SelfRACG: Enabling LLMs to Self-Express and Retrieve for Code Generation

Qian Dong[1,3], Jia Chen[2], Qingyao Ai[3,1]*, Hongning Wang[1], Haitao Li[1],
Yi Wu[2], Yao Hu[2], Yiqun Liu[1], Shaoping Ma[1]

[1]DCST, Tsinghua University, Beijing, China

[2]Xiaohongshu Inc., China [3]Quan Cheng Laboratory, China

{dq22,liht22}@mails.tsinghua.edu.cn, {aiqy,hw-ai,yiqunliu,msp}@tsinghua.edu.cn,

{chenjia2,xiaohui,xiahou}@xiaohongshu.com

Abstract

Existing retrieval-augmented code generation (RACG) methods typically use an external retrieval module to fetch semantically similar code snippets used for generating subsequent fragments. However, even for consecutive code fragments, the content often diverges due to logical progression, resulting in a content gap. This gap undermines the performance of current RACG methods, as *external* retrieval modules based on content matching fail to infer the specific information need of LLMs to generate the next code fragment. Therefore, we propose **SelfRACG**, a novel paradigm that enables large language models (LLMs) to **Self**-express their information needs to enhance **RACG**. Specifically, SelfRACG includes an information need expression module and a two-stage information need-guided training strategy, which encourages LLMs to express their information need. Extensive experiments demonstrate that SelfRACG can retrieve external knowledge that better aligns with the LLM's own information needs, resulting in superior generation performance compared to vanilla RACG. Moreover, both the training and deployment costs for retrieval in our framework are much lower than those of the strongest retrieval model. [1]

1 Introduction

Current retrieval-augmented code generation (RACG) techniques typically employ external retrieval modules, such as BM25 (Robertson et al., 2009) or embedding-based retrieval models (Muennighoff et al., 2024; Lee et al., 2024), to identify code fragments that closely align in content with preceding codes, thereby enhancing the generation of subsequent fragments.

However, even consecutive code fragments can differ significantly in content due to the progression of logic. As shown in Figure 1, the preceding

---

* Corresponding author.

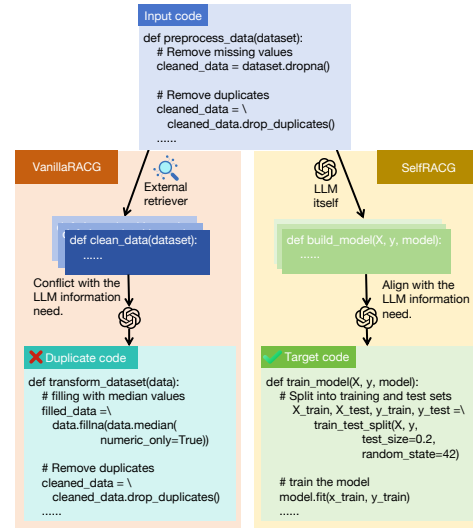[1]https://github.com/CSQianDong/SelfRACG

Figure 1: Comparison of vanilla RACG vs. SelfRACG.

code fragment might implement a function for data pre-processing, such as cleaning or normalizing a dataset, while the subsequent fragment could define an entirely different function for training a machine learning model on the processed data. Although these two fragments are part of the same workflow, their content differs substantially, as each addresses a distinct functionality. Consequently, existing RACG methods struggle to effectively improve overall performance, as the *external* retrieval modules often fail to accurately fetch the knowledge required for code generation. These external retrievers typically rely on similarity to retrieve code fragments for RACG. However, the retrieved fragments are often functionally similar to the existing code, rather than specifically aligned with the LLM's actual generation needs. Building on the observation that LLMs inherently possess the capability to generate the next token, we argue that the hidden states associated with the next token already encapsulate the information needs required for future content generation. Therefore, a critical research question is: **Can LLMs express their own information needs for RACG?**

10694

To enable LLMs to **Self**-express their information needs and perform **RACG**, our **Self-RACG** paradigm introduces two core components: an Information Need Expression (INE) module and a two-stage Information Need-Guided (ING) training strategy. Specifically, INE module is achieved by a parameter-efficient fine-tuning technique named Layer-wise Low-Rank Adaptation (L-LoRA). Through L-LoRA, we implement a retrieval-aware attention operation parallel to self-attention at each layer of the LLM. With minimal additional parameters and training costs, we extract the LLM's information needs from the hidden state of the next token. Since the retrieval-aware attention operates completely in parallel with the LLM's original self-attention, the next-token hidden states remain unaffected. Consequently, the original generation capabilities of the LLM can be fully preserved. The second key technique of SelfRACG is ING, a two-stage training strategy that enables LLMs accurately retrieve the information required for subsequent code generation. The first stage of ING leverages existing code from GitHub to create training samples, providing the LLM with foundational code retrieval skills. The second stage then uses a small amount of LLM-synthesized data to further train the model, aligning its retrieval capabilities with its own generation preferences.

We conduct extensive experiments to validate our method on two comprehensive benchmarks using various top-tier code LLMs. The experimental results demonstrate that SelfRACG effectively bridges the content gap between retrieved and subsequent code fragments, thereby improving the quality of code generation. Moreover, both the training and deployment costs of our INE module for retrieval are much lower than those of the strongest retrieval module.

The key contributions of our work can be summarized as follows:

- We propose SelfRACG, a novel paradigm that enables LLMs to self-express their own information needs for RACG.

- We introduce the INE module and ING training strategy to equip LLMs with the capability of expressing information needs at a low cost, without compromising the LLMs' original capabilities.

- We conduct comprehensive experiments on two benchmarks using multiple code LLMs, demonstrating effectiveness of our method.

2 Related Work

2.1 Code Large Language Models

Recent advancements in code large language models (LLMs), such as Qwen2.5-Coder (Hui et al., 2024), OpenCoder (Huang et al., 2024), and DeepSeek-Coder (Guo et al., 2024), have demonstrated impressive capabilities in code generation. These models leverage vast amounts of code data to provide functional code snippets across diverse tasks. However, their performance often degrades when dealing with repository-level code completion, which requires fine-grained understanding of repository-specific contexts and dependencies (Zhang et al., 2023; Wang et al., 2024). To mitigate this issue, various retrieval-augmented generation methods have been proposed (Zhang et al., 2023; Liu et al., 2024b; Lu et al., 2022).

2.2 Retrieval-Augmented Code Generation

RACG has emerged as a promising paradigm for enhancing code generation by integrating external context into the generation pipeline (Tan et al., 2024; Parvez et al., 2021; Gao et al., 2023). Traditional RACG methods rely on retrievers like BM25 (Robertson et al., 2009) or embedding-based models (Muennighoff et al., 2024; Lee et al., 2024) to fetch contextually relevant code snippets from repositories. For example, RepoCoder (Zhang et al., 2023) employs an iterative retrieval-generation framework to iteratively refine retrieved code snippets. Similarly, Repohyper (Phan et al., 2024) introduces a semantic graph-based retrieval method to capture broader contextual relationships within repositories. RACG could address the limitations of parametric knowledge in LLMs. Despite the advantage, existing RACG approaches often struggle to bridge the content gap between retrieved codes and the model's information needs, leading to suboptimal generation performance.

2.3 LLM-based Embedding Models

Beyond generative tasks, LLMs have also been adapted for text embedding tasks, enabling them to serve as powerful embedding models. GritLM (Muennighoff et al., 2024) unifies generative and embedding tasks within a single model through generative representational instruction tuning through extensive training costs. This approach enables GritLM to excel in both embedding and generation. Besides, gte-Qwen (Li et al., 2023b),

instructor (Su et al., 2022) and NV-Embed (Lee et al., 2024) extend embedding tasks with instruction tuning, enabling fine-grained control over embedding generation by explicitly encoding task-specific instructions. These works highlight the potential of LLMs in embedding tasks, presenting new possibilities for retrieval and semantic understanding.

3 Method

In this section, we introduce our proposed Self-RACG framework. Section 3.1 introduces the Information Need Expression (INE) module, which enables LLMs to express their information needs from hidden states for retrieval. Section 3.2 outlines the two-stage Information Need-Guided (ING) training strategy, which is designed to train the INE module, ensuring that the retrieved code snippets align with the LLM's generation needs. Figure 2 illustrates the workflow of SelfRACG.

3.1 Information Need Expression Module

The design of the INE module is guided by two key principles: (1) introducing minimal additional parameters to reduce training costs, and (2) equipping the LLM to produce retrieval representations without compromising its original generation capabilities. To achieve this, the INE module is implemented as a parallel component to the traditional self-attention mechanism, incorporating a retrieval-aware attention operation through our proposed Layer-wise Low-Rank Adaptation (L-LoRA).

Specifically, for each transformer layer $l \in [1, L]$, the projections $\mathcal{X}^{(l)}$ of original self-attention are first computed as

$$\mathcal{X}^{(l)} = h^{(l)} W_{\mathcal{X}}^{(l)}, \qquad (1)$$

where $\mathcal{X}$ is a wildcard representing the $\{Q, K, V\}$ projections in self-attention. $W_{\mathcal{X}}^{(l)} \in \mathbb{R}^{d \times d}$ are the weight matrices of the $\{Q, K, V\}$ projections. Then, the scaled dot-product self-attention at layer l is computed as

$$h^{(l+1)} = \text{Softmax}\left(\frac{Q^{(l)} K^{(l)\top}}{\sqrt{d}}\right) V^{(l)}. \quad (2)$$

To enable parameter-efficient tuning of LLMs, the vanilla LoRA (Hu et al., 2021) decomposes weight updates $\Delta W_{\mathcal{X}}^{(l)} \rightarrow \alpha A_{\mathcal{X}}^{(l)} B_{\mathcal{X}}^{(l)\top}$. For each projection $\{Q, K, V\}$, the computation becomes

$$\tilde{\mathcal{X}}^{(l)} = \mathcal{X}^{(l)} + \alpha h^{(l)} A_{\mathcal{X}}^{(l)} B_{\mathcal{X}}^{(l)\top}, \qquad (3)$$

where α is a scaling factor that controls the magnitude of the adaptation, and $rank$ represents the rank of low-rank matrices $A_{\mathcal{X}}^{(l)}, B_{\mathcal{X}}^{(l)} \in \mathbb{R}^{d \times rank}, rank \ll d$. Subsequently, $h^{(l+1)}$ is computed by Equation (2) by replacing $\mathcal{X}$ with $\tilde{\mathcal{X}}$.

Vanilla LoRA shifts the hidden states of the subsequent layers, potentially affecting the generation quality of the LLM. However, there is a significant gap between retrieval and generation tasks, making it challenging to optimize both simultaneously with limited training resources.

Therefore, as illustrated in Figure 2, our approach computes retrieval-related projections using L-LoRA and introduces a parallel attention mechanism alongside the original self-attention to produce representations specifically for the retrieval task. The computation of retrieval-related projections is similar to vanilla LoRA, which can be defined as

$$\mathcal{X}_{\text{ret}}^{(l)} = \mathcal{X}^{(l)} + \underbrace{\alpha h^{(l)} A_{\mathcal{X},\text{ret}}^{(l)} B_{\mathcal{X},\text{ret}}^{(l)\top}}_{\text{Extra computational cost.}}. \qquad (4)$$

Only $\alpha h^{(l)} A_{X,\text{ret}}^{(l)} B_{X,\text{ret}}^{(l)\top}$ represents the additional computational cost of calculating the retrieval-related projections, while $\mathcal{X}^{(l)}$ can be directly reused from the original self-attention, as shown in the left of Figure 2. Notably, we only require the query projection of the last token t to compute the layer-wise retrieval representation. The layer-wise retrieval representation can be computed by

$$r^{(l)} = \text{Softmax}\left(\frac{Q_{ret,t}^{(l)} K_{ret}^{(l)\top}}{\sqrt{d}}\right) V_{ret}^{(l)}, \quad (5)$$

where $Q_{ret,t}^{(l)}$ is query projection for last token t.

The information need of the LLM can be expressed as the mean pooling of each layer's retrieval representations, which can be computed as

$$r = \frac{1}{L} \sum_{l=1}^{L} r^{(l)}. \qquad (6)$$

This mean-pooling aggregation helps refine the retrieval process by leveraging both shallow and deep contextual features from self-attention.

L-LoRA and vanilla LoRA primarily differ in how the computed projections are used. After obtaining the retrieval-related projections $\mathcal{X}_{ret}^{(l)}$
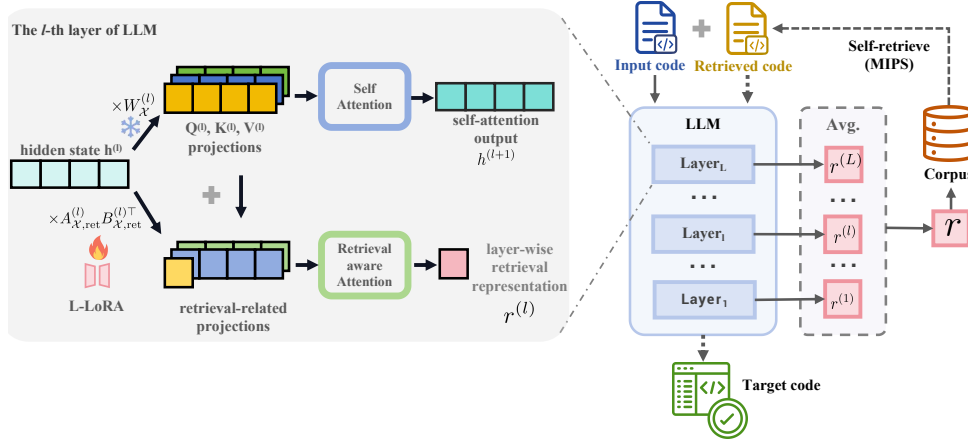
Figure 2: The workflow of SelfRACG. We use the l-th layer as an example and provide a detailed explanation of the relationship between retrieval-aware attention and self-attention within the gray box at the left of figure.

through L-LoRA, they are employed in a parallel retrieval-aware attention operation alongside the original self-attention. In contrast, projections $\tilde{\mathcal{X}}^{(l)}$ in Equation (3) obtained through vanilla LoRA are used within the original self-attention operation. As a result, L-LoRA effectively decouples the generation and retrieval tasks, enabling the rich information encoded in the LLM's hidden states to be efficiently extracted for retrieval, while preserving the model's original generation capabilities.

3.2 Information Need-Guided Training

To efficiently align the LLM's retrieval capability with its generation needs, we propose a two-stage training strategy, namely Information Need-Guided (ING) training, as illustrated in Figure 3.

Stage 1: Retrieval Learning. In Stage 1 of ING training, we leverage existing GitHub repositories to construct extensive unsupervised training pairs. For a code fragment q, the positive sample is the next code fragment p, while the negative pool $\mathcal{N}$ comprises code fragments from in-batch training pairs. After stage 1 training, the INE module gains the ability to capture general logical patterns, such as common design structures and typical coding habits. However, it is not aligned with the LLM's generation preferences.

Stage 2: Preference Alignment. To align retrieval capability with the LLM's generation preference, we synthesize candidate fragments $\{g_1, g_2, ..., g_k\}$ via LLM itself for a code fragment q. The positive sample g_1 corresponds to the LLM's immediate next-step generated code fragment, while the negative pool $\mathcal{N}$ combines synthetic negatives $\{g_2, ..., g_k\}$ and in-batch negatives. This stage ad-
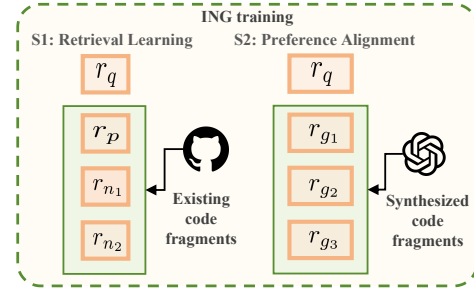

Figure 3: The illustration of two stage in ING training.

dresses permutation ambiguities by training the INE module to identify the precise fragment required for next-step generation.

Both stages employ the contrastive learning objective, which can be defined as

$$\mathcal{L} = -\log \frac{e^{s(r_q, r_p)}}{e^{s(r_q, r_p)} + \sum_{r_n \in \mathcal{N}} e^{s(r_q, r_n)}}, \quad (7)$$

where r_q denotes the context representation from INE. $s(\cdot, \cdot)$ means the inner production operation. r_p and r_n are the positive and negative candidate representations, respectively. Following prior works (Muennighoff et al., 2024; Lee et al., 2024; Li et al., 2023b), when calculating r_p and r_n through INE, we prepend a prefix to denote them as retrieval candidates. In contrast, r_q is computed directly, without any prefix or suffix. This distinction ensures that the representation of the context and the candidates are produced in different formats.

In summary, the INE module and the ING training strategy enable LLMs to express their information needs and retrieve code fragments that are useful for subsequent generation at low computational cost.

3.3 Inference

To effectively leverage the retrieval representations produced by the INE module during inference, we follow a straightforward yet efficient process that ensures efficiently code retrieval for RACG.

First, we pre-process the code repository by encoding each code fragment using the INE module. These representations are then stored in a vector database, creating a dense embedding index that can be efficiently searched at inference time.

During inference, the incomplete code fragment is first processed by the same INE module to generate its retrieval representation. This representation is then used to perform Maximum Inner Product Search (MIPS) (Karpukhin et al., 2020), allowing LLM to identify the code snippets that align with its generation needs. The inference process is illustrated on the right side of Figure 2. The solid lines represent the process by which the LLM expresses its information needs, while the dashed lines denote the subsequent step where the retrieved code fragments (via MIPS) are combined with the original code fragment to form a new input, which is then used to generate the target code.

4 Experimental Setup

4.1 Baselines

The improvement of the SelfRACG paradigm lies in integrating the retrieval module with the LLM and ensuring that the retrieved results align closely with the LLM's information needs. Therefore, we validate the effectiveness of our method by comparing it with different retrieval models. We refer to the RACG paradigm with an additional retrieval model as VanillaRACG. For VanillaRACG, we use the following retrieval models:

- **BM25** (Robertson et al., 2009): A traditional sparse retrieval method based on term frequency-inverse document frequency.

- **OpenAI-Small/Large** [2]: OpenAI-Small/Large are the two strongest closed-source embedding models from OpenAI, specifically `text-embedding-3-small` and `text-embedding-3-large`. These models can be used for effective code fragment retrieval.

- **GritLM** (Muennighoff et al., 2024): GritLM is the closest baseline to SelfRACG, as it enables LLMs to support both embedding and generation tasks simultaneously. However, it relies on full parameter fine-tuning, which requires substantial computational resources and training data.

- **NV-Embed-v2** (Lee et al., 2024): A state-of-the-art embedding model ranked top on the MTEB (Muennighoff et al., 2022) benchmark. However, NV-Embed-v2 can only be used to produce embeddings and is not capable of handling generation tasks.

4.2 Benchmarks

We evaluate our method on two benchmarks:

- **RepoEval** (Zhang et al., 2023): This benchmark is designed for repository-level code completion tasks and comprises three sub-tasks:

  - **Line-level Completion**: Tasks that require the model to complete a single line of code based on the context.
  - **API-level Completion**: Tasks that involve completing function calls or API invocations.
  - **Function-level Completion**: Tasks that need the model to complete the body of a function based on its definition or usage.

- **CrossCodeEval** (Ding et al., 2024): This benchmark focuses on cross-file contextual understanding. Built upon real-world code repositories, it challenges models to utilize dependencies and contextual information across multiple files to generate accurate code completions.

4.3 Evaluation Metrics

We adopt multiple metrics to comprehensively evaluate both retrieval and generation performances:

4.3.1 Code Completion Metrics

- **Exact Match (EM)**: Measures the percentage of predictions that exactly match the ground-truth code. This metric is applied to all sub-tasks.

- **Edit Similarity (ES)**: Evaluates the similarity between the generated and ground-truth code fragments using a normalized Levenshtein distance. It is also employed for all sub-tasks.

- **Pass@1**: It assesses whether the code generated via greedy decoding passes all relevant test cases when replacing the original code. This metric is exclusively utilized for function-level completion task in RepoEval (Zhang et al., 2023).

---

[2] https://platform.openai.com/docs/api-reference/embeddings

4.3.2 Code Retrieval Metrics

- **Recall@K**: Evaluates the proportion of ground-truth code fragments that are successfully retrieved within the top K results.

- **MRR@10**: Mean Reciprocal Rank at 10, calculated as the mean of the reciprocal ranks of the ground-truth code fragment for each sample. The reciprocal rank is given by $\frac{1}{pos}$, with pos being the position of the ground-truth code fragment.

4.4 Specific Settings

To thoroughly evaluate the effectiveness of our method, we select multiple top-tier code LLMs across various sizes and versions. We use the instructed version of Qwen2.5-Coder and (Hui et al., 2024) the base version of OpenCoder (Huang et al., 2024). For Qwen2.5-Coder, we experiment with both 3B and 7B parameter sizes, and for Open-Coder, we use the 1.5B and 8B parameter sizes.

For the Stage 1 of ING, the code data is taken from the GitHub provided by CodeRAG-Bench (Wang et al., 2024), which contains about 1.7 million code files. We filtered out repositories that are already used in benchmarks as well as files containing less than 20 lines of code. After filtering, we constructed a final corpus of about 1.3 million files. Following prior work (Zhang et al., 2023; Liu et al., 2024b), we split the corpus into code fragments with a 20-line interval. The same splitting method was applied to the code in benchmarks. Notably, we did not use any supervised training datasets (Xie et al., 2023; Nguyen et al., 2016; Chen et al., 2025), which are widely adopted for training embedding models in prior works (Dong et al., 2023; Li et al., 2023a).

In the L-LoRA module, the rank is set to 16, and the alpha parameter is set to 32. For synthetic data generation, we use the vLLM (Kwon et al., 2023) inference library for acceleration. Starting with the first 20 lines of the preprocessed code corpus, we prompt the LLM to generate and complete the subsequent code fragments. In total, we generated 500k synthetic samples for the Stage 2 of ING.

We run all experiments on 8 NVIDIA A100 80GB GPUs. By utilizing multi-GPU parallelism, the largest model for synthetic data generation takes approximately 3.5 hours. The total data generation and training time for the largest model in the second phase of ING is 48 hours, corresponding to 384 GPU hours. In the Stage 2 of ING, we use 4 synthetic negative samples per code fragment.

5 Experimental Results

5.1 Code Completion Performance

In this section, we compare the code completion performance of VanillaRACG at different retrieval settings with SelfRACG. The results are presented in Table 1 and Table 6. As the trends in Table 1 and Table 6 are similar, we have moved Table 6 to Appendix E. From these tables, we can draw the following conclusions:

- Compared to no retrieval, RACG significantly improves the generation performance. Even the simplest retrieval model, BM25, leads to substantial improvements across all code completion sub-tasks and metrics.

- Embedding-based retrieval models outperform BM25, further enhancing code completion performance. This indicates that embedding-based retrieval models have stronger expressive capabilities than statistical methods, making them better at capturing the information needs of LLMs.

- LLM-based retrieval models deliver better generation results, highlighting the superior understanding power of LLMs in capturing complex information needs.

- Among all VanillaRACG configurations, GritLM-7B achieves the best generation performance. Fine-tuned on a large dataset, it is designed to integrate both retrieval and generation capabilities into one LLM. Compared to NV-Embed-v2, which focuses solely on the retrieval task, the data retrieved by GritLM-7B align more closely with the information needs of generation, resulting in better performance. However, GritLM-7B requires significant computational resources to integrate both generation and embedding capabilities, according to its paper (Muennighoff et al., 2024). In contrast, SelfRACG outperforms GritLM-7B while using 1/8 of the GPU hours.

- SelfRACG outperforms VanillaRACG based on GritLM-7B. This performance gain can be attributed to its unified design. SelfRACG uses the same LLM for both retrieval and generation, allowing it to extract retrieval representations directly from the next-token hidden states. This approach results in a closer alignment between the retrieved information and the LLM's own generation needs.

Table 1: The code completion experimental results of OpenCoder-Base

	RepoEval-API		RepoEval-Line		RepoEval-Function			CrossCodeEval		MEAN	
	EM	ES	EM	ES	EM	ES	Pass@1	EM	ES	EM	ES
OpenCoder-1.5B-Base											
w/o Retrieval	0.258	0.557	0.344	0.603	0.050	0.388	19.8%	0.033	0.421	0.171	0.492
VanillaRACG											
w/ BM25	0.297	0.604	0.396	0.636	0.065	0.414	24.8%	0.055	0.449	0.203	0.526
w/ OpenAI-small	0.303	0.607	0.408	0.647	0.074	0.430	24.8%	0.074	0.457	0.215	0.535
w/ OpenAI-large	0.303	0.611	0.415	0.652	0.084	0.421	26.3%	0.072	0.454	0.218	0.534
w/ NV-Embed-v2	0.328	0.626	0.423	0.650	0.074	0.415	26.9%	0.077	0.463	0.225	0.538
w/ GritLM-7B	0.322	0.624	0.428	0.657	0.081	0.427	27.6%	0.086	**0.470**	0.229	0.545
SelfRACG	**0.342**	**0.636**	**0.439**	**0.668**	**0.093**	**0.439**	**29.4%**	**0.088**	0.465	**0.241**	**0.552**
OpenCoder-8B-Base											
w/o Retrieval	0.292	0.591	0.377	0.628	0.059	0.414	24.8%	0.048	0.437	0.194	0.518
VanillaRACG											
w/ BM25	0.346	0.633	0.444	0.675	0.081	0.445	29.1%	0.084	0.477	0.238	0.557
w/ OpenAI-small	0.349	0.633	0.443	0.675	0.090	0.467	29.4%	0.104	0.483	0.247	0.565
w/ OpenAI-large	0.338	0.631	0.450	0.685	0.099	0.464	31.9%	0.106	0.487	0.248	0.567
w/ NV-Embed-v2	0.365	0.648	0.469	0.685	0.087	0.459	29.1%	0.114	0.494	0.259	0.572
w/ GritLM-7B	0.360	0.650	0.468	0.689	0.099	0.463	30.7%	0.128	0.501	0.264	0.576
SelfRACG	**0.391**	**0.680**	**0.481**	**0.705**	**0.124**	**0.481**	**33.8%**	**0.127**	**0.507**	**0.281**	**0.593**

Table 2: Retrieval performance of different retrieval models.

Model	#Params	#Dims	Recall@1	Recall@3	Recall@5	Recall@10	MRR@10
External Retriever							
OpenAI-small	-	1536	0.136	0.359	0.473	0.626	0.291
OpenAI-large	-	3072	0.149	0.359	0.484	0.644	0.304
GritLM-7B	7B	4096	0.174	0.444	0.592	0.748	0.356
NV-Embed-v2	7B	4096	0.178	0.465	0.584	0.712	0.357
SelfRACG							
OpenCoder-1.5B	1.5B	1536	0.208	0.433	0.528	0.664	0.358
Qwen2.5-Coder-3B	3B	1536	0.239	0.517	0.621	0.748	0.411
Qwen2.5-Coder-7B	7B	3584	**0.244**	0.508	0.616	0.746	**0.413**
OpenCoder-8B	8B	4096	0.237	**0.524**	**0.642**	**0.756**	**0.413**

5.2 Code Retrieval Performance

In this subsection, we evaluate the retrieval performance of various retrieval models. Since both API-level and Function-level completions involve the permutation of implementations, it is difficult to evaluate the retrieval results. Theoretically, given the context, the line-level ground truth and the LLM's information need for generating the next step are consistent. Therefore, we use the line-level completion samples from RepoEval for retrieval performance evaluation. Specifically, we treat the context as the query and the code fragment containing the target line as the positive ground truth, using Recall@K and MRR@10 to assess the retrieval performance. The results are summarized in Table 2. From the table, we observe the following trends: (1) SelfRACG outperforms other retrieval models in terms of MRR@10 and Recall@1. This indicates that by aligning the retrieval with the generation task, SelfRACG's retrieval capabilities are consistent with the generation logic of LLM, leading to more refined ranking results. (2) Among the

Table 3: Retrieval strategies comparison.

Retrieval Strategy	API	Line	Func
VanillaRACG (GritLM)	0.650	0.690	0.463
w/ next fragment	0.657	0.692	0.472
w/ next query	0.673	0.705	0.476
SelfRACG	0.680	0.705	0.481

different settings of SelfRACG, we observe that larger model sizes generally yield better retrieval performance. Scaling up the model in terms of parameters may further enhance its ability to retrieve information that aligns with the generation preferences of the LLM.

5.3 Alternative Retrieval Strategies

We evaluate the generation performance of different retrieval strategies using the three levels (API, line and function) of completion tasks from Repo-Eval and the ES metric. The base model used in this experiment is OpenCoder-8B-Base. The comparison results are presented in Table 3. Directly using similar code fragments to augment in generating the next code segment yields the weakest

Table 4: Ablation studies of SelfRACG.

Method	OpenCoder-1.5B		OpenCoder-8B	
	MRR@10	ES	MRR@10	ES
SelfRACG	0.358	0.6678	0.413	0.7048
w/o Stage 2.	0.316	0.6531	0.364	0.6912
w/o L-LoRA	0.296	0.6518	0.338	0.6847

Table 5: Overhead comparison of different methods.

Method	Training	Deployment
GritLM-7B	3,072 GPU hours	14GB
NV-Embed-v2	-	14GB
SelfRACG	384 GPU hours	0.02GB

performance. The *w/ next fragment* strategy directly uses the subsequent code fragment following the retrieved code fragment as external knowledge for RACG. However, the improvements are limited, as the subsequent fragment may not always align with the LLM's generation needs.

The *w/ next query* strategy leverages the LLM itself to generate a explicit query, which is then used to retrieve external knowledge by GritLM. As shown in Table 3, this strategy achieves better results, as the LLM-generated queries are more aligned with its own generation needs. However, it still performs worse than SelfRACG on API and function-level completion tasks. This is because SelfRACG directly produces a implicit query embedding from the next-token hidden states, which is more expressive than an explicit query. Additionally, it avoids the extra step of converting an explicit query into an embedding via GritLM, reducing information loss. Therefore, SelfRACG achieves the best generation performance.

It is worth noting that the performance of different strategies on line-level tasks is relatively similar. This is because line-level completions are generally more deterministic given the context, making the LLM's information needs easier to capture compared to more complex API or function-level tasks.

5.4 Ablation Studies

We conduct a step-by-step ablation of SelfRACG to analyze the impact of key techniques on retrieval and generation performance. Due to limited computational resources, we performed ablation experiments on OpenCoder using the RepoEval line-level completion data. The results are presented in Table 4. The first row shows the performance of SelfRACG using all techniques.

From the table, we can draw the following conclusions: The preference alignment in stage 2 of ING plays a crucial role in the model's performance. Without alignment with the generation preferences, knowledge conflicts occur, which negatively affect the model's generation quality. In the ablation experiment for L-LoRA, we fine-tuned the corresponding OpenCoder for embedding tasks using full parameter tuning. However, due to com-

putational limitations, full parameter fine-tuning results in smaller batch sizes. In the full parameter fine-tuning setup, the batch sizes for 1.5B and 8B OpenCoder are 32 and 4, respectively. A smaller batch size leads to fewer negative samples in contrastive learning, thus decreasing the retrieval performance. Therefore, under limited resources, L-LoRA could achieve better performance.

5.5 Overhead Comparison

In this section, we present a comparison between SelfRACG and Vanilla RACG with an LLM-based embedding model, analyzing their overhead from both training and deployment perspectives.

Table 5 presents a comparison of training and deployment overhead for different methods. For a fair comparison, the results for SelfRACG are reported using Qwen-2.5-Coder-7B. Notably, GritLM-7B requires substantial training overhead. It is designed to handle both generation and embedding tasks simultaneously, requiring 8 nodes with 8 NVIDIA A100 80GB GPUs. NV-Embed-v2 did not report its exact training resource consumption. SelfRACG based on Qwen-2.5-Coder-7B requires only a single node with 8 NVIDIA A100 80GB GPUs and a total of 384 GPU hours, equipping the LLM with self-expressive information need capabilities without collapsing its generation ability. Both NV-Embed-v2 and GritLM-7B are standalone embedding models, requiring an additional 14GB of VRAM for deployment in half-precision mode. In comparison, SelfRACG shares the same LLM backbone as the generation model, with only a small number of additional parameters introduced.

6 Conclusion

In this work, we introduced SelfRACG that allows LLMs to self-express their own information needs for code generation. SelfRACG integrates an information need expression module and a two-stage information need-guided training strategy, improving retrieval performance while preserving generation capabilities. Experiments on two benchmarks show that SelfRACG outperforms existing retrieval models at less than 1/8 of the training cost, providing a more efficient solution for RACG.

Acknowledgments

References

Jia Chen, Qian Dong, Haitao Li, Xiaohui He, Yan Gao, Shaosheng Cao, Yi Wu, Ping Yang, Chen Xu, Yao Hu, et al. 2025. Qilin: A multimodal information retrieval dataset with app-level user sessions. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3670–3680.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36.

Qian Dong, Yiding Liu, Qingyao Ai, Haitao Li, Shuaiqiang Wang, Yiqun Liu, Dawei Yin, and Shaoping Ma. 2023. I3 retriever: incorporating implicit interaction in pre-trained language models for passage retrieval. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, pages 441–451.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming– the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *EMNLP (1)*, pages 6769–6781.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2024. Nv-embed: Improved techniques for training llms as generalist embedding models. *arXiv preprint arXiv:2405.17428*.

Haitao Li, Qingyao Ai, Jia Chen, Qian Dong, Yueyue Wu, Yiqun Liu, Chong Chen, and Qi Tian. 2023a. Sailer: structure-aware pre-trained language model for legal case retrieval. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1035–1044.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023b. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024a. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024b. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003*.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seungwon Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722*.

Niklas Muennighoff, Hongjin Su, Liang Wang, Nan Yang, Furu Wei, Tao Yu, Amanpreet Singh, and Douwe Kiela. 2024. Generative representational instruction tuning. *arXiv preprint arXiv:2402.09906*.

Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2022. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*.

Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. Ms marco: A human-generated machine reading comprehension dataset.

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

Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. 2024. Repohyper: Better context retrieval is all you need for repository-level code completion. *arXiv preprint arXiv:2403.06095*.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A Smith, Luke Zettlemoyer, and Tao Yu. 2022. One embedder, any task: Instruction-finetuned text embeddings. *arXiv preprint arXiv:2212.09741*.

Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *arXiv preprint arXiv:2405.07530*.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*.

Xiaohui Xie, Qian Dong, Bingning Wang, Feiyang Lv, Ting Yao, Weinan Gan, Zhijing Wu, Xiangsheng Li, Haitao Li, Yiqun Liu, et al. 2023. T2ranking: A large-scale chinese benchmark for passage ranking. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2681–2690.

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

A Limitation

We acknowledge several limitations in this study and aim to address them in future work. First, due to limited computational resources, our method has been validated only on LLMs with up to 8B parameters. Second, our approach has been evaluated exclusively on the RACG task; in future work, we plan to extend it to general retrieval-augmented generation scenarios to validate its broader applicability. In fact, the potential of L-LoRA extends beyond retrieval tasks. We will explore its application to various embedding tasks, enabling low-cost embedding capabilities for every LLM.

B Legal and Ethical Considerations

In our research, ethical considerations are given top priority throughout the entire development process. We ensure that our work is free from any discriminatory elements and does not violate personal privacy. We are committed to using only publicly accessible and authorized datasets, avoiding any data that could lead to biased or harmful outcomes. All models, datasets, and code associated with this research are publicly available, enhancing transparency and enabling the community to conduct further research and validation.

C Generation Performance with Different Number of Fragments

In this section, we analyze the impact of the number of retrieved code fragments on the generation performance for the line-level completion task, and present the results in Figure 4. By jointly analyzing Table 2 and Figure 4, we can draw the following conclusions. Although OpenCoder-1.5B has a lower Recall@10 compared to GritLM-7B and NV-Embed-v2, it performs better in terms of Recall@1 and MRR@10. This indicates that SelfRACG is able to rank relevant code fragment higher, leading to better generation results. Due to the "lost-in-the-middle" issue (Liu et al., 2024a), LLMs tend to ignore information from the middle of the context. Thus, a more accurate relevance modeling can significantly improve generation performance, as useful information is placed at the beginning of the context. As shown in the left part of Figure 4, SelfRACG benefits from an increasing number of recalled code fragments, with performance improving rapidly as more fragments are included. When only five code fragments are used, SelfRACG using OpenCoder-1.5B and OpenCoder-8B already outperforms other retrieval models that use ten fragments.

D Prompt Template

Following prior studies (Zhang et al., 2023; Wang et al., 2024; Liu et al., 2024b), we use the prompt template as presented in Figure 5.

E The results of Qwen2.5-Coder-Instruct.

To further evaluate the generalizability of Self-RACG, we conduct additional experiments using the Qwen2.5-Coder-Instruct (Hui et al., 2024) models, including both 3B and 7B variants. The results are summarized in Table 6.

Overall, the performance trends observed with Qwen2.5-Coder-Instruct are consistent with those reported in the main experiments using OpenCoder. These results further reinforce the robustness and
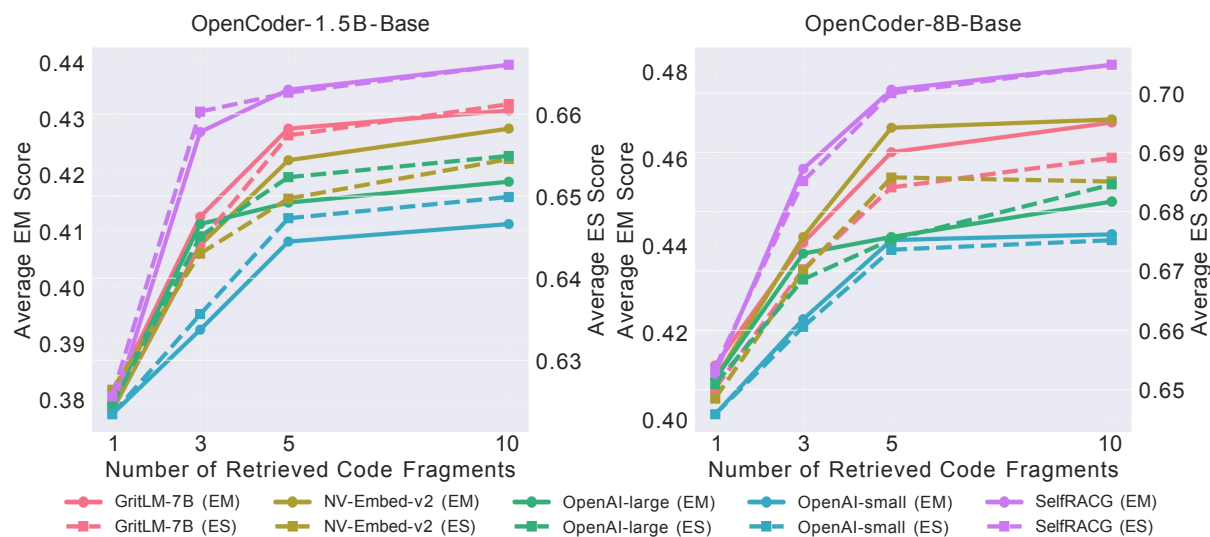
Figure 4: The generation performance with different number of retrieved code fragments.

general applicability of SelfRACG across different
model backbones.

```
# Here are some relevant code fragments from other files of the repo:
# The below code fragment 1 can be found in:
# file_handling/read_file.py
# -------------
# def read_file_lines(file_path):
#     with open(file_path, 'r') as file:
#         lines = file.readlines()
#     return lines
# -------------
# The below code fragment 2 can be found in:
# data_processing/parse_data.py
# -------------
# def parse_data(lines):
#     result = []
#     for line in lines:
#         parts = line.split(',')
#         result.append(parts)
#     return result
# -------------
# Based on above, complete the next statement of the following codes:
```
Retrieved Code Snippets

```
file_path = "data.txt"
read_lines = read_file_lines(file_path)
cleaned_lines = [clean_line(line) for line in read_lines]
# Now we want to parse the cleaned lines
```
Context Code

```
parsed_result = parse_data(cleaned_lines)
```
Predicted Statement

Figure 5: Prompt template used in SelfRACG.

Table 6: The code completion experimental results of Qwen2.5-Coder-Instruct.

	RepoEval-API		RepoEval-Line		RepoEval-Function			CrossCodeEval		MEAN	
	EM	ES	EM	ES	EM	ES	Pass@1	EM	ES	EM	ES
Qwen2.5-Coder-3B-Instruct											
w/o Retrieval	0.271	0.576	0.348	0.604	0.056	0.404	23.5%	0.040	0.426	0.179	0.502
VanillaRACG											
w/ BM25	0.311	0.609	0.406	0.643	0.065	0.405	25.7%	0.068	0.458	0.212	0.529
w/ OpenAI-small	0.313	0.609	0.418	0.660	0.084	0.425	26.9%	0.086	0.465	0.225	0.540
w/ OpenAI-large	0.317	0.611	0.426	0.667	0.084	0.432	27.2%	0.082	0.462	0.227	0.543
w/ NV-Embed-v2	<u>0.343</u>	0.623	0.436	0.669	0.084	0.425	27.9%	0.098	0.474	0.240	0.548
w/ GritLM-7B	0.335	<u>0.626</u>	<u>0.450</u>	<u>0.676</u>	<u>0.096</u>	<u>0.436</u>	<u>30.0%</u>	<u>0.103</u>	<u>0.480</u>	<u>0.246</u>	<u>0.555</u>
SelfRACG	**0.364**	**0.647**	**0.459**	**0.688**	**0.115**	**0.456**	**32.2%**	**0.105**	**0.484**	**0.261**	**0.569**
Qwen2.5-Coder-7B-Instruct											
w/o Retrieval	0.274	0.575	0.343	0.602	0.056	0.404	24.8%	0.054	0.428	0.182	0.502
VanillaRACG											
w/ BM25	0.316	0.603	0.410	0.645	0.059	0.412	27.9%	0.080	0.460	0.216	0.530
w/ OpenAI-small	0.324	0.611	0.412	0.652	0.087	0.436	28.2%	0.098	0.474	0.230	0.543
w/ OpenAI-large	0.321	0.605	0.427	0.663	0.090	**0.441**	29.1%	0.099	0.474	0.234	0.546
w/ NV-Embed-v2	<u>0.344</u>	0.619	0.438	0.668	0.084	0.428	26.9%	0.112	0.484	0.244	0.550
w/ GritLM-7B	0.339	<u>0.623</u>	<u>0.443</u>	0.668	<u>0.096</u>	<u>0.437</u>	<u>30.7%</u>	<u>0.115</u>	<u>0.496</u>	<u>0.248</u>	<u>0.556</u>
SelfRACG	**0.371**	**0.645**	**0.443**	**0.668**	**0.099**	0.436	**31.6%**	**0.132**	**0.499**	**0.261**	**0.562**