Chain-of-Function: Inducing Sequential Plans Over Decomposed Code Indexes for Code Generation

Jongyoon Kim, Seung-won Hwang*
Seoul National University, Seoul, Republic of Korea

{ john.jongyoon.kim, seungwonh }@snu.ac.kr

Abstract

Large Language Models (LLMs) have shown performance gains in code generation tasks via Retrieval Augmented Generation (RAG). However, these methods, known as Code-RAG, implicitly rely on retrieving ground-truth code, so their gain mostly comes from this. This reflects the issue of the Code-RAG pipeline itself, which has not been designed for reusability. When the gold snippet is absent, LLMs easily fail to leverage the retrieved codes. To overcome this, we propose Chain-of-Function (CoF): treat code as a composition of reusable functions. We decompose the retrieval pool into individual functions and encode them as pseudocode for a modality match between code and problem description. At inference, we generate a pseudocode plan based on the problem description, decompose it into required functions, and retrieve each independently, enabling function reuse without a gold snippet. On LiveCodeBench, our method achieves up to a +2% pass rate over PERC and improves the reuse rate, the number of retrieved codes referred on generated code increases to 2.3% from 0.69%, with shorter retrieved context compared to PERC.

Ⅰ. Introduction

Retrieval-augmented generation (RAG) for code aims to improve large language models (LLMs) by supplying relevant external code during inference. While effective in some cases, recent benchmarks such as CodeRAG-Bench show that substantial gains occur mainly when the retrieved snippet is a *gold snippet*, a complete, ground-truth solution. Without such a snippet, models often ignore the retrieved content.

A key challenge highlighted by recent empirical studies is the **low reusability of retrieved code** [1,2]. Current pipelines index code at coarse granularity, such as entire code file or large code blocks, collapsing multiple uncrelated functions into a single emberdding. Even wehn retrievers find partially relevant snippets, models often struggle to disentagle useful components from the surrounding irrelevant code blocks. Real codebases are compositional: a solution can be expressed as $C = f_1 \circ f_2 \circ \dots \circ f_n$, where each is a distinct, reusable function. When retrieval ignores this **chain-of-function** structure, relevant functions are buried in noise, making partial reuse difficult.

Our **chain-of-function retrieval** framework redefines the retrieval pool by decomposing code into function-level units, which are then represented as pseudocode and indexed independently. This enables partial reuse without relying on a gold snippet, improving both accuracy and reuse rate. The query process mirrors this decomposition: we generate a high-level pseudocode plan for the problem, break it down into functional steps, and use each step as a precise query to retrieve a corresponding function. These retrieved functions are then aggregated and supplied as context to an LLM to generate the final solution.

II. Related Works

Prior work in Code-RAG has advanced along two primary axes: retrieval indexing and query formation. Initial approaches to **indexing** typically treated entire files or large code blocks as single documents for retrieval [3], which often dilutes the semantic signal of individual functions. Recognizing this limitation, more recent work has explored finer-grained indexing strategies, such as retrieving at the function level [4] or leveraging structured representations like programming knowledge graphs [5].

Similarly, **query formation** techniques have grown more sophisticated, moving from simple similarity search [3] toward structured approaches like PERC (plan-as-query) [6] or iterative refinement [7,8]. However, despite these parallel advancements, a fundamental disconnect persists: the granularity of the retrieved items often fails to match the compositional nature of the code generation task. Our work directly addresses this gap by synchronizing both the retrieval and query processes at the function level, ensuring that each piece of retrieved information is a targeted, reusable component for the final code assembly.

Ⅲ. Methodology

A. Problem Formulation

In a standard Code-RAG setting, the goal is to generate a solution code C_s for a given problem description p, using a set of retrieved code contexts E. The generation is conditioned on a prompt that includes the task query (q) and the retrieved examples: $c_s = LLM(p, E;\ Prompt)$ The retrieved contexts E are selected from a large retrieval pool D by scoring each candidate $c \in D$ based on its similarity to a query q:

$$E = Top_{c \in D} sim(\psi(q), \psi(c))$$

where ψ is an encoding function. This formulation highlights the dependency on retrieving a highly relevant code snippet$c$.

B. Stage 1: Function-Level Retrieval Pool Construction

Our approach begins by reformulating the retrieval pool D. Instead of indexing entire code solutions, we use an LLM to refactor each code snippet (c) into a set of single-responsibility functions $\{ f_1, f_2, \dots, f_n \}$. Each function is then converted into natural language pseudocode. This process builds a retrieval pool that focuses on the functional essence of the code rather than its syntax, enhancing retrieval accuracy.

* Corresponding author

B. Stage 2: Plan-Guided Function Induction and Retrieval

At inference, we first generate a high-level pseudocode plan for the given problem description (p). This plan is then decomposed into a list of required functions, $\{g_1, g_2, \ldots, g_n\}$. For each required function, we generate a detailed pseudocode description, which serves as a fine-grained query. Unlike PERC, that uses the entire plan as one query, our function-specific queries allow for the precise retrieval of relevant, reusable components.

IV. EXPERIMENTS AND RESULTS

A. Experimental Setup

- **Dataset:** LiveCodeBench [7] v5 (880 problems) with the programming-solutions corpus from CodeRAG-bench [8].
- **Models:** nomic-embed-text for retrieval; llama3.1-8B, 70B [9], and gpt-4o-mini [10] for generation.
- **Metric:** Pass@1, which measures the percentage of test cases passed by the generated solution.

B. Results

Table 1 shows that CoF consistently improves the Pass@1 score by +1.6% to 2% over the state-of-the-art PERC model across various LLMs. This indicates that CoF retrieves more relevant functions that are useful for solution generation.

	llama3.1-8b	**llama3.1-70b**	**gpt-4o-mini**
No Retrieval	23.64	36.05	52.61
Problem-as-query	22.84	36.7	53.52
PERC [6]	24.2	35.57	52.84
CoF (ours)	**25.8**	**37.62**	**54.02**

Table 1: Pass@1 score of LiveCodeBench [3] problem with programming-solution corpus from CodeRAG-Bench [2].

C. Analysis

Table 2 highlights CoF's superior reusability. CoF reused 2.3% of retrieved functions for solving 37 questions, compared to only 0.69% for PERC for 11 questions. As CoF retrieves more functions while shortening the length of the retrieved context, we can validate that the retrieved context from CoF enables LLMs to leverage the context more effectively.

	PERC [6]	CoF (ours)
Average Number of codes retrieved	3	**4.71**
Average length of retrieved codes	857.56	**677.83**
Number of retrieved codes referred	0.69% (11 questions)	**2.3% (37 questions)**

Table 2: Reusability Analysis between retrieved codes and generated codes.

V. Conclusion

We addressed the limitations of Code-RAG by decomposing the retrieval pool into pure, single-responsibility functions and aligning queries via pseudocode decomposition. Our CoF approach significantly enhances retrieval precision and solves more problems on a Pass@1 basis. Future work includes exploring the dynamic assembly of retrieved units via an agentic approach to close remaining performance gaps.

ACKNOWLEDGMENT

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) [NO.RS-2021-II211343, Artificial Intelligence Graduate School Program (Seoul National University)] and also supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by ICT R&D program of MSIT/IITP (2022-0-00995, Automated reliable source code generation from natural language descriptions).

References

[1] W. Gu *et al.*, "What to Retrieve for Effective Retrieval-Augmented Code Generation? An Empirical Study and Beyond," *arXiv preprint arXiv:2503.20589*, 2025.

[2] Z. Z. Wang *et al.*, "Coderag-Bench: Can Retrieval Augment Code Generation?," *arXiv preprint arXiv:2406.14497*, 2024.

[3] F. Zhang *et al.*, "Repocoder: Repository-Level Code Completion Through Iterative Retrieval and Generation," *arXiv preprint arXiv:2303.12570*, 2023.

[4] Y. Ding *et al.*, "Function-Level Code Retrieval with Multi-Faceted Representations," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023.

[5] I. Saberi and F. Fard, "Context-Augmented Code Generation Using Programming Knowledge Graphs," *arXiv preprint arXiv:2410.18251*, 2024.

[6] J. Yoo, H. Han, Y. Lee, J. Kim, and S.-W. Hwang, "PERC: Plan-As-Query Example Retrieval for Underrepresented Code Generation," *arXiv preprint arXiv:2412.12447*, 2024.

[7] D. Zan *et al.*, "When Language Models Meet Code: A Survey on Retrieval-Augmented Code Generation," *arXiv preprint arXiv:2312.17581*, 2023.

[8] W. Gu *et al.*, "What to Retrieve for Effective Retrieval-Augmented Code Generation? An Empirical Study and Beyond," *arXiv preprint arXiv:2503.20589*, 2025.

[7] N. Jain *et al.*, "Livecodebench: Holistic and Contamination Free Evaluation of Large Language Models for Code," *arXiv preprint arXiv:2403.07974*, 2024.

[9] AI@Meta, "Llama 3 Model Card," 2024. [Online]. Available: https://huggingface.co/docs/hub/en/model-cards

[10] J. Achiam *et al.*, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.