CODERAG-BENCH: Can Retrieval Augment Code Generation?

Anonymous ACL submission

Abstract

While language models (LMs) excel at generating code, many programs are difficult to generate using only parametric knowledge. Despite the success of retrieval-augmented generation (RAG) in text-centric tasks, its potential for code generation remains under-explored. This work introduces CODERAG-BENCH, a holistic retrieval-augmented code generation benchmark covering tasks like basic programming, open-domain, and repository-level problems and provide reproducible evaluations on both retrieval and end-to-end code generation performance. We further create a diverse, open datastore for code retrieval, aggregating sources such as competition solutions, tutorials, library documentation, StackOverflow posts, and GitHub repositories. Based on CODERAG-BENCH, we conduct large-scale evaluations of 10 retrievers and 10 LMs and systematically analyze when retrieval can benefit code generation models and identify remaining challenges. We find that while retrieving high-quality contexts improves code generation, retrievers often struggle to fetch useful contexts, and generators face limitations in using those contexts effectively. We hope CODERAG-BENCH encourages further development in code-oriented RAG methods.[1]

1 Introduction

Generating code from natural language has rapidly advanced with language models (LMs; Chen et al. 2021; Li et al. 2022, 2023; Roziere et al. 2023). However, most models follow an NL (Natural Language)-to-code approach without integrating external context, which is crucial in complex scenarios like using unfamiliar libraries (Zhou et al., 2023; Jimenez et al., 2024). Relying solely on parametric knowledge also limits adaptation to new data distributions at test time, such as evolving public libraries or private code bases not seen during training (Zhang et al., 2023; Jimenez et al., 2024).

Retrieval-augmented generation (RAG; Lewis et al. 2020; Guu et al. 2020) addresses this by retrieving relevant documents at inference time, reducing reliance on model parameters (Asai et al., 2024) and improving accuracy across tasks (Izacard et al., 2022b). Despite success in text-based tasks, its application to diverse coding problems and retrieval sources remains under-explored (Zhou et al., 2023; Su et al., 2024).

We present CODERAG-BENCH, a holistic benchmark designed to advance research in retrieval-augmented code generation (RACG; §2). CODERAG-BENCH (as in Figure 1) covers six programming tasks across four categories: basic programming, open-domain coding, repository-level, and code retrieval tasks. For each task, we manually annotate canonical documents as references for evaluating RACG systems. We also compile a diverse corpus of documents from five sources: programming solutions, online tutorials, Python library documentation, StackOverflow posts, and GitHub files. In total, CODERAG-BENCH has 9k coding tasks and 25 million retrieval documents, providing a robust foundation for reproducible and reliable evaluations in retrieval and RACG.

We conduct holistic evaluations in retrieval, generation, and RACG (§3). Code generation models significantly benefit from access to canonical documents (i.e., from the canonical retrieval corpus) in various scenarios. For example, GPT-4o achieves a 27.4% gain on SWE-Bench and a 6.9% gain on the harder ODEX subset when canonical documents are provided. In RACG settings, where models retrieve top relevant documents, some even surpass their performance when using gold documents, highlighting the strong potential of retrieval-augmented approaches for enhancing code generation. However, current retrieval models face challenges in selecting useful documents, particularly

[1] Our code is available at https://anonymous.4open.science/r/code-rag-bench-F383.

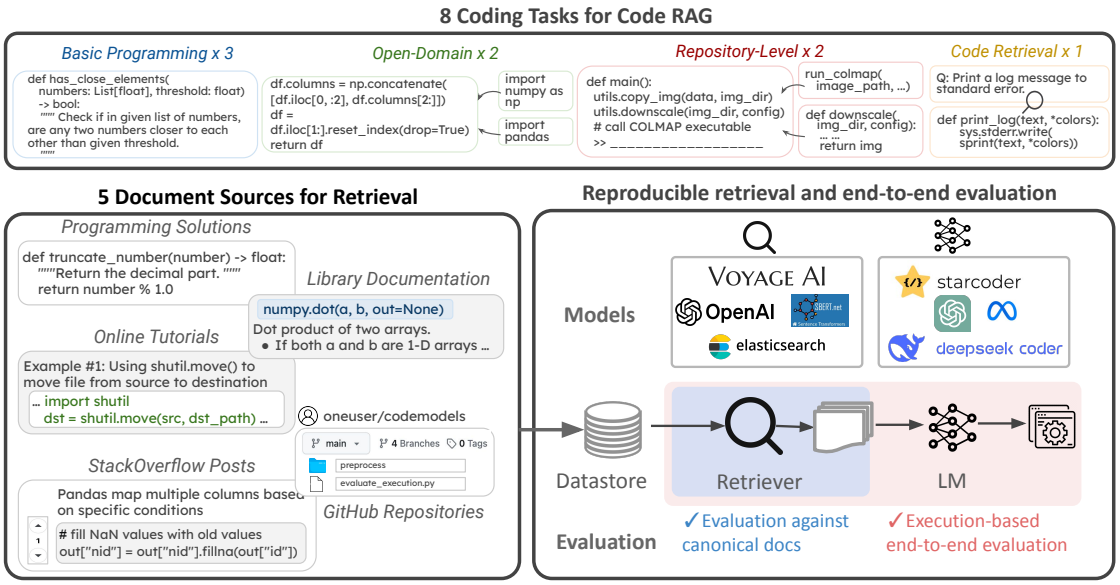

Figure 1: Overview of CODERAG-BENCH.

for open-domain and repository-level tasks. Additionally, generation models with limited context windows exhibit smaller improvements, suggesting considerable room for future advancements.

Beyond canonical retrieval, we also explore RACG with open retrieval, i.e., retrieving documents from various sources with different chunking strategies (§4). We find that models can benefit from functionally relevant snippets from certain sources, and chunking documents to 200–800 tokens often gives the best results. For instance, by retrieving from StackOverflow or online tutorials, both StarCoder and GPT4o can significantly improve, while on repository-level tasks, the gains are rather limited. Overall, we hope CODERAG-BENCH can serve as a testbed for future work exploring, analyzing, and improving RACG systems.

2 The CODERAG-BENCH

For CODERAG-BENCH (Figure 1), the curation is driven by three factors: (i) **Diverse tasks**: Code generation spans multiple levels (line, function, repository) across closed and open domains. (ii) **Rigorous evaluation**: We offer high-quality ground-truth annotations for retrieval and execution-based evaluation to measure functional correctness. (iii) **Unified interface**: Our codebase provides a consistent interface for retrieval, augmented generation, and evaluation, unlike current datasets with varied pipelines.

In this section, we introduce the creation process of CODERAG-BENCH: programming problem integration (§2.1), retrieval source collection (§2.2), canonical document annotation (§2.3), and the evaluation pipeline (§2.4). Examples with canonical documents are available in §A.

2.1 Programming Problems

We categorize existing Python-based coding datasets into four types:[2] code retrieval, basic programming, open-domain problems, and repository-level problems. To ensure the diversity of datasets, we choose and unify multiple frequently adopted datasets for each category, as listed in Table 1.

Basic programming problems This category includes interview-style problems that mostly require Python built-in operations and pose algorithmic challenges. We select the two most widely used datasets: HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), which ask the model to complete a function from an NL problem description. However, due to limited public knowledge about model training data, it is unclear whether models suffer from data contamination on HumanEval and MBPP (Jain et al., 2024). Hence, we also include LiveCodeBench (Jain et al., 2024) with problems collected from coding websites after the training cutoff of LMs that we consider, to decrease the risk of contamination.

Open-domain problems Open-domain coding problems require Python libraries beyond the standard libraries used in basic programming problems. We adopt the DS-1000 (Lai et al., 2023) and ODEX (Wang et al., 2023b) datasets that cover data-science and general open-domain coding problems. DS-1000 collects data science problems

[2] In this work we focus on Python-related tasks because it is the most widely-used programming language for benchmarking code generation. We leave extensions to other programming languages for future work.

Type	Dataset	# Examples	# Corpus	Ground-Truth Docs	Evaluation
Basic programming	HumanEval	164	164	program solutions	execution
	MBPP	500	500	program solutions	execution
	LiveCodeBench	400	-	-	execution
Open-domain	DS-1000	1000	34,003	docs	execution
	ODEX	945	34,003	docs, stackoverflow	execution
Repository-level	RepoEval (function)	373	237	github repository	execution
	SWE-bench-Lite	300	40,868	github repository	execution
Code retrieval	CodeSearchNet-Py	22,177	22177	CSN functions	ndcg@10

Table 1: Overview of the datasets in CodeRAG-Bench. CSN stands for CodeSearchNet.

with programs using seven common data-related libraries such as `pandas` and `numpy`. ODEX covers problems using a broader range of 79 libraries, such as web requests with `requests` and database operations with `sqlalchemy`.

Repository-level coding problems Beyond function-level, some problems require editing files in the context of an entire GitHub repository. We thus adopt RepoEval (Zhang et al., 2023) and SWE-bench (Jimenez et al., 2024) for repository-level code generation and issue-solving tasks. We integrate all three splits of RepoEval but only report its function split, as it is the only split supporting execution-based evaluation.[3] Notably, our codebase is the first to enable reproducible execution evaluation on RepoEval. SWE-bench focuses on resolving GitHub issues by asking models to edit multiple files that pass the required test cases. We use SWE-bench-Lite,[4] a 300-problem subset whose results can be reproduced, with a packaged Docker container (Wang et al., 2024).

Code retrieval problems In addition to retrieval for augmenting generations, we adopt the Python split of CodeSearchNet (CSN) as a code retrieval task. CSN searches for the correct implementation of an NL query from a pool of functions collected from GitHub repositories. Instead of monitoring how generation changes with various retrieval results, CSN can directly measure retrieval quality.

2.2 Retrieval Sources

We collect retrieval documents from five commonly used resources for program developers, listed in Table 2. CODERAG-BENCH supports two retrieval setups: **canonical retrieval**—retrieves documents from only the canonical datastore (§2.3), and **open retrieval**—retrieves documents from any datastore.

Programming solutions We create one document from each basic programming problems that have canonical solutions (i.e., HumanEval and MBPP), following VoyageAI (2024), by concatenating its NL problem and program solution.

Online tutorials We collect tutorials from multiple websites including GeeksforGeeks, W3Schools, tutorialspoint, and Towards Data Science,[5] via the raw HTML pages obtained from ClueWeb22 (Overwijk et al., 2022), a large-scale crawled web corpus. Each page contains code snippets and their text explanations, covering topics from basic programming techniques to advanced library usage.

Library documentation We collect the official documentation provided by `devdocs.io` for all Python libraries following (Zhou et al., 2023). These could be especially useful for open-domain and repository-level problems that use some library functions to realize complex setups.

StackOverflow posts StackOverflow (SO) is among the most frequently visited sites for developers. We collect all SO posts from the RedPajama-1T (Computer, 2023) `stackexchange` split. We treat each post as a retrievable document, that has a question, code responses, and textual explanations.

GitHub repository We collect high-quality repositories from GitHub, using the `github` split of RedPajama-1T (Computer, 2023), as developers often refer to popular repositories when writing their programs. Following this practical paradigm, we enable LMs to retrieve files from other repositories as contexts to write the current program.

2.3 Canonical Document Annotation

To ensure reliable retrieval evaluation and estimate the upper bound of a RACG system with an ideal retriever, it's essential that all examples include *canonical documents*–the documents containing the necessary context to solve the programming

[3] Two other splits (API and line) are evaluated by lexical measures that have been shown as ineffective in signifying functional correctness (Chen et al., 2021; Wang et al., 2023b).

[4] https://www.swebench.com/lite.html

[5] https://geeksforgeeks.org; https://www.w3schools.com/; https://www.tutorialspoint.com/; https://towardsdatascience.com

Resource	Corpus size	Avg. length
Programming solutions	$1.1k$	194.6
Online tutorials	$79.4k$	1502.5
Library documentation	$34k$	953.4
StackOverflow posts	$23.5M$	689.2
Github files	$1.7M$	5135.4

Table 2: Five sources to form our retrieval datastore.

problem. As most existing datasets lack these canonical documents, we annotate them from the corresponding retrieval pool, as shown in Table 1.

Basic programming problems The canonical document for examples in HumanEval and MBPP is the documents we created in §2.2 in the *programming solutions* pool. Since LiveCodeBench does not provide solutions to its problems, we do not annotate canonical documents for it.

Open-domain problems Since open-domain problems require libraries, we annotate the canonical *library documentation* for DS-1000 and ODEX examples. We first automatically parse out the library functions used in each program, and find their corresponding documentation entries. Then, we manually verify the functions and remove incorrect ones. This yields an average of 1.4 and 1.2 entries for DS-1000 and ODEX.

Repository-level problems We adopt *canonical code* from the original dataset as our canonical documents: 20-line code snippets of the missing functions in RepoEval, and the ground-truth edited files in SWE-bench. We obtain these from the completed local repositories from the original datasets.

2.4 Evaluation Metrics

For retrieval, we evaluate NDCG, Precision and Recall (Thakur et al., 2021) and use NDCG@10 percentage as our primary metric, following prior work (Izacard et al., 2022a). For code generation, we adopt the pass@k metric (Chen et al., 2021) to measure the execution correctness of programs. We evaluate the final RAG performance both in canonical and open retrieval setups.

3 Canonical RACG

We evaluate 10 top retrieval and 10 generation models on CODERAG-BENCH with canonical data sources. We report results of document retrieval (§3.2), direct NL-to-code generation (§3.3), and end-to-end RACG with retrieved context (§3.4).

3.1 Experimental Setup

Retrieval baselines We adopt 10 top-performing retrievers from three categories: sparse, dense, and proprietary APIs. For sparse retrievers, we use BM25 (Robertson and Zaragoza, 2009), known for its robustness in domain adaptation (Thakur et al., 2021). Dense retrievers include BGE-base/large (Xiao et al., 2023), GIST-base/large (Solatorio, 2024), and SFR-Embedding-Mistral (Meng et al., 2024), all top-ranked on the MTEB leaderboard (Muennighoff et al., 2022). We also include open code embedding models, Codesage-small (Zhang et al., 2024) and Jina-v2-code Günther et al., 2023, which are specifically trained for code retrieval. Proprietary APIs include `voyage-code-2`(VoyageAI, 2024), optimized for code retrieval, and `openai-text-embedding-small-03`, selected for its cost-effectiveness. Finally, we apply reranking with BGE-reranker-base(Xiao et al., 2023) on top-100 `openai` results before generation.

Generation baselines We adopt both code-specific LMs and strong general text-oriented LMs. For code-specific LMs, we use StarCoder2 (Lozhkov et al., 2024), CodeGemma (Team, 2024), CodeLlama (Roziere et al., 2023), and DeepSeek-Coder (Guo et al., 2024) in various sizes. For general text LMs, we include three top-performing models: Llama3 (Meta, 2024), Command-R (CohereAI, 2024) specially optimized for RAG, and proprietary GPT models `gpt-3.5-turbo-0125` and `gpt-4o`. We use the instruct version of all generation models if available, since they often perform better than the base versions.

Experimental setup For retrieval, we implement BM25 retrievers using pyserini (Lin et al., 2021) with parameter $k_1 = 1.2$ and $b = 0.75$, and use sentence-transformers (Reimers and Gurevych, 2019)[6] for all dense models with open checkpoints. We prepend the top-5 retrieved documents to the original problems (we study the number of documents in §E), and do not include other contexts such as few-shot examples. For code generation, we use temperature $t = 0.2$, $top_p = 0.95$ and sample one response for all generations. Specifically on SWE-bench-Lite, we adopt the $n = 21$ way sampling and majority-vote reranking strategy proposed by Agentless (Xia et al., 2024).[7]

3.2 Retrieval Results

Table 3 shows retrieval results on six tasks.

[6] https://sbert.net/

[7] We found that without these approaches, performance even with state-of-the-art GPT4o remains around 1-2%.

Method	Problem Solutions			Library Docs		In-Repository Files		Avg.
	HumanEval	MBPP	CSN	DS-1000	ODEX	RepoEval	SWE-bench-Lite	All
BM25	**100.0**	98.6	89.1	5.2	6.7	93.2	43.0	57.7
GIST-base (768)	98.0	98.0	89.9	12.0	12.1	81.2	46.8	58.0
GIST-large (1024)	**100.0**	98.9	89.6	13.6	28.0	82.9	47.8	61.7
BGE-base (768)	99.7	98.0	90.0	10.8	22.0	77.5	44.9	58.8
BGE-large (1024)	98.0	**99.0**	90.6	8.9	11.5	80.4	40.1	56.3
SFR-Mistral (4096)	**100.0**	**99.0**	-	19.3	**37.1**	83.8	**62.7**	**67.0**
Codesage-small (768)	100.0	96.3	**90.7**	8.9	14.3	94.1	47.1	60.1
Jina-v2-code (768)	**100.0**	97.7	-	26.2	19.9	90.5	58.3	65.4
OpenAI-03 (1536)	**100.0**	98.9	-	18.2	16.5	93.0	43.3	61.7
Voyage-code (1536)	**100.0**	**99.0**	-	**33.1**	26.6	**94.3**	29.1	63.7

Table 3: Retrieval performance (NDCG@10) on code generation datasets. LiveCodeBench is excluded due to lack of ground-truth solutions. RepoEval is at the function level with 2k context tokens. Embedding dimension sizes are listed next to method names. Bold indicates best performance, underline indicates second-best. Highlighted models are specifically trained for code domains. Avg. reflects overall scores, excluding CodeSearchNet.

Comparison of lexical and neural retrievers BM25 has been widely used as a primary retrieval model in recent RACG work (Zhou et al., 2023; Jimenez et al., 2024), yet comprehensive comparisons against diverse retrieval systems are often under-explored. While prior studies indicate that neural retrieval systems often underperform BM25 baselines in out-of-domain scenarios (Thakur et al., 2021), our analysis of CODERAG-BENCH reveals that dense embedding models frequently surpass BM25. We hypothesize that this is because many competitive retrieval models are trained on diverse tasks across various domains, including code data (Asai et al., 2023; Su et al., 2023), enhancing their robustness in code retrieval setups.

Do code retrieval models perform better? At similar parameter scales, models specifically trained for code retrieval tasks typically show superior performance. Notably, Jina-v2-code outperforms GIST-base and BGE-base by 7.4 and 6.6 average NDCG@10, respectively, while Voyage-code significantly outperforms OpenAI-03.

Do larger retrieval models perform better? Among dense retrieval models, increasing model size often leads to better retrieval performance, similar to the trends observed in LMs (Brown et al., 2020). In particular, GIST-large ($340M$) constantly outperforms GIST-base ($110M$), and SFR-Mistral ($7B$) achieves the best among all open sparse and dense models on all tasks, surpassing proprietary embedding models on several tasks.

Efficiency While larger retrieval models often outperform smaller ones, they often introduce significant costs. We analyze efficiency, focusing on (i) *encoding latency*: latency to encode documents offline, and (ii) *search latency*: latency to encode queries/documents and calculate their similarities, (iii) *model storage requirements*, and (iv) *index storage requirements*. We conduct efficiency analysis on sampled CodeSearchNet Python data.[8] See experimental details in §B. As shown in Table 4, BM25 indexing and searching takes only seconds to finish. Compared to base-size GIST-base, the SFR-Mistral model is more powerful in retrieval, yet requires over 5× larger index storage, and adds nearly 100× latency to encode documents, suggesting that the efficiency aspect should also be carefully studied for RAG pipelines.

Method	Encoding	Search	Model	Index
BM25	0.15ms	0.02ms	-	141MB
GIST-base	3.7ms	9.7ms	440MB	307MB
GIST-large	13ms	18ms	1300MB	409MB
SFR-Mistral	316ms	113ms	14220MB	1638 MB
Voyage-code	22ms	40ms	-	1172MB
OpenAI-03	31ms	47ms	-	1172MB

Table 4: Efficiency analysis for document retrieval.

3.3 Generation with Canonical Documents

We first evaluate possible lower- and upper-bounds on RACG results by testing generation (i) without any retrieval, and (ii) with ground-truth documents. We report both results in Table 5. Compared to the base generation without contexts, incorporating canonical contexts improves in most setups, and substantially so on *basic programming* problems.

On open-domain tasks, most code-specific LMs increase up to 5.2 points, signifying that most models can benefit from indirectly helpful documents. In contrast, GPTs show no gains with retrieval. We hypothesize that this is because both datasets mostly test on common Python libraries, which

[8] Due to the costs, we randomly sample 10k queries and 100k from CodeSearchNet Python split. For API models, we use a batch size of 64 for encoding.

Method	Basic Programming					Open-Domain						Repo-Level			
	HumanEval		MBPP		LCB	DS-1000		ODEX		ODEX-hard		RepoEval		SWE-bench	
	w/o	gold	w/o	gold	w/o	w/o	gold	w/o	gold	w/o	gold	w/o	gold	w/o	gold
StarCoder2-7B	31.7	94.5	10.4	34.8	1.5	29.2	30.0	14.6	17.5	10.3	17.2	26.5	42.0	0.0	0.7
CodeGemma-7B	49.4	77.4	48.0	52.2	21.5	20.1	19.8	18.9	18.2	13.8	13.8	24.7	32.2	0.0	0.3
CodeLlama-7B	34.8	87.2	23.8	42.8	13.5	21.8	26.1	35.8	41.0	27.6	31.0	24.1	38.3	0.0	0.0
CodeLlama-34B	42.7	84.8	51.2	88.0	5.8	34.7	37.0	34.9	38.0	17.2	27.6	29.8	42.6	0.0	0.0
DeepSeekCoder-7B	70.1	87.8	60.8	63.6	30.5	41.4	43.2	39.2	41.7	17.2	24.1	28.2	43.7	0.0	0.0
DeepSeekCoder-33B	78.0	95.7	61.0	92.2	33.8	40.2	40.1	28.0	28.9	24.1	31.0	32.4	45.3	0.3	0.7
Llama3-8B	57.9	65.2	35.6	52.8	2.8	28.9	31.1	37.4	33.7	13.8	17.2	26.0	43.2	0.0	0.3
Command-R	43.3	51.2	37.2	37.8	10.0	25.8	28.5	35.5	36.0	20.7	20.7	23.9	37.0	0.0	0.3
GPT-3.5-turbo	72.6	91.5	70.8	72.6	35.3	43.7	42.9	41.7	40.3	17.2	24.1	23.9	39.1	0.7	6.3
GPT-4o	75.6	92.6	79.4	81.4	43.8	52.7	51.2	44.6	44.2	20.7	27.6	32.4	46.1	2.3	30.7

Table 5: Code generation pass@1 without additional contexts (*w/o*), and with ground-truth documents (*gold*). We only report *w/o* for LCB because LCB does not have ground-truth documents. We highlight results showing *gold* > *w/o* with green (darker green when having 10+ increases), and with red if *gold* < *w/o*.

powerful models may have already memorized, similar to their memorization of factual knowledge (Mallen et al., 2023; Kandpal et al., 2023), thereby reducing the need for retrieval. To verify this hypothesis, we build an ODEX subset of examples with the 20 least used libraries, i.e., **ODEX-hard**. As shown in Table 5, adding documents retrieved with most methods improves the results by 20.3–40.1%, showing the effectiveness of RACG on challenging coding tasks using unfamiliar libraries.

Repository-level challenges All models show gains of 7.5–17.2 points with canonical snippets in RepoEval, but SWE-bench Lite proves much more challenging — only GPT-3.5-turbo and GPT-4o achieve non-trivial results, consistent with previous findings (Yang et al., 2024). Notably, GPT-4o shows a 27.4% increase when using gold documents on SWE-bench, indicating that retrieval significantly enhances performance when paired with strong core generation capabilities, even in highly challenging coding tasks.

3.4 Retrieval-Augmented Code Generation

We now experiment with top-performing retrieval and generation models in the full RACG setting, which requires both retrieve documents and generating conditioned on the documents. We select the best retrieval models from each type: BM25, GIST-large, Voyage, and OpenAI embeddings. For generation, we select (i) StarCoder2-7B: a weaker model that benefits the most from contexts; (ii) DeepSeekCoder-7B: one of the strongest open code LMs; and (iii) GPT-3.5-turbo: one of the top proprietary models. For each dataset, we retrieve the most relevant contexts from its canonical source marked in Table 1, and retrieve *programming solutions* for LiveCodeBench. Table 6 shows the results. Note that we exclude canonical docs (answers) from the retrieval corpora for basic programming tasks.

Overall, the best retrieval models vary depending on the task and underlying LMs. In some cases, top-performing retrieval models do not lead to the best RACG outcomes, highlighting the need to evaluate RACG systems holistically across varied tasks.

Basic programming problems Most retrieved contexts help StarCoder2 generations. On MBPP, RACG even outperforms canonical setup by 15.6–17.8. However, RACG does not improve DeepSeekCoder generations, which we observe is due to over-complicated and ungrammatically repetitive generations when with additional contexts. In comparison, GPT-3.5-turbo can effectively improve with added contexts, showing its better ability to leverage augmented contexts.

Open-domain problems The weaker StarCoder2 benefits from retrieved library documentation across all datasets, while DeepSeekCoder and GPT-3.5 show gains mainly on ODEX-hard problems. This aligns with findings from the canonical document setup, indicating that RACG is particularly effective for less popular libraries. Interestingly, despite relatively low NDCG@10 scores, the best-performing RACG combinations match their canonical results on ODEX-hard.

Repository-level problems All models show improvements with retrieved code snippets on RepoEval, with RACG using strong retrievers like `openai-embeddings` performing on par with—or even surpassing—the canonical setup, likely due to the additional context provided to the models. On SWE-Bench, the best-performing combination, Retrieval-then-Rerank and GPT4o, yields a 21-point improvement over the no-retrieval baseline. However, there remains a 9-point gap compared to the gold setup, indicating room for improvement on the retrieval side, as reflected in the limited code retrieval performance shown in Table 3.

Method	Basic Programming			Open-Domain			Repo-Level	
	HumanEval	MBPP	LCB	DS-1000	ODEX	ODEX-hard	RepoEval	SWE-bench
w/ StarCoder2-7B								
None	31.7	2.4	1.5	29.2	14.6	10.3	26.5	0.0
BM25	**43.9**	51.8	1.0	**36.7**	14.1	13.8	36.7	0.0
GIST-large	38.7	50.4	0.5	35.9	**17.3**	13.8	40.8	0.3
Voyage, code	39.0	52.6	0.3	36.0	15.3	10.3	45.8	0.3
OpenAI, small	39.0	52.6	**1.5**	35.5	15.9	**17.2**	51.2	0.0
OpenAI, rerank	34.8	**53.4**	0.5	33.4	14.1	**17.2**	**53.9**	0.3
Gold	94.5	34.8	-	30.0	17.5	17.2	42.0	0.7
w/ DeepseekCoder-7B-instruct								
None	70.1	60.8	30.5	41.4	39.2	17.2	28.2	0.7
BM25	**68.9**	60.0	31.8	**36.6**	37.8	20.7	37.3	0.0
GIST-large	66.3	56.6	**33.8**	35.9	34.9	20.7	44.5	0.3
Voyage, code	66.5	56.4	31.8	35.9	**39.4**	17.2	46.6	0.3
OpenAI, small	**68.9**	58.6	32.0	35.5	37.1	20.7	55.2	0.3
OpenAI, rerank	53.0	**60.6**	31.5	36.5	37.1	**24.1**	**55.5**	0.3
Gold	87.8	63.6	-	43.2	41.7	24.1	48.1	0.0
w/ GPT-3.5-turbo								*GPT-4o*
None	72.6	70.8	35.3	43.7	41.7	17.2	23.9	0.7
BM25	73.2	72.4	**35.5**	36.9	**41.0**	**24.1**	30.8	6.7
GIST-large	73.2	68.2	34.8	36.7	36.2	13.8	38.3	19.3
Voyage, code	**75.0**	66.8	34.5	**37.4**	**41.0**	20.7	43.2	15.7
OpenAI, small	73.8	68.4	35.8	36.9	40.3	17.2	48.0	21.0
OpenAI, rerank	64.0	**72.6**	33.5	**37.4**	40.5	17.2	**49.6**	**21.7**
Gold	91.5	72.6	-	42.9	40.3	24.1	39.1	30.7

Table 6: Performance of retrieval-augmented code generation, with top retrieval and generation models. We bold-type the best RACG results. We test gpt-4o on SWE-bench to show non-trivial results than gpt-3.5-turbo. Note that we exclude code canonical answer from the retrieval corpora for basic programming tasks.

Model	Retriever	HumanEval							ODEX						
		w/o	Prog	Tut	Docs	SO	GitHub	All	w/o	Prog	Tut	Docs	SO	GitHub	All
StarCoder	BM25	31.7	97.6	27.4	**29.3**	32.9	30.5	97.6	14.6	18.2	13.4	14.1	**11.6**	15.9	16.2
	GIST		67.1	**34.8**	26.7	32.3	**32.9**	69.1		14.6	**15.7**	**17.3**	11.4	15.5	**17.1**
	OpenAI		97.6	29.3	24.4	**36.0**	31.1	97.6		**18.7**	14.1	15.9	10.9	**16.9**	15.3
GPT-4o	OpenAI	75.6	94.5	90.2	90.9	91.5	84.8	95.1	44.6	**49.2**	44.2	47.6	40.3	39.4	39.6

Table 7: Comparing five retrieval sources on HumanEval and ODEX, using StarCoder2 (top) and GPT-4o (bottom).

4 RACG with Open Retrieval

Besides retrieving documents from the canonical source, we explore RACG with open retrieval from all sources (§2.2) on three category-representative datasets: HumanEval, ODEX, and RepoEval. We also study *mixed* retrieval with documents from all sources, where we aggregate the top-1 documents from all five sources as augmented contexts.[9] We use the three top retrievers along with StarCoder2 and OpenAI retrieval with GPT-4o generation, to study open RACG with weak and strong LMs.

General programming: HumanEval Among all sources, SO posts can improve the results by 1.8–4.3, regardless of the choice of retrievers. Tutorials can improve results by 2.1 only with the GIST retriever.From manual examinations of the results, many retrieved posts and tutorials are *about the same programming problem* as the HumanEval example, with code and detailed textual explanations, hence could hint or disclose the answer. Other retrieval sources do not often contain relevant content thus do not bring improvements. Surprisingly, generation with mixed documents performs as well as using the gold documents, suggesting that the model can *discern and integrate the most useful content* from a mixture of texts.

Open-domain: ODEX Programming solutions are the most helpful source by bringing 3.8–4.3 gains, even surpassing gains of canonical documentation. Notably, both GPT-4o and StarCoder using OpenAI retrieval from programming solutions, outperform their variants retrieving from documentation by 3.2 and 1.6 points. Although the retrieved content is only sometimes functionally relevant to the ODEX examples, they can *exemplify the correct usage* of libraries such as `regex` in solutions and `requests` in GitHub files, thus guiding the generation to be more functionally correct. Similar to HumanEval, GIST-large is particularly good

[9] We use the first 500 tokens of each document for all experiments in this section, which we show to be optimal in ablation studies (§4), and satisfies all model context limits.

at retrieving tutorials, while BM25 and OpenAI embeddings find higher-quality program solutions, indicating their respective domain advantages.

Repository-level: RepoEval Open sources are less useful than code snippets in the local repository. Understanding local code contexts is crucial and irreplaceable than external resources. When using both local and open-source contexts (*L+O*), models surpass the no-retrieval baseline, yet are still only comparable with *Local*, suggesting more efforts and insights to benefit from both sources.

Method	w/o	Local	Prog	Tut	Docs	SO	GitHub	Open	L+O
				StarCoder2-7B					
BM25		36.7	23.6	25.2	23.9	23.6	25.5	23.6	31.4
GIST	26.5	40.8	24.1	23.3	21.7	24.7	24.4	24.1	41.8
OpenAI		51.2	23.9	24.1	24.1	23.1	22.8	24.9	**50.9**
				GPT-4o					
OpenAI	32.4	62.2	35.4	28.7	27.8	29.0	28.2	30.3	54.2

Table 8: RACG with open retrieval on RepoEval.

Exploring optimal chunking strategies Adding many documents may exceed model context limits hence impairing RACG, we thus explore various chunking strategies to better integrate retrieval. Compared to the no-chunking baseline, we study (i) post-retrieval chunking that takes the first N-tokens of each document, (ii) post-retrieval with reranking using BGE-reranker-base (§3.1) to find the most relevant N-token chunk from each document, and (iii) pre-retrieval chunking that chunks documents beforehand and retrieves N-token pieces directly.[10]

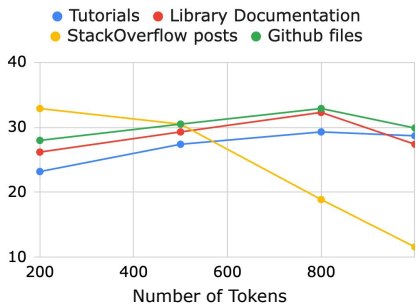

Figure 2: Performance with different chunking sizes.

We compare (i) using the first N-tokens for N from 200 to 1500 (Figure 2). Most sources are best represented by the first 800 tokens except for SO posts. However, we find (ii) reranking within this optimal range of 200–800 tokens greatly degrades the results, showing limited utility of current rerankers. Lastly, (iii) pre-retrieval achieves the highest scores on almost all document sources (Table 9).

[10]We do not chunk programming solutions since they are typically short (average <200 tokens as in Table 2).

Method	Tutorials	Docs	SO	GitHub
Full text	6.7	17.7	28.0	3.7
First chunk	27.4	29.3	30.5	**30.5**
w/ reranking	9.1	9.1	14.0	13.4
Pre-retrieval	**31.1**	**32.9**	**33.5**	29.3

Table 9: Comparing chunking strategies on HumanEval.

5 Related Work

Code generation Neural code generation has been a crucial task (Lu et al., 2021), and increasingly strong code LMs have been created (Roziere et al., 2023; Li et al., 2023; Guo et al., 2024; Team, 2024) to solve various tasks (Chen et al., 2021; Lai et al., 2023; Jimenez et al., 2024). However, most LMs generate code solely based on NL queries and model parametric knowledge, without using external programming sources (e.g., tutorials). To fill in this gap and allow systematic studies of RACG, we integrate various datasets and retrieval sources to build CODERAG-BENCH.

Retrieval augmented generation (RAG) RAG has been widely used in knowledge-intensive tasks (Lewis et al., 2020; Guu et al., 2020), however, mostly on text-centric tasks using general domain corpora such as Wikipedia (Asai et al., 2024). Some works used programming context retrieved from repositories (Ding et al., 2023; Yang et al., 2024) or documentations (Zhou et al., 2023), yet none of them considered RACG across varied coding tasks and knowledge sources. In text-centric tasks, unified benchmarks such as BEIR (Thakur et al., 2021) and KILT (Petroni et al., 2020) aggregate retrieval and generation tasks and facilitate its progress (Muennighoff et al., 2022). To similarly enable systematic studies of RACG across coding tasks and retrieval sources, we curate a unified benchmark and release its RACG codebase.

6 Conclusion

In this work, we propose CODERAG-BENCH, a benchmark for retrieval-augmented code generation with various coding tasks and retrieval sources. With our experiments with top-performing retrieval and generation models, we show that retrieving external documents can greatly benefit code generation. However, current retrieval models struggle to find useful documents, and generation models have limited context capacity and RAG abilities, both leading to suboptimal RACG results. We hope CODERAG-BENCH can serve as a solid testbed to advance future endeavors in this direction.

Limitations

We propose a new paradigm, retrieval-augmented code generation, equipped with a comprehensive benchmark CODERAG-BENCH. However, as an initial exploration in this field, our work could be extended in task and language diversity, as well as model and methodological improvements.

We aggregate various existing code generation tasks, but many interesting scenarios such as code debugging remain under-explored. Meanwhile, we focus on coding tasks using Python programming language, but extrapolating to other languages may bring additional challenges.

Meanwhile, for benchmarking purposes, we mostly experimented with vanilla retrieval, reranking, and generation methods, but better backbone models and advanced methods for each RACG component are yet fully explored. Our results may not represent all model behaviors, and we encourage future works to build methods that break certain limitations we observe in current systems.

References

Akari Asai, Timo Schick, Patrick Lewis, Xilun Chen, Gautier Izacard, Sebastian Riedel, Hannaneh Hajishirzi, and Wen-tau Yih. 2023. Task-aware retrieval with instructions. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 3650–3675, Toronto, Canada. Association for Computational Linguistics.

Akari Asai, Zexuan Zhong, Danqi Chen, Pang Wei Koh, Luke Zettlemoyer, Hannaneh Hajishirzi, and Wen-tau Yih. 2024. Reliable, adaptable, and attributable language models with retrieval. *arXiv preprint arXiv:2403.03187*.

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*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *ArXiv*, abs/2005.14165.

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*.

CohereAI. 2024. Command r.

Together Computer. 2023. Redpajama: An open source recipe to reproduce llama training dataset.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, et al. 2023. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents. *arXiv preprint arXiv:2310.19923*.

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

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2022a. Unsupervised dense information retrieval with contrastive learning. *Transactions on Machine Learning Research*.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane A. Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022b. Few-shot learning with retrieval augmented language models. *ArXiv*, abs/2208.03299.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. 2023. Large language models struggle to learn long-tail knowledge. In *International Conference on Machine Learning*, pages 15696–15707. PMLR.

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 *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*. Reproducibility Certification.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9802–9822, Toronto, Canada. Association for Computational Linguistics.

Rui Meng, Ye Liu, Shafiq Rayhan Joty, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. 2024. Sfr-embedding-mistral:enhance text retrieval with transfer learning.

Meta. 2024. Introducing meta llama 3: The most capable openly available llm to date.

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

Arnold Overwijk, Chenyan Xiong, Xiao Liu, Cameron VandenBerg, and Jamie Callan. 2022. Clueweb22: 10 billion web documents with visual and semantic information. *arXiv preprint arXiv:2211.15848*.

Fabio Petroni, Aleksandra Piktus, Angela Fan, Patrick Lewis, Majid Yazdani, Nicola De Cao, James Thorne, Yacine Jernite, Vladimir Karpukhin, Jean Maillard, et al. 2020. Kilt: a benchmark for knowledge intensive language tasks. *arXiv preprint arXiv:2009.02252*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Stephen E. Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3:333–389.

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*.

Aivin V. Solatorio. 2024. Gistembed: Guided in-sample selection of training negatives for text embedding fine-tuning.

Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. Arks: Active retrieval in knowledge soup for code generation. *arXiv preprint arXiv:2402.12317*.

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.

CodeGemma Team. 2024. Codegemma: Open code models based on gemma.

Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A heterogeneous benchmark for zero-shot evaluation of information retrieval models. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

VoyageAI. 2024. voyage-code-2: Elevate your code retrieval.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023a. Learning to filter context for retrieval-augmented generation. *arXiv preprint arXiv:2311.08377*.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023b. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding. *arXiv*.

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*.

Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024. Code representation learning at scale. *arXiv preprint arXiv:2402.01935*.

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

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 Example Illustrations

A.1 Example with Canonical Documents

To present our canonical document annotation (§2.3) more concretely, we illustrate examples with their annotated canonical documents. Figure 3 shows the general-programming examples, with one HumanEval and one MBPP example, respectively. Figure 4 shows two open-domain coding examples with canonical library documentation from DS-1000 and ODEX, respectively.

A.2 RACG with Helpful and Distracting Documents

Beyond the numerical numbers reported in experiment sections, here we provide some concrete examples that: (i) benefit from RACG when relevant documents are retrieved, and (ii) distracted by irrelevant documents retrieved hence results in degraded performance.

B Additional Details about Retrieval Efficiency

For open access models, we use the same single A100 GPU with 80 GRAM, with a batch size of 64 for GIST base and large, and 8 for SFR-Mistral. For proprietary models, we estimate their efficiency using a batch size of 64. We then average the time for each batch for each query and document. For Voyage-code, we apply a "dynamic-batching" technique that make sure the total tokens in the batch won't exceed the token limit. For both open and

Canonical Document

```
def truncate_number(number: float) -> float:
    """ Given a positive floating point number, it can
    be decomposed into  and integer part (largest
    integer smaller than given number) and decimals
    (leftover part always smaller than 1).

    Return the decimal part of the number.
    >>> truncate_number(3.5)
    0.5
    """
    return number % 1.0
```

Problem

```
def truncate_number(number: float) -> float:
    """ Given a positive floating point number, it can
    be decomposed into  and integer part (largest
    integer smaller than given number) and decimals
    (leftover part always smaller than 1).

    Return the decimal part of the number.
    >>> truncate_number(3.5)
    0.5
    """
```

Canonical Document

```
# Write a python function to remove first and last
occurrence of a given character from the string.

def remove_Occ(s,ch):
  for i in range(len(s)):
    if (s[i] == ch):
      s = s[0 : i] + s[i + 1:]
      break
  for i in range(len(s) - 1,-1,-1):
    if (s[i] == ch):
      s = s[0 : i] + s[i + 1:]
      break
  return s
```

Problem

```
# Write a python function to remove first and last
occurrence of a given character from the string.
```

Figure 3: HumanEval (left) and MBPP (right) examples with annotated canonical solutions.

proprietary models, we define the search efficiency as the time it takes to embed individual query and the time to calculate similarities. Note that the time for both can be optimized by tokenizing all documents and all queries, then taking the dot product. The actual runtime for API models varies for each organization with different rate limits and the batch size. For this experiment, we set the maximum context length to match the maximum length of the original models. This notably increases the encoding latency of SFR Mixtral, which has a longer maximum context window size than smaller embedding models.

C Result Reproduction

In Table 5 in §3, we are able to reproduce most results reported in the original papers, but with minor variances. Here we explain the differences in implementation and (potential) reasons that lead to these small performance variances.

Our approach To keep a fair comparison among all models, we use the same prompt for each dataset when evaluating all models. Meanwhile, we use zero-shot prompts without any additional instructions, i.e., only input the original problem description of the example, to prevent unknown effects on the model performance when using different instructions and/or in-context examples.

According to this setup, we next describe the differences in prompts used by the original works and how they may affect the results.

StarCoder2 The StarCoder2 technical report (Lozhkov et al., 2024) reported results on the HumanEval, MBPP, and DS-1000 datasets. On HumanEval, our reproduced results (31.7) is slightly lower than their number (35.4), possibly because the original paper additionally input the test cases as additional information in the prompt, whereas in our basic NL-to-code setup, no test cases are provided. This additional information may cause their results to be higher.

On MBPP dataset, they adopt a subset of MBPP, i.e., 399 out of 427 examples that have additional test cases populated by Liu et al. (2023). In contrast, we evaluate on the entire dataset, which is likely to cause the variance in results.

On DS-1000, the original paper samples 40 generations and report the pass@1 rate, while we only generate one program with greedy decoding. This difference in decoding strategy may cause slight variance in the results.

CodeGemma The CodeGemma technical report (Team, 2024) reported results on HumanEval and MBPP datasets, but does not provide any details about the instructions, few-shot examples, or other parts of the prompt that they use. We were able to roughly reproduce their reported results, but with 3-5 points less in pass@1.

Canonical Document

pandas.reference.api.pandas.dataframe.groupby
pandas.DataFrame.groupby DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=NoDefault.no_default, observed=False, dropna=True)[source]
… …

pandas.reference.api.pandas.dataframe.squeeze
pandas.DataFrame.squeeze
DataFrame.squeeze(axis=None)[source]
Squeeze 1 dimensional axis objects into scalars. Series or DataFrames with a single element are squeezed to a scalar.
… …

Problem

What is best way to achieve this ? closest I got was with the zip function but haven't managed to make it work for more then one level (two columns).
A:
<code>
import pandas as pd
df = pd.DataFrame({'name': ['A', 'A', 'B', 'C', 'B', 'A'], 'v1': ['A1', 'A2', 'B1', 'C1', 'B2', 'A2'], 'v2': ['A11', 'A12', 'B12', 'C11', 'B21', 'A21'], 'v3': [1, 2, 3, 4, 5, 6]})
</code>
result = ... # put solution in this variable
BEGIN SOLUTION
<code>

Canonical Document

python.library.socket#socket.socket.send

socket.send(bytes[, flags])
Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the Socket Programming HOWTO. Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an InterruptedError exception (see PEP 475 for the rationale).

Problem

sending http headers to `client`

Figure 4: DS-1000 (left) and ODEX (right) examples with annotated canonical library documentation.

CodeLlama The CodeLlama technical report (Roziere et al., 2023) reports results on HumanEval and MBPP datasets. We were able to perfectly reproduce their results on the HumanEval dataset under the zero-shot setting. However, for MBPP experiments, they use 3-shot prompting, which could potentially explain that our zero-shot results are 4 points lower in pass@1.

DeepSeekCoder The DeepSeekCoder technical report (Guo et al., 2024) reports results on HumanEval and MBPP for the 7B-instruct-v1.5 and the 33B-instruct models, the report additionally report DS1000 results for the 33B-instruct model. We could reproduce the original results on HumanEval and DS-1000, but got slightly worse results on MBPP because they used few-shot prompting, which should outperform our zero-shot method.

Llama3 Since there is no technical report available yet, the official blog post [11] report results on HumanEval, without any descriptions on prompting construction or the inference process. Our reproduced results are about 4 points lower than their original results.

[11] https://ai.meta.com/blog/meta-llama-3/

D Analysis on Open-Domain Coding Problems

In §3.4, providing the documentation of required libraries brings limited benefits, especially with strong proprietary models such as GPT and Gemini. While we hypothesize that these strong models are sufficiently familiar with the required libraries and in turn barely benefit from additional information about them, in this section, we quantitatively investigate this issue and verify its validity.

Concretely, we construct a subset of ODEX containing only examples with less common libraries. We use the real-world distribution of all libraries involved in ODEX and select examples that use the top 20 least common libraries (e.g., `sqlite3`, `ftplib`, `flask`). We then evaluate model performance on this subset and compare the results with and without documentation in model contexts.

With varied retrieval models Aligning with §3.4, we examine the RACG results using documentation retrieved by different retrieval models. As shown in Table 10, augmenting documentation retrieved with most methods improves the results by 20.3–40.1%. Compared to the entire ODEX set where most queries require common libraries, this hard-library split more clearly demonstrates the effectiveness of augmenting library documentation.

Question

```
Split string 'Words, words, words.' using a regex '(\\W+)'
```

Solution

```
re.split('(\\W+)', 'Words, words, words.')
```

Context (helpful)

```
# Write a function to put spaces between words starting
with capital letters in a given string by using regex.
import re
def capital_words_spaces(str1):
 return re.sub(r"(\w)([A-Z])", r"\1 \2", str1)
```

Model Prediction

```
import re
def f_2195340():
    """split string 'Words, words, words.' using a regex
'(\W+)'"""
   text = 'Words, words, words.'
   return re.split(r'(\W+)', text)
```

Question

```
Find intersection data between series `s1` and series `s2`
```

Solution

```
pd.Series(list(set(s1).intersection(set(s2))))
```

Context (distracting)

```
skimage.segmentation.join_segmentations(s1, s2)
[source]

Return the join of the two input segmentations. The join J
of S1 and S2 is defined as the segmentation in which two
voxels are in the same segment if and only if they are in
the same segment in both S1 and S2.  Parameters
class Intersection(expr1, expr2, **extra)
set_extremes(e1, e2)[source]
```

Model Prediction

```
from skimage.segmentation import join_segmentations
def f_18079563(s1, s2):
   """Find intersection data between series `s1` and series
`s2`
   """
   return join_segmentations(s1, s2)
```

Figure 5: RACG helps with relevant contexts (left) and hurts with distracting contexts (right).

This result verifies our hypothesis that strong GPT models are familiar with most common libraries, and can only benefit from additional library information when harder libraries are required.

Model	none	BM25	GIST	Voyage	OpenAI	Gold
GPT-3.5-turbo	17.2	**24.1**	13.8	20.7	17.2	**24.1**
GPT-4	20.7	24.1	17.2	27.6	24.1	27.6

Table 10: RACG results on the subset of ODEX examples using the least common libraries.

E How Many Documents to Augment?

Different models have varied context length limits and context utilization abilities. Therefore, we study how model performance varies when providing different numbers of documents in the context. We experiment with one representative dataset for each task category: HumanEval since it is the most commonly used dataset, ODEX for its broad domain coverage, and RepoEval for its solvable difficulty. We compare RACG performance when providing top-1, 2, 5, and 10 documents.

As shown by Figure 6, adding five documents yields the best results in most settings, except for StarCoder2 on RepoEval which best uses 8 documents. Despite the drastic variance in length limits of StarCoder2 ($16k$) and DeepseekCoder ($4k$), the sweet spot is consistently 5 documents. While adding a few documents brings helpful contexts, adding more low-ranked documents may introduce noise and deteriorate generation due to the imperfections of retrieval systems (Wang et al., 2023a).

F Does RACG Help Stronger Models?

We have shown that RACG with open retrieval improves a relatively weaker model, StarCoder2 (§4). To see if this improvement of RACG with open retrieval generalizes to stronger models, we experiment with a series of top-performing proprietary models: GPT-4o, Claude-3-haiku and sonnet, and Gemini-1.5-flash and pro.

Basic programming: HumanEval RACG can consistently improve the performance of GPT-4 and Claude-3-sonnet when leveraging all sources of documents. However, for weaker models such as Claude-3-haiku and Gemini-1.5-flash, RACG only helps when aggregating multiple sources yet falls short when grounding on one source (even the canonical solution source). Interestingly, the stronger Claude-3-sonnet performs worse than the weaker Claude-3-haiku, but can benefit from all retrieval sources and outperform haiku with documents from the canonical programming source, suggesting its potentially better RAG ability. While the stronger Claude effectively benefits from extra contexts, the stronger Gemini-1.5-pro behaves similarly to its weaker counterpart and cannot do RACG effectively with non-canonical sources.

Open domain: ODEX All models experience limited improvements by leveraging library documentation to complex the ODEX task, with the only exception that GPT-4o improves 4.6 points by incorporating programming solutions into the context.

As results degrade in most cases, we conduct

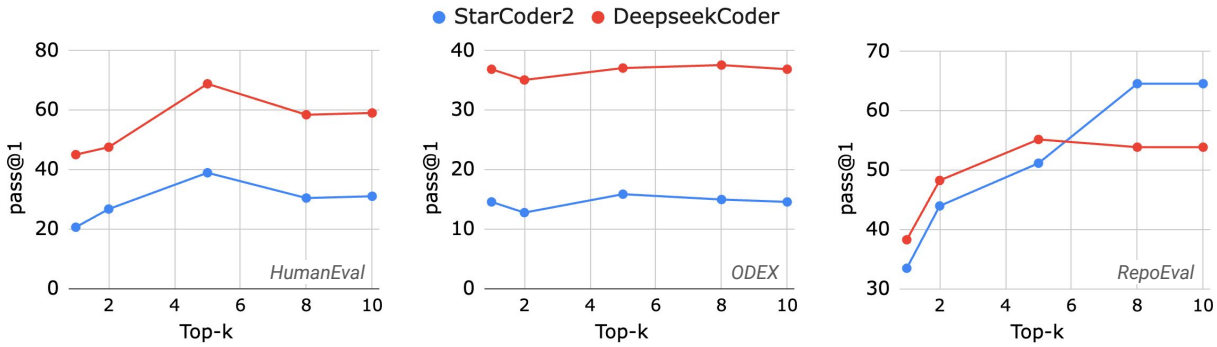

Figure 6: Comparing RACG performance with various numbers of documents.

a manual analysis to examine when most models fail. We find that most models tend to copy functions in the context, sometimes even overwriting the function being queried, thus failing all the test cases specific to the queried function. Further, possibly affected by the plethora of programs in context, models tend to generate over-complicated programs which, however, do not often pass the test cases.

In general, most models can be easily distracted or disturbed by additional contexts (Wang et al., 2023a), and fail to conduct the designated code generation task, indicating much room for improvement for RACG.

Repository level: RepoEval While GPT-4o can solve the RepoEval task with a reasonable success rate, all Claude models are challenged by the task and achieve less than 10% pass@1 for most scenarios. We find Claude models mostly respond with explanations of the incomplete input code, instead of the to-be-completed code even with proper instructions, possibly caused by some properties of the unknown training data. Gemini-1.5-flash also barely solves the task and often generates textual explanations; however its stronger pro variant gets about 10–25 point improvements, demonstrating its stronger repository-level code completion abilities.

Method	Baseline	Program	Tutorial	Docs	SO	GitHub	All
GPT-4o	75.6	94.5	90.2	90.9	91.5	84.8	95.1
Claude-3-haiku	74.4	77.4	77.4	71.3	67.7	73.2	82.9
Claude-3-sonnet	65.9	78.7	66.5	68.9	70.7	73.8	80.5
Gemini-1.5-flash	72.0	91.5	75.0	70.1	68.9	68.9	95.1
Gemini-1.5-pro	82.9	95.7	79.9	77.4	79.9	80.5	86.6

Table 11: RACG on HumanEval with strong code LMs.

Method	Baseline	Program	Tutorial	Docs	SO	GitHub	All
GPT-4o	44.6	49.2	44.2	47.6	40.3	39.4	39.6
Claude-3-haiku	48.5	42.6	39.2	44.6	33.7	40.5	35.1
Claude-3-sonnet	41.0	37.6	35.3	38.0	34.2	42.4	38.0
Gemini-1.5-flash	50.6	48.3	46.7	46.2	41.9	44.9	43.1
Gemini-1.5-pro	57.2	54.4	45.6	51.0	46.5	39.6	46.0

Table 12: RACG on ODEX with strong code LMs.

Method	Baseline	Local	Program	Tutorial	Docs	SO	GitHub	All	L+E
GPT-4o	32.4	62.2	35.4	28.7	27.8	29.0	28.2	30.3	54.2
Claude-3-haiku	9.1	0.5	0.5	0.5	0.5	0.5	0.2	0.2	0.5
Claude-3-sonnet	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Gemini-1.5-flash	1.3	16.9	4.0	2.1	3.2	2.1	3.2	2.7	11.8
Gemini-1.5-pro	10.5	39.1	15.1	13.4	15.8	15.3	11.8	12.3	33.0

Table 13: RACG on RepoEval with strong code LMs.