ARKS: Active Retrieval in Knowledge Soup 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]

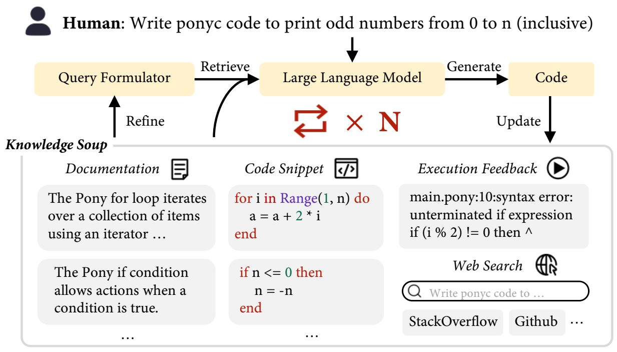
Abstract

Recently the retrieval-augmented generation (RAG) paradigm has raised much attention for its potential in incorporating external knowledge into large language models (LLMs) without further training. While widely explored in natural language applications, its utilization in code generation remains under-explored. In this paper, we introduce Active Retrieval in Knowledge Soup (ARKS), an advanced strategy for generalizing large language models for code. In contrast to relying on a single source, we construct a knowledge soup integrating web search, documentation, execution feedback, and evolved code snippets. We employ an active retrieval strategy that iteratively refines the query and updates the knowledge soup. To assess the performance of ARKS, we compile a new benchmark comprising realistic coding problems associated with frequently updated libraries and long-tail programming languages. Experimental results on ChatGPT and CodeLlama demonstrate a substantial improvement in the average execution accuracy of ARKS on LLMs. The analysis confirms the effectiveness of our proposed knowledge soup and active retrieval strategies, offering rich insights into the construction of effective retrieval-augmented code generation (RACG) pipelines. Our model, code, and data are available at `https://arks-codegen.github.io`.

1. Introduction

The retrieval-augmented generation (RAG) paradigm has raised significant attention in the era of large language models (LLMs) (Guu et al., 2020; Karpukhin et al., 2020; Izacard et al., 2023; Borgeaud et al., 2022; Asai et al., 2023). It empowers LLMs to effectively assimilate external knowledge (Khattab et al., 2022), providing reference (Menick et al., 2022), and boosting overall performance (Jiang et al., 2023b). While the realm of text-based applications has witnessed extensive advancements in RAG over the years, the progress in retrieval-augmented code generation (RACG) has not kept pace with that of language applications (Zhou et al., 2023; Zhang et al., 2023a). This may be partially attributed to the fact that existing LLMs have already shown strong abilities in code generation (Chen et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Fried et al., 2022; Xu et al., 2022; Li et al., 2023b). Unlike textual corpora, which tends to be consistently stable, code repositories undergo frequent updates, which often encompass modifications aimed at disabling undesirable functions, the integration of new features, and the introduction of fresh syntactic sugar (Walters et al., 2009; Ali et al., 2023; Bugden & Alahmar, 2022). Considering the dynamic nature of code, it is crucial to tackle the challenges in RACG.

Building on the success of practices in RAG, earlier efforts on RACG have primarily centered around generating code through retrieval from the web content (e.g., Stack Overflow) (Parvez et al., 2021; Wang et al., 2022), the code documentations (Zan et al., 2022; Zhou et al., 2023), or the source code in the same repository (Zhang et al., 2023a;



The Pony for loop iterates over a collection of items using an iterator …

The Pony if condition allows actions when a condition is true.

```
for i in Range(1, n) do
  a = a + 2 * i
end
```

```
if n <= 0 then
  n = -n
end
```

```
main.pony:10:syntax error:
unterminated if expression
if (i % 2) != 0 then ^
```

Figure 1. Instead of depending solely on human instructions for knowledge soup retrieval and completing the pipeline in a single round, we actively engage in the retrieval process through the following steps: (1) generating code and updating the knowledge soup; (2) refining the query; (3) retrieving information from the knowledge soup and back to (1). Through active retrieval for N iterations, both the query and knowledge soup undergo improvements, ultimately enhancing the quality of the generated code.

[1]The University of Hong Kong [2]Fudan University [3]Sea AI Lab. Correspondence to: Hongjin Su <hjsu@cs.hku.hk>.

Shrivastava et al., 2023). While these approaches have proven effective for code generation, they generally treat it as a one-time process and only employ a single-source (i.e., either documents or code) for retrieval. However, we argue that exclusively relying on single-source retrieval may not be the optimal choice , especially in the context of code generation. For example, when the compiler identifies a syntax error for the generated code, leveraging the error message to initiate a web search can unveil relevant coding examples. LLMs can then utilize such information to generate code with higher quality.

Motivated by the above example and the recently proposed active RAG (Jiang et al., 2023b), in this paper, we posit that *RACG should be conceptualized as a multi-round process with active exploration*. Specifically, we advocate for a more active retrieval of all available information we can leverage, which we use "knowledge soup" to note below. The knowledge soup here includes the web content from the search engine, the related documentation, the feedback from code execution, and the dynamically added code snippets. Different from single-source, these different information build the multi-source retrieval pipeline, which facilitate the diversity of the retrieved information during code generation. Meanwhile, as depicted in Figure 1, ARKS leverages a multi-round procedure which generates and retrieves information iteratively. Different from the traditional single-round code generation, our retrieval process stands out as both the query and the knowledge soup undergo continuous updates after each iteration. This strategic refinement aims to facilitate the recall of more pertinent information. Through multiple-round interactions with the knowledge soup, the LLM progressively improves the generated code.

To evaluate our system in the generalization setting with LLMs pre-trained on massive public data, we propose a novel benchmark comprising four datasets designed to simulate updates in code repositories. 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, and open-source models like CodeLlama (Roziere et al., 2023). Experimental results across these four datasets demonstrate that our method can yield a significant improvement in the average performance of different LLMs. For example, on the Ring dataset, our method showcases an impressive boost, elevating the performance of ChatGPT from 3.7% to 35.5%. The further analysis unveils an unexpected finding: web search, the most widely utilized method in RAG, is not as effective as expected. In contrast, the effectiveness of RACG is notably influenced by code snippets, code documentation, and execution feedback. Finally, we showcase the notable advancements attained through our active retrieval approach, demonstrating an improvement of up to 27.1% when compared to the results achieved through single-round retrieval. We will release the code and the new benchmark upon acceptance.

2. Active Retrieval in Knowledge Soup

Given a question n in natural language, the objective of retrieval-augmented code generation is to first retrieve 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 uses a question to retrieve from a static knowledge base, and finishes the pipeline in a single round, we continue the retrieval process by actively refining queries and updating the knowledge soup. 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 formulation (§2.1), the construction of knowledge soup (§2.2), the mechanism of active retrieval (§2.3) and the datasets to benchmark our system (§2.4).

2.1. Query Formulation

While it is natural to employ the natural language question as Q, we argue that this is not optimal for retrieving relevant information to solve coding problems. In traditional open-domain QA tasks, where the RAG pipeline is extensively used, the question and the information source usually form strong linkage (e.g., the gold passage of the question "What is the widest highway in North America" explicitly mentions "Highway 401 that passes through Toronto is one of the widest" (Kwiatkowski et al., 2019)). However, realistic code generation tasks only give a high-level intent, e.g., "Write a function that calculates the n^{th} Fibonacci number", without mentioning the exact algorithm or syntax to use.

Therefore, to establish more informative queries in the scenario of RACG, we explore more promising options. In numerous RAG pipelines, the natural language **question** that describes the coding task is the default query. We therefore include it as a baseline for our query formulation. Frequently, human programmers use the error message provided by the executor to search the web for pertinent information. Inspired by this, we consider the **execution feedback** as a candidate to serve as the query. Moreover, the code produced by LLMs, despite not being accurate, frequently provides clues about the syntax or grammar employed, guiding the retrieval model toward locating pertinent information. This motivates us to add the LLM-generated **code** to the candidate set of query formulation. Finally, since some code is abstract and the syntax is hidden in easily overlooked details, translating the code to natural language through LLM explanation probably provides more explicit signals to retrievers. For example, retrieval models will find it easier to locate corresponding descriptions when given the query ”check equality of two variables”, compared to ”a==b”. We finalize our candidate pool of query formulation with the inclusion of **explained code**.

Different from the question, which is directly used to retrieve external knowledge in the traditional RAG pipeline, the other three query formulations are dynamically updated based on the LLM generations. In the first iteration, when no previous LLM generation is available, LLMs draft a code solution based on the problem description. The execution feedback, code, and the explained code are then obtained from the draft solution. In general, except formulated as the question, the query in the $(i+1)^{th}$ iteration is developed from the code LLMs generate in the i^{th} iteration, which implies that the query is refined every round to solicit the information that LLMs require in the current state. In §3.3, we comprehensively study the effectiveness of each query formulation. While it is possible to combine several query formulations to seek better retrieval performance, we leave it to future research to develop more delicate mechanisms of query formulation.

2.2. Knowledge Soup

Intuitively, the documentation of updated libraries/new programming languages serves as a good knowledge source, as it is succinct and comprehensively covers syntax descriptions. Despite its effectiveness, we further explore other options that may also help to generalize LLMs:

Web search is a general and popular resource applied in traditional RAG applications. 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 grappling 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 webbrowsing 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 F, we include more discussions about the content in 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. Unlike web search, which may contain a mix of diverse information sources with noise (e.g., unrelated texts or redirection links), the documentation is homogeneous and clean. Acting as a detailed guide, it allows programmers to efficiently comprehend and use the language or library. 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 new programming languages in code generation, without making any parameter update.

[1]https://pypi.org/project/google/

[2]https://pypi.org/project/html2text/

Execution feedback is a specific knowledge type for code generation. Human programmers routinely turn to execution feedback, a process that precisely identifies errors within the code, facilitating targeted fixes. Unlike the retrieval of information from knowledge sources, which often involves the use of a retrieval model, obtaining execution feedback entails directly accessing information provided by the code executor, such as a Python interpreter. This feedback is generated instantly and guaranteed to be both accurate and pertinent. These features separate the it apart from the traditional RAG knowledge sources.

Frequently used by human programmers to debug their code, it potentially not only serves as a good query but also an informative source that helps LLMs fix syntax errors and write correct code. Earlier studies have shown the efficacy of LLMs in auto-correcting programs in domains they have been trained for(Yasunaga & Liang, 2020; Le et al., 2022). In this paper, we investigate whether LLMs retain such capability to rectify mistakes in unfamiliar target libraries or programming languages.

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 programming language naturally align with the LLM generation objective and provide concrete examples of input, output, 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 grammars. We construct the set of code snippets by accumulating the code solutions generated by LLMs verified to be free of syntax errors. For each code snippet generated by LLMs, we execute it with several sample inputs (Appendix C). If the executor does not complain in all of the execution, we count the code snippet as ”syntax-correct”, which can serve as a demonstration for other instances to refer to. Throughout the process of active retrieval, the set of code snippets is gradually enlarged with more and more newly added examples.

2.3. Active Retrieval Pipeline

After constructing the diverse knowledge soup, the next question is, how to perform effective retrieval to support LLM generation. We tackle this problem by proposing a novel pipeline. As shown in Figure 1, given a human instruction, LLM first drafts a code solution, which is executed with sample inputs, and used to update execution feedback and code snippets in the knowledge soup (§2.2). Next, we formulate the query as one of execution feedback, LLM-generated code, or the explained code for the examples with incorrect syntax. The formulated query retrieves from the knowledge soup, which augments LLM for the next round generation. We terminate the iteration when the syntax accuracy (the percentage of examples where the execution on sample inputs does not report error) of the evaluation set remains the same for consecutive two rounds.

In addition, we also experiment with the question as the query formulation. Following the standard RAG pipeline, the question is directly used to retrieve from the knowledge soup in the first iteration, which augments LLM to generate code. Additionally, we also apply the idea of ARKS, where we execute the generated code to update the knowledge soup. The next iteration starts with using the same query (question) to retrieve from the updated knowledge soup, and the active retrieval process continues.

2.4. Datasets

Since LLMs are extensively trained on public data, we curate a new benchmark to evaluate their generalization capability with ARKS. 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. By default, we use execution accuracy (pass@1) as the metric to evaluate LLM generalization performance. We present the dataset statistics in Table 1. More details about our curation process can be found in Appendix A.

[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/

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.

3. Experimental Results and Analysis

To verify the effectiveness of ARKS, we conduct extensive experiments with both the proprietary model ChatGPT (gpt-3.5-turbo-1106) and the open-source model CodeLlama (CodeLlama-34b-Instruct). In §3.1, we comprehensively compare the effectiveness of the four proposed knowledge sources and their compositional benefits. By default, we apply the active retrieval to elicit the best performance of LLM generalization (§3.2). Primarily, we employ the INSTRUCTOR (Su et al., 2022) as the retrieval model and the explained code as the query among various alternatives (§3.3). In contrast to large context windows, we allow the maximum length of 4096 by default, as the further increase incurs a higher cost, but fails to provide additional improvement (§3.4). More experimental settings can be found in Appendix E.

3.1. Knowledge Soup

As introduced in §2.2, we argue the necessity for a diverse group of information to be retrievable in RACG. To address this, we first evaluate the performance of individual performance of single source retrieval, wherein web search, execution feedback, code snippets, and documentation are considered as distinct sources. Subsequently, we explore the benefits arising from the integration of these sources into a comprehensive knowledge soup.

Single Source Retrieval. As shown in Table 2, each type of information contributes varying degrees of performance improvement when utilized as a retrieval source for the model. Surprisingly, among the four resources, the impact of incorporating web search on performance of both ChatGPT and CodeLlama is the least substantial, which is contradictory to the success of web search applied in RAG.

Knowledge	Model: ChatGPT					Model: CodeLlama				
	Scipy-M	Tensor-M	Ring	Pony	Avg.	Scipy-M	Tensor-M	Ring	Pony	Avg.
None	17.6	11.1	3.7	1.8	8.6	11.3	17.8	0.0	0.0	7.3
Single Source Retrieval (Baseline)										
Web	19.0	13.3	4.7	1.8	9.7	12.0	17.8	0.0	0.0	7.5
Exec	21.1	20.0	4.7	4.4	12.6	12.7	17.8	0.0	0.0	7.6
Code	28.2	31.1	24.3	7.1	22.7	17.6	35.6	26.2	15.9	23.8
Doc	35.2	48.9	26.2	5.3	28.9	26.1	44.4	0.9	10.6	20.5
Knowledge Soup Retrieval (Ours)										
Exec + Code	33.1	40.0	26.2	8.9	27.1	18.4	35.6	26.2	15.9	24.0
Exec + Doc	36.6	53.3	27.1	7.1	31.0	26.8	46.7	0.9	11.5	21.5
Doc + Code	37.3	48.9	34.6	8.9	32.4	30.3	48.9	26.2	16.8	30.6
Doc + Code + Exec	**37.3**	**53.3**	**35.5**	**12.4**	**34.6**	**30.3**	**51.1**	**26.2**	**16.8**	**31.1**

Table 2. ChatGPT and CodeLlama execution accuracy with different knowledge sources. Tensor-M refers to Tensorflow-M and Avg. refers to the average score across four benchmarks. Web denotes the web search content; Exec denotes the execution feedback from compiler/interpreter; Code denotes the code snippets generated by LLMs in previous rounds that are verified to be free of syntax error; Doc refers to the documentation. Adding more knowledge sources consistently enhances the performance, which demonstrates the advantage of a diverse knowledge soup in the RACG pipeline.

To understand the underlying reasons, we manually investigate the retrieved content from web search (Appendix F). Particularly in the domain of less-common programming languages, we observe that top-ranking results are not always relevant, and LLMs face the challenge of filtering out unrelated content. The obstacle impede the efficacy of web search as a valuable resource for current LLMs in RACG. Then for the effectiveness of execution feedback, we observed a strong dependency on the quality of error messages. The executors behind the well-established libraries Scipy-M and Tensorflow-M consistently pinpoints issues with precision, such as missing parameters. In contrast, Ring and Pony often produce more ambiguous errors, e.g., simply ”syntax error”, providing less clarity on the errors in the generated code. In comparison to utilizing web search and execution feedback alone, both ChatGPT and CodeLlama demonstrate substantial improvements by including code snippets or the code documentations as the retrieval source. This suggests that, in addition to traditional documentation, code snippets prove to be a viable alternative for boosting the performance of LLMs on code generation, especially on less-common programming languages.

Retrieval	Scipy-M	Tensor-M	Ring	Pony	Avg
Model: ChatGPT					
No Retrieval	17.6	11.1	3.7	1.8	8.6
One-Time Retrieval	32.4	33.3	8.4	2.7	19.2
Active Retrieval	37.3	53.3	35.5	12.4	34.6
Model: CodeLlama					
No Retrieval	11.3	17.8	0.0	0.0	7.3
One-Time Retrieval	16.9	37.8	4.7	4.4	16.0
Active Retrieval	30.3	51.1	26.2	16.8	31.1

Table 3. The comparison of LLM performance without retrieval (None Retrieval), One-Time Retrieval, and Active Retrieval from the diverse knowledge soup (execution feedback, code snippets, and documentation). Tensor-M denotes Tensorflow-M . The significant improvement enhanced by active retrieval implies the necessity for LLMs to iteratively refine the query and interact with the knowledge soup in generalized code generation.

Knowledge Soup Retrieval. The experimental results of knowledge soup retrieval are presented at the bottom of Table 2. It is evident that substantial improvements can be realized through the integration of diverse knowledge sources, exemplified by the notable advancements observed in the combination. For example, compared to single source retrieval with Code, Exec+Code yields a 8.9% improvement on the dataset Tensorflow-M for ChatGPT. Notably, the augmentation of LLMs with a comprehensive mix of documentation, code snippets, and execution feedback [5] consistently produces the best results across all datasets. The complementary strengths achieved by incorporating all available knowledge sources are particularly noteworthy. When compared to vanilla code generation, the combination of Doc+Code+Exec elevates the performance of ChatGPT from 8.6% to the impressive 34.6% on average. Similar patterns can be also observed in CodeLlama. This significant improvement underscores the effectiveness of leveraging diverse knowledge sources as a knowledge soup. In summary, the experimental results indicate that the diverse knowledge sources are likely to benefit from each other, and their integration leads to synergistic advantages.

[5]Noe that the exclusion of web search from our knowledge soup is deliberate, as it does not exhibit superior performance in our single source retrieval experiments.

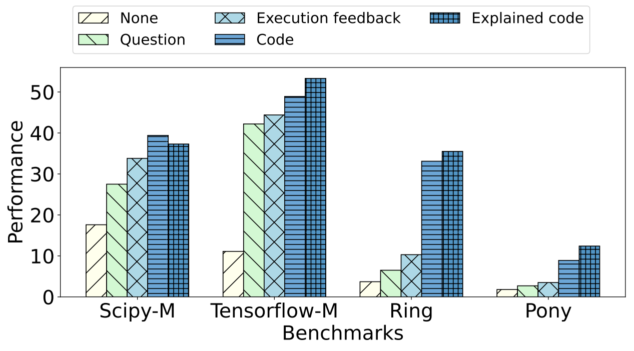


Figure 2. The generalization performance of ChatGPT when questions, execution feedback, code, or the explained code are used as the query to retrieve external knowledge. In general, the explained code is the most effective, enabling LLMs to achieve the best results in three out of four datasets.

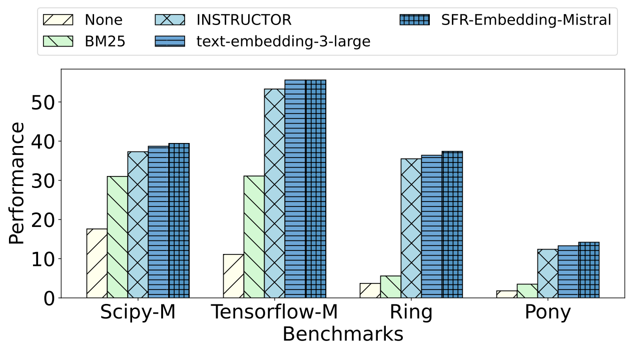


Figure 3. Comparison of ChatGPT generalization performance when the sparse retriever (BM25), or the dense retriever (INSTRUCTOR, text-embedding-3-large, SFR-Embedding-Mistral) is employed. The results show that dense retrievers significantly outperform their sparse counterpart, BM25. In general, ChatGPT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

3.2. Active Retrieval

Besides constructing a diverse knowledge soup, we demonstrate that it is critical to adopt an effective retrieval pipeline for generalizing LLMs. Table 3 shows that both ChatGPT and CodeLlama achieve significant improvement by actively refining the query and updating the knowledge soup.

Incorporating all of the execution feedback, code snippets, and documentation as the knowledge source, ChatGPT and CodeLlama improve the average accuracy by 10.6% and 8.7% with one-time RAG respectively. An additional substantial increase of 10.6% and 8.7% for both models is observed with continuous active retrieval. This demonstrates that, for LLMs to optimize the utilization of accessible data and attain superior efficacy in generalized code generation, it is essential to meticulously refine the query and actively update the knowledge in an iterative retrieval process. In Appendix D, we include more discussions about the trade-off between the performance and the cost.

3.3. Retrieval-Augmented Generation

In exploring how the generalization capabilities of LLMs are significantly influenced by the proficient acquisition of external knowledge, we investigate two primary design elements: the query formulation and the choice of the retrieval model. By default, we use ChatGPT with the optimal recipe of knowledge soup to include execution feedback, code snippets, and documentation (Table 2).

Query Formulation. To fairly compare the effectiveness of four query formulations introduced in §2.1, we experiment with each of them to retrieve from the diverse knowledge soup. Figure 2 demonstrates that the explained code is the most effective in general, which excels in three out of four datasets. Although each of the four formulations demonstrates efficacy in retrieving pertinent information, the notable disparities in LLM generalization performance underscore the considerable differences in their effectiveness. In particular, in the dataset Ring, the utilization of the question as the query enables ChatGPT to attain a mere 6.5% accuracy. In contrast, employing the explained code as the query substantially enhances its accuracy to 35.5%, marking a notable improvement of 29%. This implies that the query formulation plays a critical role in constructing a successful RACG pipeline.

Retrieval Model. We experiment with a representative sparse retriever, BM25, and several competitive dense retrievers. INSTRUCTOR (Su et al., 2022) is an embedding model fine-tuned to follow instructions for efficient adaptation. text-embedding-3-large[6] 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 3, 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. However, the gap between different dense retrievers is not significant. Throughout the experiment, we use the INSTRUCTOR as the default retrieval model for better efficiency.

Retrieval Accuracy Besides evaluating the effectiveness

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

	Scipy-M	Tensor-M	Ring	Pony	Average
BM25-R	42.3	45.6	9.4	7.8	26.3
BM25-G	31.0	31.1	5.6	3.5	17.8
INST-R	62.7	71.1	56.1	48.7	59.7
INST-G	37.3	53.3	35.5	12.4	34.6
text-R	66.2	77.8	63.6	52.2	65.0
text-G	38.7	55.6	36.4	13.3	36.0
SFR-R	69.1	77.8	59.8	54.0	65.2
SFR-G	39.4	55.6	37.4	14.2	36.7

Table 4. The retrieval accuracy (percentage of examples where all the gold documentation has been retrieved, denoted as ”-R”) and generalization performance (”-G”) of ChatGPT when BM25, INSTRUCTOR (INST), text-embedding-3-large (text) or SFR-Embedding-Mistral (SFR) is used as the retrieval model. There is a big gap between the two accuracies, indicating that in a lot of cases, even if the gold documentation is included in the prompt, ChatGPT still fails to generate correct code.

of the retrieval model using the generalization performance of ChatGPT, we investigate the retrieval accuracy, which is defined as the percentage of examples where all the gold documentation has been retrieved, i.e., ChatGPT has sufficient information to write the code. Table 4 demonstrates a big gap between the retrieval accuracy and ChatGPT generalization performance. This suggests that frequently, even when the gold standard documentation is part of the prompt, ChatGPT cannot generate accurate code. Furthermore, there is a notable positive correlation between the two measures of accuracy, meaning that improved retrieval accuracy is linked to enhanced generalization performance. In particular, when utilizing the BM25 retrieval model, ChatGPT attains a mere 3.5% accuracy in the Pony dataset, likely due to its insufficient retrieval accuracy of 7.8%, in contrast to the highest score of 54.0% obtained by using SFR-Embedding-Mistral. Therefore, the performance of LLM generalization is highly correlated to the preciseness of the retrieval model and the LLM’s capability to comprehend and integrate external knowledge in code generation.

3.4. 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[7] and GPT-4[8] have reached 200k and 128k tokens respectively, which questions 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.

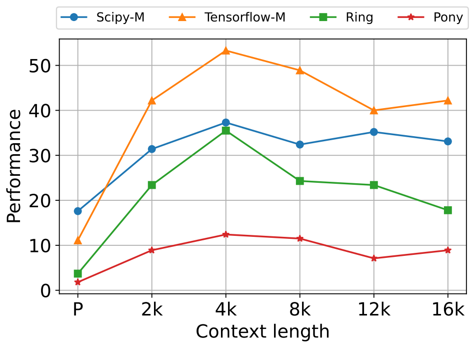


Figure 4. ChatGPT performance with various maximum allowed context lengths. P refers to the baseline where no external knowledge is included. Although the model supports the context length up to 16k, the results reveal that the execution accuracy ceases to enhance when the context window is expanded 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.

We use the explained code as the query and retrieve the top documentation up to 2k, 4k, 8k, 12k, and 16k tokens, which augments ChatGPT for code generation. Figure 4 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.

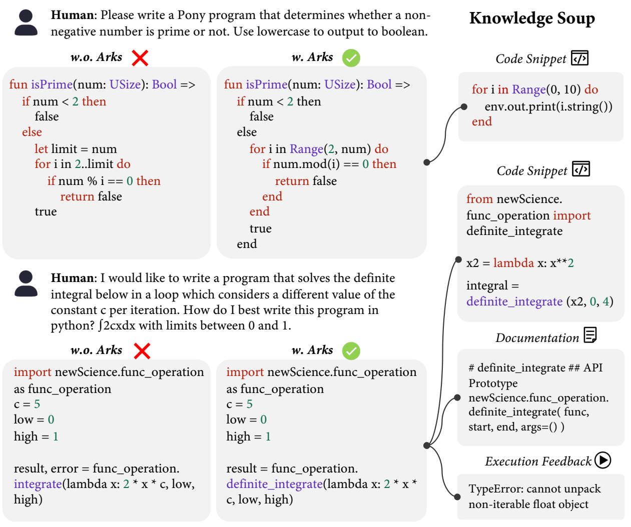
In summary, the naive addition of extensive knowledge through the utilization of larger context windows may not guarantee the enhancement of the effectiveness in RACG. We leave it to future research to design a more delicate retrieval system that can appropriately regulate the content utilized for LLM generation.

3.5. Case Study

To provide a more intuitive understanding of the improvement achieved by ARKS on top of LLMs, we present two different cases in Figure 5. The first showcases the transi-

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

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


Human: Please write a Pony program that determines whether a non-negative number is prime or not. Use lowercase to output to boolean.

```
fun isPrime(num: USize): Bool =>
  if num < 2 then
    false
  else
    let limit = num
    for i in 2..limit do
      if num % i == 0 then
        return false
    true
```

```
fun isPrime(num: USize): Bool =>
  if num < 2 then
    false
  else
    for i in Range(2, num) do
      if num.mod(i) == 0 then
        return false
      end
    end
    true
  end
```

```
for i in Range(0, 10) do
  env.out.print(i.string())
end
```

```
from newScience.
func_operation import
definite_integrate

x2 = lambda x: x**2
integral =
definite_integrate (x2, 0, 4)
```

Human: I would like to write a program that solves the definite integral below in a loop which considers a different value of the constant c per iteration. How do I best write this program in python? ∫2cxdx with limits between 0 and 1.

```
import newScience.func_operation
as func_operation
c = 5
low = 0
high = 1

result, error = func_operation.
integrate(lambda x: 2 * x * c, low,
high)
```

```
import newScience.func_operation
as func_operation
c = 5
low = 0
high = 1

result = func_operation.
definite_integrate(lambda x: 2 * x *
c, low, high)
```

```
# definite_integrate ## API
Prototype
newScience.func_operation.
definite_integrate( func,
start, end, args=() )
```

```
TypeError: cannot unpack
non-iterable float object
```

Figure 5. A comparison between model results without our method (*w.o.* ARKS) and with our method (*w.* ARKS) in the case study. The incorporation of the knowledge soup visibly improves the performance of the model.

tion from an incorrect to a correct state in the Pony dataset using code snippets alone. In contrast, the second example demonstrates how the generated code becomes correct by leveraging the knowledge soup on the Scipy-M dataset.

4. 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 pretrained on extensive code corpus have exhibited impressive abilities in the domain of code 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., 2023b; 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 Code Generation The retrieval-augmented generation (RAG) is an appealing paradigm that allows LLMs to efficiently utilize external knowledge (Shi et al., 2023; Izacard & Grave, 2020; Xu et al., 2023a; Jiang et al., 2023b). Recent works have applied it to the code generation task (Patel et al., 2023; Guo et al., 2023; Parvez et al., 2021; 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 APIRetriever 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. To the best of our knowledge, we are the first to adopt the active retrieval strategy with a diverse knowledge soup to explore the setting where LLMs need to incorporate external knowledge in code generation.

Code Execution Previous works have extensively employed executors (Interpreters/Compilers) in code-related tasks (Wang et al., 2022; Pi et al., 2022; Liu et al., 2023a; Olausson et al., 2023; 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 execution feedback both as a query to retrieve from the diverse knowledge soup and also as one of the information sources that can be directly incorporated into the LLM prompt for code repair.

5. Conclusion

Much recent work illustrated the ability of LLMs to incorporate external knowledge with retrieval-augmented generation. We present a comprehensive study on the implication of this ability for code generation in updated libraries and long-tail programming languages. The best pipeline explored in this paper, ARKS, adopts active retrieval in the diverse knowledge soup consisting of documentation, execution feedback, code snippets, and web search. Distinct from the traditional RAG, which usually retrieves from a fixed knowledge base and finishes generation in a single round, active retrieval iteratively refines the query and updates the knowledge soup. Our extensive experiments demonstrate substantial enhancement provided by active retrieval and diverse knowledge soup. Through an in-depth analysis, we further show the critical role of the query formulation and the retrieval model, and the inefficiency of large context windows in RACG. We hope that our findings will inspire researchers and practitioners to develop effective strategies in their customized code-generation tasks with LLMs.

6. Impact Statements

This paper presents work whose goal is to advance the field of Large Language Model code generation in updated libraries and long-tail programming languages. There are many potential societal consequences of our work, none of which we feel must be highlighted here.

References

Ali, M. S., Manjunath, N., and Chimalakonda, S. X-cobol: A dataset of cobol repositories. *arXiv preprint arXiv:2306.04892*, 2023.

Asai, A., Min, S., Zhong, Z., and Chen, D. Acl 2023 tutorial: Retrieval-based language models and applications. *ACL 2023*, 2023.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., van den Driessche, G., Lespiau, J., Damoc, B., Clark, A., de Las Casas, D., Guy, A., Menick, J., Ring, R., Hennigan, T., Huang, S., Maggiore, L., Jones, C., Cassirer, A., Brock, A., Paganini, M., Irving, G., Vinyals, O., Osindero, S., Simonyan, K., Rae, J. W., Elsen, E., and Sifre, L. Improving language models by retrieving from trillions of tokens. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvári, C., Niu, G., and Sabato, S. (eds.), *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pp. 2206–2240. PMLR, 2022. URL https://proceedings.mlr.press/v162/borgeaud22a.html.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Bugden, W. and Alahmar, A. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503*, 2022.

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

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Guo, Y., Li, Z., Jin, X., Liu, Y., Zeng, Y., Liu, W., Li, X., Yang, P., Bai, L., Guo, J., et al. Retrieval-augmented code generation for universal information extraction. *arXiv preprint arXiv:2311.02962*, 2023.

Guu, K., Lee, K., Tung, Z., Pasupat, P., and Chang, M. REALM: retrieval-augmented language model pre-training. *CoRR*, abs/2002.08909, 2020. URL https://arxiv.org/abs/2002.08909.

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

Huang, B., Lu, S., Chen, W., Wan, X., and Duan, N. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*, 2023.

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

Izacard, G., Lewis, P. S. H., Lomeli, M., Hosseini, L., Petroni, F., Schick, T., Dwivedi-Yu, J., Joulin, A., Riedel, S., and Grave, E. Atlas: Few-shot learning with retrieval augmented language models. *J. Mach. Learn. Res.*, 24:251:1–251:43, 2023. URL http://jmlr.org/papers/v24/23-0037.html.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023a.

Jiang, Z., Xu, F., Gao, L., Sun, Z., Liu, Q., Dwivedi-Yu, J., Yang, Y., Callan, J., and Neubig, G. Active retrieval augmented generation. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7969–7992, Singapore, December 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.495. URL https://aclanthology.org/2023.emnlp-main.495.

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. Dense passage retrieval for open-domain question answering. In Webber, B., Cohn, T., He, Y., and Liu, Y. (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL https://aclanthology.org/2020.emnlp-main.550.

Key, D., Li, W.-D., and Ellis, K. I speak, you verify: Toward trustworthy neural program synthesis. *arXiv preprint arXiv:2210.00848*, 2022.

Khattab, O., Santhanam, K., Li, X. L., Hall, D., Liang, P., Potts, C., and Zaharia, M. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *CoRR*, abs/2212.14024, 2022. doi: 10.48550/ARXIV.2212.14024. URL https://doi.org/10.48550/arXiv.2212.14024.

Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., Toutanova, K., Jones, L., Kelcey, M., Chang, M.-W., Dai, A. M., Uszkoreit, J., Le, Q., and Petrov, S. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, W.-t., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35: 21314–21328, 2022.

Li, C., Liang, J., Zeng, A., Chen, X., Hausman, K., Sadigh, D., Levine, S., Fei-Fei, L., Xia, F., and Ichter, B. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*, 2023a.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023b.

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

Liu, C., Lu, S., Chen, W., Jiang, D., Svyatkovskiy, A., Fu, S., Sundaresan, N., and Duan, N. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*, 2023a.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023b.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023c.

Menick, J., Trebacz, M., Mikulik, V., Aslanides, J., Song, H. F., Chadwick, M. J., Glaese, M., Young, S., Campbell-Gillingham, L., Irving, G., and McAleese, N. Teaching language models to support answers with verified quotes. *CoRR*, abs/2203.11147, 2022. doi: 10.48550/ARXIV.2203.11147. URL https://doi.org/10.48550/arXiv.2203.11147.

Muennighoff, N., Tazi, N., Magne, L., and Reimers, N. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*, 2022.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., von Werra, L., and Longpre, S. Octopack: Instruction tuning code large language models. *CoRR*, abs/2308.07124, 2023. doi: 10.48550/ARXIV.2308.07124. URL https://doi.org/10.48550/arXiv.2308.07124.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Olausson, T. X., Inala, J. P., Wang, C., Gao, J., and Solar-Lezama, A. Demystifying gpt self-repair for code generation. *arXiv preprint arXiv:2306.09896*, 2023.

Parvez, M. R., Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 2719–2734, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.232. URL https://aclanthology.org/2021.findings-emnlp.232.

Patel, A., Reddy, S., Bahdanau, D., and Dasigi, P. Evaluating in-context learning of libraries for code generation. *arXiv preprint arXiv:2311.09635*, 2023.

Pi, X., Liu, Q., Chen, B., Ziyadi, M., Lin, Z., Fu, Q., Gao, Y., Lou, J.-G., and Chen, W. Reasoning like program executors. In Goldberg, Y., Kozareva, Z., and Zhang, Y. (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 761–779, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.48. URL `https://aclanthology.org/2022.emnlp-main.48`.

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

Shao, Z., Gong, Y., Shen, Y., Huang, M., Duan, N., and Chen, W. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. *arXiv preprint arXiv:2305.15294*, 2023.

Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., and Wang, S. I. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*, 2022.

Shi, W., Min, S., Yasunaga, M., Seo, M., James, R., Lewis, M., Zettlemoyer, L., and Yih, W.-t. Replug: Retrieval-augmented black-box language models. *arXiv preprint arXiv:2301.12652*, 2023.

Shrivastava, D., Kocetkov, D., de Vries, H., Bahdanau, D., and Scholak, T. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*, 2023.

Su, H., Shi, W., Kasai, J., Wang, Y., Hu, Y., Ostendorf, M., Yih, W.-t., Smith, N. A., Zettlemoyer, L., and Yu, T. One embedder, any task: Instruction-finetuned text embeddings. *arXiv preprint arXiv:2212.09741*, 2022.

Walters, R., Fritchey, G., and Taglienti, C. Transact-sql. 01 2009. doi: 10.1007/978-1-4302-2414-3_5.

Wang, L., Yang, N., Huang, X., Yang, L., Majumder, R., and Wei, F. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*, 2023a.

Wang, W., Wang, Y., Joty, S., and Hoi, S. C. 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*, pp. 146–158, 2023b.

Wang, Z., Zhou, S., Fried, D., and Neubig, G. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*, 2022.

Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.

Xu, F., Shi, W., and Choi, E. Recomp: Improving retrieval-augmented lms with compression and selective augmentation. *arXiv preprint arXiv:2310.04408*, 2023a.

Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Xu, P., Ping, W., Wu, X., McAfee, L., Zhu, C., Liu, Z., Subramanian, S., Bakhturina, E., Shoeybi, M., and Catanzaro, B. Retrieval meets long context large language models. *arXiv preprint arXiv:2310.03025*, 2023b.

Yang, J., Prabhakar, A., Narasimhan, K., and Yao, S. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *arXiv preprint arXiv:2306.14898*, 2023.

Yasunaga, M. and Liang, P. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pp. 10799–10808. PMLR, 2020.

Zan, D., Chen, B., Lin, Z., Guan, B., Wang, Y., and Lou, J.-G. When language model meets private library. *arXiv preprint arXiv:2210.17236*, 2022.

Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., and Chen, W. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL `https://aclanthology.org/2023.emnlp-main.151`.

Zhang, K., Wang, D., Xia, J., Wang, W. Y., and Li, L. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *arXiv preprint arXiv:2305.14591*, 2023b.

Zhou, S., Alon, U., Xu, F. F., Jiang, Z., and Neubig, G. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2022.

Zhou, S., Alon, U., Xu, F. F., Jiang, Z., and Neubig, G. Docprompting: Generating code by retrieving the docs.

In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ZTCxT2t2Ru.

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 [9]. 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.

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. Private 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

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

	Model: ChatGPT				Model: CodeLlama			
Knowledge	Monkey	BeatNum	TorchData	Avg.	Monkey	BeatNum	TorchData	Avg.
None	38.6	27.7	42.0	36.1	80.2	70.3	54.0	68.2
Web	50.5	45.5	52.0	49.3	84.2	78.2	66.0	76.1
Exec	47.5	44.6	60.0	50.7	81.2	77.2	72.0	76.8
Code	63.4	50.5	68.0	60.6	92.1	79.2	88.0	86.4
Doc	59.4	68.3	62.0	63.2	90.1	86.1	84.0	86.7
Exec + Code	64.4	52.5	70.0	62.2	92.1	81.2	88.0	87.1
Exec + Doc	61.4	68.3	66.0	65.2	91.1	88.1	86.0	88.4
Doc + Code	66.3	70.3	72.0	69.5	93.1	89.1	92.0	91.4
Doc + Code + Exec	**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 a similar pattern as that in Table 2, the exceptionally high accuracy of CodeLlama 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.

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. Cost Analysis

Table 3 demonstrates the significant performance gain provided by active retrieval in both ChatGPT and CodeLlama. Despite significant enhancement 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

		Scipy-M		Tensorflow-M		Ring		Pony		Average	
Model	Retrieval	Acc	Tokens	Acc	Tokens	Acc	Tokens	Acc	Tokens	Acc	Tokens
ChatGPT	None	17.6	423	11.1	322	3.7	206	1.8	222	8.6	293
	Single-time-Q	25.4	3687	33.3	3728	4.7	3826	1.8	3763	16.3	3751
	Single-time	32.4	4826	33.3	4924	8.4	4528	2.7	4584	19.2	4716
	Active	37.3	13631	53.3	12568	35.5	24987	12.4	13819	34.6	16251
CodeLlama	None	11.3	476	17.8	381	0.0	263	0.0	314	7.3	386
	Single-time-Q	14.1	3923	33.3	4012	0.0	4050	0.0	3978	11.9	3991
	Single-time	16.9	5124	37.8	5023	4.7	4987	4.4	4823	16.0	4989
	Active	30.3	14564	51.1	12323	26.2	29384	16.8	14592	31.1	17716

Table 7. The comparison of LLM performance and consumed tokens per example without retrieval (None retrieval), single-time retrieval from the documentation using the question as query (Single-time-Q), single-time (Single-time) and active (Active) retrieval from the knowledge soup (execution feedback, code snippets, and documentation) using the explained code as the query. By default, we use INSTRUCTOR as the embedding model. Single-time-Q simulates the traditional RAG setting, where the question is directly used as the query to retrieve documentation. 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.

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.

E. 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 it is included in the prompt for LLM generation: (1). For web search content, include it until the maximum allowed length, e.g., 4096, 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 4096, we will allocate a maximum length of 300 to code snippets and a maximum length of 4096-300-400=3396 to the documentation.

F. 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 6, we present an example of the top-3 web search results returned by Google search to the query ”In the programing 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, mixing 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.

~~~~
Web content 1:
#  Minimum number of increment-other operations to make all array elements equal.

__ Report

We are given an array consisting of n elements. At each operation you can
select any one element and increase rest of n-1 elements by 1. You have to
make all elements equal performing such operation as many times you wish. Find
the minimum number of operations needed for this.

......

## What kind of Experience do you want to share?
 [ Interview Experiences ](https://write.geeksforgeeks.org/posts- new?cid=e8fc46fe-75e7-4a4b-be3c-0c862d655ed0)
[ Admission Experiences ](https://write.geeksforgeeks.org/posts- new?cid=82536bdb-84e6-4661-87c3-e77c3ac04ede)
[ Engineering Exam Experiences ](https://write.geeksforgeeks.org/posts- new?cid=fbed2543-6e40-4f77-b460-
e962cc55c315) [ Work Experiences ](https://write.geeksforgeeks.org/posts- new?cid=22ae3354-15b6-4dd4-a5b4-
5c7a105b8a8f) [ Campus Experiences ](https://write.geeksforgeeks.org/posts- new?cid=c5e1ac90-9490-440a-a5fa-
6180c87ab8ae) [ Add Other Experiences ](https://write.geeksforgeeks.org/#experiences)

Web content 2:
# Overview There are 3 normal tasks accompanied with 2 challenge tasks in div 1 as we usually do. You can check the
[ Statistics ](//codeforces.com/blog/entry/13271) by By [ DmitriyH ](/profile/DmitriyH "Expert DmitriyH") for detail. Problem
B, C is by [ sevenkplus ](/profile/sevenkplus "Grandmaster sevenkplus") , problem D is by [ xlk ](/profile/xlk "International
Master xlk") and problem A, E is by me.
......
[ Codeforces ](https://codeforces.com/) (c) Copyright 2010-2024 Mike Mirzayanov The only programming contests Web
2.0 platform Server time: Jan/29/2024 15:27:43 (h1). Desktop version, switch to [ mobile version ](?mobile=true) .

Web content 3:
Skip to main content Open menu Open navigation [ ](/) Go to Reddit Home r/adventofcode A chip A close button Get
app Get the Reddit app [ Log In ](https://www.reddit.com/login) Log in to Reddit
......
### Get the Reddit app Scan this QR code to download the app now Advent of Code is an annual Advent calendar of
small programming puzzles for a variety of skill sets and skill levels that can be solved in any programming language you
like.

Problem:
Write a pony function to find the largest and smallest number from an array.
Wrap your code with ```.
```
fun minMax(arr: Array[ISize]): (ISize, ISize) =>
...
(min, max)
```
~~~~

Figure 6. 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.