EvoR: Evolving Retrieval for Code Generation

Hongjin Su[1], Shuyang Jiang[2], Yuhang Lai[2],
Haoyuan Wu[1], Boao Shi[1], Che Liu[1], Qian Liu[3], Tao Yu [1],

[1]The University of Hong Kong, [2]Fudan University, [3]Sea AI Lab,
Correspondence: hjsu@cs.hku.hk

Abstract

Recently the retrieval-augmented generation (RAG) has been successfully applied in code generation. However, existing pipelines for retrieval-augmented code generation (RACG) employ static knowledge bases with a single source, limiting the adaptation capabilities of Large Language Models (LLMs) to domains they have insufficient knowledge of. In this work, we develop a novel pipeline, EvoR, that employs the synchronous evolution of both queries and diverse knowledge bases. On two realistic settings where the external knowledge is required to solve code generation tasks, we compile four new datasets associated with frequently updated libraries and long-tail programming languages, named EvoR-bench. Extensive experiments demonstrate that EvoR achieves two to four times of execution accuracy compared to other methods such as Reflexion (Shinn et al., 2024), DocPrompting (Zhou et al., 2023), etc. We demonstrate that EvoR is flexible and can be easily combined with them to achieve further improvement. Further analysis reveals that EvoR benefits from the synchronous evolution of queries and documents and the diverse information sources in the knowledge base. We hope that our studies will inspire more insights into the design of advanced RACG pipelines in future research.

1 Introduction

The retrieval-augmented generation (RAG) paradigm has raised significant attention due to its efficiency in adapting large language models (LLMs) without training (Guu et al., 2020; Karpukhin et al., 2020; Izacard et al., 2023; Borgeaud et al., 2022; Asai et al., 2023). Recent research has demonstrated its successful applications in code generation. They implement the retrieval-augmented code generation (RACG) pipelines either using a given query (Parvez et al., 2021b), or a rewritten version (Jiang et al.,
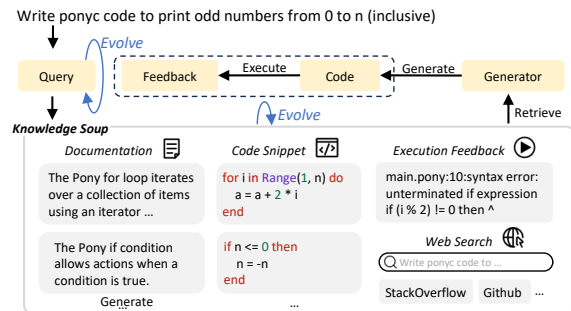

Figure 1: Instead of using a given query to retrieve from a static knowledge base, we design a novel pipeline to dynamically evolve both queries and knowledge soup in retrieval-augmented code generation.

2023b) to retrieve from a static knowledge base with a single type of information, e.g., syntax documentation (Zan et al., 2022; Zhou et al., 2023), code repositories (Zhang et al., 2023a; Shrivastava et al., 2023), etc.

However, more knowledge sources are potentially helpful to generalization, e.g., web content (Parvez et al., 2021a; Wang et al., 2022), code snippets generated by LLM (Zhang et al., 2023b), etc. This information is easily obtained and can enrich knowledge bases, which are shared among all instances of the same task. Furthermore, the unique characteristic of execution in code generation enables more information collected on-the-fly. For instance, if a code snippet generated by LLMs is successfully executed without reporting error messages, it is guaranteed to be syntactically correct and can serve as a concrete example to demonstrate the corresponding grammar or function usage.

In this work, we introduce EvoR, a novel pipeline that applies synchronous evolution of both queries and documents in RACG. In the traces of multi-round interactions among retrievers, LLMs and executors, both queries and knowledge bases are updated based on the execution feedback and LLM outputs in every iteration. This strategic re-

finement aims to facilitate the extraction of the most pertinent information. Apart from the given library documentation, we construct a diverse knowledge soup to further integrate the web search content, execution feedback, and code snippets generated by LLMs in the inference time.

To prevent the issue of data leakage associated with large language models pretrained on massive public datasets, and assess EVOR under a reliable generalization setting, we compile a new benchmark, EVOR-BENCH, comprising four datasets designed to simulate realistic scenarios in RACG. Specifically, two of these datasets focus on modifications made to widely-used Python libraries, Scipy and Tensorflow. The remaining two datasets simulate the introduction of new grammars, with the help of two less-common programming languages Ring and Pony. To conduct thorough experiments, we employ both proprietary models, such as ChatGPT (OpenAI, 2022), and open-source models like CodeLlama (Roziere et al., 2023). Experimental results across these four datasets demonstrate that our method yields a significant improvement in the average performance over existing code generation methods. For example, EVOR outperforms DocPrompting (Zhou et al., 2023) by 18.6% on average using CodeLlama (§3). Further analysis unveils that both synchronous evolution and diverse sources in knowledge bases are critical to the success of EVOR (§4.1, §4.2). We demonstrate that EVOR is flexible to integrate with many other code generation approaches including the agent-based one, e.g., swe-agent, offering further performance enhancement in both EVOR-BENCH and existing benchmarks (§4.3). Finally, we showcase EVOR is a more effective approach to using tokens, and demonstrates superior results in all levels of token consumption ranging from 4k to 24k (§4.4). In summary, our contributions are:

- We propose a novel pipeline, EVOR, highlighting the complementary strength of synchronous evolution of queries and diverse knowledge bases in RACG.

- We compile a new benchmark, EVOR-BENCH, on two realistic RACG settings related to frequently updated libraries and long-tail programming languages.

- We conduct extensive analyses and find that EVOR can be easily combined with existing code generation approaches including agent-based ones to provide further improvements.

2 Evolving Retrieval

Given a question n in natural language, the objective of retrieval-augmented code generation is to first retrieve relevant information K^+ from external knowledge K and then augment large language models to generate a program p in the target library/programming language, which LLM M is not familiar with. Distinct from the classical retrieval-augmented generation, which usually focuses on static knowledge bases, we propose synchronous evolution of both queries and diverse knowledge bases. Intuitively, this helps the retrieval model identify more relevant information and thus improves the quality of LLM generation (Shao et al., 2023). In this section, we present the process of query evolution (§2.1), the knowledge base construction and evolution (§2.2), the EVOR pipeline (§2.3) and the collection of EVOR-BENCH (§2.4).

2.1 Query evolution

Starting from the given question n, we first go through a warmup iteration i_0 where $q_0 = n$ is used as the query in retrieval. Conditioned on both n and the retrieved knowledge K_r, LLM M then generates a draft program p^0. We apply the a compiler or interpreter to execute p^0 on LLM-generated inputs $I = [i_1, i_2, ..., i_n]$ (more details on Appendix C), which provides execution feedback $F^0 = [f_1^0, f_2^0, ..., f_n^0]$. Based on n, p^0, I, and F^0, we prompt an LLM M_q to write a new query q_1 on what knowledge is currently required. In general, given n, p^i, I, and F^i in the iteration i, M_q writes q_{i+1}, which is used for retrieval in the iteration $i + 1$.

2.2 Knowledge Soup

In this section, we first introduce the four components included in the construction of the knowledge soup K and then describe the process of its evolution.

2.2.1 Construction

We consider four types of knowledge as follows:

Web search is a general and popular resource applied in traditional RAG applications. Human programmers frequently refer to it when they try

to understand some syntax or fix a bug. It contains diverse information including blogs, tutorials, and community Q&A discussions relevant to solving coding problems. Intuitively, it is valuable as human programmers frequently rely on search engines to seek assistance when struggling with coding challenges. Previous work (Nakano et al., 2021) has fine-tuned GPT-3 (Brown et al., 2020) to answer long-form questions using a text-based web-browsing environment. In this study, we investigate the efficacy of LLMs in utilizing web search content to solve unfamiliar coding problems without further training. We use the Python API of Google search [1] to retrieve top-ranking websites and further convert the HTML page to markdown using the package html2text [2]. In Appendix G, we include more discussions about the content in the web search.

Documentation is commonly accessible upon the release of a new programming language or an updated library version. Official documentation serves to thoroughly elucidate the essential syntax and grammar required for coding. Zhou et al. (2022) demonstrated that language models can effectively leverage code documentation after fine-tuning. In this work, we focus on understanding the capability of LLMs in utilizing the documentation of updated libraries or long-tail programming languages in code generation, without making any parameter update.

Execution feedback is a specific knowledge type for code generation. It exposes syntax mistakes and locates code errors, which are frequently referenced by human programmers to debug. While multiple types of execution can provide feedback (e.g., execution by LLMs), we focus on the compiler or interpreter execution in this work. Previous works (Shinn et al., 2024) have demonstrated that LLMs are capable of repairing buggy code using the execution feedback. Instead of only leveraging the error messages obtained from executing the generated faulty programs, we further enrich the knowledge base by preparing sample code-error pairs. More details can be found in Appendix D.

Code snippets are the short pieces of code that demonstrate sample usage of certain functions or syntax. Different from other types of knowledge that involve natural language, code snippets in pro-

---

Algorithm 1 EVOR Pipeline

1: **Input:** n: the coding problem description; M: the LLM to generate the code answer; M_q: the LLM to evolve queries; M_t: the LLM to generate test inputs; R: the retriever to output a list of relevant passages; K: the knowledge base; m: the maximum number of iterations; E: the compiler or interpreter to execute programs
2: **Initialization:** $I = [], p = null$.
3: **for** $i = 0, \ldots, m$ **do**
4:     **if** $i = 0$ **then**
5:         $q_i \leftarrow n$
6:     **else**
7:         $q_i \leftarrow M_q(n, p^{i-1}, I, F^{i-1})$ ▷ Evolve query
8:     **end if**
9:     $K_r \leftarrow R(q_i, K)$ ▷ Retrieve relevant knowledge
10:     $p^i \leftarrow M(n, K_r)$ ▷ Generate program
11:     $p \leftarrow p^i$
12:     $F^i = E(p^i, I)$ ▷ Execute and get feedback
13:     **if** F^i is sucess **then**
14:         $K \leftarrow K \cup \{p^i\}$ ▷ Evolve knowledge base
15:     **else**
16:         $K \leftarrow K \cup \{(p^i, F^i)\}$ ▷ Evolve knowledge base
17:     **end if**
18:     **if** $i = 0$ **then**
19:         $I \leftarrow M_t(n, p^i)$ ▷ Generate test inputs
20:     **end if**
21:     **if** terminate condition is satisfied **then**
22:         **break**
23:     **end if**
24: **end for**
25: **Return:** p: output code.

---

gramming language naturally align with the LLM generation objective and provide concrete examples of inputs, outputs, and parameters. Furthermore, they serve as a means to convey information about the programming language itself, providing crucial details such as bracket placement, utilization of special tokens, and other grammar. Before evaluation, we collect a set of code snippets verified to be free of syntax errors (more details in Appendix D). Additionally, we also accumulate code solutions generated by LLMs.

2.2.2 Evolution

The evolution of knowledge bases is primarily contributed by the execution feedback and code snippets. In each iteration, we execute the generated program with sample inputs (Appendix C). If the execution successfully exits, we classify the code snippet as "syntax-correct", which can serve as a demonstration for other instances to refer to. Otherwise, we add the (`code snippet, error messages`) pair to the knowledge base. Throughout the process of iterative generation, the knowledge base evolves to include increasingly rich information.

2.3 EVOR Pipeline

Algorithm 1 demonstrates the EVOR pipeline. In every iteration i, we first formulate the query q_i (§2.1), and use it to retrieve relevant information K_r from the knowledge base K. The program p^i is then generated conditioned on both n and K_r. We get the execution feedback F^i by executing p^i on LLM-generated test inputs I using the compiler or interpreter E. The knowledge base K is then evolved to include p^i if the execution is successful, or the pair of (p^i, F^i) otherwise. The pipeline exits either upon reaching the termination condition or the maximum iteration steps. In the experiments, we set the maximum iterations to 30 and the termination condition to be the same execution feedback in consecutive 3 iterations, i.e., the algorithm exits if the program is successfully executed or results in the same error in consecutive 3 iterations.

2.4 Datasets

Since LLMs are extensively trained on public data, we curate a new benchmark to evaluate their generalization capability with EVOR. Specifically, we introduce four datasets where two focus on updated libraries and two are about long-tail programming languages.

We first modified two popular Python libraries, Scipy and Tensorflow, to simulate the real updates [3], and denote them as Scipy-M and Tensorflow-M respectively. We then collect problems of the Scipy and Tensorflow split from DS-1000 (Lai et al., 2023) and adapt them to our modified version. For the long-tail programming languages, we select Ring and Pony. They have little public data and are excluded from the StarCoder training set, which involves 88 mainstream programming languages (Li et al., 2023b). We make use of the problems in LeetCode [4] for these two datasets. For each problem in modified libraries or long-tail programming languages, we manually write the ground truth solution and annotate the oracle documentation based on it. We present the dataset statistics in Table 1. More details about our curation process can be found in Appendix A.

3 Experiment

To verify the effectiveness of EVOR, we conduct extensive experiments with both the proprietary

Dataset	# P	# D	A.T	A.P.L	A.S.L	A.D.L
Scipy-M	142	3920	3.1	322.6	44.1	499.7
Tensor-M	45	5754	4.1	234.5	39.0	517.6
Ring	107	577	18.2	108.3	98.3	334.0
Pony	113	583	18.4	116.9	129.8	3204.0

Table 1: Data statistics of four benchmarks. We report the number of problems (# P), the number of official documentation files (# D), the average number of test cases (A.T), the average problem length (A.P.L), the average solution length (A.S.L) and the average gold documentation length (A.D.L). Tensor-M refers to Tensorflow-M . Problem length, solution length and document length are calculated by the tiktoken (https://pypi.org/project/tiktoken/) package with model gpt-3.5-turbo-1106.

model ChatGPT (gpt-3.5-turbo-1106[5]) and the open-source model CodeLlama [6]. In §3.1, we describe 5 baseline settings of other code generation approaches and specify the default configuration of EVOR in §3.2. In §3.3, we compare the results of EVOR, existing code generation methods, as well as their combinations. By default, we use the execution accuracy (pass@1) as the metric throughout the paper.

3.1 Baselines

We compare EVOR to the vanilla generation and four recent methods that demonstrate significant performance improvement in code generation tasks:

Vanilla: we implement the vanilla generation baseline where we directly get the outputs from LLMs based on the coding question n without augmenting external knowledge.

MPSC: Huang et al. (2023) proposed Multi-Perspective Self-Consistency (MPSC) incorporating both inter- and intra consistency. Following the original implementation, we prompt LLMs to generate diverse outputs from three perspectives: Solution, Specification and Test case, construct the 3-partite graph, and pick the optimal choice of solutions based on confidence scores.

ExeDec: Shi et al. (2023a) introduced a decomposition-based synthesis strategy, where they employ a subgoal model to predict the subgoal of the desired program state for the next part of the program and use another synthesizer model to generate the corresponding subprogram to achieve that

---

[3]We do not use a real library update version because it is potentially exposed to LLM training data, which deviates from our purpose to evaluate LLMs' generalization ability.

[4]https://leetcode.com/problemset/

[5]https://platform.openai.com/docs/models/gpt-3-5-turbo

[6]https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf

Method	Model: ChatGPT					Model: CodeLlama				
	Scipy-M	Tensor-M	Ring	Pony	Avg.	Scipy-M	Tensor-M	Ring	Pony	Avg.
Baseline										
Vanilla	17.6	11.1	3.7	1.8	8.6	11.3	17.8	0.0	0.0	7.3
MPSC	18.3	11.1	4.1	1.8	8.8	11.6	17.8	0.0	0.0	7.4
ExeDec	22.5	17.8	4.5	3.6	12.1	13.2	17.8	0.0	0.0	7.8
Reflexion	23.2	22.2	5.3	4.7	13.9	14.5	20.0	0.0	0.9	8.9
DocPrompting	32.4	33.3	8.4	2.7	19.2	16.9	37.8	4.7	4.4	16.0
Ours										
EVOR	37.9	53.3	36.6	13.5	35.3	31.2	53.3	26.7	17.4	32.2
EVOR + MPSC	38.6	55.6	37.8	15.6	36.9	33.6	55.6	27.3	18.4	33.7
EVOR + ExeDec	39.2	55.6	40.0	16.3	37.8	34.1	57.8	27.9	19.1	34.7
EVOR + Reflexion	39.4	55.6	39.2	17.3	37.9	35.3	55.6	28.6	18.8	34.6

Table 2: The performance of baseline methods, EVOR and their combinations in EVOR-BENCH. EVOR demonstrates significantly superior results, with further improvement when combined with other baseline methods.

subgoal. Subprograms are finally combined as the output answer to solve the original coding problem. In experiments, we use ChatGPT as the subgoal model and compare LLMs to synthesize programs following the subgoal predictions.

Reflexion: Shinn et al. (2024) uses a framework to reinforce LLMs through linguistic feedback. It employs an iterative optimization process. In each iteration, the actor model produces a trajectory conditioned on the instructions and memories. The evaluator model then evaluates the trajectory and calculates a scalar reward. Self-reflection model generates verbal experience feedback on the pairs of trajectories and rewards, which are stored in the memory. Throughout experiments, we use the compiler or interpreter as the evaluator model, which returns 0 upon execution errors, and 1 otherwise. By default, we use ChatGPT as the self-reflection model and compare the capabilities of LLMs to generate programs as actor models.

DocPrompting: Zhou et al. (2022) proposed to explicitly leverage code documentation by first retrieving the relevant documentation pieces given a natural language (NL) intent, and then generating code based on the NL intent and the retrieved documentation. It can be viewed as a degraded version of EVOR where neither queries nor knowledge bases evolve and the retrieval pool encompasses the documentation as a single source.

3.2 Default EVOR Configuration

By default, we incorporate the documentation, execution feedback, and code snippets in the knowledge soup K for EVOR, as the content of web search contains large portions of noisy information (Appendix G) and only marginally improves

the results (§4.2). We use ChatGPT for both M_q to evolve queries and M_t to generate test inputs, and vary M between ChatGPT and CodeLlama to output code answers. We employ the INSTRUCTOR-xl (Su et al., 2023) as the primary retrieval model (Appendix H) and allow a maximum context length of 4,096 for both ChatGPT and CodeLlama, as the further increase incurs a higher cost, but fails to provide additional improvements (Appendix I).

3.3 Results

Table 2 shows that existing code generation approaches perform poorly on EVOR-BENCH. With CodeLlama, the improvements of MPSC, ExeDec, and Reflexion are smaller than 2% on average, compared to the vanilla generation. In particular, the execution accuracy remains 0 in Ring across three methods. This indicates that, even though existing approaches excel in code generation tasks that do not require external knowledge (e.g., HumanEval (Chen et al., 2021)), they cannot be directly applied to the setting of RACG without designing extra mechanisms to retrieve and utilize the external information. In contrast, by explicitly using documentation, DocPrompting significantly surpasses MPSC, ExeDec, and Reflexion by a large margin, further confirming that domain knowledge is critical to solving tasks in EVOR-BENCH.

Furthermore, EVOR achieves 16.1% and 16.2% absolute gain with ChatGPT and CodeLlama respectively on top of DocPrompting. This can be explained by the fact that DocPrompting only uses the documentation as a single retrieval source, without evolution in both queries and knowledge. By combining EVOR with MPSC, ExeDec, or Reflex-

evolution	Scipy-M	Tensor-M	Ring	Pony	Avg
	Model: ChatGPT				
No evolution	32.6	40.0	11.7	5.2	22.4
Evolve query	32.9	44.4	27.8	8.5	28.4
Evolve knowledge	33.5	42.2	13.5	6.1	23.8
EVOR (Evolve both)	37.9	53.3	36.6	13.5	35.3
	Model: CodeLlama				
No evolution	23.9	42.2	8.2	7.3	20.4
Evolve query	26.6	44.4	11.7	12.8	23.9
Evolve knowledge	25.8	44.4	12.6	8.3	22.8
EVOR (Evolve both)	31.2	53.3	26.7	17.4	32.2

Table 3: The performance of ChatGPT and CodeLlama when neither queries nor knowledge evolves, only the query evolves, only the knowledge evolves and when both evolve (EVOR). Results show that evolving both is consistently better across all datasets.

evolution ($\rightarrow$)	w/o evolution		w/ evolution	
Knowledge ($\downarrow$)	ChatGPT	CodeLlama	ChatGPT	CodeLlama
None	8.6	7.3	-	-
	Single Source Retrieval			
Web	9.7	7.5	10.6	8.2
Exec	11.7	7.4	13.8	9.1
Code	15.6	16.2	23.5	24.8
Doc	19.2	16.0	28.9	20.5
	Knowledge Soup Retrieval			
Exec + Code	18.3	15.9	27.8	25.3
Exec + Doc	20.7	17.2	32.4	23.0
Code + Doc	21.8	19.6	33.5	31.2
Exec + Code + Doc	22.4	20.4	35.3	32.2

Table 4: The average performance of ChatGPT and CodeLlama with different knowledge sources. Web refers to web search, Exec refers to execution feedback, Code refers to code snippets and Doc refers to documentation. The results show that diverse types of knowledge enhance RACG performance, where the improvement is larger under the setting with evolution.

ion, we observe further performance increase by up to 2.6% on average with ChatGPT. This suggests that EVOR is flexible to be integrated with existing approaches to further push forward the boundary of LLM performance in RACG.

4 Analysis

4.1 Synchronous evolution

We investigate how the synchronous evolution of queries and knowledge influences the RACG performance of LLMs. We compare EVOR to the setting where we only evolve queries (skip line 13-20 in Algorithm 1), only evolve knowledge (skip line 7 in Algorithm 1), and evolve neither of them (skip line 7, 13-20 in Algorithm 1, and terminate in a single iteration). We adopt the default setting in §3.2 except the specified changed in the algorithm. Table 3 shows evolving either queries or knowledge significantly enhances the results, highlighting that knowledge evolution also contributes to improving RACG in addition to the query rewriting. By applying the synchronous evolution of both queries and knowledge, EVOR consistently outperforms evolving either of them by large margins across all datasets in EVOR-BENCH. This suggests the complementary strength of synchronous evolution for eliciting the best performance of LLMs in RACG.

4.2 Diverse Knowledge

To understand the influence of diverse knowledge sources on EVOR, we conduct an ablation study by constraining the types of knowledge in the retrieval pool. Specifically, we construct the knowledge soup K with only one of web search, execution feedback, code snippets and documentation. We also consider the pairwise combination of execution feedback, code snippets and documentation,

and the setting where all of them are included. For each constructed knowledge soup, we experiment with evolving neither queries nor knowledge, and evolving both of them. We skip line 14 in Algorithm 1 when the code snippets are not included in the knowledge soup and skip line 16 when the execution feedback is not incorporated. Exceptions occur when the knowledge soup consists solely of web search content or documentation, where we only evolve queries.

In Table 4, we present the average performance of ChatGPT and CodeLlama using different types of knowledge sources, under two settings where evolution is and is not involved. The results show that, when augmenting with code snippets or syntax documentation, the performance of ChatGPT and CodeLlama is significantly higher than those using the web search or execution feedback. In particular, both models achieve less than 1% improvement when only using the web search as the knowledge source without evolving queries. This indicates that the general web search may not provide the most effective information to adapt LLMs in RACG. Compared to single-source retrieval, LLMs consistently achieve better results when more types of knowledge are integrated. For example, without queries and knowledge evolution, ChatGPT archives 6.2% higher average performance by using both code snippets and documentation as the knowledge sources, compared to only employing the code snippets. This indicates the advantage of diverse knowledge soup in enhancing the RACG performance of LLMs.

On the other hand, by evolving both queries and

knowledge or only evolving queries, both Chat-GPT and CodeLlama achieve significantly larger improvements when the knowledge soup becomes more diverse. For example, when the documentation is further included in the knowledge soup on top of the execution feedback and the code snippets, CodeLlama enhances the average result from 15.9% to 20.4% (+4.5%) in the setting where queries and knowledge are not evolved, but enhances from 25.3% to 32.2% (+6.9%) when both are evolved. This suggests that synchronous evolution is critical to fully exploit the advantage of diverse knowledge soup in adapting LLMs in RACG.

4.3 Repo-level Code Generation

Apart from updated libraries and long-tail programming languages, repo-level code generation is also a natural and realistic scenario for RACG, where LLMs are instructed to solve issues with reference to the Github repository code. Different from the documentation in EVOR-BENCH, the repository code could be much more complex with intertwined variable dependencies, customized function calls, etc. To solve an issue, the LLM usually needs to act as an agent to explore directories, use tools, make decisions, and more. Recent efforts have demonstrated the success of such agent-based methods (OpenDevin Team, 2024; Yang et al., 2024).

We explore the applicability of EVOR in this challenging setting. Specifically, we employ the popular SWE-bench-Lite (Jimenez et al., 2023) as the testbed, use all the repository content as the documentation, and adopt the configuration in §3.2. Due to the difficulty of the tasks, we experiment with two settings: (1) use GPT-4-1106 for all LLMs in Algorithm 1; (2) use Claude-3-opus. Figure 2 shows that EVOR outperforms the traditional RAG by a large margin, and is comparable with SWE-agent. This highlights the generalizability of EVOR with successful application in repo-level code generation.

Furthermore, we integrate EVOR with SWE-agent where we augment the search space of SWE-agent to include the execution feedback and code snippets without syntax errors, and dynamically update queries and the knowledge base in every iteration of generation. Figure 2 demonstrates additional performance improvements on top of both EVOR and SWE-agent, further proving EVOR's flexibility in its integration to agent-based approaches.
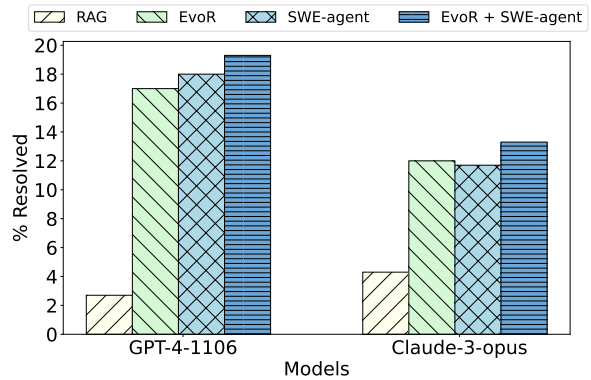

Figure 2: Performance of RAG, EVOR, SWE-agent (Agent) and combination of EVOR and SWE-agent.

4.4 Effective Token Usage

Olausson et al. (2023b) argues that iterative code generation, e.g., self-repair, may not yield higher pass rates when taking the cost into account. We conduct additional experiments to check the performance of EVOR at different levels of token budgets. In Algorithm 1, we adopt the termination condition as the limit of maximum token consumption, i.e., the algorithm exits when the tokens used by LLMs throughout iterations exceed a given threshold. Additionally, we compare EVOR to DocPrompting, the best baseline approach in Table 2. Following Olausson et al. (2023b), we sample LLMs multiple times until the token limit, using the concatenation of the given question n and the retrieved documentation K_r as the prompt. We calculate pass@t and set the token threshold to 4,000, 8,000, 12,000, 16,000, 20,000 and 24,000. Figure 3 shows that EVOR achieves significantly higher performance at all token levels for both ChatGPT and CodeLlama. With the increase of consumed tokens, EVOR demonstrates larger improvements compared to DocPromting, indicating the more effective token usage of EVOR in generalizing LLMs in RACG.

5 Related works

Since the focus of our work is to enhance code generation with retrieval, our work is closely related to code generation and retrieval-augmented code generation. Additionally, we are connected to the line of code execution since we also leverage it as an important retrieval source.

LLM-based Code Generation LLMs that have been pre-trained on extensive code corpus have exhibited impressive abilities in the domain of code
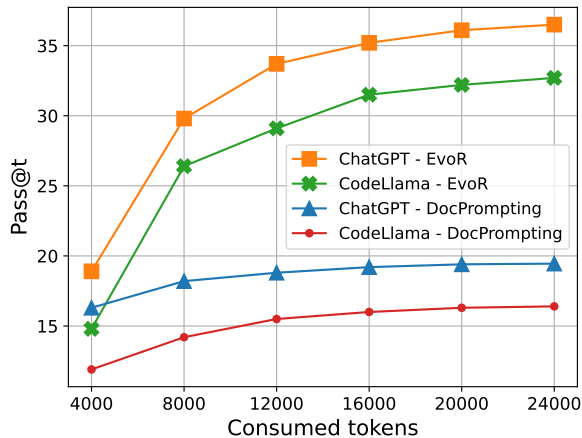
Figure 3: The pass rate of ChatGPT and CodeLlama at different token consumption levels. The results show that EvoR achieves a more significant increase compared to DocPrompting when the consumed tokens increase.

generation (Li et al., 2022; Nijkamp et al., 2022; Li et al., 2023b; Roziere et al., 2023; Wei et al., 2023). Numerous techniques have been suggested to improve the coding capabilities of LLM without the need to adjust its parameters (Chen et al., 2022; Huang et al., 2023; Li et al., 2023a; Zhang et al., 2023c; Chen et al., 2023; Key et al., 2022) However, most of these works set up the evaluation in scenarios LLMs are familiar with, e.g., HumanEval (Chen et al., 2021), HumanEvalPack (Muennighoff et al., 2023) and MBPP (Austin et al., 2021), where they are capable of demonstrating superior zero-shot performance by only utilizing internal knowledge. In this work, we focus on evaluating the capabilities of LLMs to incorporate external knowledge for the purpose of code generation in updated libraries or less-common programming languages. Our task reflects a more realistic yet challenging scenario for LLMs.

Retrieval-Augmented Generation The retrieval-augmented generation (RAG) is an appealing paradigm that allows LLMs to efficiently utilize external knowledge (Shi et al., 2023b; Izacard and Grave, 2020; Xu et al., 2023a; Jiang et al., 2023c). Recent works have applied it to the code generation task (Patel et al., 2023; Guo et al., 2023; Parvez et al., 2021a; Wang et al., 2023b). Specifically, Zhou et al. (2022) explored the natural-language-to-code generation approach that explicitly leverages code documentation. Zan et al. (2022) introduced a framework designed to adapt LLMs to private libraries, which first utilizes an APIRe-

triever to find useful APIs and then leverages an APICoder to generate code using these API docs. Zhang et al. (2023a) employed the iterative generate-retrieval procedure to do repository-level code completion. There are also recent efforts showing much enhanced performance by simply rewriting the queries (Ma et al., 2023; Anand et al., 2023; Chan et al., 2024) To the best of our knowledge, we are the first to adopt the synchronous evolution of queries and diverse knowledge to explore the setting where LLMs need to incorporate external information in code generation.

Code Execution Previous works have extensively employed executors (Interpreters/Compilers) in code-related tasks (Wang et al., 2022; Liu et al., 2023a; Olausson et al., 2023a; Chen et al., 2023). Shi et al. (2022) introduced execution result– based minimum Bayes risk decoding for program selection. Yang et al. (2023) established an interactive coding benchmark by framing the code as actions and execution feedback as observations. Chen et al. (2023) use the execution result as feedback to help LLM refine the code. In this work, we utilize the executor to provide feedback and check code outputs on syntax errors, which contributes to evolve both queries and knowledge in RACG.

6 Conclusion

Much recent work illustrated the ability of LLMs to incorporate external knowledge with retrieval-augmented generation. We propose a novel pipeline, EvoR, which achieves two to four times execution accuracy compared to existing code generation methods. Extensive experiments demonstrate that EvoR can be easily combined with them to provide further improvements, including the agent-based ones to solve challenging tasks such as repo-level code generation. Through an in-depth analysis, we further show the complementary strength of synchronous evolution of queries and documents in RACG, which enhances the model performance by larger margins with more diverse knowledge sources. We hope that our findings will inspire researchers and practitioners to develop efficient and effective strategies in their customized code-generation tasks with LLMs.

7 Limitations

Despite the effectiveness of EvoR in RACG, one limitation is that it requires multiple rounds of interactions among retrievers, LLMs, and executors

to output the code answer. This iterative process can lead to longer latency and increased energy consumption, which are critical concerns in real-time applications and energy-constrained environments. We hope that future work will design more efficient architectures or approaches to integrate LLMs seamlessly while maintaining or improving performance in RACG. Such advancements could significantly enhance the practicality and scalability of LLMs in realistic scenarios.

8 Potential Risk

The use of retrieval-augmented code generation with large language models introduces several potential risks, primarily centered around the quality and relevance of the retrieved code snippets. There is a risk of biased or incorrect information being retrieved, which could propagate errors or introduce vulnerabilities into generated code. Additionally, there are concerns about privacy and security if sensitive code snippets are inadvertently included in the retrieval process. Addressing these risks requires careful curation of retrieval sources, robust validation mechanisms, and continuous monitoring to ensure the integrity and safety of the generated code.

References

Abhijit Anand, Vinay Setty, Avishek Anand, et al. 2023. Context aware query rewriting for text rankers using llm. *arXiv preprint arXiv:2308.16753*.

Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Acl 2023 tutorial: Retrieval-based language models and applications. *ACL 2023*.

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

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2022. Improving language models by retrieving from trillions of tokens. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 2206–2240. PMLR.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Chi-Min Chan, Chunpu Xu, Ruibin Yuan, Hongyin Luo, Wei Xue, Yike Guo, and Jie Fu. 2024. Rq-rag: Learning to refine queries for retrieval augmented generation. *arXiv preprint arXiv:2404.00610*.

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

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

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Yucan Guo, Zixuan Li, Xiaolong Jin, Yantao Liu, Yutao Zeng, Wenxuan Liu, Xiang Li, Pan Yang, Long Bai, Jiafeng Guo, et al. 2023. Retrieval-augmented code generation for universal information extraction. *arXiv preprint arXiv:2311.02962*.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. REALM: retrieval-augmented language model pre-training. *CoRR*, abs/2002.08909.

Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. 2023. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*.

Gautier Izacard and Edouard Grave. 2020. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*.

Gautier Izacard, Patrick S. H. Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2023. Atlas: Few-shot learning with retrieval augmented language models. *J. Mach. Learn. Res.*, 24:251:1–251:43.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023a. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023b. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907*.

Zhengbao Jiang, Frank Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023c. Active retrieval augmented generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7969–7992, Singapore. Association for Computational Linguistics.

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

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

Darren Key, Wen-Ding Li, and Kevin Ellis. 2022. I speak, you verify: Toward trustworthy neural program synthesis. *arXiv preprint arXiv:2210.00848*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023a. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

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

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023a. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023c. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting for retrieval-augmented large language models. *arXiv preprint arXiv:2305.14283*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *CoRR*, abs/2308.07124.

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

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023a. Demystifying gpt self-repair for code generation. *arXiv preprint arXiv:2306.09896*.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023b. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. Website. https://openai.com/blog/chatgpt.

OpenDevin Team. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. https://github.com/OpenDevin/OpenDevin. Accessed: ENTER THE DATE YOU ACCESSED THE PROJECT.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021a. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

Arkil Patel, Siva Reddy, Dzmitry Bahdanau, and Pradeep Dasigi. 2023. Evaluating in-context learning of libraries for code generation. *arXiv preprint arXiv:2311.09635*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. *arXiv preprint arXiv:2305.15294*.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*.

Kensen Shi, Joey Hong, Manzil Zaheer, Pengcheng Yin, and Charles Sutton. 2023a. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *arXiv preprint arXiv:2307.13883*.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. 2023b. Replug: Retrieval-augmented black-box language models. *arXiv preprint arXiv:2301.12652*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2023. One embedder, any task: Instruction-finetuned text embeddings. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1102–1121, Toronto, Canada. Association for Computational Linguistics.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023a. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*.

Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023b. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 146–158.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. 2023a. Recomp: Improving retrieval-augmented lms with compression and selective augmentation. *arXiv preprint arXiv:2310.04408*.

Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. 2023b. Retrieval meets long context large language models. *arXiv preprint arXiv:2310.03025*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *arXiv preprint arXiv:2306.14898*.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When language model meets private library. *arXiv preprint arXiv:2210.17236*.

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

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

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023c. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *arXiv preprint arXiv:2305.14591*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

A Dataset curation

We introduce more details about our dataset curation process for updated library (§A.1) and long-tail programming languages (§A.2). In §A.3, we describe our implementation of the test case construction for the dataset Ring and Pony.

A.1 Library-oriented data collection

Following Zan et al. (2022), we use the synonyms of original API names and API arguments in the updated library, such as converting `stack` to `pile`. Additionally, we combine two similar APIs into one, with newly added arguments to distinguish the authentic functionalities, e.g., `linear_interpoloate` integrates two APIs, `griddata` and `interp1d`. Finally, we create new class objects and align methods with the original class. For instance, a new `SparseMatrix` object is created to include all sparse matrix objects in Scipy. We rewrite the ground truth solution for each example with new APIs.

To construct the documentation of the updated libraries, we first collect the original libraries [7]. We then replace the old documentation files with our modified version. For each question, we annotate the oracle documentation by checking the ground truth answer. We grasp the corresponding documentation pages and concatenate them to serve as the minimum documentation required for answering the problem. We reuse the test cases introduced in DS-1000 to evaluate LLM generalization performance.

A.2 Language-oriented data collection

For each programming problem collected from LeetCode, we rewrite the function signatures to adapt them to the target programming language. We collect the whole documentation for Ring and Pony from their websites: https://ring-lang.github.io/doc1.19/ and https://www.ponylang.io/. For each question, we labeled the oracle documentation of the specific grammar used in the ground truth, such as data structures or branching syntax. We concatenate the document for each new syntax used in the ground truth to obtain a minimum document that contains the required syntaxes for answering the question.

---

[7] https://docs.scipy.org/doc/, https://www.tensorflow.org/api_docs

A.3 Test-case generation for language-oriented data

To accurately evaluate the performance of LLM in writing code of long-tail programming languages, we follow (Liu et al., 2023b) to construct a comprehensive set of test cases for each problem. Specifically, we first prompt ChatGPT to write a validation script and solution script using Python. The validation script will check for the input constraints (e.g. single line of a positive integer, two binary strings, etc.) of the problem. The solution script is supposed to generate the correct answer given a valid input. We then manually check both scripts and modify them if necessary for all problems. Next, we prompt ChatGPT to create complex and corner cases until there are 20 test cases for each problem. We then apply mutation-based strategies to extend the number of test cases for each problem to 200. The mutation-based strategy works as follows. We first parse the input into the appropriate format and types (e.g. list of strings, tuple of integers, etc.) We will then randomly mutate the test cases multiple times to create a new input based on the types. For instance, we may add 1 or subtract 1 from an integer to mutate it. All generated test cases added are checked by both the validation script and solution script. A test case is considered as valid if the following three conditions are met: (1). Both scripts do not report any error; (2). The solution script terminates within 1 second; (3). The answer returned by the solution script matches that in the test case.

The final step is to apply test-suite reduction which selects a subset of all input test cases while preserving the original test effectiveness (i.e. the reduced set of test cases marks a code solution as right/wrong if and only if the original set marks it as right/wrong). We employ the three strategies proposed by (Liu et al., 2023b): code coverage, mutant killing, LLM sample killing. Code coverage evaluates how each test case covers different branch conditions in the solution script. Mutant killing employees a mutation testing tool for Python to create mutant codes from the solution script. LLM sample killing prompts llama-2-70b to generate several incorrect solutions to the problem. We run all test cases against these different codes to perform the test-suite reduction. Finally, we generate the answer using the solution scripts.

B Prvate Library

We notice that Zan et al. (2022) crafted three benchmarks named TorchDataEval, MonkeyEval, and BeatNumEval to evaluate the capability of language models in code generation with private libraries. Their benchmarks share some similarities with our two datasets on updated libraries, where we both modified popular Python libraries to explore the setting for LLM generalization. Different from them, our datasets are built with increased complexity, where we not only use the simple synonym to update the API names, but additionally combine two APIs and create new class objects. This indicates that our datasets are likely to cover broader scenarios of library updates in real life.

Nonetheless, we also benchmark our system on their datasets with varied knowledge source. Table 5 shows that CodeLlama achieves exceptionally high score in all three datasets, with zero-shot accuracy 80.2% in Monkey. Since the three datasets were available in Github as early as 2022, which is well ahead of the time CodeLlama was released, we suspect that CodeLlama has been trained on the three datasets. Although our system still looks to be effective in their benchmarks with performance gain by including more knowledge sources, we are concerned that these datasets may not be able to reflect the generalization capabilities of LLM.

C LLM-generated program inputs

To verify the syntax of the generated program, one effective way is to execute it with test cases. To simulate the scenario where no test case is available, we investigate whether it is possible to generate program inputs with LLMs. Specifically, we prompt ChatGPT and CodeLlama to generate 5 test cases for each problem, and only save the inputs for evaluating the syntax of other programs. As an ablation study, we execute the gold program of each problem with the generated inputs and count a generated input as valid if no error is reported during execution. We calculate the accuracy as the percentage of examples where all the generated test inputs are valid. Table 6 shows that both ChatGPT and CodeLlama exhibit superior performance in generating test inputs. This indicates that LLM-generated test inputs serve as good resources as syntax verifiers.

D Sample code snippets and execution feedback

We collect sample code snippets and execution feedback in constructing the knowledge base. Specifically, we prompt LLMs to write short scripts of sample usage of each function in the documentation corpus. We then execute those scripts. If the execution of a code snippet reports errors, we include it as a pair of (code, error); otherwise, we regard it as a code snippet that could demonstrate the syntax and function usage.

E Cost Analysis

Despite significant enhancement of EVOR in the generalization results, the iterative process that involves multiple LLM generations incurs large costs. In this section, we discuss the trade-off between the cost and the performance. To measure the cost, we count the total tokens processed by LLM throughout the process in each example.

From Table 7, we can see that, the exceptional performance is linked to the extensive processing of tokens. Compared to employing Single-time-Q, which simulates the traditional RAG pipeline and directly uses the question as the query to retrieve documentation, ChatGPT and CodeLlama achieve 2.9% and 4.1% performance gain in average execution accuracy by using Single-time, which formulates the query as the explained code and retrieves from diverse knowledge soup. This enhancement is at the expense of around 25% more processed tokens for both models. With active retrieval, the average performance further increases by 15.4% and 15.1% for ChatGPT and CodeLlama respectively. However, the processed tokens increase by more than 2 times for both models. With a notable increase in both cost and performance, there arises a trade-off for practitioners to carefully weigh and adjust according to their specific requirements.

F More Experimental Setting

In all settings, we leave a length of 400 for generation and adopt ChatGPT as the LLM to explain the code, i.e., all the code is fairly translated into the explained code. In every iteration of active retrieval and LLM generation, we add the examples with correct syntax (judged by executors with sample inputs) to the set of code snippets and only rectify the code with syntax error.

For each of the knowledge sources considered in this paper, we adopt the following principles if

	Model: ChatGPT				Model: CodeLlama			
Method	Monkey	BeatNum	TorchData	Avg.	Monkey	BeatNum	TorchData	Avg.
Vanilla	38.6	27.7	42.0	36.1	80.2	70.3	54.0	68.2
EVOR	**67.3**	**70.3**	**74.0**	**70.5**	**93.1**	**90.1**	**92.0**	**91.7**

Table 5: We evaluate the zero-shot ChatGPT and CodeLlama on three private libraries. Although we observe significant improvements of EVOR, the exceptionally high accuracy of CodeLlama Vanilla (zero-shot) performance suggests the risk of data leakage, making it less reliable to assess model generalization capabilities.

	Model: ChatGPT				Model: CodeLlama		
Scipy-M	Tensor-M	Ring	Pony	Scipy-M	Tensor-M	Ring	Pony
89.2	93.3	100.0	100.0	86.8	91.1	95.6	96.8

Table 6: The accuracy of ChatGPT (left) and CodeLlama (right) in generating valid program inputs. Although LLMs cannot guarantee to write accurate test cases, their performance in generating only program inputs is exceptionally high.

it is included in the prompt for LLM generation: (1). For web search content, include it until the maximum allowed length, e.g., 4,096, as we do not merge it without other knowledge sources; (2). For execution feedback, include the error message and the line of the code that leads to the error; (3). For code snippets, allocate a maximum length of 300 to them, as they are usually short; (4). For documentation, always include other types of knowledge first, and include documentation to fill in the rest length. For example, if we want to include both documentation and code snippets as the knowledge source and the maximum context length is 4,096, we will allocate a maximum length of 300 to code snippets and a maximum length of 4,096-300-400=3,396 to the documentation.

G Web Search

As the artificially modified libraries are not available online, we replace the documentation returned by web search with our modified version. In addition, we heuristically update the content from web search based on our modifications, e.g., map keywords to the synonyms we use.

In Figure 4, we present an example of the top-3 web search results returned by Google search to the query "In the programming language Pony, checks if the current element is less than the previous element in the nums array". Due to the infrequent usage of the programming language Pony, there is little available resource online. The web search fails to identify the relevant knowledge piece. Even the specific instruction "programming language Pony" is given in the query, a guidance to solve the problem in C++ is included. In addition, the returned texts are long, complex, and diverse, mix-

ing various types of knowledge sources including tutorials, blogs, and community Q&A discussions. LLMs may find it challenging to process and effectively utilize all of the information simultaneously. Finally, although we empirically remove some unrelated information, e.g., remove the line that starts with ∗ that is likely to be an irrelevant item listing, there is more that is hard to remove with just heuristics. This poses a great challenge to LLMs as they are burdened to filter the unrelated content and avoid getting distracted by it.

Figure 4: A web content example covering tutorials (web content 1), blogs (web content 2) and Q&A discussions (web content 3). We show the top-3 results returned by google search and cut each webpage for brevity.

H Retrieval Model

We experiment with a representative sparse retriever, BM25, and several competitive dense re-

Model	Retrieval	Scipy-M		Tensorflow-M		Ring		Pony		Average	
		Acc	Tokens	Acc	Tokens	Acc	Tokens	Acc	Tokens	Acc	Tokens
ChatGPT	Vanilla	17.6	423	11.1	322	3.7	206	1.8	222	8.6	293
	DocPrompting	32.4	3687	33.3	3728	8.4	3826	2.7	3763	19.2	3751
	RK	32.6	4826	40.0	4924	11.7	4528	5.2	4584	22.4	4716
	EVOR	37.9	13631	53.3	12568	36.6	24987	13.5	13819	35.3	16251
CodeLlama	Vanilla	11.3	476	17.8	381	0.0	263	0.0	314	7.3	386
	DocPrompting	16.9	3923	37.8	4012	4.7	4050	4.4	3978	16.0	3991
	RK	23.9	5124	42.2	5023	8.2	4987	7.3	4823	20.4	4989
	EVOR	31.2	14564	53.3	12323	26.7	29384	17.4	14592	32.2	17716

Table 7: The comparison of LLM performance and consumed tokens per example without retrieval (Vanilla), retrieval without evolution from only documentation (DocPrompting), retrieval without evolution from knowledge soup (RK) and EVOR retrieval. By default, we use INSTRUCTOR-xl as the embedding model. The results in the table demonstrate the association between superior results and the massively processed tokens, which implies the trade-off between the performance and the cost.
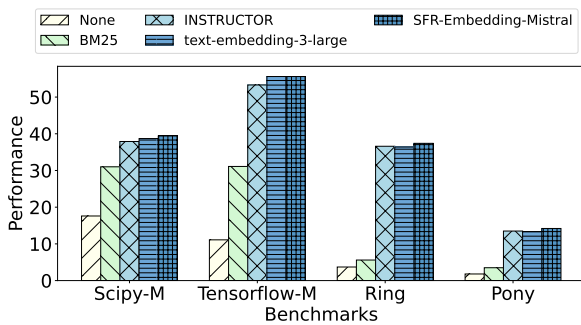

Figure 5: Comparison of ChatGPT generalization performance when the sparse retriever (BM25), or the dense retriever (INSTRUCTOR, text-embedding-3-large, SFR-Embedding-Mistral) is employed. The results show that dense retrievers significantly outperform their sparse counterpart, BM25. In general, ChatGPT achieves the best performance when SFR-Embedding-Mistral is used as the retrieval model.
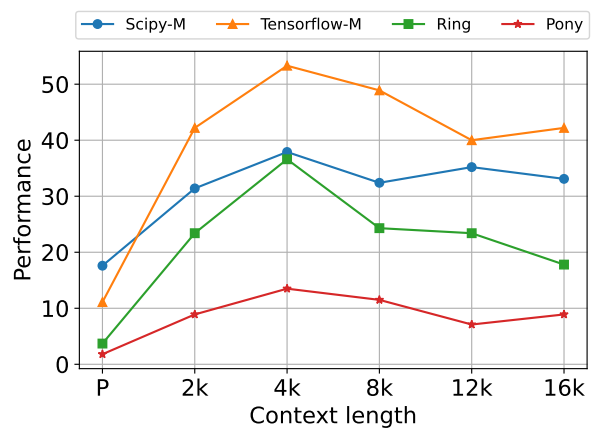

Figure 6: ChatGPT performance with various maximum allowed context lengths. P refers to the baseline where no external knowledge is included. Although the model supports the context length up to 16k, the results reveal that the execution accuracy ceases to enhance when the context window is expanded from 4k to 16k. This suggests that augmenting ChatGPT with external knowledge beyond the 4k context does not yield further improvement in the generalization performance.

trievers using the default setting of EVOR, except for changing the retrieval model. INSTRUCTOR-xl (Su et al., 2023) is an 1.3B embedding model fine-tuned to follow instructions for efficient adaptation. text-embedding-3-large[8] is OpenAI's latest embedding model, showcasing competitive performance. SFR-Embedding-Mistral is trained on top of E5-mistral-7b-instruct (Wang et al., 2023a) and Mistral-7B-v0.1 (Jiang et al., 2023a) and achieves state-of-the-art performance in MTEB leaderboard (Muennighoff et al., 2022).

As shown in Figure 5, across four datasets, when utilizing dense retrievers, ChatGPT significantly enhances the performance achieved with a sparse retriever. Aligned with the results in the retrieval benchmark (MTEB), ChatGPT consistently achieves the best performance when using SFR-Embedding-Mistral as the retrieval model. How-

ever, the gap between different dense retrievers is not significant. After considering both the performance and the cost, we opt for INSTRUCTOR-xl for efficient and cost-effective development of EVOR.

I Long-context Model

Besides the retrieval-based pipelines, long-context models are another alternative for LLMs to incorporate massive external knowledge. The context window of Claude 2.1[9] and GPT-4[10] have reached 200k and 128k tokens respectively, which ques-

---

[8]https://platform.openai.com/docs/guides/embeddings

[9]https://www.anthropic.com/news/claude-2-1

[10]https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo

tions the necessity to adopt RACG, where only a small portion of knowledge is retrieved and exposed to LLMs. Intuitively, LLMs benefit from larger context windows, as they can utilize more external knowledge to enhance their coding. However, our experiments do not imply the case.

We adopt the default setting of EVOR, but only change the maximum context length of LLMs to 2k, 4k, 8k, 12k, and 16k tokens. Figure 6 indicates that ChatGPT achieves the best performance when only using external knowledge of 4k tokens. This aligns with the findings in Xu et al. (2023b). With extended context lengths, i.e., more retrieved content is included in the prompt, the performance does not further increase.

The potential reasons to explain this situation include: (1). Only a few documents are required to answer a specific question. As shown in Table 1, the length of gold documentation, i.e., minimum required syntax descriptions, never surpasses 4k, which does not even surpass 1k in Scipy-M , Tensorflow-M and Ring. This implies that the retriever has a good chance to include the gold documentation within 4k context length; (2). LLMs have low attention in the middle of long contexts (Liu et al., 2023c). With long contexts, LLMs may fail to identify the relevant content in the middle that can help solve the problem.

We leave it to future research to design a more delicate retrieval system that can appropriately regulate the content utilized for LLM generation.

J Personally Identifying Infomation

We collect data from the domain of code generation. We authors carefully reviewed all the collected data, and confirm that the data that was collected/used does not contain any information that names or uniquely identifies individual people or offensive content.

K Intended use

EVOR is an advanced pipeline for RACG, and it is expected to be applied in customized code generation. EVOR-BENCH consists of four realistic benchmarks for RACG, and is expected to be used as an evaluation benchmark to evaluate RACG systems.

L License

Our code and data will be released under Apache-2.0 license.

M Data Documentation

EVOR-BENCH is collected from LeetCode and adapted from DS-1000 (Lai et al., 2023), which was originally collected from StackOverflow. The data in EVOR-BENCH is all in English.

N Machines

We run all experiments on Nvidia A100 GPUs. It takes around 1 hour to finish one dataset in EVOR-BENCH. To complete all the experiments in the paper, it tasks around 24 hours.

O Packages

For all the packages we use in the code, we employ the pip or conda implementation in the latest version.

P Instructions for Human Annotators

- Write the code in the corresponding programming languages to the problem

- Find the documentation of the syntax used in the code.

Q Data Consent

We adapt data from DS1000 (Lai et al., 2023), which is released under CC-BY-SA-4.0 license. Some of the data is collected from LeetCode, which is a public platform for practicing coding skills. By the copyright law 107: Notwithstanding the provisions of sections 106 and 106A, the fair use of a copyrighted work, including such use by reproduction in copies or phonorecords or by any other means specified by that section, for purposes such as criticism, comment, news reporting, teaching (including multiple copies for classroom use), scholarship, or research, is not an infringement of copyright. In determining whether the use made of a work in any particular case is a fair use the factors to be considered shall include— (1)the purpose and character of the use, including whether such use is of a commercial nature or is for non-profit educational purposes; (2)the nature of the copyrighted work; (3)the amount and substantiality of the portion used in relation to the copyrighted work as a whole; and (4)the effect of the use upon the potential market for or value of the copyrighted work.[11]

We should be eligible to use the data.

---

[11]https://copyright.gov/fair-use/

R Annotators

We authors, computer science undergraduate, master or PhD students collect the data

S AI Usage

We do not use AI for research, coding or writing in this paper.