Retrieval-Augmented Code Generation: Literature Survey and Baseline Reproduction

Zian Pan
LTI
Carnegie Mellon University
zianp@andrew.cmu.edu

Dunhan Jiang
CBD
Carnegie Mellon University
dunhanj@andrew.cmu.edu

Zhilan Wang
LTI
Carnegie Mellon University
nicolewa@andrew.cmu.edu

1 Introduction

Retrieval-Augmented Generation (RAG) systems (Lewis et al., 2021) combine a generative model with an external knowledge source to produce better outputs. For code generation, RAG helps solve problems that require knowledge beyond what a model knows. For example, when writing code with unfamiliar libraries or APIs, developers often need to consult documentation or study previous code examples. While traditional code generation models and even large language models often struggle with specialized or rarely used programming components, RAG addresses this limitation by dynamically retrieving relevant code snippets, API documentation, or question-and-answer content from developer forums. By including retrieved information as an additional context during generation, RAG systems can produce code that is more accurate and more adaptable to newly released libraries and programming paradigms. In this project, we try to reproduce the REDCODER framework described by (Parvez et al., 2021) and validate it on the CodeXGlue-CodeSearchNet (Lu et al., 2021b). Our code is made public on Github: https://github.com/zianpan/fancy_retriever.git

2 Related Work

Unlike generating human spoken languages, code generation presents unique challenges because code snippets are often highly structured. While natural language allows for flexibility in expression, programming languages require strict adherence to syntax and semantics. Additionally, the evaluation method often requires checking functional correctness rather than simply checking textual similarity. Recent work has explored retrieval-augmented approach for various coding tasks, including:

- Code synthesis (generating executable code from natural language descriptions, requiring translation of human intent into programming constructs)
- Code completion (predicting and filling in partial code given surrounding context)
- Code translation (converting code from one programming language to another)

Some innovative RAG approaches have been developed specifically for code generation, each addressing different aspects of the challenge.

- RepoCoder (Zhang et al., 2023a), which expanded traditional retrieval methods by implementing repository-level retrieval. This approach iteratively fetches relevant context from across an entire codebase to assist in multifile code completions, reflecting how professional developers navigate complex projects. However, this approach requires extensive computational resources and struggles with very large repositories where the signal-to-noise ratio in retrieved content can diminish generation quality.
- ReACC (Lu et al., 2022), a framework that helps finish code by finding and using similar code from existing programs. Unlike earlier methods that limited context to the current file, ReACC searches a comprehensive database for related code examples when it encounters unfinished code. One limitation is that ReACC's effectiveness decreases considerably for new or specialized tasks where the retrieval database contains few similar examples.

These methods have contributed significantly to the field, but they also face some shared challenges. Most rely on matching words or phrases, which can overlook examples that mean the same thing but are written differently. Moreover, many current approaches don't keep up with changes in programming libraries and APIs. As a result, they often return outdated code examples.

There are also methods that aim to address the challenge of syntactic correctness in code generation.

- kNN-TRANX (Zhang et al., 2023c) addresses the challenge by adding syntax constraints for the retrieval of data stores. This could reduce the impact of retrieval noise. By filtering out examples that do not match the syntax rules, this approach helps prevent code from being compiled with errors. However, this syntactic filtering can be overly restrictive, potentially eliminating semantically relevant but structurally different solutions, especially in languages with multiple idioms to solve the same problem.
- CodeGRAG (Du et al., 2024) took a novel approach by extracting code structure graphs for cross-language code generation, making it easier to translate code between different programming languages while keeping the code structure intact. However, this approach introduces significant computational overhead and struggles with languages that have fundamentally different paradigms, such as translating between functional and object-oriented languages.

One major limitation in current methods is the lack of attention to how well the retrieved examples actually being used within the code generation process. Most systems simply add these examples to the prompt without checking how relevant they are or adjusting their approach based on how confident they are in the retrieval. This can result in misleading examples being used.

In addition, the methods used to evaluate RAG-based code generation systems are still quite limited. Many benchmarks focus on solving algorithmic puzzles, rather than on real-world programming tasks that involve complex tools, libraries, and frameworks.

Therefore, we choose to implement the RECODER framework (Parvez et al., 2021) combined with CodeXGlue-CodeSearchNet (Lu et al., 2021b) as it provides a comprehensive evaluation benchmark that encompasses various code generation tasks in three categories. Using this approach, we can precisely measure and further improve retrieval strategies, clearly guiding improvements in retrieval quality, context integration, and model generalization. In the next section, we will provide a detailed description of the framework.

3 Methodology

3.1 REDCODER Framework

One of the first systems to apply retrieval to code generation was REDCODER (Parvez et al., 2021). It works like how programmers actually code. The REDCODER framework uses a simple two-step approach:

Finding Similar Code (SCODE-R): This part of the system helps to find the most relevant code or descriptions from a large database, based on the input of a user, which could be a sentence in plain English or a code piece. What makes REDCODER's retriever special is that it is built specifically for working with code. It is based on a well-known search model called the Dense Passage Retriever (DPR) (Karpukhin et al., 2020). However, instead of starting from scratch, they use models that already understand the code. For example, they use GraphCodeBERT (Guo et al., 2021) to understand code; it is a smart model that knows both the text and the structure of code. In addition, for text, they use something similar, like CodeBERT (Feng et al., 2020). Because of this set-up, the retriever can find meaningfully similar code or descriptions, even if the exact words do not match. Most importantly, in their tests, they searched through a mix of code-only entries, text-only entries(e.g. code summaries), and entries that have both code and its summary. As a result, when you want to generate a piece of code, the retriever can bring up similar code snippets. Similarly, if you want to summarize the code, it can find similar functions that already have summaries.

Code Generator (SCODE-G): Once the system finds useful examples using the retriever (SCODE-R), it adds those examples to the original input and sends everything into a code generation model to create the final output. The model they use for this is called PLBART (Ahmad et al., 2021). It is a powerful model that was trained in a lot of code using a method in which parts of the input are 'noised' or hidden, and the model learns to fill in the blanks. In addition to PLBART's seq2seq architecture, their retrieval is providing additional context that is added to the input. They tried two ways to add this extra context:

- **Simple method (REDCODER base):** Just stick the retrieved examples (code or summaries) in front of the input, one after the other.
- **Smarter method (REDCODER-EXT):** If a

retrieved code snippet has a summary, they pair them together and give both to the model. In this way, the generator sees not only the code but also how a human described it, which can help it write a better summary or generate better code.

3.2 CodeXGLUE

To validate our reproduced results, we adopt the same benchmark dataset used by the REDCODER paper (Parvez et al., 2021), which is CodeXGlue(Lu et al., 2021b). CodeXGLUE is a comprehensive benchmark dataset that encompasses 10 distinct tasks across 14 curated datasets for program understanding and generation. These tasks include code completion, code repair, code translation, natural language code search, text-to-code generation, and so forth. In this paper, we focus on the text-to-code generation. Therefore, we evaluated both retriever and generator on the CodeXGlue-CodeSearchNet (Lu et al., 2021b), which is a subset for text-to-code generation in python, following the paper's style.

3.3 CodeRAG-BENCH

We have also identified a good benchmark: CodeRAG-Bench(Wang et al., 2025), which addresses the need for systematic evaluation of when and how retrieval can benefit code generation tasks. CodeRAG-Bench includes:

- Basic programming problems (HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021)): Tests involving simple coding tasks.
- Open-Domain Coding Problems (DS-1000(Lai et al., 2022), ODEX(Wang et al., 2023)): Tasks requiring specialized API usage.
- Repository-Level Tasks (RepoEval(Zhang et al., 2023b), SWE-Bench(Jimenez et al., 2024)): Tasks that require context from multiple code files.

Their retrieval approach combines multiple retrieval sources (e.g., website posts, library documentation, and GitHub repositories) to provide a comprehensive context for the generative model. They systematically evaluated different retrieval strategies and baseline models, highlighting scenarios where retrieval significantly improves code generation performance. The benchmark also includes canonical document annotations to evaluate retrieval quality. So far, I also tested our reimplemented pipeline on the MBPP subset.

3.4 Implementation Details

Our first and primary goal is to reproduce the SCODE-R retriever as discussed in the REDCODER (Parvez et al., 2021) paper. We carefully follow the methodology presented in the paper by employing a DPR architecture (Karpukhin et al., 2020) consisting of two distinct encoders: one for natural language summaries and another for source code. The process is smooth since the paper release checkpoints for python code retriever for the text-to-code task. Furthermore, we utilize pre-trained GraphCodeBERT (Guo et al., 2021) to encode the code-description pairs in our implementation. The model was downloaded from HuggingFace.

As for the second generator part, our primary objective was to reproduce the functionality of the original SCODE-G pipeline, which was originally based on a fine-tuned PLBART model for generating code from natural language prompts augmented with retrieved code examples. Due to the unavailability of the original PLBART checkpoint and the prohibitive computational expense of re-finetuning it on our dataset, we designed an alternative retrieval-augmented code generation system. In our pipeline, each text prompt is enriched by concatenating several top-retrieved code instances, obtained from our reproduced SCODE-R pipeline, using a special delimiter token. This formulation provides the subsequent generation model with rich contextual cues to guide the synthesis of the desired code output. We evaluated our system on two distinct datasets: CodeXGlue-CSNet (scn) and CodeRagBench-MBPP (crb) . For the generative component, we tested both an official version of the PLBART model that is not fine-tuned and another Llama3.1-8B-Instruct model in standalone and retrieval-augmented configurations. These models are tested with and without the retrieved code instance augmentations, where their generated code instances are further evaluated by first the exact match, second the BLEU, and third the CodeBLEU (Ren et al., 2020).

4 Results

The preliminary results are shown in 1. As indicated in Table 1, while the pure generative models (e.g., PLBART on csn and crb, Llama3.1-8B-Instruct on csn and crb) achieved modest BLEU and CodeBLEU (Ren et al., 2020) scores, the retrieval-augmented setups—particularly Llama3.1-8B-Instruct (Touvron

Type	Method	EM	BLEU	CodeBLEU
SCODE-R Retriever	SCODE-R on csn	0.00	19.50	20.12
	SCODE-R on crb	0.00	21.75	22.68
Generative	PLBART on csn	0.00	3.52	10.39
	PLBART on crb	0.00	3.28	11.23
	Llama3.1-8B-Instruct on csn	0.00	8.62	14.95
	Llama3.1-8B-Instruct on crb	0.00	9.37	16.04
Retrieval Augmented SCODE-G	PLBART on csn	0.02	5.83	12.37
	PLBART on crb	0.05	5.35	12.45
	Llama3.1-8B-Instruct on csn	8.46	21.95	25.69
	Llama3.1-8B-Instruct on crb	9.31	22.76	26.01

Table 1: Evaluation results on two benchmark datasets—CodeXGlue-CSNet (csn) and CodeRagBench-MBPP (crb)—for various code generation approaches. The table compares retrieval-based methods (SCODE-R on csn and crb), pure generative models (PLBART and Llama3.1-8B-Instruct on csn and crb), and their retrieval-augmented counterparts within our SCODE-G framework. Performance is measured using Exact Match (EM), BLEU, and CodeBLEU (Ren et al., 2020), highlighting the significant improvements achieved by incorporating retrieved code instances into the generation process.

et al., 2023) on both datasets—showed remarkable improvements, achieving exact match scores of 8.46 and 9.31, BLEU improvements to approximately 22, and CodeBLEU scores reaching up to 25.69 and 26.01. These results demonstrate that our pipeline effectively captures the retrieval-augmented code generation methodology originally proposed in SCODE-G, thereby generating high-quality, task-specific code even in the absence of the original PLBART model and without the intensive cost of re-finetuning it. In general, our implementation roughly achieves the performance compared with the baseline paper.

5 Next Step

In the upcoming phase of our project, we plan to refine our retrieval system with two innovative approaches. First, we will implement a self-refining Retriever that uses large-language model feedback to iteratively improve search results. This works like having an expert programmer analyze the initial code snippets and suggest more targeted searches to find missing components. Second, we will develop a multimodal ranking that looks at code from multiple perspectives, analyzing not just the text but also the code structure (AST), data flow patterns, and semantic meaning. By combining these techniques, we expect to deliver more relevant code examples to the generation model, particularly for complex programming tasks that require specialized components. After these steps, we will validate our improvements using the CodeXGLUE benchmark(Lu et al., 2021a) and CodeRagBench (Wang et al., 2025), which offers a diverse set of real-world programming tasks. This will allow us to consistently compare and measure how each enhancement impacts code quality across different challenges. We believe that this approach reflects how experienced developers search for and adapt existing code to solve new problems.

6 Conclusion

Retrieval-Augmented Generation (RAG) shows a lot of potential to make code generation more accurate and flexible. Through our review of previous work and our reproduction of the REDCODER framework (Parvez et al., 2021), we highlighted both the strengths and current challenges of using retrieval-based methods. Although RAG can improve tasks like code synthesis, completion, and translation by introducing helpful examples, it still struggles to judge how useful those examples are and apply them in real-world coding scenarios. Using the CodeRAG-Bench framework (Wang et al., 2025) and improving both our retriever and generator, our work is closer to supporting real-world development. We believe it is important to build systems that use smarter retrieval signals, better understand when and how to include context, and can generate both code and clear explanations to go with it.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *Preprint*, arXiv:2103.06333.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Kounianhua Du, Jizheng Chen, Renting Rui, Huacan Chai, Lingyue Fu, Wei Xia, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. 2024. Codegrag: Bridging the gap between natural language and programming language via graphical retrieval augmented generation. *Preprint*, arXiv:2405.02355.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

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 Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *Preprint*, arXiv:2211.11501.

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. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Preprint*, arXiv:2005.11401.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021a. Codexglue: A machine learning benchmark dataset for code understanding and generation. *Preprint*, arXiv:2102.04664.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021b. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *Preprint*, arXiv:2108.11601.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. *Preprint*, arXiv:2212.10481.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. Coderag-bench: Can retrieval augment code generation? *Preprint*, arXiv:2406.14497.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023b. Repocoder: Repository-level code completion through iterative retrieval and generation. *Preprint*, arXiv:2303.12570.

Xiangyu Zhang, Yu Zhou, Guang Yang, and Taolue Chen. 2023c. Syntax-aware retrieval augmented code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1291–1302, Singapore. Association for Computational Linguistics.