Code-aware Re-ranking for Retrieval in Low Documentation Settings

Dhruv Gupta
MIIS, LTI
dhruvgu2@andrew.cmu.edu

Gayathri Ganesh Lakshmy
MIIS, LTI
gganeshl@andrew.cmu.edu

Daniel Chechelnitsky
PhD, LTI
dchechel@andrew.cmu.edu

1 Introduction

Code retrieval systems have become increasingly important as software development grows in complexity, but their effectiveness heavily depends on the quality and availability of documentation. Recent work has shown that retrieval-augmented generation (RAG) can significantly improve code generation by incorporating external knowledge at inference time. However, real-world codebases often lack comprehensive documentation, presenting challenges for effective retrieval. To systematically study these scenarios, we investigate code retrieval in low-documentation settings using the CodeRAG-Bench corpus. We emulate varying levels of documentation sparsity through progressive normalization techniques: removing docstrings, standardizing function names, and normalizing variable names, allowing us to analyze how retrieval performance degrades with reduced documentation.

To address the challenges of retrieving code in these constrained conditions, we propose two complementary approaches. First, we introduce a code-aware reranking mechanism that leverages large language models to evaluate the semantic relevance of retrieved code snippets, going beyond traditional lexical similarity measures. Our reranking approach shows substantial improvements, increasing recall@10 from 73.8% to 81.1% for function name normalization. Second, we develop a pseudocode generation technique that bridges the semantic gap between natural language queries and code representations by generating intermediate pseudocode that captures the essential algorithmic structure. This approach demonstrates even more dramatic improvements in severe low-documentation settings, improving recall@50 from 60.4% to 79.9% when both function and variable names are normalized. Through these approaches, we show that effective code retrieval is possible even in scenarios with limited documentation, though significant challenges remain when multiple types of documentation are removed simultaneously.

2 Datasets

In recent years, researchers have developed various datasets to evaluate code generation capabilities. Our experiments focus on two widely-used benchmarks for general programming tasks: HumanEval and MBPP. These datasets form the foundation of the general programming problems in CodeRAG-Bench, providing a diverse set of algorithmic challenges with varying levels of documentation.

2.1 Benchmark Datasets

HumanEval, introduced by Chen et al.[4], consists of 164 hand-written programming problems designed to test the ability of a model to generate functionally correct code. Each problem contains a function signature, docstring, and several unit tests. This dataset focuses on algorithmic problem-solving and basic programming concepts, making it particularly suitable for our study of code retrieval in low-documentation settings. The varying format and detail level of its docstrings closely mirrors real-world software documentation practices. MBPP (Mostly Basic Python Programming), developed by Austin et al.[2], contains 974 crowd-sourced Python programming problems. Each problem includes a natural language description and several test cases to verify code correctness. A key feature of MBPP is its consistency in providing exactly three input/output examples as assert statements for each problem, contrasting with HumanEval's variable documentation format. This standardized structure makes MBPP valuable for analyzing how documentation consistency affects retrieval performance. Together, these datasets provide a comprehensive testbed for evaluating code retrieval systems under different documentation conditions. Their complementary characteristics—HumanEval's real-world documentation variability and MBPP's structured consistency—enable us to thoroughly assess our proposed retrieval enhancement methods across different documentation patterns and levels of detail.

3 Related Works

As language models advance, their ability to generate functional code from natural language descriptions has improved dramatically. In this section, we examine key approaches in code generation, retrieval-augmented generation, and code embeddings.

3.1 State-of-the-art Code Generation Methods

Recent approaches to code generation have demonstrated significant improvements through various methodologies. Dong et al. [6] introduced a collaborative framework where multiple LLM agents operate in distinct roles (analyst, coder, tester), achieving impressive results with 74.4% Pass@1 on HumanEval and 68.2% on MBPP. Similarly, CodeT [3] improved functional correctness through self-debugging mechanisms, reaching 65.8% and 67.7% Pass@1 scores on HumanEval and MBPP respectively. These approaches demonstrate the power of incorporating multiple evaluation steps in code generation, inspiring our re-ranking methodology. AlphaCode [15], WizardCoder [16], and TaskWeaver [18] represent another direction in code generation advancement. AlphaCode achieved competitive results using large-scale sampling and filtering, while WizardCoder demonstrated substantial improvements through evolved instruction tuning, achieving a 22.3 point increase on HumanEval. TaskWeaver introduced a novel code-first framework specifically for data analytics tasks. These approaches' success in filtering and evaluating code solutions particularly influenced our re-ranking strategy.

3.2 Retrieval Augmented Generation

The Retrieval Augmentation Generation (RAG) framework, introduced by Lewis et al. [14], combines retrieval and generation steps to enhance text generation tasks. RAG has been successfully applied across various domains, from multiple-choice question answering to multilingual and multimodal applications [19, 22, 5, 24]. These diverse applications demonstrate RAG's versatility and potential for code-specific tasks. The Code-RAG benchmark [21] represents a significant advancement in code-specific RAG evaluation, encompassing various programming domains and problem types. Building upon earlier benchmarks like HumanEval and SWE-Bench [4, 13], Code-RAG addresses the unique challenges of code retrieval, including output consistency and performance considerations [11]. Recent work in bug detection [10] and data analysis [20] demonstrates the framework's expanding applications.

3.3 Code Embeddings

Code embedding techniques have evolved along several paths, with contrastive pre-training emerging as a particularly effective approach. Studies by [12] and [17] demonstrated that encoder-only models trained contrastively can learn code functionality rather than just form, with "cpt-code" achieving 93.5% accuracy on CodeSearchNet across multiple languages. These findings significantly influenced our approach to code retrieval in low-documentation settings. Alternative approaches include tree-based methods like code2vec [1] and ASTNN [25], which use abstract syntax trees to capture code structure. More recent work combines these approaches with contrastive learning [23] or multimodal representations [8]. Token-based methods like CodeBERT [7] and graph-based approaches like GraphCodeBERT [9] offer additional perspectives, though with varying degrees of success. These diverse approaches to code representation inform our understanding of how to effectively retrieve and rank code snippets.

4 Methodology

In this section, we first describe the techniques we used to normalize the original CodeRAG-Bench corpus to simulate a low-documentation setting for our retrieval tasks. We then describe the two methods, Query Pseudocode Generation and Code-aware Re-ranking, which were used to improve gold-standard code retrieval in our low-documentation corpus.

4.1 Normalizing the corpus

We perform four levels of corpus normalization on the original programming-solutions corpus provided by the original CodeRAG-Bench paper. Namely, function docstring removal, function name normalization, variable normalization, and both function and variable name normalization.

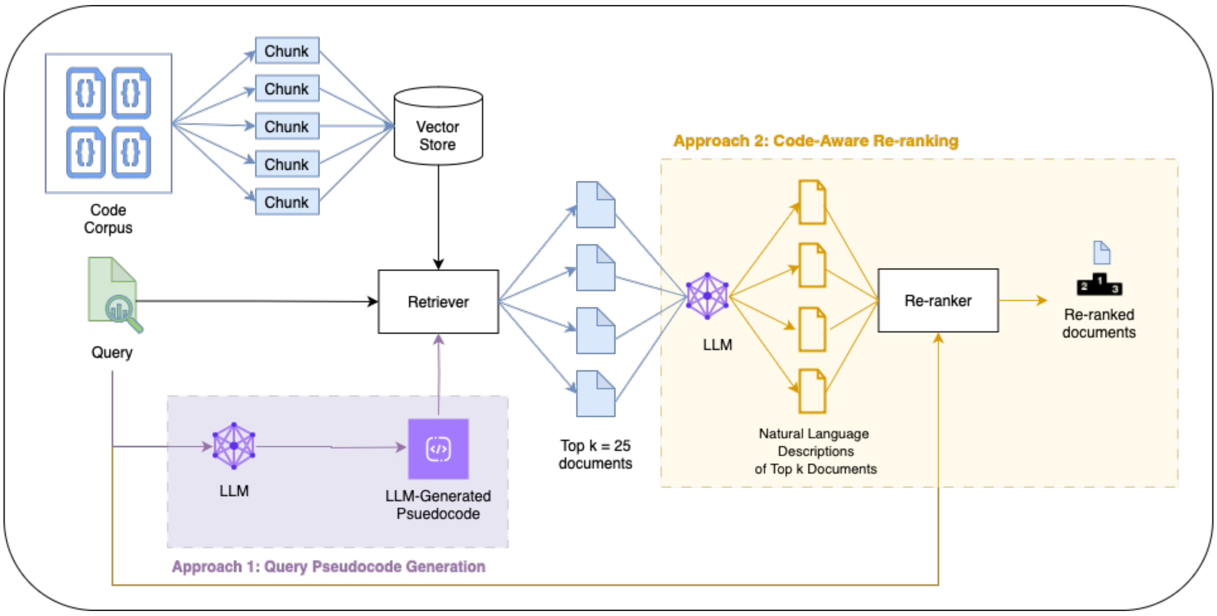

Figure 1: Architecture of the two new approaches: Query Pseudocode Generation and Code-aware Re-ranking

Our motivation for docstring removal from function headers was to remove the exact match of the HumanEval and MBPP task query text from our corpus. We do so using simple regex to remove comments and docstring from the corpus documents which correspond to queries in the original respective datasets. Next, we normalize function and variable names to see how invariant basic text embedding models are towards these features of code. Function name replacement was also done using basic regex to identify the string following a python function declaration. On the other hand, we used AST parsing methods to identify and rename variables in the code. Examples of the corpus normalization can be found in the Appendix as Table 3.

We experiment with a number of combinations of these levels of normalizations:

- No corpus normalization (Original)
- Function Docstring Removal
- Docstring and function name normalization
- Docstring and variable name normalization
- Docstring, function and variable name normalization

4.2 Code-aware Reranking

The reasoning behind adding code-aware reranking is to introduce a second layer of refinement to the code retrieval process, aiming to address the limitations of traditional similarity-based approaches. While vector-based retrieval systems are effective at capturing general semantic similarities between queries and the stored corpus, they often lack the ability to deeply understand the correctness or completeness of the retrieved solutions. This gap in understanding can lead to retrieving seemingly relevant but ultimately flawed or incomplete code, which significantly impacts the utility of the system in programming contexts. Code-aware reranking tackles this challenge by integrating semantic evaluation through a large language model (LLM) to reassess and refine the rankings of the initially retrieved results.

The process begins with a baseline retrieval of code snippets using vector similarity, where each snippet is scored based on its closeness to the query in the embedding space. From these results, a subset of the top-k candidates undergoes reranking based on a task-specific evaluation. This evaluation is guided by a prompt that directs the LLM to rate the relevance of a given code snippet compared to the query based on specific dimensions such as implementation completeness, algorithm correctness, edge case handling, and efficiency. By assigning a relevance score to each candidate, the reranking process ensures that the final results prioritize code snippets that are not only similar in terms of semantics but also meet the more nuanced requirements of the programming task. Examples of descriptions of documents generated by our re-ranking model can be found in Table 4 in the appendix.

The impact of the initial similarity score and the LLM-assigned relevance score on the final score are then balanced through a weighted combination of the two, parameterized by a tunable scalar α. This hybrid scoring approach provides a systematic approach to integrating the strengths of both vector-based retrieval and LLM-driven semantic evaluation, ensuring that the final ranking reflects both broad semantic alignment and task-specific quality.

4.3 Query Psuedocode Generation

Our motivation for leveraging pseudocode in code retrieval stems from the limitations of traditional retrieval methods, which often rely on textual similarity between natural language queries and raw code representations. These conventional approaches struggle to capture the logical structure and semantics of programming tasks, leading to suboptimal performance when surface-level differences such as syntax or naming conventions exist. Pseudocode provides a high-level abstraction that encapsulates the problem-solving logic inherent in programming tasks. By focusing on structural and semantic equivalence, pseudocode bridges the gap between the intent expressed in natural language descriptions and the implementation details in code, facilitating a more meaningful alignment between queries and stored solutions.

The approach begins with generating pseudocode for queries using a large language model trained to translate task descriptions into structured logical steps. This pseudocode adheres to standard conventions, explicitly detailing data structures, operations, and control flows while abstracting away superficial differences. Once generated, the pseudocode serves as input to a retrieval system based on vector similarity search. To enhance retrieval robustness, we preprocess the stored code corpus through normalization techniques that strip away docstrings and replace variable and function names, ensuring that the retrieval process focuses on the essence of the code rather than its stylistic elements.

By emphasizing semantic similarity over syntactic alignment, this methodology facilitates better retrieval of solutions that are logically equivalent to the query, even when the implementations diverge in style or detail. Distilling programming queries into pseudocode steps not only enhances alignment but also demonstrates the potential to significantly improve the robustness and effectiveness of code retrieval systems. Examples of such code and pseudocode pairs produced by our technique can be found in Appendix's Table 5 .

5 Results

In this section, we present the results of the baseline retriever in our normalized corpus to demonstrate its deteriorating performance in low documentation settings. We then present the boosts in performance observed with our Code-aware Re-ranking and Query Psuedocode Generation approaches. Currently, all of our plots and analyses are focused on retrieval for the HumanEval tasks on normalized versions of the programming solutions corpus provided by the original CodeRAG-Bench paper.

5.1 Evaluation Metrics

We evaluate our retrieval performance using Recall@k, which measures the percentage of queries for which the correct code solution appears in the top k retrieved documents. For each query in our evaluation set, we consider a retrieval successful if any of the top k retrieved documents contains the ground truth solution, matching the evaluation methodology used in CodeRAG-Bench. This metric is particularly relevant for retrieval-augmented generation systems, where success depends on retrieving at least one correct implementation within the top k results that will be used to augment the generation process.

5.2 Baseline Corpus Normalization Results

Our analysis reveals a clear pattern of performance degradation in the GIST large model's retrieval capabilities as we progressively remove documentation elements from the programming solutions corpus. As shown in Figure 2, while the model maintains near-perfect recall scores across all metrics with no normalization and minimal degradation with docstring removal, performance drops significantly as we apply more aggressive normalization techniques. The most striking observation is the substantial impact of function name normalization on retrieval performance. Recall@1 drops from nearly 100% to 36% when function names are standardized, suggesting that the model heavily relies on semantic information encoded in function names for retrieval. This dependence on function naming is particularly interesting as it indicates that current embedding and retrieval systems may be

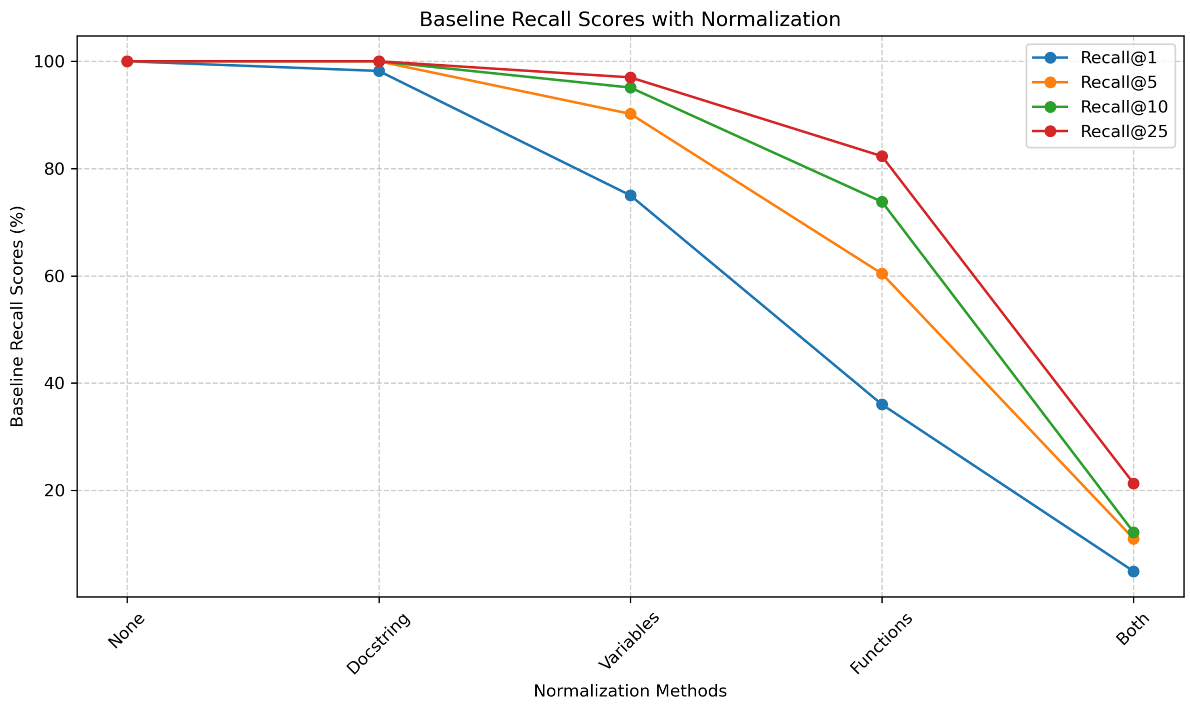

Figure 2: Deterioration of Baseline recall scores of the GIST large model on HumanEval Retreival from the programming solutions corpus with different levels of corpus normalization.

Normalization	**Recall@1**	**Recall@5**	**Recall@10**	**Recall@25**
None	100.0/100.0	100.0/100.0	100.0/100.0	100.0/-
Docstring	98.2/97.6	100.0/100.0	100.0/100.0	100.0/-
Functions	36.0/56.1	60.4/79.9	73.8/81.1	82.3/-
Variables	75.0/80.5	90.2/96.3	95.1/97.0	97.0/-
Both	4.9/12.8	11.0/20.1	12.2/21.3	21.3/-

Table 1: Baseline/Reranked Recall Scores (%) using Llama-3.1-70B-Instruct for description generation and GIST large for embeddings. For Recall@25, only baseline scores are shown since we re-rank only top-25 documents. Dark green: >10 points improvement, Light green: 0-10 points improvement, Light red: 0-10 points decrease, Dark red: >10 points decrease. For Recall@25, only baseline scores are shown.

paying too much attention to superficial naming patterns rather than understanding the underlying functionality. Variable name normalization shows a more moderate impact, with Recall@1 decreasing to approximately 75%. When both function and variable names are normalized, we observe the most severe degradation, with Recall@1 plummeting to below 5%, highlighting the compounding effect of removing multiple documentation signals. Overall, these results underscore the challenges of code retrieval in low-documentation settings and motivate our proposed enhancements to the retrieval pipeline.

5.3 Code-Aware Reranking Results

Our code-aware reranking approach demonstrates significant improvements across all normalization settings, with particularly strong gains in the more challenging scenarios. Since we only rerank the top-25 retrieved documents, the baseline Recall@25 scores represent the theoretical upper bound for our reranking performance. This constraint means that while we can improve the ranking within these 25 documents, we cannot recover relevant documents that fall outside this initial retrieval set. As observed in Table 1, the most notable improvements appear in function name normalization, where our reranking approach increases Recall@1 from 36.0% to 56.1% and Recall@5 from

Normalization	**Recall@10**	**Recall@25**	**Recall@50**	**Recall@100**
None	100.0/100.0	100.0/100.0	100.0/100.0	100.0/100.0
Docstring	99.4/99.4	100.0/100.0	100.0/100.0	100.0/100.0
Functions	76.2/81.7	87.2/91.5	93.3/95.7	98.2/98.8
Variables	95.1/94.5	97.6/97.6	99.4/97.6	100.0/100.0
Both	34.1/49.4	47.0/67.1	60.4/79.9	77.4/90.9

Table 2: Baseline/Pseudocode Recall Scores (%) using Llama-3.1-70B-Instruct for pseudocode generation and all-mpnet-base-v2 for embeddings. Dark green: >10 points improvement, Light green: 0-10 points improvement, Light red: 0-10 points decrease, Dark red: >10 points decrease. For Recall@25, only baseline scores are shown.

60.4% to 79.9%. This substantial improvement suggests that our code-aware reranking effectively captures functional similarities that the baseline retriever misses when function names are normalized. Similarly, we observe consistent gains across other normalization levels, with variable name normalization showing improvements from 75.0% to 80.5% at Recall@1, and the most challenging scenario (both function and variable normalization) seeing improvement from 4.9% to 12.8% at Recall@1. The consistent pattern of improvement across all settings indicates that our reranking approach successfully leverages semantic understanding of code functionality rather than relying solely on superficial textual similarities.

5.4 Query Pseudocode Results

Referring to Table 2, our pseudocode generation approach shows particularly strong performance in highly normalized settings, though with varying effectiveness across different normalization levels. Most notably, in the most challenging scenario where both function and variable names are normalized, this approach achieves substantial improvements, with Recall@10 increasing from 34.1% to 49.4% and Recall@50 improving from 60.4% to 79.9%. This significant gain suggests that pseudocode generation effectively bridges the semantic gap when traditional documentation signals are minimal.

However, the results reveal an interesting pattern in less aggressive normalization settings. For function name normalization, we observe modest improvements (76.2% to 81.7% at Recall@10), while variable name normalization actually shows a slight performance degradation (95.1% to 94.5% at Recall@10). This pattern suggests that pseudocode generation may be most beneficial in scenarios with severe documentation limitations, but could potentially introduce noise when sufficient documentation signals are still present in the code. The approach performs consistently well at higher retrieval depths (Recall@50 and Recall@100), indicating that it helps surface relevant documents that might be missed by traditional retrieval methods, albeit sometimes at the cost of precision in the top results.

6 Analysis and Limitations

In this section, we look at the effect of varying code-aware model sizes and embedding models sizes in our re-ranking technique. Moreover, we discuss the limitations of our current methodology.

6.1 Code-aware Model Size Effects on Re-ranking Performance

The results presented in Figure 3 highlight the nuanced relationship between model size and re-ranking performance in code retrieval tasks across varying degrees of normalization. A key observation is that increasing the number of model parameters does not guarantee superior Recall@k scores. For instance, while Llama-3.1-70B, the largest model in the study, performs slightly better than its smaller counterpart (Llama-3.1-8B) in some cases, the improvements are not substantial or consistent. Similarly, Mistral-8x7B, despite its larger parameter count, occasionally performs worse than both smaller Llama models. These findings suggest that model size alone is insufficient to drive significant gains in re-ranking performance. However, the ability of a model to deeply understand and reason about code, such as its semantics and structure, may play a more decisive role in determining re-ranking effectiveness.

6.2 Embedding Model Size Effects on Re-ranking Performance

The analysis of the results shown in Figure 4 highlights the relative improvements in Recall@k

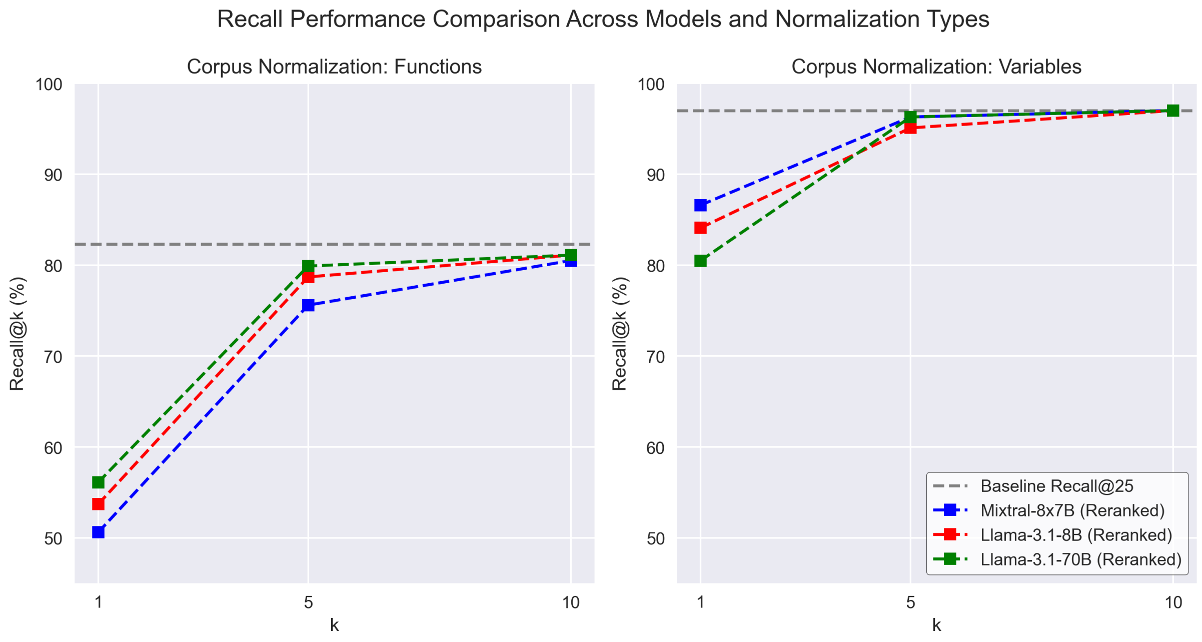

Figure 3: Comparing progression of the Recall@k scores across various code-aware models with different kinds of normalization, using Recall@25 as the theoretical upper bound performance

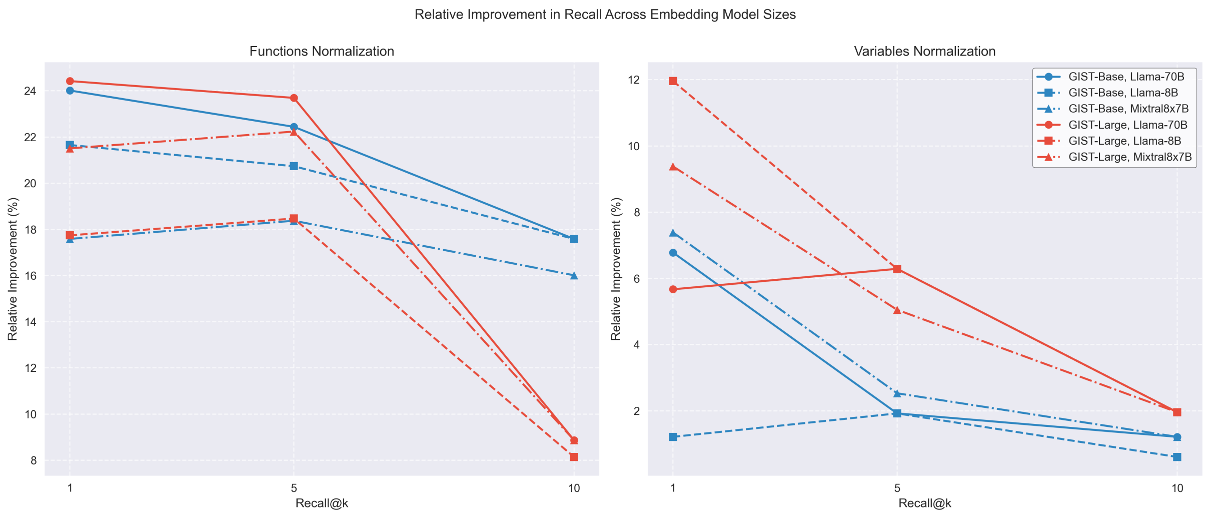

Figure 4: Relative improvement in Recall@k scores across different embedding model sizes (GIST-base and GIST-Large) under varying normalization techniques

scores achieved by embedding models of different sizes under varying normalization techniques. Specifically, the figure compares GIST-base and GIST-Large embeddings across normalization strategies focused on functions and variables.

Across the experiments, GIST-Large consistently demonstrates higher relative improvements in Recall@k scores compared to GIST-base, suggesting that larger embedding models have an advantage in capturing the nuanced semantics necessary for effective re-ranking. For lower values of Recall@k, the performance gap between GIST-base and GIST-Large is often less pronounced, indicating that smaller models can still perform well in simpler retrieval scenarios. However, as Recall@k increases, the benefits of larger embeddings become more apparent, particularly for complex queries that require a deeper understanding of the embedding space.

6.3 Limitations of our current approach

Our code-aware reranking methodology, while effective in improving retrieval performance in low documentation settings, faces several important limitations. A significant drawback is the computational overhead introduced by generating descriptions for each candidate code snippet during the reranking process. This additional inference step with large language models increases both the time and computational resources required for retrieval. One potential solution to this limitation would be to train a smaller, specialized model specifically for generating code descriptions. Such a model could be fine-tuned on code-description pairs, potentially offering similar quality at reduced computational cost.

The pseudocode generation technique also presents limitations, particularly in scenarios with higher levels of documentation. Our results show that while this approach excels in severely limited documentation settings, it can actually introduce noise when sufficient documentation signals are already present. This performance discrepancy likely stems from the limitations of current text embedding models, which may not have a sophisticated understanding of the relationship between code and pseudocode representations. To address this limitation, future work could explore using specialized code embedding models that are specifically trained to understand the semantic relationships between code, pseudocode, and natural language descriptions. Such models might better capture the structural and logical similarities between these different representations, potentially improving retrieval performance across all documentation levels.

7 Conclusion

In this work, we investigated the challenge of code retrieval in low-documentation settings and proposed two complementary approaches to address this problem. Our code-aware reranking mechanism demonstrated significant improvements in retrieval performance, particularly when dealing with normalized function names, increasing recall@10 from 73.8% to 81.1%. The pseudocode generation technique proved especially effective in the most challenging scenarios, improving recall@50 from 60.4% to 79.9% when both function and variable names are normalized.

Our analysis revealed several important insights about code retrieval systems. First, current embedding models heavily rely on superficial features like function names, with performance degrading significantly when these signals are removed. Second, while larger language models can improve code-aware re-ranking performance, the relationship between model size and effectiveness is not straightforward, suggesting that specialized training might be more important than raw model size. Third, our pseudocode generation approach showed that bridging the semantic gap between natural language and code is possible, though careful consideration must be given to when this technique is applied.

Future work could explore several promising directions. First, developing more efficient methods for code-aware re-ranking, possibly through distilled models specifically trained for code description generation. Second, investigating specialized code embedding models that better capture the relationship between code, pseudocode, and natural language. Finally, extending these approaches to other programming languages and more complex codebases would help validate their generalizability. As software systems continue to grow in complexity, improving code retrieval in low-documentation settings remains a crucial challenge for the field.

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[5] Nadezhda Chirkova, David Rau, Hervé Déjean, Thibault Formal, Stéphane Clinchant, and Vassilina Nikoulina. Retrieval-augmented generation in multilingual settings. *arXiv preprint arXiv:2407.01463*, 2024.

[6] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[8] Jian Gu, Zimin Chen, and Martin Monperrus. Multimodal representation for neural code search. *CoRR*, abs/2107.00992, 2021.

[9] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366, 2020.

[10] Nima Shiri Harzevili, Mohammad Mahdi Mohajer, Jiho Shin, Moshi Wei, Gias Uddin, Jinqiu Yang, Junjie Wang, Song Wang, Zhen Ming, Nachiappan Nagappan, et al. Checker bug detection and repair in deep learning libraries. *arXiv preprint arXiv:2410.06440*, 2024.

[11] Pengfei He, Shaowei Wang, Shaiful Chowdhury, and Tse-Hsun Chen. Exploring demonstration retrievers in rag for coding tasks: Yeas and nays! *arXiv preprint arXiv:2410.09662*, 2024.

[12] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. Contrastive code representation learning. *CoRR*, abs/2007.04973, 2020.

[13] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[14] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[15] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[16] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

[17] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. Text and code embeddings by contrastive pre-training. *CoRR*, abs/2201.10005, 2022.

[18] Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*, 2023.

[19] Shangeetha Sivasothy, Scott Barnett, Stefanus Kurniawan, Zafaryab Rasool, and Rajesh Vasa. Ragprobe: An automated approach for evaluating rag applications. *arXiv preprint arXiv:2409.19019*, 2024.

[20] Zifeng Wang, Benjamin Danek, Ziwei Yang, Zheng Chen, and Jimeng Sun. Can large language models replace data scientists in clinical research? *arXiv preprint arXiv:2410.21591*, 2024.

[21] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. Coderag-bench: Can retrieval augment code generation?, 2024.

[22] Peng Xia, Kangyu Zhu, Haoran Li, Hongtu Zhu, Yun Li, Gang Li, Linjun Zhang, and Huaxiu Yao. Rule: Reliable multimodal rag for factuality in medical vision language models. *arXiv preprint arXiv:2407.05131*, 2024.

[23] Zixiang Xian, Rubing Huang, Dave Towey, Chunrong Fang, and Zhenyu Chen. Transformcode: A contrastive learning framework for code embedding via subtree transformation. *IEEE Transactions on Software Engineering*, 50(6):1600–1619, June 2024.

[24] Jianhao Yuan, Shuyang Sun, Daniel Omeiza, Bo Zhao, Paul Newman, Lars Kunze, and Matthew Gadd. Rag-driver: Generalisable driving explanations with retrieval-augmented in-context learning in multi-modal large language model. *arXiv preprint arXiv:2402.10828*, 2024.

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

A Appendix

Normalization	Code Example

Original	`from typing import List`
	`def has_close_elements(numbers: List[float], threshold: float) -> bool:`
	`   """ Check if in given list of numbers, are any two numbers closer`
	`   to each other than given threshold."""`
	`  »> has_close_elements([1.0, 2.0, 3.0], 0.5)`
	`  False`
	`  »> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)`
	`  True`
	`  """`
	`  for idx, elem in enumerate(numbers):`
	`    for idx2, elem2 in enumerate(numbers):`
	`      if idx != idx2:`
	`        distance = abs(elem - elem2)`
	`        if distance < threshold:`
	`          return True`
	`  return False`

Docstring	`from typing import List`
	`def has_close_elements(numbers: List[float], threshold: float) -> bool:`
	`  for idx, elem in enumerate(numbers):`
	`    for idx2, elem2 in enumerate(numbers):`
	`      if idx != idx2:`
	`        distance = abs(elem - elem2)`
	`        if distance < threshold:`
	`          return True`
	`  return False`

Variables (AST)	`from typing import List`
	`def has_close_elements(var_0: List[float], var_1: float) -> bool:`
	`  for var_2, var_3 in enumerate(var_0):`
	`    for var_4, var_5 in enumerate(var_0):`
	`      if var_2 != var_4:`
	`        var_6 = abs(var_3 - var_5)`
	`        if var_6 < var_1:`
	`          return True`
	`  return False`

Functions	`from typing import List`
	`def func_0(numbers: List[float], threshold: float) -> bool:`
	`  for idx, elem in enumerate(numbers):`
	`    for idx2, elem2 in enumerate(numbers):`
	`      if idx != idx2:`
	`        distance = abs(elem - elem2)`
	`        if distance < threshold:`
	`          return True`
	`  return False`

Both (AST)	`from typing import List`
	`def func_0(var_0: List[float], var_1: float) -> bool:`
	`  for var_2, var_3 in enumerate(var_0):`
	`    for var_4, var_5 in enumerate(var_0):`
	`      if var_2 != var_4:`
	`        var_6 = abs(var_3 - var_5)`
	`        if var_6 < var_1:`
	`          return True`
	`  return False`

Table 3: Comparison of Different Code Normalization Techniques using AST-based Variable Renaming

Code	Description

`for idx, elem in enumerate(numbers):`
`    for idx2, elem2 in enumerate(numbers):`
`        if idx != idx2:`
`            distance = abs(elem - elem2)`
`            if distance < threshold:`
`                return True`
`return False`
	This function checks if any two numbers in a given list are closer to each other than a specified threshold. It uses a nested loop to calculate the absolute difference between each pair of numbers and returns True as soon as it finds a pair with a difference less than the threshold, otherwise it returns False.

`from math import floor, ceil`
`if value.count('.') == 1:`
`    # remove trailing zeros`
`    while (value[-1] == '0'):`
`        value = value[:-1]`
`    num = float(value)`
`    if value[-2:] == '.5':`
`        if num > 0:`
`            res = ceil(num)`
`        else:`
`            res = floor(num)`
`    elif len(value) > 0:`
`        res = int(round(num))`
`    else:`
`        res = 0`
`return res`
	This function takes a string representing a number and returns the closest integer to it. If the number is equidistant from two integers, it rounds away from zero, returning the integer farthest from zero. It handles decimal numbers, removing trailing zeros and using the ceil or floor function for numbers ending in .5, and the round function for other decimal numbers.

Table 4: Examples of Python Code and Their Descriptions

Query (Docstring)	Generated Pseudocode

Check if in given list of numbers, are any two numbers closer to each other than given threshold.	SORT numbers in ascending order
	FOR i FROM 0 TO LENGTH(numbers) - 2
	CALCULATE difference = numbers[i + 1] - numbers[i]
	IF difference < threshold THEN
	RETURN True
	END FOR
	RETURN False

You're given a list of deposit and withdrawal operations on a bank account that starts with zero balance. Your task is to detect if at any point the balance of account falls below zero, and at that point function should return True. Otherwise it should return False.	INITIALIZE balance TO 0
	FOR EACH operation IN operations
	UPDATE balance BY ADDING operation TO balance
	IF balance IS LESS THAN 0
	RETURN True
	RETURN False

Table 5: Examples of Query Docstrings and Their Generated Pseudocode