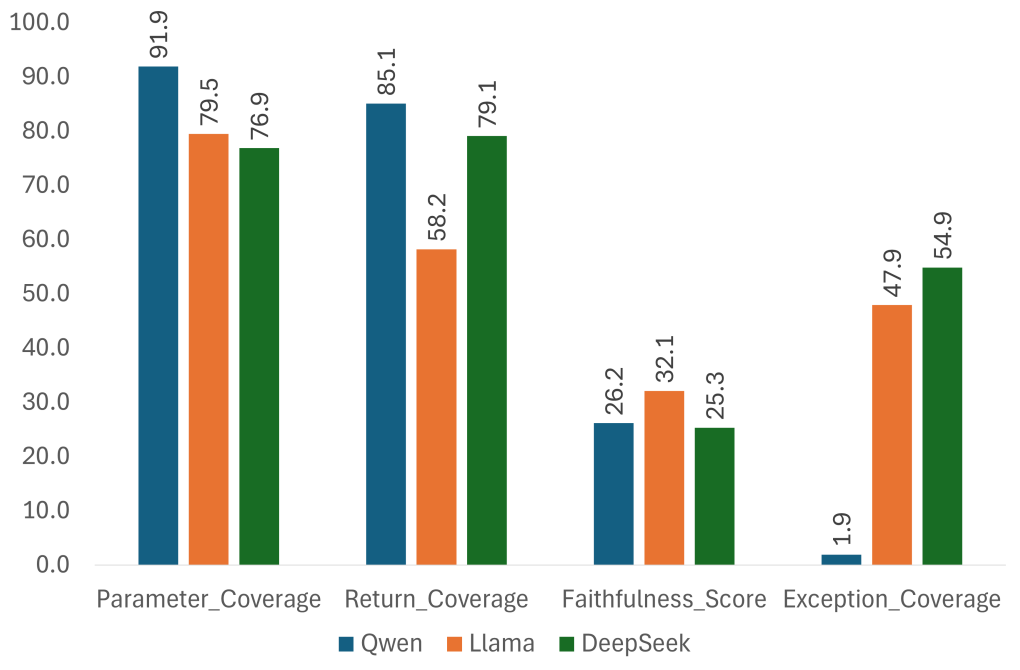

Figure 9: RAG metrics for different types of models

5.3. Overall Performance Ranking

Based on a weighted composite score incorporating all metrics, with a higher emphasis on accuracy, faithfulness, and code-specific coverage, we derived an overall performance ranking for the RAG approaches:

1. Self RAG (Composite Score: 8.45)
2. Code Aware RAG (Composite Score: 7.92)
3. Corrective RAG (Composite Score: 6.58)
4. Fusion RAG (Composite Score: 4.23)
5. Simple RAG (Composite Score: 3.78)

This ranking confirms that the more advanced, agentic, and context-aware RAG strategies provide a significant performance advantage over a simple, direct retrieval pipeline.

$$\begin{aligned}\text{Score} = \frac{1}{10}\big(&2 \times \text{accuracy} + 2 \times \text{faithfulness} \\ &+ 1.5 \times \text{ROUGE-1} + 1.5 \times \text{BLEU} \\ &+ 1.5 \times \text{param. cov.} + 1.5 \times \text{return cov.}\big)\end{aligned} \tag{2}$$

6. Discussion

Our comparative analysis of the RAG architectures yields significant insights into their respective strengths and weaknesses for automated docstring generation. The results highlight the trade-offs between several competing goals: generating high-quality prose, creating comprehensive documentation, and ensuring the output is factually grounded in the retrieved context. This section provides a detailed interpretation of these findings, discusses their implications for developers and researchers, and offers data-driven recommendations for selecting the optimal RAG strategy.

6.1. Interpretation of Architectural Performance

Our results show that the choice of RAG architecture is more than a minor detail; it is a critical factor that determines the quality of the final docstring. Two strategies in particular, Self RAG and Code Aware RAG, demonstrated unique and complementary strengths.

- **Self RAG and the Pursuit of Faithfulness:** The superior performance of Self RAG, which achieved the highest overall composite score (8.45), is

the most significant finding. Its iterative self reflection mechanism appears to be uniquely effective at enhancing both the semantic quality and, most critically, the faithfulness of the generated docstrings. With a Faithfulness Score of 51.34% nearly double that of any other approach Self RAG demonstrates that a "generate-then-critique" loop is a powerful mechanism for ensuring the model's output is grounded in the retrieved context. This suggests that for tasks where preventing factual errors is paramount, simply providing context is insufficient; a reflective or corrective step is necessary to enforce compliance.

Finding 1: Self RAG emerges as the most effective approach for docstring generation, achieving the highest overall performance score of 8.45.

- **Code Aware RAG and the Power of a Precise Query:** In contrast, Code Aware RAG excelled at the task of comprehensive documentation coverage, ranking a close second overall (composite score: 7.92). It achieved the highest scores in Parameter Coverage (85.88%), Return Coverage (75.12%), and Exception Coverage (64.43%). This suggests that investing computational effort before retrieval by parsing the code to create a highly specific and enriched query provides the LLM with a clean, targeted context. This focused context appears to be optimal for the more structured aspects of docstring generation, allowing the model to accurately document the code's functional interface without being distracted by broader, potentially noisy information.

Finding 2: Code Aware RAG demonstrates distinctive strengths in comprehensive documentation coverage, ranking second overall with a composite score of 7.92.

6.2. Trade-off Between Conciseness and Comprehensiveness

Our findings also highlight a clear trade-off between the conciseness and the comprehensiveness of the generated documentation.

- **Corrective RAG:** produced the most ideally concise docstrings (100.71% of baseline length) while maintaining competitive performance, demonstrating that effective documentation does not require verbosity. This makes it an appealing choice for development environments that favor minimalist documentation styles.

- **Fusion RAG:** outputs (119.91% of baseline) did not translate to improved quality or coverage, as it ranked fourth overall. This result challenges a common assumption. Simply providing more information from diverse sources does not guarantee a better outcome. The additional 20% in docstring length appears to add redundancy rather than value, suggesting that for a structured task like docstring generation, the quality and relevance of the context are more important than the sheer quantity.

Finding 3: Advanced RAG strategies present a clear trade-off between comprehensiveness and conciseness. Corrective RAG reveals an interesting balance between documentation quality and brevity, producing the most concise docstrings at 100.71% of baseline length while maintaining competitive performance metrics.

6.3. Practical Implications and Recommendations

The performance-to-computation ratio varies significantly across the different architectures. Based on their design, we can infer their relative costs. Simple RAG is the most computationally efficient, but its low performance makes it a suboptimal choice. At the other end of the spectrum, Fusion RAG is likely the most expensive due to its parallel retrieval paths, and its poor performance-to-cost ratio makes it an unattractive option.

The most compelling trade-offs are between the top two performers:

- For production systems where trustworthiness and documentation quality are paramount, the superior performance of Self RAG justifies its higher computational overhead. The significant improvements in accuracy and, most importantly, faithfulness make it the recommended choice for high-stakes applications.

- For resource-constrained environments or tools where efficiency and structural completeness are the primary goals, Code Aware RAG emerges as the most practical and efficient alternative. It offers nearly the same level of documentation coverage as Self RAG but with a significantly lower computational footprint.

Ultimately, selecting a RAG approach for efficient code documentation requires balancing computational cost with quality requirements. Our findings provide a clear framework to help make that decision.

7. Conclusion

This paper presented a comprehensive benchmarking framework for Retrieval-Augmented Generation (RAG) systems, specifically evaluating their performance

on the high-fidelity task of automated Python docstring generation. By implementing and testing five distinct RAG architectures - Simple, Corrective, Self, Code Aware, and Fusion - across three state-of-the-art language model pairs, we achieved a nuanced assessment of their respective strengths and weaknesses.

Our findings consistently demonstrate that all evaluated RAG strategies significantly outperform a non-augmented baseline, confirming the fundamental value of grounding LLMs in external context for this task. The results reveal a critical architectural trade-off. Self RAG emerged as the clear leader in generating trustworthy documentation, achieving the highest overall performance and a Faithfulness Score of 51.34%, nearly double that of any other approach. This underscores the importance of self-reflection mechanisms in mitigating model hallucination. Conversely, Code Aware RAG proved to be the most effective strategy for achieving structural completeness and efficiency, attaining the highest scores in Parameter Coverage (85.88%) and producing the most ideally concise docstrings.

Ultimately, our study reveals that the choice of RAG architecture has a more pronounced impact on the quality of the generated docstring than the specific underlying language model. This work contributes a reproducible framework for benchmarking RAG systems for code documentation and provides clear, empirical guidelines for practitioners to select the optimal pipeline based on whether the primary goal is maximizing factual grounding or achieving structural completeness and efficiency.

8. Future Direction

Building upon the insights gained from this study, several promising avenues for future research emerge:

- **Exploration of Graph-based Retrieval:** Future work should investigate the use of Graph RAG, where the knowledge base is a structured knowledge graph of code entities and their relationships. This could provide a more powerful and contextually aware retrieval mechanism than the standard vector search used in this study.

- **Advanced Evaluation with LLM-as-a-Judge:** To further enhance the reliability of our benchmarks, future evaluations could incorporate more sophisticated LLM-as-a-Judge frameworks. Using a state-of-the-art LLM to assess the quality of generated docstrings across multiple dimensions could provide a more nuanced and human-aligned measure of performance.

References

AI, D. et al. (2024). DeepSeek-V3 Technical Report. Working paper.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Asai, A., Wu, Z., Wang, Y., Sil, A., and Hajishirzi, H. (2024). Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

Bai, J. et al. (2023). Qwen Technical Report.

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P. S., Yang, Q., and Xie, X. (2024). A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3).

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. (2023). Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE Computer Society.

Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T.-S., and Li, Q. (2024). A survey on RAG meeting LLMs: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6491–6501.

Goodger, D. and van Rossum, G. (2001). PEP 257 – docstring conventions. Technical Report 257, Python Software Foundation.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H., editors, *Advances in Neural Information Processing Systems 34*, pages 9459–9474. Curran Associates, Inc.

Lin, C.-Y. (2004). ROUGE: A package for automatic evaluation of summaries. In *Proceedings of the ACL-04 Workshop on Text Summarization (WAS 2004)*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Liu, Y., Iter, D., Xu, Y., Wang, S., Xu, R., and Zhu, C. (2023). G-Eval: NLG evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2511–2522, Singapore. Association for Computational Linguistics.

Meta, A. . (2024). The Llama 3 herd of models.

Pan, J. J., Wang, J., and Li, G. (2024). Survey of vector database management systems. *The VLDB Journal*, 33(5):1591–1615.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Parvez, M. R., Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Poudel, B., Cook, A., Traore, S., and Ameli, S. (2024). DocuMint: Docstring generation for Python using small language models.

Rackauckas, Z. (2024). RAG-Fusion: A new take on retrieval-augmented generation.

Rani, S. J., Deepika, S. G., Devdharshini, D., and Ravindran, H. (2024). Augmenting code sequencing with retrieval-augmented generation (RAG) for context-aware code synthesis. In *2024 First International Conference on Software, Systems and Information Technology (SSITCON)*, pages 1–7.

Reimers, N. and Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using siamese BERT-networks.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. (2020). CodeBLEU: A method for automatic evaluation of code synthesis.

Singh, P. K. and Sharma, S. (2022). A literature review on automatic comment generation using deep learning techniques. In *2022 3rd International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 1–7.

Sun, W., Miao, Y., Li, Y., Zhang, H., Fang, C., Liu, Y., Deng, G., Liu, Y., and Chen, Z. (2025). Source Code Summarization in the Era of Large Language Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 419–431. IEEE Computer Society.

Tian, Y., Liu, A., Dai, Y., Nagato, K., and Nakao, M. (2024). Systematic synthesis of design prompts for large language models in conceptual design. *CIRP Annals*, 73(1):85–88.

Tong, W. and Zhang, T. (2024). CodeJudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051, Miami, Florida, USA. Association for Computational Linguistics.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Von Luxburg, U., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Yan, S.-Q., Gu, J.-C., Zhu, Y., and Ling, Z.-H. (2024). Corrective Retrieval Augmented Generation. OpenReview.

Yang, Z., Chen, S., Gao, C., Li, Z., Hu, X., Liu, K., and Xia, X. (2025). An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology*. Just Accepted.

Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., and Artzi, Y. (2020). BERTScore: Evaluating text generation with BERT. In *International Conference on Learning Representations*.