SRACG: A Code Generation Framework with Selective Retrieval Augmentation

Mengzhen Wang[1,2], Shukai Ma[1], Songwen Gong[1,2], Jiexin Wang[1,2], Ruolin Chen[1,2], Liuwen Cao[1,2], Yi Cai[1,2*]

[1]School of Software Engineering, South China University of Technology, Guangzhou, China
[2]Key Laboratory of Big Data and Intelligent Robot (SCUT), Ministry of Education of China
{202311089439, 202230481493, 202421045674}@mail.scut.edu.cn, jiexinwang@scut.edu.cn, {rolachen1999, caoliuwencc}@163.com, ycai@scut.edu.cn

Abstract

Large Language Models (LLMs) have demonstrated remarkable performance in code generation, offering new possibilities for translating natural language into executable programs. To further enhance LLMs' code generation capabilities, Retrieval-Augmented Generation (RAG) has emerged as a promising strategy by retrieving code examples aligned with the generation intent to guide the process. However, existing RAG-based methods often suffer from unnecessary augmentation, preference misalignment, and surface-level mimicry, which undermine the effectiveness of retrieved examples in guiding LLMs toward accurate code generation. To address these challenges, we propose SRACG, a Selective Retrieval-Augmented Code Generation framework. SRACG begins with a necessity-aware selection mechanism to identify generation intents that genuinely require retrieval support, thereby avoiding degradation from indiscriminate augmentation. For intents identified as needing enhancement, it first employs a multi-objective retrieval strategy to select examples that are semantically aligned with the intent. These candidates are then further filtered by assessing their consistency with the LLM's inherent generation preferences, ensuring alignment in both style and structure. Finally, it extracts execution plans from the filtered examples to uncover their underlying logic, guiding the LLM to better comprehend the examples instead of merely mimicking surface-level content. Experimental results on widely used benchmarks show that SRACG significantly improves the success rate of LLM-generated code and outperforms existing approaches.

Introduction

In recent years, LLMs (Roziere et al. 2023; Achiam et al. 2023; Wang et al. 2025; Anthropic 2024) have achieved remarkable success in code generation, advancing the research frontier of translating natural language into executable programs. These models exhibit impressive generalization capabilities across various programming languages and tasks, reshaping the automation landscape of software development (Dong et al. 2025; Yang et al. 2024; Chen et al. 2024). Meanwhile, RAG has emerged as a powerful paradigm for incorporating external knowledge into generative models, with proven effectiveness in question answering, dialogue, and summarization (Gao et al. 2023; Asai et al. 2023; Wang et al. 2024; Yuan et al. 2023). Inspired by this success, a growing body of work has applied RAG to code generation, retrieving relevant code examples from large-scale repositories to assist LLMs in generating more accurate code. These early explorations demonstrate the potential of RAG in programming tasks (Parvez et al. 2021; Zhou et al. 2022; Nashid, Sintaha, and Mesbah 2023; Ma et al. 2024).

However, existing RAG methods still face significant challenges when applied to code generation:

i) Unnecessary augmentation. Most approaches introduce retrieved examples indiscriminately for all generation intents, lacking an effective mechanism to assess whether augmentation is truly necessary. In many cases, LLMs are already capable of generating high-quality code on their own, and the unnecessary retrieval examples may introduce noise and even interfere with the LLM's generation process. As shown in Figure 1(a), the LLM generates correct code by counting character frequencies to find the first non-repeating character. However, after incorporating a retrieved example that identifies the first repeated character, it generates incorrect output. The mismatch between the example and the intent causes the LLM to follow an incorrect reasoning path.

ii) Preference misalignment. Current studies predominantly focus on improving the semantic relevance between retrieved examples and the generation intent, typically by enhancing retrieval precision, expanding retrieval corpora, or incorporating online retrieval. However, examples retrieved based on semantic similarity may not align with the LLM's intrinsic generation preferences, such as favored structural patterns or typical statement execution order. When retrieved examples significantly deviate from these internal preferences, they may confuse the model or disrupt its reasoning process, ultimately degrading generation quality rather than enhancing it. As illustrated in Figure 1(b), although the retrieved example is semantically aligned with the generation intent, it adopts an explicit for loop to construct the result list step by step, whereas the code directly generated by the LLM prefers a more concise list comprehension approach. While both implementations are logically correct, guiding the LLM with the retrieved example leads it to blend the two styles, which unexpectedly results in incorrect code generation.

iii) Surface-Level Mimicry. Existing approaches often

*Corresponding author: Yi Cai (ycai@scut.edu.cn)

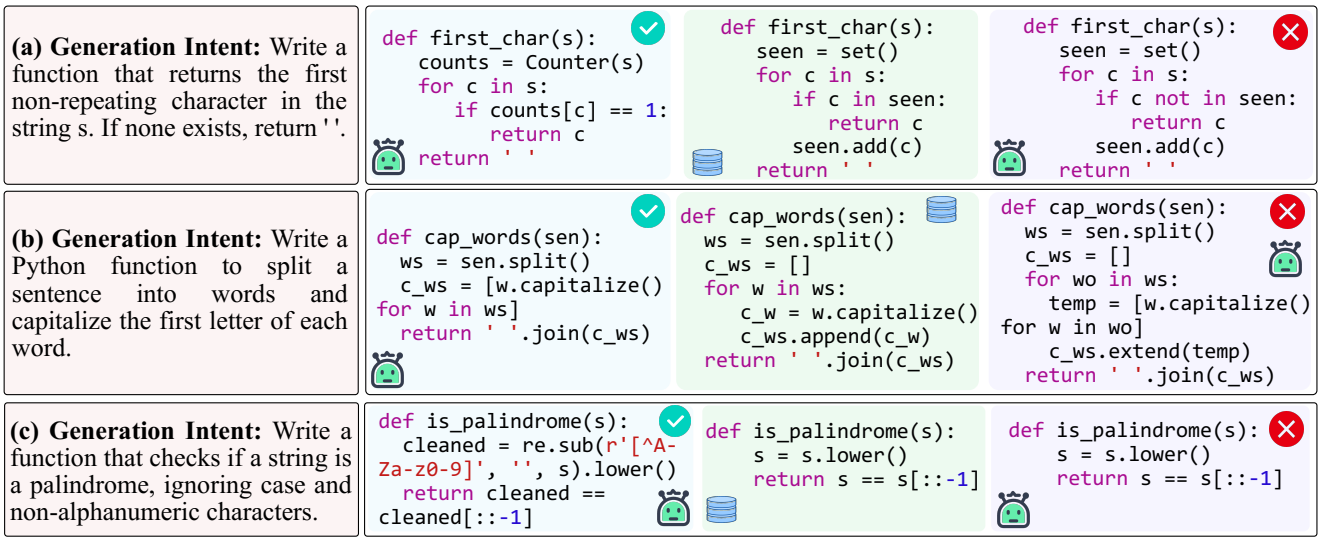

Figure 1: Illustration of failure cases in retrieval-augmented code generation. Blue regions denote LLM-generated code, green regions indicate retrieved examples, and purple regions show code produced under their influence.

incorporate retrieved examples directly into the prompt. In typical text-based RAG tasks, the retrieved passages usually contain explicit and readily usable information, enabling LLMs to easily extract key content. However, code is inherently more structured and logically complex than natural language text, which may cause LLMs to focus only on the surface-level information of the retrieved examples, failing to extract deeper and more useful implementation logic. As illustrated in Figure 1(c), although the retrieved example is semantically similar to the generation intent, it lacks the required handling of case insensitivity and non-alphanumeric characters, resulting in incomplete logic. Nevertheless, the LLM fails to perform deeper reasoning and instead copies the flawed logic directly into its output.

To address the aforementioned challenges, we propose a novel **Selective Retrieval-Augmented Code Generation (SRACG)** framework that aims to maximize the positive guidance of retrieved code examples on LLMs. Unlike prior approaches that apply example augmentation indiscriminately, our method introduces a necessity-aware selection mechanism combined with multi-dimensional enhancement strategies to achieve more accurate code generation. The overall framework consists of two key stages:

The first stage is necessity-aware intent selection. Inspired by CODET (Chen et al. 2022), we introduce an execution-based evaluation strategy using shared test cases to determine whether a generation intent requires retrieval augmentation. Unlike CODET, which relies on a limited set of test cases, we design diverse boundary conditions to ensure broader coverage and higher-quality testing. Specifically, for each intent, the LLM is prompted to generate multiple candidate codes along with corresponding test cases. These are then dual-executed, and the confidence score for each intent is computed based on the overall success rate across test cases. Intents with scores below a predefined threshold are selected for enhanced generation in the next stage. The rationale behind this strategy is that if multiple candidate codes consistently pass a diverse set of generated test cases, the model is likely reliable for that intent. In contrast, if most candidates fail or behave inconsistently, it indicates uncertainty in the model's generation. For such intents, relying on the model alone is insufficient, and external retrieval is needed to support generation.

The second stage is retrieval-augmented generation, in which SRACG optimizes the enhancement process along three strategies. For each intent needing augmentation, the framework first applies a multi-objective retrieval strategy that evaluates the semantic relevance between the intent and retrieval code examples from three perspectives: function signatures, code comments, and overall code content. This design improves retrieval robustness and yields an initial pool of candidate examples. Next, a preference consistency filtering strategy is introduced to further refine the candidates by assessing their alignment with the LLM's generation preferences. This evaluation considers structural patterns and statement execution order, ensuring that the selected code examples align with the model's inherent generation style. Finally, an execution plan extraction strategy is applied to the refined code examples to identify and distill their underlying execution logic. This encourages the LLM to incorporate more executable reasoning patterns during generation, rather than engaging in surface-level mimicry of the retrieved examples.

In summary, our contributions are as follows:

- We propose SRACG, a novel framework for selective retrieval-augmented code generation. By introducing a necessity-aware selection mechanism, it avoids performance degradation of LLMs caused by the indiscriminate inclusion of retrieved examples.
- We introduce multi-dimensional enhancement strategies that jointly consider semantic relevance and model gen-

eration preferences to select examples better aligned with the LLM's generation patterns, thereby enhancing the effectiveness of retrieval-based augmentation.

- We conduct comprehensive experiments on several widely used benchmarks. Results show that SRACG significantly improves the success rate of code generated by LLMs and consistently outperforms existing methods.

Related Works

Inspired by the success of RAG in natural language tasks, recent studies have extended this paradigm to code generation, aiming to mitigate the limitations of LLMs by retrieving relevant code examples. Early efforts such as REDCODER (Parvez et al. 2021) retrieved relevant code snippets and supplied them as auxiliary inputs to code generation models. Building on this idea, DocPrompting (Zhou et al. 2022) incorporated retrieved documentation to enhance code generation performance. To improve the structural quality of generated code, SKCoder (Li et al. 2023) retrieved similar codes, extracted key components to form a sketch, and refined the sketch into the final output. Going beyond single-source retrieval, ARKS (Su et al. 2024) constructed a "knowledge soup" by integrating web search results, documentation, and evolved code snippets. Further extending the retrieval paradigm, CAPIR (Ma et al. 2024) used an LLM-based decomposer to break down high-level tasks into subtasks, and applied a retriever to identify relevant APIs for each. Similarly, CONLINE (He et al. 2024) enhanced code generation by combining planned online retrieval with automated testing to enable iterative refinement.

While existing methods have improved retrieval augmentation, they neither judge when retrieval is needed nor ensure alignment between retrieved examples and model preferences. In contrast, SRACG introduces a necessity-aware decision mechanism and multi-dimensional enhancements for more accurate code generation.

Methodology

Given a generation intent x, the goal of code generation is to produce a code solution y that satisfies this intent.[1] Formally, an LLM aims to learn a mapping function $P_\theta(y|x,p)$, where θ denotes model parameters and p represents the prompt guiding the generation. To enhance this process, RAG introduces code examples $R(x) = \{r_1, r_2, \ldots, r_k\}$ retrieved from a codebase $\mathcal{D}$, and generates code via $P_\theta(y|x,p,R(x))$. As shown in Figure 2, SRACG enhances code generation through four strategies: necessity-aware intent selection, multi-objective retrieval, preference consistency filtering, and execution plan extraction. See Appendix A for prompt details.

Necessity-aware intent selection. To avoid unnecessary augmentation, we draw inspiration from CODET (Chen et al. 2022) and introduce a dual-execution evaluation method that uses shared test cases generated by the LLM to determine whether an intent x benefits from retrieval. However, unlike manually crafted private test cases, LLM-generated test cases may suffer from limited coverage due to their inherent randomness. To address this, we design prompts for test case generation that explicitly consider multiple aspects such as functional correctness, boundary conditions, and numerical precision, aiming to ensure broader coverage and higher-quality testing.

Specifically, for each intent x, the LLM is prompted to produce N candidate code solutions $\{y_1, y_2, \ldots, y_N\}$, along with a shared set of N test cases $\{t_1, t_2, \ldots, t_N\}$. Each y is executed against all test cases to evaluate correctness. We identify the set of code solutions S_y that pass exactly the same test cases as y, and denote the passed cases as S_t. Thus, for each x, we can derive multiple execution combinations, each defined as a consensus set $S = \{(y,t) \mid y \in S_y, t \in S_t\}$. For example, if x_1 and x_2 both pass test cases t_2, t_6, and t_8, then the corresponding consensus set consists of $S_x = \{x_1, x_2\}$ and $S_t = \{t_2, t_6, t_8\}$. Similarly, if x_1 and x_3 pass test cases t_3 and t_5, we have another consensus set with $S_x = \{x_1, x_3\}$ and $S_t = \{t_3, t_5\}$. Each consensus set is scored by $f(S) = |S_y| \cdot |S_t|$, where $|S_y|$ denotes the number of code solutions and $|S_t|$ the number of test cases. The highest consensus set score is used as the confidence score of x, denoted as $Conf(x)$.

For $X = \{x_1, x_2, \ldots, x_M\}$, we rank all intents based on their confidence scores $Conf(x)$ in ascending order. We then select the confidence score at the $\frac{1}{\alpha}$-quantile ($\alpha \in \mathbb{Z}^+$) as the threshold. Intents with scores below this threshold indicate a high degree of uncertainty in the LLM's generation and are grouped into the retrieval-required subset X^-, which are expected to benefit from retrieval-augmented generation. The remaining intents form the retrieval-unnecessary subset X^+ and are directly handled by the LLM. This strategy enables targeted augmentation for intents that truly benefit from external guidance, leading to more effective and efficient code generation.

Multi-object retrieval. For each $x^- \in X^-$, we retrieve relevant code examples from $\mathcal{D}$ using a retrieval model E. However, due to differences in retrieval model training strategies and objectives, directly computing similarity between x^- and holistic code examples often fails to capture fine-grained semantic alignment. To address this, we decompose each code example $r_j \in \mathcal{D}$ into three distinct semantic components: function signature r_j^f, code comments r_j^c and overall code content r_j^o. We then compute the semantic similarity between x^- and each component using cosine similarity: $s_j^f = \cos(h_{x^-}, h_j^f)$, $s_j^c = \cos(h_{x^-}, h_j^c)$, $s_j^o = \cos(h_{x^-}, h_j^o)$, where $h_{x^-} = E(x^-)$, $h_j^f = E(r_j^f)$, $h_j^c = E(r_j^c)$ and $h_j^o = E(r_j^o)$.

To ensure a robust match under a single retrieval model, we take the maximum similarity score among the three as the semantic relevance score for example r_j:

$$S_{sem}(x^-, r_j) = \max\{s_j^f, s_j^c, s_j^o\} \tag{1}$$

Finally, we rank all r by their semantic relevance scores with respect to x^-, and select the top-K code examples to form

[1] Following standard evaluation protocols, we generate N candidate solutions per intent to compute Pass@k, which better reflects the model's practical performance.

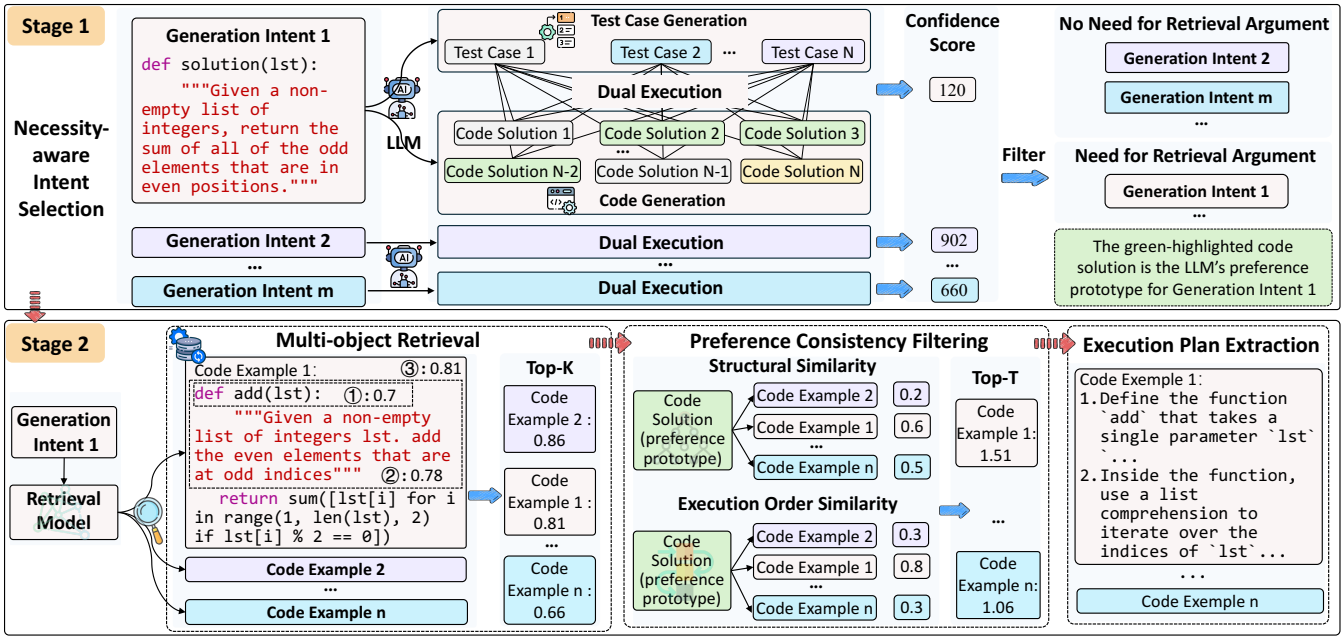

Figure 2: Overview of the SRACG framework.

the initial retrieval pool:

$$R(x^-) = \{r_1, r_2, \ldots, r_K\} \tag{2}$$

This strategy allows us to better utilize a single model's capacity by exploiting complementary semantic cues, leading to a more semantically aligned candidate pool.

Preference consistency filtering. To ensure retrieved examples align with the LLM's generation preferences, we propose a preference consistency strategy. We define "preferences" as a combination of code structure and typical statement execution orders that the LLM tends to produce in zero-shot scenarios. Specifically, for each x^-, we consider its N code solutions $\{y_1, y_2, \ldots, y_N\}$ via zero-shot generation. Among these, the most frequently generated solution reflects the LLM's most probable generation mode under x^-. We therefore designate this solution as the LLM's preference prototype, denoted by y^*. We then compute a preference consistency score to evaluate the alignment between y^* and each candidate $r_j \in R(x^-)$, based on their structural similarity and statement execution order consistency.

Structural similarity. We define a structural feature vector $\phi(\cdot)$ based on syntactic and structural patterns extracted from the code's Abstract Syntax Tree (AST). Specifically, we consider the key set of 24 syntactic node types (the full list is provided in Appendix B):

$$\mathcal{T} = \{call, dictionary, \ldots, lambda\} \tag{3}$$

as well as four global metrics: maximum AST depth, and the maximum nesting depths of if, for, and $while$ constructs.

Each code c is transformed into a fixed-length feature vector $\phi(c) \in \mathbb{R}^{28}$, where the first 24 dimensions correspond to counts of node types in $\mathcal{T}$, and the remaining 4 dimensions represent depth statistics. The structural similarity score is then computed via cosine similarity:

$$S_{struct}(y^*, r_j) = \frac{\phi(y^*)^\top \phi(r_j)}{\|\phi(y^*)\|_2 \cdot \|\phi(r_j)\|_2} \tag{4}$$

where $\|\cdot\|_2$ is the Euclidean norm. This score quantifies the structural alignment between the model's preferred generation pattern and the retrieved example, promoting those that better conform to the LLM's internal preferences.

Execution order similarity. To capture consistency in execution semantics, we extract the sequence of high-level executable statements from each code sample, focusing on: *assignment, for, if, while, return, expression*. Each code sample is then mapped to a symbolic sequence (e.g., $[A, F, I, R]$), representing the order of these statement types. We compute the normalized Longest Common Subsequence (LCS) length to quantify execution order similarity:

$$S_{order}(y^*, r_j) = \frac{LCS(s(y^*), s(r_j))}{\max(|s(y^*)|, |s(r_j)|)} \tag{5}$$

where $s(\cdot)$ denotes the statement sequence extracted from a code. The two scores are combined into a unified preference score via weighted summation:

$$S_{pref}(y^*, r_j) = \frac{1}{2}\left(S_{struct}(y^*, r_j) + S_{order}(y^*, r_j)\right) \tag{6}$$

This preference score is then combined with the original semantic relevance score $S_{sem}(x^-, r_j)$ from the multi-objective retrieval stage:

$$S_{final}(x^-, r_j) = S_{sem}(x^-, r_j) + S_{pref}(y^*, r_j) \tag{7}$$

Top-T examples with the highest S_{final} scores are selected as the final retrieval set for x^-, ensuring both semantic relevance and consistency with the LLM's preferences.

This set is denoted as $R(x^-) = \{r_1, r_2, \ldots, r_T\}$.

Execution plan extraction. In contrast to natural language passages, which often contain directly usable information, code tends to be structurally complex and logically dense. As a result, LLMs may struggle to capture the underlying implementation logic from retrieved code samples, often resorting to shallow pattern copying that leads to incorrect or suboptimal outputs. To address this, rather than directly appending the top-T retrieved code examples to the prompt, we extract an execution plan z for each example, which guides the LLM to attend to the code's underlying implementation logic. Each z is generated by prompting the LLM to produce a step-by-step explanation of the code's behavior, grounded in its corresponding primary inline comments.

Consequently, for each x^-, the LLM regenerates N candidate code solutions $\{y'_1, y'_2, \ldots, y'_N\}$ based on the prompt p, the retrieved examples $R(x^-)$, and their execution plans $Z(x^-) = \{z_1, z_2, \ldots, z_T\}$, following the distribution: $P_\theta(y \mid x^-, p, R(x^-), Z(x^-))$.

Experiment Setting

Benchmarks. We conduct experiments on two fundamental programming problem datasets, HumanEval+ and MBPP+ (Liu et al. 2023), and one competitive programming dataset, CodeContests (Li et al. 2022). These datasets are commonly adopted in prior work and serve as established standards for assessing the performance of code generation methods.

Metrics. We use Pass@k (Chen et al. 2021) to evaluate performance, leveraging ground-truth test cases to assess the functional correctness of code solutions.

Baselines. Our experiments are based on the following models: CodeGen-Mono-16B (Li et al. 2022), LLaMA-3.3-70B-Instruct (Dubey et al. 2024), GPT-3.5-Turbo, GPT-4o-Mini (Achiam et al. 2023), DeepSeek-V3 (Liu et al. 2024) , Gemini-Flash-1.5 (Team et al. 2024) and Qwen-Turbo (Bai et al. 2023). We also compare SRACG with several state-of-the-art code generation methods: CODET(Chen et al. 2022), RAT(Wang et al. 2024), and CAPIR(Ma et al. 2024).

Retrieval model. For retrieval, we adopt UniXcoder (Guo et al. 2022) as the retrieval model. We fine-tune it on the Python subset of the CodeSearchNet (Husain et al. 2019) by formulating a multi-objective retrieval task.

Codebase. We merge the two fundamental problem datasets to form the codebase for HumanEval+ and MBPP+, and combine CodeContests with APPS (Hendrycks et al. 2021) to build the codebase for CodeContests. During retrieval, we exclude ground-truth solutions to avoid data leakage. We extract execution plans from the codebase using GPT-4o-Mini in an offline manner to minimize runtime overhead.

Implementation Details. For each intent, we generate 100 candidate code solutions along with a shared set of test cases, using a temperature of 0.8, a top-p value of 0.95, and a maximum output length of 768 tokens. The confidence threshold α is set to 3. For retrieval, the top 5 candidates (K = 5) are considered, and the highest-ranked example (T = 1) is selected for augmentation. More implementation details can be found in Appendix C.

Experiment Results

Effectiveness of SRACG. To evaluate the effectiveness of SRACG, we evaluate seven widely-used LLMs on three benchmarks. As shown in Table 1, we compare each model's vanilla performance and its SRACG-enhanced variant using Pass@k (k = 1, 3, 5) as the metric.

Across all models and datasets, SRACG consistently improves generation performance. On the HumanEval+, SRACG yields notable improvements, with gains of over 7 points across all metrics for GPT-3.5 and over 5.5 points for Qwen. Even strong models like DeepSeek benefit consistently, indicating SRACG's ability to enhance generation even on high-performing bases. On MBPP+, SRACG continues to improve all models, achieving +5.62 Pass@1 for DeepSeek and up to +4.79 Pass@3 for GPT-3.5. Models such as Gemini and LLaMA also show stable gains. Although CodeContests poses greater challenges due to its competitive nature and complex reasoning demands, SRACG still achieves notable relative improvements, including +4.45 Pass@5 for LLaMA and +4.10 Pass@3 for Gemini. Even the weakest model, CodeGen, exhibits smaller but consistent gains. These results demonstrate SRACG's broad effectiveness. By selectively enhancing low-confidence cases, it significantly boosts code generation performance in both average and challenging scenarios.

Comparison with other code generation methods. We compare SRACG with several representative code generation baselines. As shown in Table 2, SRACG consistently outperforms all competing methods across all metrics.

RACG represents a standard RAG pipeline that directly incorporates the top-1 retrieved example into the prompt. Its performance lags behind DIRECT, highlighting the potential harm of indiscriminate augmentation. CODET introduces a sample mechanism based on execution results. SRACG builds upon this method by enhancing test case diversity and selection quality, leading to superior performance. Compared to CODET, SRACG achieves +4.74 Pass@1 on GPT-3.5 and +2.72 on DeepSeek, demonstrating the advantage of more robust confidence estimation. RAT improves generation through iterative retrieval-enhanced reasoning. Although effective, it mainly focuses on long-range dependency tasks. In contrast, SRACG focuses on generation reliability through necessity-aware augmentation and preference alignment, outperforming RAT by +3.49 Pass@3 on GPT-3.5 and +2.30 Pass@1 on DeepSeek. CAPIR enhances generation by decomposing coarse-grained descriptions into sub-tasks and retrieving API-level references. However, it lacks fine-grained control over sample-specific retrieval necessity and example compatibility. SRACG addresses these gaps, outperforming CAPIR by +5.25 Pass@1 on GPT-3.5 and +4.75 on DeepSeek. These results validate the effectiveness of SRACG in addressing the key challenges of retrieval-augmented code generation.

Ablation study. To evaluate the contribution of each component in SRACG, we perform an ablation study by removing four key strategies: necessity-aware intent selection, multi-objective retrieval, preference consistency filtering, and execution plan extraction.

As shown in Table 3, removing any single component

Model	HumanEval+			MBPP+			CodeContests		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
CodeGen-Mono-16B	21.01	23.64	24.33	35.86	36.62	37.40	0.33	0.33	0.33
CodeGen-Mono-16B (+*SRACG*)	25.26$^{+4.25}$	27.32$^{+3.68}$	29.44$^{+5.11}$	38.21$^{+2.35}$	39.06$^{+2.44}$	39.98$^{+2.58}$	0.58$^{+0.25}$	0.60$^{+0.27}$	0.60$^{+0.27}$
LLaMA-3.3-70B	70.11	75.63	77.46	61.78	66.17	67.55	20.55	28.42	31.31
LLaMA-3.3-70B (+*SRACG*)	73.85$^{+3.74}$	80.38$^{+4.75}$	81.19$^{+3.73}$	65.72$^{+3.94}$	70.54$^{+4.37}$	71.70$^{+4.15}$	23.86$^{+3.31}$	31.04$^{+2.62}$	35.76$^{+4.45}$
GPT-3.5-Turbo	59.82	67.00	67.93	60.49	66.34	67.97	2.39	4.08	4.99
GPT-3.5-Turbo (+*SRACG*)	66.88$^{+7.06}$	74.49$^{+7.49}$	76.02$^{+8.09}$	64.40$^{+3.91}$	71.13$^{+4.79}$	71.56$^{+4.01}$	5.61$^{+3.22}$	6.22$^{+2.14}$	8.83$^{+3.84}$
GPT-4o-Mini	73.41	78.22	79.56	64.59	67.55	68.56	7.93	12.40	14.32
GPT-4o-Mini (+*SRACG*)	77.79$^{+4.38}$	81.95$^{+3.73}$	82.89$^{+3.33}$	67.74$^{+3.15}$	71.10$^{+3.55}$	71.69$^{+3.13}$	10.25$^{+2.32}$	15.89$^{+3.49}$	18.09$^{+3.77}$
DeepSeek-V3	82.80	85.80	86.38	78.03	81.59	82.71	7.47	15.14	19.33
DeepSeek-V3 (+*SRACG*)	85.97$^{+3.17}$	88.04$^{+2.24}$	88.92$^{+2.54}$	83.65$^{+5.62}$	86.77$^{+5.18}$	87.67$^{+4.96}$	11.21$^{+3.74}$	18.33$^{+3.17}$	22.05$^{+2.72}$
Gemini-Flash-1.5	70.46	72.66	73.37	63.02	65.26	66.00	10.18	13.09	14.20
Gemini-Flash-1.5 (+*SRACG*)	72.86$^{+2.40}$	75.52$^{+2.86}$	75.52$^{+2.15}$	66.90$^{+3.88}$	68.75$^{+3.49}$	69.16$^{+3.16}$	13.96$^{+3.78}$	17.19$^{+4.10}$	17.31$^{+3.11}$
Qwen-Turbo	76.83	81.47	83.01	64.71	66.98	67.73	2.99	5.45	6.78
Qwen-Turbo (+*SRACG*)	82.59$^{+5.76}$	88.51$^{+7.04}$	88.75$^{+5.74}$	68.59$^{+3.88}$	71.79$^{+4.81}$	72.33$^{+4.60}$	5.64$^{+2.65}$	8.38$^{+2.93}$	9.17$^{+2.39}$

Table 1: Effectiveness of SRACG on different LLMs.

Method	GPT-3.5-Turbo			DeepSeek-V3		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
DIRECT	59.82	67.00	67.93	82.80	85.80	86.38
RACG	56.21	65.08	65.33	80.22	83.22	85.19
CODET	62.14	69.89	70.53	83.25	86.81	86.88
RAT	64.40	71.00	72.86	83.67	85.99	87.98
CAPIR	61.63	66.01	69.60	81.22	86.01	86.55
SRACG	**66.88**	**74.49**	**76.02**	**85.97**	**88.04**	**88.92**

Table 2: Comparison of SRACG and code generation methods on HumanEval+ using GPT-3.5 and DeepSeek-V3. "DIRECT" denotes generation without retrieval augmentation.

Method	GPT-3.5-Turbo			DeepSeek-V3		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
w/o N	62.37	69.62	70.09	82.11	86.64	87.52
w/o M	63.71	69.59	72.66	83.06	87.12	87.54
w/o P	63.50	71.17	73.03	83.11	86.95	87.07
w/o E	64.41	72.02	74.96	83.91	87.02	87.56
SRACG	**66.88**	**74.49**	**76.02**	**85.97**	**88.04**	**88.92**

Table 3: Ablation study of SRACG components on HumanEval+ using GPT-3.5-Turbo and DeepSeek-V3.

leads to performance degradation across both GPT-3.5-Turbo and DeepSeek-V3, confirming their complementary roles. Notably, removing the necessity-aware intent selection (w/o N) results in the largest performance drop. This confirms our motivation that indiscriminate augmentation can introduce noise and harm generation. The removal of multi-objective retrieval (w/o M) and preference filtering (w/o P) also leads to consistent declines. This suggests that aligning retrieved examples with the LLM's intrinsic generation preferences helps avoid confusion during generation. Excluding execution plan extraction (w/o E) leads to the smallest performance drop, but still highlights that deeper implementation logic improves generation fidelity and reduces surface-level mimicry. These findings support the overall design of SRACG and its effectiveness in addressing the key challenges of retrieval-augmented code generation.

Deeper Analysis

Effect of confidence threshold in necessity-aware intent selection. To investigate how the threshold α affects the overall effectiveness of SRACG, we evaluate the performance of several models across a range of α values on the HumanEval+ and MBPP+ datasets. As shown in Figure 3, most models on both HumanEval+ and MBPP+ achieve optimal or near-optimal performance when α = 3. On HumanEval+, models like GPT-4o-Mini, and Qwen exhibit noticeable performance drops as α increases beyond 3, indicating that too few samples are being selected for augmentation. On MBPP+, similar trends are observed, though some models like CodeGen and GPT-3.5 remain relatively stable over a broader range of α. These results confirm the importance of carefully balancing augmentation coverage and selectivity: moderate values of α yield the best trade-off between enhancing uncertain cases and avoiding unnecessary intervention. This reinforces the utility of SRACG's necessity-aware selection mechanism and highlights its adaptability across diverse models and datasets.

Impact of retrieval quantity on SRACG performance. To assess how the number of retrieved examples T influences the performance of SRACG, we evaluate settings where T ranges from 1 to 5 on two datasets: MBPP+ and CodeContests. Pass@1 is used as the primary evaluation metric. As shown in Figure 4, most models on both MBPP+ and CodeContests achieve their highest performance when T =

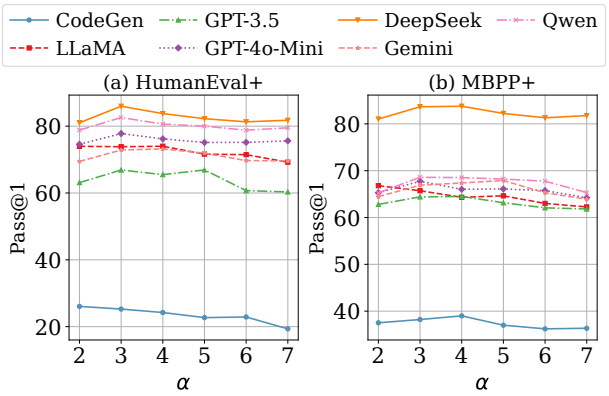

Figure 3: Effect of α on SRACG performance.

1. Notably, Qwen maintains relatively stable performance across different values of T, indicating stronger robustness. These findings highlight an important insight: SRACG benefits most from high-quality, single-example guidance rather than from multi-example augmentation. Increasing T does not inherently guarantee better performance and can even be detrimental if low-quality examples are included.

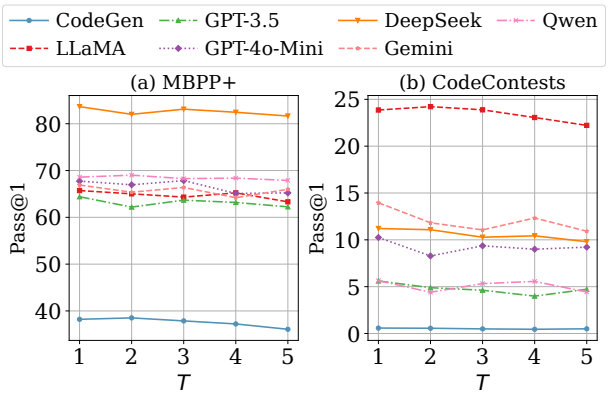

Figure 4: Effect of T on SRACG performance.

Robustness of SRACG across retriever models. To examine how different retrievers influence the effectiveness of SRACG, we evaluate three retrieval models: CodeBERT(Feng et al. 2020), GraphCodeBERT(Guo et al. 2020), and UniXcoder, on the MBPP+ dataset using four distinct code generation backbones. All three retrievers are fine-tuned using the same procedure to ensure a fair comparison. The results in Figure 5 show that SRACG consistently improves performance across all generation models, regardless of the retriever used. This demonstrates that SRACG is not highly dependent on a specific retriever and can be effectively integrated with various retrieval models, making it an adaptable augmentation framework.

Analyzing outcome shifts induced by SRACG. To further evaluate SRACG's impact, we analyze execution outcome shifts on the HumanEval+ dataset, where a task is deemed correct only if its pass@1 is 1.0. Figure 6 displays the results

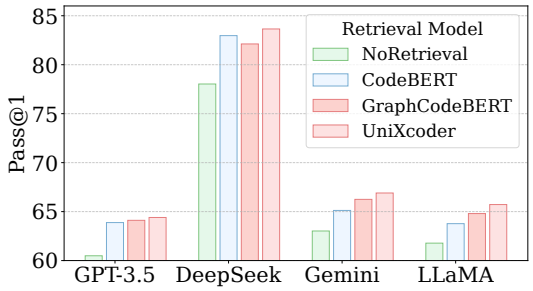

Figure 5: Performance comparison of SRACG with different retriever models on MBPP+.

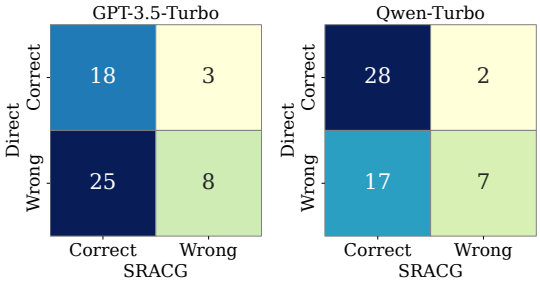

Figure 6: Outcome shifts induced by SRACG.

for GPT-3.5-Turbo and Qwen-Turbo. For GPT-3.5-Turbo, among the 54 samples requiring enhancement, 18 remained correct after SRACG, 3 were degraded, and 25 originally incorrect samples were corrected, leaving only 8 still incorrect. In comparison, Qwen-Turbo shows even greater robustness: 28 remained correct, 2 were degraded, 17 were corrected, and 7 remained incorrect. These results show that SRACG not only recovers many failed cases but also preserves original correctness. More importantly, it reveals SRACG's ability to unlock the base model's latent potential by guiding it toward solutions it previously failed to generate.

Case Study. Four cases are discussed in Appendix D.

Conclusion

This paper introduces SRACG, a selective retrieval-augmented code generation framework designed to enhance LLM performance through effective use of retrieval resources. By incorporating a necessity-aware selection mechanism and multi-dimensional enhancement strategies, SRACG effectively addresses key challenges in retrieval-augmented code generation, including unnecessary augmentation, retrieval misalignment, and shallow imitation. Extensive experiments across multiple benchmarks demonstrate that SRACG consistently improves code generation performance. These results underscore the value of selective and high-quality guidance, showing that retrieval augmentation, when applied judiciously, can significantly enhance the accuracy of LLMs in code generation.

Acknowledgments

This research is supported by the National Natural Science Foundation of China (62476097), the Fundamental Research Funds for the Central Universities, South China University of Technology (x2rjD2250190), Guangdong Provincial Fund for Basic and Applied Basic Research—Regional Joint Fund Project (Key Project) (2023B1515120078), Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (2024B1515040010), the National Natural Science Foundation of China (62402185), the Fundamental Research Funds for the Central Universities(2025ZYGXZR064).

References

Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic, A. 2024. Introducing the next generation of claude.

Asai, A.; Wu, Z.; Wang, Y.; Sil, A.; and Hajishirzi, H. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Chen, B.; Zhang, F.; Nguyen, A.; Zan, D.; Lin, Z.; Lou, J.-G.; and Chen, W. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Chen, J.; Hu, X.; Li, Z.; Gao, C.; Xia, X.; and Lo, D. 2024. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. D. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Dong, Y.; Ding, J.; Jiang, X.; Li, G.; Li, Z.; and Jin, Z. 2025. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(3): 1–22.

Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; et al. 2024. The llama 3 herd of models. *arXiv e-prints*, arXiv–2407.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Gao, Y.; Xiong, Y.; Gao, X.; Jia, K.; Pan, J.; Bi, Y.; Dai, Y.; Sun, J.; Wang, H.; and Wang, H. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1).

Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; and Yin, J. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

He, X.; Zou, J.; Lin, Y.; Zhou, M.; Han, S.; Yuan, Z.; and Zhang, D. 2024. Conline: Complex code generation and refinement with online searching and correctness testing. *CoRR*.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Husain, H.; Wu, H.-H.; Gazit, T.; Allamanis, M.; and Brockschmidt, M. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Li, J.; Li, Y.; Li, G.; Jin, Z.; Hao, Y.; and Hu, X. 2023. Skcoder: A sketch-based approach for automatic code generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2124–2135. IEEE.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36: 21558–21572.

Ma, Z.; An, S.; Xie, B.; and Lin, Z. 2024. Compositional API recommendation for library-oriented code generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 87–98.

Nashid, N.; Sintaha, M.; and Mesbah, A. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2450–2462. IEEE.

Parvez, M. R.; Ahmad, W. U.; Chakraborty, S.; Ray, B.; and Chang, K.-W. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Su, H.; Jiang, S.; Lai, Y.; Wu, H.; Shi, B.; Liu, C.; Liu, Q.; and Yu, T. 2024. Arks: Active retrieval in knowledge soup for code generation. *arXiv preprint arXiv:2402.12317*.

Team, G.; Mesnard, T.; Hardin, C.; Dadashi, R.; Bhupatiraju, S.; Pathak, S.; Sifre, L.; Rivière, M.; Kale, M. S.; Love, J.; et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*.

Wang, M.; Huang, X.; Xie, J.; Ma, S.; Men, J.; Liang, D.; and Cai, Y. 2025. From Model Diagram to Code: A Benchmark Dataset and Multi-Agent Framework. In *Proceedings of the 33rd ACM International Conference on Multimedia*, 1754–1763.

Wang, Z.; Liu, A.; Lin, H.; Li, J.; Ma, X.; and Liang, Y. 2024. Rat: Retrieval augmented thoughts elicit context-aware reasoning in long-horizon generation. *arXiv preprint arXiv:2403.05313*.

Yang, Z.; Liu, F.; Yu, Z.; Keung, J. W.; Li, J.; Liu, S.; Hong, Y.; Ma, X.; Jin, Z.; and Li, G. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE): 1585–1608.

Yuan, L.; Cai, Y.; Wang, J.; and Li, Q. 2023. Joint multimodal entity-relation extraction based on edge-enhanced graph alignment network and word-pair relation tagging. In *Proceedings of the AAAI conference on artificial intelligence*, volume 37, 11051–11059.

Zhou, S.; Alon, U.; Xu, F. F.; Wang, Z.; Jiang, Z.; and Neubig, G. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv: 2207.05987*.