REPOFORMER: Selective Retrieval for Repository-Level Code Completion

Di Wu [1] [*]   **Wasi Uddin Ahmad** [2]   **Dejiao Zhang** [2]   **Murali Krishna Ramanathan** [2]   **Xiaofei Ma** [2]

Abstract

Recent advances in retrieval-augmented generation (RAG) have initiated a new era in repository-level code completion. However, the invariable use of retrieval in existing methods exposes issues in both efficiency and robustness, with a large proportion of the retrieved contexts proving unhelpful or harmful to code language models (code LMs). In this paper, we propose a selective RAG framework to avoid retrieval when unnecessary. To power this framework, we design a self-supervised learning approach to enable a code LM to accurately self-evaluate whether retrieval can improve its output quality and robustly leverage the potentially noisy retrieved contexts. Using this LM as both the selective RAG policy and the generation model, our framework achieves state-of-the-art repository-level code completion performance on diverse benchmarks including RepoEval, CrossCodeEval, and CrossCodeLongEval, a new long-form code completion benchmark. Meanwhile, our analyses show that selectively retrieving brings as much as 70% inference speedup in the online serving setting without harming the performance. We further demonstrate that our framework is able to accommodate different generation models, retrievers, and programming languages. These advancements position our framework as an important step towards more accurate and efficient repository-level code completion.

1 Introduction

Automatic code completion has attracted long-lasting research efforts due to its high practical value in improving programmer productivity (Ye & Fischer, 2002; Hill & Rideout, 2004; Hellendoorn & Devanbu, 2017). One particularly

challenging scenario is *repository-level code completion*, where a system is required to complete lines, API invocations, or functions in a file from user repositories. For this task, language models for code (code LMs) have emerged as a promising solution due to their ability to leverage the context of the current file to generate coherent code of flexible granularity (Tu et al., 2014; Svyatkovskiy et al., 2020; Chen et al., 2021). However, these approaches fail to capture the holistic repository knowledge spanning beyond the current file, such as user-defined APIs and inter-module dependencies (Zan et al., 2022; Zhang et al., 2023; Ding et al., 2023). Recently, the *retrieval-augmented generation* (RAG) paradigm was proposed to bridge the gap: cross-file contexts such as relevant code snippets or documentations are retrieved and provided to code LMs as augmentations to the current file. This approach has shown strong empirical performance and was further advanced by recent literature through designing better retrieval mechanisms for prompting black-box code LMs (Lu et al., 2022; Shrivastava et al., 2023b; Zhang et al., 2023) and adapting the LM to better leverage structured retrieved contexts such as classes, functions, or APIs (Ding et al., 2024; Zan et al., 2022).

Despite their encouraging performance, existing RAG-based approaches largely ignore to address a critical question:

> *Should we always perform retrieval augmentation*?

Our findings suggest that the answer is predominantly negative. First, in various code completion tasks, we discover that up to 80% of the retrievals performed by a standard RAG method *do not enhance the performance* of common code LMs such as CodeGen (Nijkamp et al., 2023b) and StarCoder (Li et al., 2023b), and many degrade the performance by introducing irrelevant information (Section 5.1). Second, always retrieving introduces notable *inefficiencies*. For moderately sized repositories, sparse retrieval is already as time consuming as code completion with a 3B code LM (Section 5.3 and Section 6). This inefficiency is more pronounced with dense retrieval, enterprise-scale repositories, and iterative RAG methods such as Zhang et al. (2023).

In this paper, we challenge the assumption of always retrieving by proposing a novel repository-level code completion framework underpinned by a *selective retrieval* mechanism: the system proactively abstains from performing unneces-
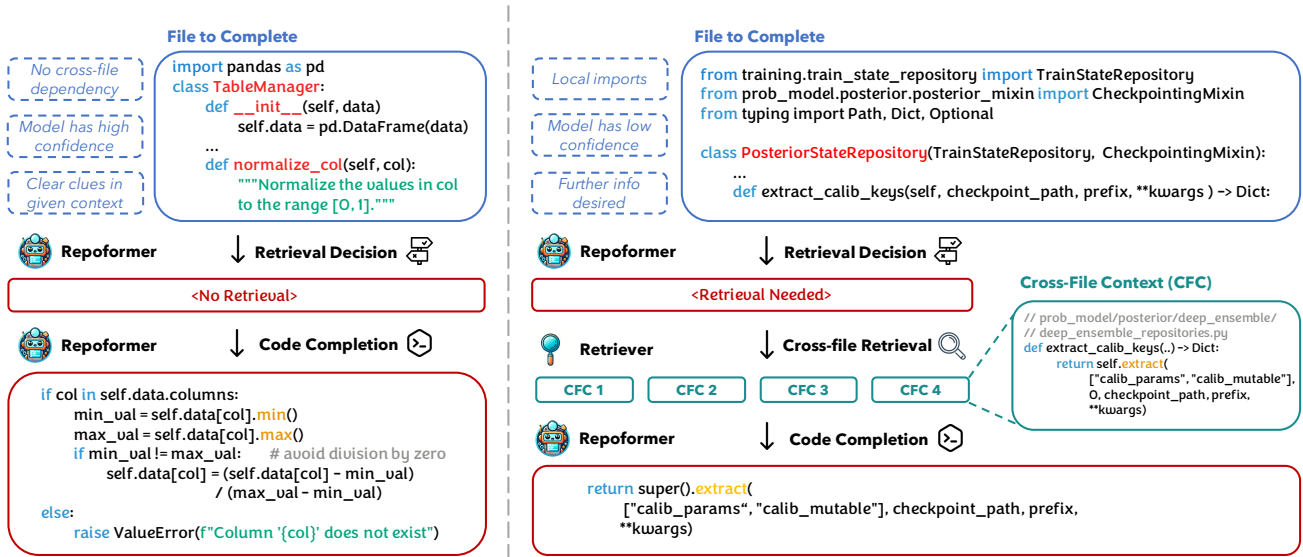

Figure 1. An overview of the proposed selective RAG framework. Given the current file context, the system first assesses whether retrieval is required and triggers the retriever if the question can likely be benefited from retrieval (right), abstaining from retrieval otherwise (left). Then, the code LM generates with optional retrieved contexts. With REPOFORMER, the two stages are streamlined via self-assessment.

sary or potentially detrimental retrievals (Figure 1 (a)). At the core of our framework is REPOFORMER, an intelligent code LM fine-tuned for *robust code completion with self-triggered retrieval augmentation*. REPOFORMER reflects three core principles:

1. **Performance-oriented self-evaluation.** After observing the current file, REPOFORMER explicitly expresses the likelihood that its prediction quality could be improved by cross-file retrieval. Our training strategy enables the model to combine two factors in this decision: the *code LM* already knowing the answer without retrieval (Kadavath et al., 2022) and the *code completion question* not depending on cross-file information and thus retrieval is likely uninformative.

2. **Robustness to retrieved contexts.** REPOFORMER learns to use the retrieved contexts to improve the quality of its output and avoid performance drops caused by potentially noisy retrieved information.

3. **Generalizability.** The aforementioned two abilities must generalize to any completion granularity, programming language, and retriever choice. In addition, REPOFORMER should be able to function as a plug-and-play selective retrieval policy when other models are employed as the generation model.

We posit that these abilities can be faithfully obtained by learning from *simulations of RAG*. Specifically, we leverage a large number of permissively licensed repositories, sample diverse blanks to complete, and pair them with the retrieved repository-level cross-file contexts. Then, for a given code LM, the ground-truth label for selective retrieval is obtained by contrasting the quality of its outputs with and without retrieval augmentation. With this dataset, we design a self-supervised objective to jointly train code LMs to accurately self-evaluate the need for retrieval and robustly complete the code with the optional retrieval augmentation (Section 3.3).

We perform comprehensive evaluations on a range of repository-level code completion tasks from RepoEval (Zhang et al., 2023), CrossCodeEval (Ding et al., 2023), and CrossCodeLongEval a new large-scale benchmark focusing on code chunk and function completion. Results show that REPOFORMER achieves strong performance, outperforming always retrieving with the same-sized StarCoderBase by more than 3 absolute points for edit similarity across multiple tasks. The 3B REPOFORMER performs on par with always retrieving using the 16B StarCoder, and the 16B REPOFORMER achieves state-of-the-art performance across all the tasks (Section 5.2). Furthermore, our framework allows for up to 70% inference speedup without harming accuracy. We also establish that REPOFORMER can accelerate RAG with larger black-box LMs as a plug-and-play selective RAG policy, improving the performance while reducing the latency of line and API completion to 75% (Section 5.3).

Finally, in Section 6, we provide comprehensive analyses on REPOFORMER's generalization ability. We show that RE-POFORMER makes precise retrieval abstention decisions, is robust to retrieved contexts, and performs well when tested in other languages or with other retrievers. To facilitate future research on repository-level code completion, we will release our implementation and the CrossCodeLongEval benchmark at https://repoformer.github.io/.

2

2 Related Work

Repository-level Code Completion Accurately completing the code in repositories has been a challenging research problem due to cross-file dependency patterns caused by modular design (Parnas, 1972; Tu et al., 2014). Early works propose application-specific training methods for n-gram LMs (Tu et al., 2014), RNNs (Hellendoorn & Devanbu, 2017; Wang et al., 2021), and Transformers (Svyatkovskiy et al., 2020) to leverage structured knowledge beyond current file's context. Recent studies investigate fine-tuning powerful pre-trained code LMs (Chen et al., 2021; Nijkamp et al., 2023b; Li et al., 2023b) to better leverage retrieved knowledge provided in context such as code and documentation snippets (Zan et al., 2022; Ding et al., 2024; Shrivastava et al., 2023a). Concurrently, other studies show that black-box code LMs can already take advantage of in-context knowledge, depending on how well the knowledge is retrieved and formatted (Lu et al., 2022; Zhou et al., 2023; Shrivastava et al., 2023b; Zhang et al., 2023). This approach does not require one to train the LM and thus promises better generalization. Orthogonal to these studies, this paper identifies and addresses the robustness and efficiency issues caused by invariably performing the retrieval augmentation. Our solution takes the form of selective retrieval augmentation through self-assessment.

Adaptive RAG This paper is consistent with the recent trend of making the RAG paradigm active and adaptive. A core question is finding an effective policy to decide *when to retrieve*. He et al. (2021) propose to learn to adjust the importance weight of retrieval based on language modeling performance. Drozdov et al. (2022) proposes to upweight the retrieved information when the retrieval has high quality. Li et al. (2023a) and Jiang et al. (2023) suggest that retrieval should be performed only when LMs have a high predictive uncertainty. Mallen et al. (2023) discover that retrieval can be avoided for popular facts. Concurrent to this work, two new studies approach adaptive RAG from a learning perspective. SKR (Wang et al., 2023) collects instances where retrieval is not helpful for black-box LMs and proposes several methods to predict these instances. Self-RAG (Asai et al., 2024) utilizes GPT-4 (OpenAI, 2023) as a knowledge engine to distill a smaller LM to evaluate whether answering a question can be benefited from retrieval. In comparison, this paper highlights the importance of understanding whether an LM knows the answer (Kadavath et al., 2022) in forming the retrieval policy. We introduce a simple yet effective scheme to fine-tune a code LM for faithful self-evaluation without extra modules (SKR), knowledge store (SKR), or labels generated by an oracle LM (Self-RAG). We show that our approach leads to no performance harms (Section 5.2), substantial speedup (Section 5.3), and a high decision accuracy (Section 6).

3 Approach

In this section, we first briefly formulate the repository-level code completion task and the considered RAG setup. Then, we illustrate the details of the proposed framework.

3.1 Background

Problem Formulation We denote each *repository-level code completion* task as (X_l, X_r, Y, F). Y is the ground truth completion that needs to be generated. In this paper, Y always contains one or more consecutive lines of code. X_l and X_r are the code to the left/right of Y in the same file. We will use the left/right context to refer to them. F is the set of other files in the repository. A code completion system utilizes X_l, X_r, and F to generate a hypothesis $\hat{Y}$.

Retrieval-Augmented Generation We follow the RG-1 formulation in Zhang et al. (2023) to execute RAG for code completion in four stages: *indexing*, *query formation*, *retrieval*, and *generation*. We consider two components:

- An **in-repository retriever** $\mathcal{R}$ that queries F with information from X_l and X_r and returns relevant cross-file contexts CC. CC consists of k code chunks $cc_1, cc_2, ..., cc_k$, each of which contains consecutive lines of code extracted from a file in F. We mainly use Jaccard similarity (Jaccard, 1912) as $\mathcal{R}$ due to its speed and strong performance (Zhang et al., 2023).
- A **code LM** $\mathcal{M}$ that leverages X_l, X_r, and CC to output $\hat{Y}$. The inclusion of X_r and CC is optional. In this paper, we always directly provide X_r in the prompt in addition to X_l (Shrivastava et al., 2023b; Pei et al., 2023). We provide empirical support for this design in Appendix B.

Full documentation of the RAG stages and their hyperparameters are provided in Appendix A for further reference.

3.2 Self-selective RAG for Code Completion

Central to our framework is the idea of *selective RAG*, where the system decides whether the LM's generation could benefit from retrieved contexts and abstains from retrieval augmentation when it is deemed unnecessary (Figure 1).

For this selective decision, two traditional heuristics are relevant: (1) performing a *trial retrieval* and only augmenting the high-relevance contexts (e.g., Drozdov et al. (2022)) or (2) performing a *trial generation* and conducting RAG only when the model's uncertainty is high (e.g., Jiang et al. (2023)). For repository-level code completion, these strategies are informative to some extent: in line completion and API completion from RepoEval, both heuristics can maintain the same level of performance with only 50% retrieval budget. However, we find that they fail to generalize well to

all tasks and still incur a high latency cost as they need to conduct retrieval to make the decisions (Appendix C).

Instead, our framework adopts a *self-selective RAG* formulation. After observing X_l and X_r, the LM directly self-triggers cross-file retrieval by generating a special token $<$cc$>$ or abstains from retrieval via an empty token ϕ[1]. This approach is inspired by the explorations in Kadavath et al. (2022), which show that an LM can be trained to predict whether it knows the answer or not without retrieval. Beyond this *self-knowledge*, our model also combines the *question's characteristics* (i.e., whether retrieving cross-file information can likely help or not) in its judgment, as we will discuss in the next section. Finally, after the optional retrieval, the LM proceeds with the code completion with X_l, X_r, combined with CC if retrieval is triggered.

Implementation-wise, self-selective RAG's inference is conveniently modeled as an extension to fill-in-the-middle (Bavarian et al., 2022), with the entire process executed in a single left-to-right pass (Figure 2). One advantage of this design is the *flexibility*. The LM possesses the ability for RAG and fill-in-the-middle, and can seamlessly self-switch between the two when encountering different questions. Users can also easily adjust the ratio between the two through the retrieval threshold. Another advantage is its *efficiency*. The selective decision overhead is only a single forward pass, a significant save compared to making the retrieval decision via trial generation or trial retrieval. When the LM abstains from retrieval, it can directly proceed with generation and the retrieval overhead is completely avoided.

3.3 Self-supervised Multi-task Learning

To power self-selective RAG, the LM needs two crucial abilities: accurate self-assessment and robustness to the retrieved context. We design a contrastive data labeling scheme to mine self-supervision from public repositories, followed by fine-tuning with a novel multi-task objective.

Data construction We leverage large-scale permissively licensed repositories from the Stack (Kocetkov et al., 2022) and create the fine-tuning data via a three-step procedure:

1. **Sample** target lines Y that are either (1) random code chunks of varied lengths or (2) function bodies.
2. **Retrieve** CC using the current file. We include Y in the query for 50% of the data[2].
3. **Label** whether extending the current file with CC can

---

[1]In practice, instead of greedily decoding $<$cc$>$, we check whether its probability exceeds a certain threshold.

[2]The main goal of the design is to align better with both non-iterative and iterative RAG use cases. During testing, a user may retrieve with both the in-file context and Y', a model's draft prediction, which results in a CC distribution close to that with Y in the query (Zhang et al., 2023).

(a) Fill-in-the-middle
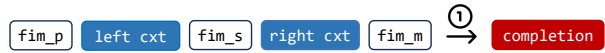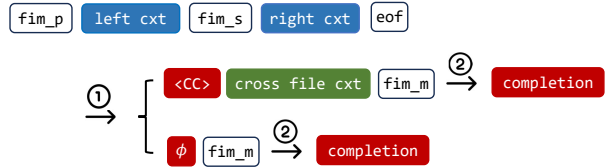
(b) Self-selective RAG

Figure 2. A comparison between fill-in-the-middle and self-selective RAG. We mark the end of the current file with a new token $<$eof$>$, which triggers the LM's self-evaluation. $\rightarrow$ denotes the invocation of the LM. We color current-file context, retrieved contexts, and LM-generated parts in blue, green, and red respectively. `fim_p`, `fim_s`, and `fim_m` refer to the special tokens for fill-in-the-middle: `fim_prefix`, `fim_suffix`, and `fim_middle`. These tokens are already learned during the pre-training.

improve a code LM $\mathcal{M}$'s code completion quality by more than a threshold T, measured by Edit Similarity (ES, definition in Section 4.1) against Y.

The full algorithms are presented in Appendix D. After running the algorithm, we obtain the fine-tuning instances, each in the form $(X_l, X_r, Y, CC, label)$.

Verbalization Each instance is verbalized into a sequence for fine-tuning. If *label* is false, only X_l and X_r are provided preceding Y. Otherwise, we additionally provide CC after the special token $<$cc$>$. The two verbalizations correspond to the two branches in Figure 2 (b).

Training Objective We introduce two losses, $\mathcal{L}_{eval}$ for self-assessment and $\mathcal{L}_{gen}$ for code generation.

1. $\mathcal{L}_{eval}$: a cross-entropy loss on predicting $<$cc$>$ immediately following $<$eof$>$.
$$\mathcal{L}_{eval} = -\log p_{\mathcal{M}}(<\text{cc}>|X_l, X_r) \qquad (1)$$

2. $\mathcal{L}_{gen}$: a cross-entropy loss on the tokens following $<$fim_middle$>$. Depending on *label*, $\mathcal{L}_{gen}$ represents either code completion with only in-file information or retrieval-augmented code completion.
$$\mathcal{L}_{gen} = \begin{cases} -\log p_{\mathcal{M}}(Y|X_l, X_r, CC), & \text{if } label \\ -\log p_{\mathcal{M}}(Y|X_l, X_r), & \text{otherwise} \end{cases} \qquad (2)$$

The final training objective is $\lambda\mathcal{L}_{eval} + \mathcal{L}_{gen}$, a weighted combination of the two losses. We do not supervise the model on predicting the other tokens in X_l, X_r, CC, or the special tokens for fill-in-the-middle. Teacher forcing is used just as in normal causal language model training.

4 Experimental Setup

4.1 REPOFORMER Implementation Details

Training Data We sample Python repositories from the Stack (Kocetkov et al., 2022). Basic filtering are applied to retain 18k repositories that have (1) at least five Python files, (2) at least three imports per file, and (3) at least two local imports per file. These criteria ensure the existence of local dependencies where RAG could be helpful. We use $\mathcal{M} =$ StarCoderBase-1B and $T = 0$ to label 240k chunk and 120k function completion instances. We reserve 500 repositories for validation and use the rest for training.

Training We fine-tune the 1B, 3B, 7B, and 16B variants of StarCoderBase with $\lambda = 1.0$, maximum sequence length 2048, learning rate 2e-5, batch size 512, 50 warmup steps, and a linear learning rate decay. The models are trained for 2 epochs, which approximately takes 8, 12, 20, and 50 hours for the 1B/3B/7B/16B models respectively with 8 Nvidia A100 GPUs (40G memory). Our implementation is based on Jain et al. (2023)[3]. We will call our models REPOFORMER-1B/3B/7B/16B. We have also applied the same method to train a multilingual version of REPOFORMER on a mixture of Python, Java, C#, and Typescript repositories. As we focus on the methodological discussion in the main text, we refer interested readers to Appendix E.2 for the detailed experiment setup and results.

Hyperparameter optimization We conduct a grid search with StarCoderBase-1B on the following search space: learning rate {1e-5, 2e-5, 5e-5}, λ {0.2, 1.0, 2.0, 5.0}, training epochs {1, 2, 5}, and warmup steps {50, 100}. The best hyperparameters are selected based on the code completion performance on the validation dataset.

4.2 Evaluation Setup

Evaluation Datasets We evaluate on RepoEval (Zhang et al., 2023), which consists of line, API, and function completion tasks created from 32 Python repositories. To investigate the generalization to other languages, we also evaluated the original CrossCodeEval (Ding et al., 2023), which features line completion instances covering four languages: Python, Java, C#, and TypeScript (Appendix E.2). Observing that RepoEval has a limited repositrory coverage and that CrossCodeEval has a limited task coverage, we additionally leverage 1500 raw Python repositories from CrossCodeEval to create a new chunk and function completion benchmark, which we call CrossCodeLongEval. We detail the dataset creation process and basic statistics in Appendix D. For the rest of this paper, we will use CCEval to refer to both CrossCodeEval and CrossCodeLongEval interchangeably, and use the specific language and task (line,

chunk, or function completion) to differentiate them.

Evaluation Metrics We evaluate $\hat{Y}$ with both *reference-based* and *execution-based* evaluation. For reference-based evaluation, exact match (EM) and edit similarity (ES) are reported. Following Zhang et al. (2023), ES is defined as

$$ES(\hat{Y}, Y) = \frac{1 - Lev(\hat{Y}, Y)}{\max(|\hat{Y}|, |Y|)}, \qquad (3)$$

where Lev is the Levenshtein distance (Levenshtein et al., 1966). We report $ES \times 100$ in all the tables following Zhang et al. (2023) for better readability. For execution-based evaluation, we report the unit test pass rate (UT). $\hat{Y}$ is said to pass the unit tests if replacing Y with $\hat{Y}$ does not cause any unit test to fail. We implement simple post-processing procedures to handle common cases such as excessive lines in model's outputs, which are documented in Appendix A.

Models We experiment on two families of strong code LMs. **CodeGen-Mono** (Nijkamp et al., 2023b) is pretrained sequentially in natural language, multilingual code, and a Python corpus. **StarCoder** and **StarCoderBase** (Li et al., 2023b) are trained with fill-in-the-middle ability on a large corpus of multilingual code, GitHub issues, Git commits, and Jupyter notebooks. StarCoder is obtained by training StarCoderBase on an additional Python corpus.

5 Results

5.1 Is retrieval always helpful?

As a proof of concept, we first show that on a range of repository-level code completion tasks, the retrieved contexts often fail to improve code LMs' generation quality.

In Table 1 and Figure 3, we evaluate four code LMs on function completion and API completion from RepoEval. For each model, we report the instance-level performance change from code completion only using X_l and X_r to retrieval-augmented code completion with X_l, X_r, and CC (detailed prompts in Appendix A).

The results reveal an intriguing pattern: for repository-level code completion, the help from cross retrieval is often sparse. Specifically, retrieval improves LMs' performance on only 20% or fewer instances. For more than 60% of the instances, retrieval augmentation does not affect the performance at all[4]. Finally, another 20% retrievals actually harm the performance, almost as often as the first case. The observed trends are consistent for both API and function completion and hold for both small-sized (1B and 2B) and moderate-to-large (around 16B) code LMs. The generality of this observation is further confirmed by an analysis of REPOFORMER's train-

---

[3]https://github.com/amazon-science/ContraCLM

[4]Upon a manual inspection, we find that most of the outputs in this category are also not changed by retrieval at all.
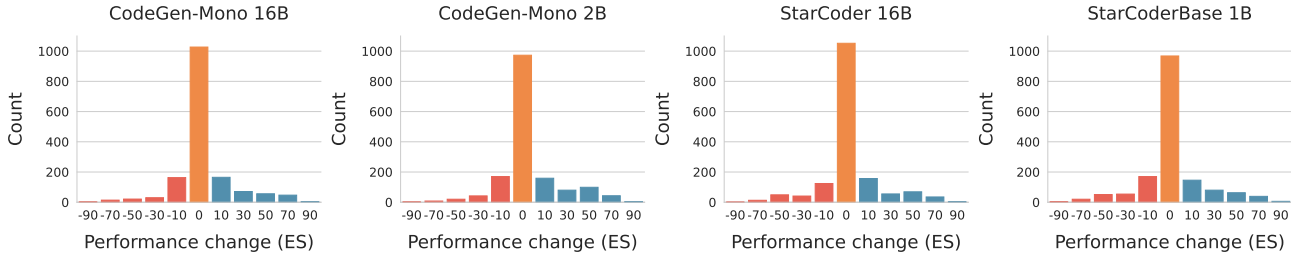
Figure 3. The performance gain on RepoEval API completion from retrieved cross-file contexts. Each bucket contains values ranging from label-10 to label+10 except for the central bucket, which corresponds to exactly 0. The retrieved contexts only improve the performance in about 20% of instances. The trend is consistent across all the evaluated LM families and sizes.

Model	Size	Performance (UT)		UT Change		
		$X_l + X_r$	$X_l + X_r + CC$	↓	=	↑
CodeGen-Mono	16B	23.74	24.18	23	407	25
CodeGen-Mono	2B	30.55	32.51	18	400	37
StarCoder	16B	34.73	42.86	16	386	53
StarCoderBase	1B	22.20	25.71	16	407	32

Table 1. The performance change on RepoEval function completion exhibited by four models from retrieved cross-file contexts. For the majority of the instances, RAG does not improve the performance. "↑", "=", "↓" denote the counts for performance increase, no performance change, and performance drop.

ing data, where we find that retrieval improves the performance for only fewer than 30% instances (Appendix D). Together, these findings highlight the suboptimality of the always retrieving and augmenting the cross-file contexts and thus motivate our selective retrieval proposal.

5.2 REPOFORMER achieves strong code completion performance via selective RAG

Next, we evaluate the code completion performance of RE-POFORMER. We compare the following three settings[5]. For the first two baselines, we use the state-of-the-art single-iteration prompting pipeline (Zhang et al. (2023), detailed in Appendix A). We use StarCoder models due to their strong performance among the open-source code LMs.

1. **No Retrieval**. This baseline only provides X_l and X_r to the model in the prompt.
2. **Always Retrieving**. This baseline always augments X_l and X_r with the retrieved CC.
3. **Selective Retrieval**. We provide REPOFORMER with X_l and X_r in the prompt, optionally augmented with CC based on two selective RAG policies:
   - **Greedy Selection**. Retrieval is performed if <cc> is the most likely token following <eof>.
   - **Threshold Selection**. If the probability of <cc>

following <eof> is greater than a threshold T, retrieval augmentation is performed[6].

The results are summarized in Table 2. Compared to no retrieval and always retrieving with StarCoderBase of the same size, REPOFORMER's selective retrieval strategy exhibits strong performance improvements across all the tasks and both lexical-based and execution-based metrics. Via the threshold selection strategy, REPOFORMER-3B can outperform StarCoderBase-7B on most of the tasks and metrics except EM for API completion, even outperforming the 5x larger StarCoder in terms of ES for API and chunk completion. Finally, The REPOFORMER-16B model outperforms the strongest StarCoder baseline by 3%, averaged across all tasks, setting up the new start-of-the-art for repository-level code completion. We also experimentally confirm that the performance improvement from our framework can generalize to three *languages beyond Python* (Appendix E.2) as well as *dense retrieval* instead of Jaccard similarity (Appendix E.3). In later sections, we demonstrate that the observed success is due to both the ability to accurately abstain from retrieval and the improved robustness to retrieval.

In terms of code completion accuracy, the threshold selection strategy outperforms the greedy selection strategy on all the tasks. In the next section, we show that the two strategies represent different ways to achieve a good balance between accuracy and inference budget.

5.3 REPOFORMER improves inference efficiency

We illustrate the benefits of REPOFORMER for saving the inference latency in a realistic "online serving" setting.

Latency Model We assume that indexing has already been done for the working repository. Given a code completion request containing the current file (X_l, X_r), the system issues three processes at the same time:

---

[5]We do not consider iterative retrieval because we find that single-iteration RAG already achieves the majority of the performance gains from multi-iteration RAG.

[6]We find that $T = 0.15$ for function completion and $T = 0.2$ for the other tasks generally work well. These two thresholds are always used unless otherwise stated.

Size	Model	RAG Policy	RepoEval (Line)		(API)		(Function)		CrossCodeLongEval (Chunk)		(Function)
			EM	ES	EM	ES	UT	ES	EM	ES	ES
1B	STARCODERBASE	No	43.44	67.77	37.81	66.54	22.20	47.65	31.08	60.09	47.49
		Always	51.19	72.30	43.94	69.17	25.71	55.64	37.22	63.73	50.50
	REPOFORMER	Selective$_G$	51.90	74.50	43.50	71.00	24.00	53.10	38.52	68.08	52.09
		Selective$_T$	**54.40**	**76.00**	**46.10**	**72.70**	**28.79**	**57.30**	**41.92**	**69.97**	**53.71**
3B	STARCODERBASE	No	49.00	72.12	40.44	69.02	24.84	51.22	36.14	64.65	49.88
		Always	56.69	76.68	47.00	72.62	29.67	57.68	42.26	67.74	53.39
	REPOFORMER	Selective$_G$	56.30	77.60	46.10	73.60	28.57	54.70	42.06	70.70	54.47
		Selective$_T$	**59.63**	**79.02**	**49.31**	**74.96**	**32.96**	**60.56**	**46.66**	**72.23**	**56.24**
7B	STARCODERBASE	No	51.88	74.03	43.31	70.79	25.49	52.28	38.88	66.61	52.45
		Always	59.44	78.15	49.56	73.65	31.43	58.51	44.44	69.53	55.41
	REPOFORMER	Selective$_G$	56.00	76.63	48.06	75.03	30.77	55.27	43.80	72.46	56.14
		Selective$_T$	**59.63**	**78.63**	**50.87**	**76.89**	**35.16**	**60.64**	**46.88**	**74.20**	**57.18**
16B	STARCODER	No	55.25	76.07	44.50	71.00	34.73	53.60	42.58	69.40	54.20
		Always	61.25	79.24	51.12	74.50	42.86	60.96	47.90	71.90	58.06
	REPOFORMER	Selective$_G$	58.13	78.81	48.69	76.23	42.42	58.42	45.00	73.36	57.71
		Selective$_T$	**61.75**	**80.34**	**51.88**	**77.93**	**44.18**	**62.58**	**49.18**	**75.50**	**58.93**

Table 2. Experiment results on RepoEval and CrossCodeLongEval. The best performance among each model size is boldfaced. We use Selective$_G$ and Selective$_T$ to denote the greedy selection and the threshold selection strategy for selective retrieval. REPOFORMER greatly outperforms STARCODERBASE of the same size while consuming a smaller retrieval budget. Among the two selective policies, threshold selection enables the best selective RAG performance.

- P_1: make a retrieval decision using REPOFORMER.
- P_2: use a code LM $\mathcal{M}$ to generate $\hat{Y}$ without CC.
- P_3: retrieve CC and generate $\hat{Y}$ with CC using $\mathcal{M}$.

Depending on the result of P_1, the system waits for either P_2 or P_3 and ignores the other process. If $\mathcal{M}$ is REPOFORMER, P_1 can be merged with P_2 by forcing $\mathcal{M}$ to generate a hypothesis without CC after collecting the retrieval decision. We consider three latency terms: (1) T_d, time required for the retrieval decision, (2) T_r, the retrieval latency, and (3) T_g, the generation latency. Then, the latency for P_1, P_2, and P_3 are T_d, T_g, and $T_r + T_g$. When $\mathcal{M}$ is REPOFORMER or a model larger than REPOFORMER, we have $T_d < T_g < T_r + T_g$. Therefore, the latency of the entire system is T_g or $T_r + T_g$ depending on P_1. Using this latency model, we benchmark the latency of various selective retrieval settings on RepoEval with the vllm library (Kwon et al., 2023) on a single Nvidia A100 GPU (80G).

First, we consider $\mathcal{M}$ = REPOFORMER and present the results in Table 3. Line and API completion are presented to cover short and moderate target lengths[7]. Both selective strategies significantly improve the latency, with a different trade-off: threshold selection results in *improvements for both accuracy and latency* compared to always retrieving, while using greedy selection results in a larger latency gain with a minor performance degradation (around 1.0 ES). It is

---

[7]We omit the function completion results as RepoEval uses very small repositories for function completion for easier unit testing.

	RAG Policy	API Completion			Line Completion		
		ES	%RAG	SU	ES	%RAG	SU
1B	Always	72.02	100%	0%	75.91	100%	0%
	Selective$_G$	71.04	18%	69%	74.50	19%	61%
	Selective$_T$	72.72	61%	28%	76.00	62%	27%
3B	Always	74.66	100%	0%	78.68	100%	0%
	Selective$_G$	73.60	19%	46%	77.60	20%	43%
	Selective$_T$	74.96	78%	17%	79.02	74%	16%

Table 3. RAG latency of REPOFORMER with two self-selective RAG paradigms. **%RAG** = ratio of instances where RAG is performed. **SU** = Speedup compared to always retrieving (the higher, the better). Compared to the always retrieving baseline, the threshold selection strategy consistently demonstrates gains in both accuracy and latency. The greedy selection strategy shows much larger latency gains with a small performance degradation.

worth mentioning that the latency improvement from selective RAG could be further enhanced with a more advanced retrieval setup. For instance, conducting dense retrieval on large repositories often consumes more than 80% of the entire RAG pipeline's latency. Then, a 20% RAG policy could translate into more than 70% speedup. We empirically verify this statement in Appendix E.3.

Next, we consider using diverse larger LMs as $\mathcal{M}$ in the code completion framework and using selection$_T$ with REPOFORMER-1B as a *plug-and-play* selective RAG policy to decide whether retrieval should be performed. We experiment on a diverse set of LMs: StarCoderBase, Code

Model	RAG Policy	API Completion		Line Completion	
		ES	SU	ES	SU
SCB-7B	Always Retrieving	73.65	0%	78.15	0%
	REPOFORMER-1B	74.10	24%	78.31	25%
SCB-16B	Always Retrieving	74.50	0%	79.24	0%
	REPOFORMER-1B	74.84	24%	79.48	24%
CG25-7B	Always Retrieving	63.07	0%	68.42	0%
	REPOFORMER-1B	63.37	20%	68.86	29%
CL-7B	Always Retrieving	58.75	0%	59.99	0%
	REPOFORMER-1B	58.91	25%	60.47	28%
CL-16B	Always Retrieving	61.08	0%	61.58	0%
	REPOFORMER-1B	62.10	32%	62.45	30%
CHATGPT	Always Retrieving	63.38	0%	61.76	0%
	REPOFORMER-1B	64.01	28%	61.92	18%

Table 4. Accuracy and latency of larger code LMs as the generation model and with REPOFORMER-1B as the policy model for selective RAG. SCB = StarCoderBase, CG25 = CodeGen25, CL = Code Llama. **SU** = Speedup compared to Always Retrieving (the higher, the better). Compared to the Always Retrieving baseline, REPOFORMER's selective decisions improve both the accuracy and latency of these larger LMs.

Llama (Roziere et al., 2023)[8], CodeGen25 (Nijkamp et al., 2023a), and ChatGPT[9]. As shown in Table 4, the selective predictions from REPOFORMER-1B successfully reduce the inference latency with different larger LMs by approximately 25% *while improving their accuracy*. Collectively, the findings indicate that REPOFORMER has acquired robust selective retrieval capabilities that could generalize to diverse types of code LMs.

6 Analysis

In this section, we present further analyses and ablation studies on REPOFORMER-1B.

Is REPOFORMER sensitive to threshold settings? In Figure 4, we present the code completion accuracy and latency of REPOFORMER as a function of the threshold. As the threshold increases, the model's code completion performance first increases due to avoiding potentially harmful retrievals. At threshold 0.4, the model still maintains similar performance compared to always retrieving, with latency reduced by 50%. This result demonstrates that REPOFORMER can accommodate various threshold settings and provide a good accuracy-latency trade-off. We provide the visualization for other tasks in Appendix E.4.

Does REPOFORMER make accurate and calibrated selective retrieval decisions? In Figure 5, we evaluate the precision of retrieval abstention decisions made by REPOFORMER's threshold selection strategy. We find that the abstentions are accurate for over 80% instances across all

[8]We accessed the model through Amazon SageMaker (https://docs.aws.amazon.com/sagemaker/).
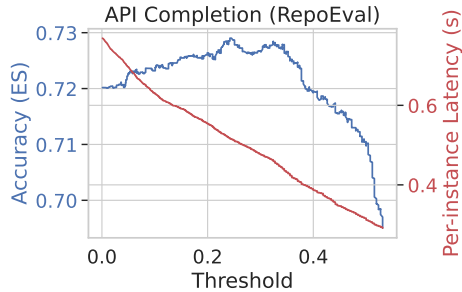[9]We use gpt-3.5-turbo-0613 via the OpenAI API.

Figure 4. The accuracy and latency change with different threshold settings. Selective Retrieval with REPOFORMER achieves better accuracy and better latency than always retrieving. In addition, this behavior is relatively insensitive to the threshold.
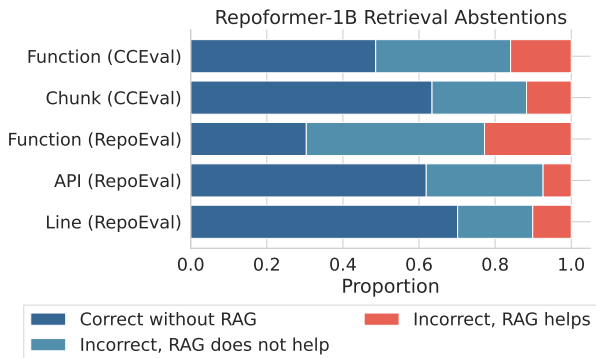

Figure 5. An analysis of the instances where REPOFORMER-1B abstains from retrieval. We divide the instances into (1) the model answering correctly without retrieval (dark blue), the model making a mistake that cannot be improved by retrieval (light blue), and the model achieving better performance when retrieval is performed (red). The precision of abstention is over 0.8 on all tasks except for Function (RepoEval), which has a precision of 0.78.

the tasks: when REPOFORMER abstains from retrieval, its code completion prediction either is already correct without retrieval or cannot be improved by retrieval. We also evaluate the calibration of the selective decisions and find REPOFORMER generally making near-calibrated predictions for line and API completion while the calibration is suboptimal for function completion with UT employed as the metric (Appendix E.1). We hypothesize that this could be caused by using ES to create the training signal and encourage future work to devise methods for labeling the quality of function completion more effectively.

Is REPOFORMER robust to retrieval? In Figure 6, we show the performance change caused by CC on the instances where REPOFORMER requests for retrieval. Compared to STARCODERBASE, REPOFORMER exhibits more and greater performance gains upon observing CC. The number of performance decreases is also significantly reduced, indicating an improved robustness to the potentially
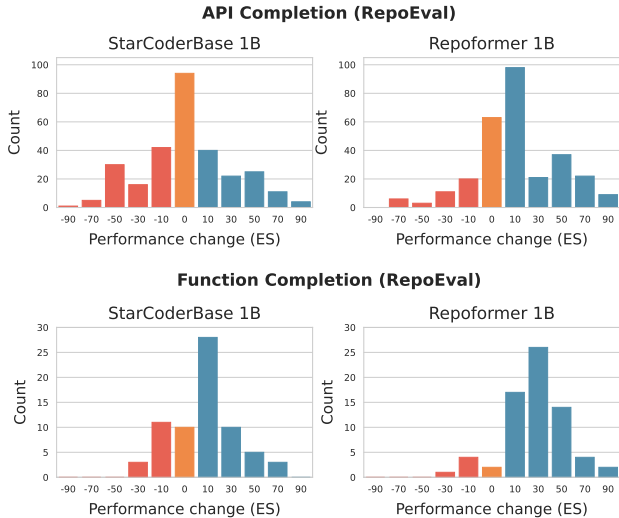
Figure 6. The performance change on RepoEval from retrieved cross-file context for the instances where REPOFORMER self-selects retrieval. Compared to StarCoderBase, REPOFORMER is better at leveraging CC to improve the generation quality.

Model	RAG Policy	Chunk Completion			Function Completion		
		T	%RAG	ES	T	%RAG	ES
SC	No	-	0%	60.09	-	0%	47.49
	Always	-	100%	63.73	-	100%	50.50
RF	No	-	0%	66.22	-	0%	49.77
	Selective$_T$	0.20	75%	69.97	0.15	76%	53.71
	Always	-	100%	69.95	-	100%	53.56
A1	No	-	0%	66.14	-	0%	49.25
	Selective$_T$	0.99	100%	70.21	0.99	100%	53.93
	Always	-	100%	70.21	-	100%	53.93
A2	No	-	0%	66.49	-	0%	49.02
	Always	-	100%	70.45	-	100%	53.90
A3	No	-	0%	66.25	-	0%	49.01
	Always	-	100%	68.85	-	100%	52.12
A4	No	-	0%	64.96	-	0%	25.44
	Selective$_T$	0.10	86%	69.35	0.10	83%	26.50
	Always	-	100%	69.19	-	100%	26.35

Table 5. Ablation study results. We report the performance on two tasks from the CCEval dataset. **SC** = StarCoderBase-1B. **RF** = REPOFORMER-1B. **T** = threshold for the Selective$_T$ policy. We found T = 0.10 works better for A4 and thus applied it to all the A4 results. **%RAG** = ratio of instances where RAG is performed.

We hypothesize that placing CC within the infilling part is detrimental due to breaking the fill-in-the-middle semantics learned in StarCoder pre-training.

7 Conclusion

In this paper, we challenge the common assumption of always performing retrieval for RAG-based repository-level code completion. In response, we propose a selective retrieval augmentation framework powered by REPOFORMER, a code LM that identifies whether cross-file context is necessary, and self-triggers retrieval. Extensive evaluations demonstrate our approach's effectiveness in enhancing accuracy while significantly reducing latency, showcasing its potential in practical coding environments.

Discussion Building upon REPOFORMER, future research may consider several important directions:

1. **Further speeding up large LMs.** Beyond as a selective retrieval policy, REPOFORMER has the potential to serve as an effective plug-in draft model in settings such as *speculative decoding* (Chen et al., 2023).
2. **More effective function completion.** To enable a good scalability, we used lexical similarity as the signal for training label creation. Although this heuristics enables improvements in function completion evaluation, designing a more accurate and scalable labeling approach is an important future direction.
3. **Personalized retrieval.** We apply a uniform selective policy across repositories. However, certain repositories could be inherently more RAG-friendly by exhibiting a higher level of duplication (Zhang et al., 2023). Adapting the selective RAG paradigm towards accurate personalized policies is an important direction.

irrelevant retrieval contexts. In Table 8 in the appendix, we further study the effect of using *dense retrieval*. Although dense retrieval returns an arguably different context distribution compared to sparse retrieval, REPOFORMER still exhibits strong improvements in both quality and latency.

Ablation Study We study several alternative designs:

- (**A1**) Combining $\mathcal{L}_{eval}$ and $\mathcal{L}_{gen}$ as a single cross-entropy loss. In general, this down-weights $\mathcal{L}_{eval}$.
- (**A2**) Removing the self-evaluation loss $\mathcal{L}_{eval}$.
- (**A3**) Further removing all the CC from A2. This amounts to only training on fill-in-the-middle.
- (**A4**) Placing $<$cc$>$ and CC after $<$fim_middle$>$ and marking its end with a new token $<$cc_end$>$. A4 mainly studies whether it is more beneficial to train the LM to treat CC as context fetched during fill-in-middle generation instead of part of the input context.

We fine-tune StarCoderBase-1B with the same setup as REPOFORMER and present the results on CCEval in Table 5. Although **A1** has slightly better RAG performance, it fails to make meaningful selective decisions due to $\mathcal{L}_{eval}$ being outweighed by $\mathcal{L}_{gen}$ in long sequences: the probability of $<$cc$>$ is almost always 1. For **A2**, we find it only slightly outperforms REPOFORMER, suggesting learning $\mathcal{L}_{eval}$ does not harm the RAG ability a lot while bringing in the strong selective retrieval ability, which in turn boosts both accuracy and latency. **A3** has the same performance for in-file completion as REPOFORMER, but exhibits worse RAG performance, indicating the necessity of training with CC. Finally, **A4** achieves reasonable chunk completion performance but performs much worse in function completion.

Acknowledgement

Impact Statement

Our research introduces a novel approach to repository-level code completion that significantly enhances efficiency and accuracy by employing selective retrieval, reducing unnecessary computational waste, and contributing to more sustainable software development practices. Although promising in streamlining development workflows and potentially applicable in various domains, it is important to consider the implications of increased automation in software development, programming education, and the potential for inadvertent biases. Ensuring the ethical use and ongoing evaluation of such code automation technologies is crucial to maximize their societal benefits while mitigating risks. In addition, as a general infrastructure, it is important to design additional mechanisms that prevent RAG systems from revealing sensitive data in the retrieval database.

In this work, we mainly rely on open-sourced, permissively-licensed repositories (the Stack, CrossCodeEval) and models (StarCoder, CodeGen) to perform the experiments. However, as mentioned by Ding et al. (2023), some of the repositories of RepoEval are with non-permissive licenses. We rely on the dataset and code distributed by the original RepoEval authors to perform the experiment and do not redistribute the dataset or adapt it for other purposes.

References

Asai, A., Wu, Z., Wang, Y., Sil, A., and Hajishirzi, H. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=hSyW5go0v8.

Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., and Chen, M. Efficient training of language models to fill in the middle. *ArXiv preprint*, abs/2207.14255, 2022. URL https://arxiv.org/abs/2207.14255.

Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling. *ArXiv preprint*,

abs/2302.01318, 2023. URL https://arxiv.org/abs/2302.01318.

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*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Ding, Y., Wang, Z., Ahmad, W. U., Ding, H., Tan, M., Jain, N., Ramanathan, M. K., Nallapati, R., Bhatia, P., Roth, D., and Xiang, B. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://arxiv.org/abs/2310.11248.

Ding, Y., Wang, Z., Ahmad, W. U., Ramanathan, M. K., Nallapati, R., Bhatia, P., Roth, D., and Xiang, B. Co-CoMIC: Code completion by jointly modeling in-file and cross-file context. pp. 3433–3445, May 2024. URL https://aclanthology.org/2024.lrec-main.305.

Drozdov, A., Wang, S., Rahimi, R., McCallum, A., Zamani, H., and Iyyer, M. You can't pick your neighbors, or can you? when and how to rely on retrieval in the kNN-LM. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 2997–3007, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-emnlp.218. URL https://aclanthology.org/2022.findings-emnlp.218.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.499. URL https://aclanthology.org/2022.acl-long.499.

He, J., Neubig, G., and Berg-Kirkpatrick, T. Efficient nearest neighbor language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 5703–5714, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.461. URL https://aclanthology.org/2021.emnlp-main.461.

Hellendoorn, V. J. and Devanbu, P. T. Are deep neural networks the best choice for modeling source code? In Bodden, E., Schäfer, W., van Deursen, A., and Zisman, A. (eds.), *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pp. 763–773.

ACM, 2017. doi: 10.1145/3106237.3106290. URL https://doi.org/10.1145/3106237.3106290.

Hill, R. and Rideout, J. Automatic method completion. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pp. 228–235. IEEE Computer Society, 2004. doi: 10.1109/ASE.2004.10034. URL https://doi.ieeecomputersociety.org/10.1109/ASE.2004.10034.

Jaccard, P. The distribution of the flora in the alpine zone.1. *New Phytologist*, 11:37–50, 1912. URL https://api.semanticscholar.org/CorpusID:85574559.

Jain, N., Zhang, D., Ahmad, W. U., Wang, Z., Nan, F., Li, X., Tan, M., Nallapati, R., Ray, B., Bhatia, P., Ma, X., and Xiang, B. ContraCLM: Contrastive learning for causal language model. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6436–6459, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.355. URL https://aclanthology.org/2023.acl-long.355.

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 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.495. URL https://aclanthology.org/2023.emnlp-main.495.

Kadavath, S., Conerly, T., Askell, A., Henighan, T., Drain, D., Perez, E., Schiefer, N., Hatfield-Dodds, Z., DasSarma, N., Tran-Johnson, E., et al. Language models (mostly) know what they know. *ArXiv preprint*, abs/2207.05221, 2022. URL https://arxiv.org/abs/2207.05221.

Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., et al. The stack: 3 tb of permissively licensed source code. *ArXiv preprint*, abs/2211.15533, 2022. URL https://arxiv.org/abs/2211.15533.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In Flinn, J., Seltzer, M. I., Druschel, P., Kaufmann, A., and Mace, J. (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Levenshtein, V. I. et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pp. 707–710. Soviet Union, 1966. URL https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.

Li, J., Tang, T., Zhao, W. X., Wang, J., Nie, J.-Y., and Wen, J.-R. The web can be your oyster for improving language models. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 728–746, Toronto, Canada, July 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.46. URL https://aclanthology.org/2023.findings-acl.46.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., V, R. M., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Moustafa-Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you!, 2023b. URL https://doi.org/10.48550/arXiv.2305.06161.

Lu, S., Duan, N., Han, H., Guo, D., Hwang, S.-w., and Svyatkovskiy, A. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6227–6240, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.431. URL https://aclanthology.org/2022.acl-long.431.

Mallen, A., Asai, A., Zhong, V., Das, R., Khashabi, D., and Hajishirzi, H. 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)*, pp. 9802–9822, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.546. URL https://aclanthology.org/2023.acl-long.546.

Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., and Zhou, Y. Codegen2: Lessons for training llms on programming and natural languages. *ICLR*, 2023a. URL https://arxiv.org/abs/2305.02309.

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. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023b. URL https://openreview.net/pdf?id=iaYcJKpY2B_.

OpenAI. Gpt-4 technical report. *ArXiv preprint*, abs/2303.08774, 2023. URL https://arxiv.org/abs/2303.08774.

Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972. doi: 10.1145/361598.361623. URL https://doi.org/10.1145/361598.361623.

Pei, H., Zhao, J., Lausen, L., Zha, S., and Karypis, G. Better context makes better code language models: A case study on function call argument completion. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4):5230–5238, Jun. 2023. doi: 10.1609/aaai.v37i4.25653. URL https://ojs.aaai.org/index.php/AAAI/article/view/25653.

Ram, O., Levine, Y., Dalmedigos, I., Muhlgay, D., Shashua, A., Leyton-Brown, K., and Shoham, Y. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics*, 11:1316–1331, 2023. doi: 10.1162/tacl_a_00605. URL https://aclanthology.org/2023.tacl-1.75.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv preprint*, abs/2009.10297, 2020. URL https://arxiv.org/abs/2009.10297.

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*, abs/2308.12950, 2023. URL https://arxiv.org/abs/2308.12950.

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*, abs/2301.12652, 2023. URL https://arxiv.org/abs/2301.12652.

Shrivastava, D., Kocetkov, D., de Vries, H., Bahdanau, D., and Scholak, T. Repofusion: Training code models to understand your repository. *ArXiv preprint*, abs/2306.10998, 2023a. URL https://arxiv.org/abs/2306.10998.

Shrivastava, D., Larochelle, H., and Tarlow, D. Repository-level prompt generation for large language models of code. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31693–31715. PMLR, 2023b. URL https://proceedings.mlr.press/v202/shrivastava23a.html.

Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. Intellicode compose: code generation using transformer. In Devanbu, P., Cohen, M. B., and Zimmermann, T. (eds.), *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pp. 1433–1443. ACM, 2020. doi: 10.1145/3368089.3417058. URL https://doi.org/10.1145/3368089.3417058.

Tu, Z., Su, Z., and Devanbu, P. T. On the localness of software. In Cheung, S., Orso, A., and Storey, M. D. (eds.), *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pp. 269–280. ACM, 2014. doi: 10.1145/2635868.2635875. URL https://doi.org/10.1145/2635868.2635875.

Wang, Y., Shi, E., Du, L., Yang, X., Hu, Y., Han, S., Zhang, H., and Zhang, D. Cocosum: Contextual code summarization with multi-relational graph neural network. *ArXiv preprint*, abs/2107.01933, 2021. URL https://arxiv.org/abs/2107.01933.

Wang, Y., Li, P., Sun, M., and Liu, Y. Self-knowledge guided retrieval augmentation for large language models. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 10303–10315, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.691. URL https://aclanthology.org/2023.findings-emnlp.691.

Ye, Y. and Fischer, G. Supporting reuse by delivering task-relevant and personalized information. In Tracz, W., Young, M., and Magee, J. (eds.), *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pp. 513–523. ACM, 2002. doi: 10.1145/581339.581402. URL https://doi.org/10.1145/581339.581402.

Zan, D., Chen, B., Lin, Z., Guan, B., Yongji, W., and Lou, J.-G. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 277–288, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/

2022.findings-emnlp.21. URL https://aclanthology.org/2022.findings-emnlp.21.

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, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151.

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, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=ZTCxT2t2Ru.

A   Detailed RAG Execution Setup

Below, we describe the four steps we follow for executing RAG as well as the related hyperparameters.

1. **Indexing.** All files in F are divided into fix-sized code chunks with a sliding window. We set the chunk size to 20 for line, API, and chunk completion and set 50 for function completion. We use half of the chunk size as the stride size. Despite the duplication caused by the overlap between adjacent chunks, this design improves retrieval accuracy with tolerable cost, as the number of files is limited in a repository compared to large open-domain code corpora.

2. **Query Formation.** A query is constructed based on X_l. We always use a fixed number of lines at the end of X_l (i.e., immediately preceding Y) as the query. The query contains the same number of lines as the chunks in the index.

3. **Retrieval.** A similarity function f is used to compare the query with every chunk and identify k most similar code chunks. We use $k = 10$ and Jaccard similarity (Jaccard, 1912) for f for the main results. Fragment alignment (Lu et al., 2022) is then applied: for each of the k most similar code chunks, the chunk immediately following is included in CC instead of the original chunk. We explored other choices mentioned in Figure 8 such as cosine similarity with UniXCoder (Guo et al., 2022) or CodeBLEU (Ren et al., 2020), but find them failing to outperform Jaccard similarity.

4. **Generation.** CC is concatenated with the in-file context as a prompt for $\mathcal{M}$. The prompt is provided below.

Prompt   Recent literature demonstrates the effectiveness of directly providing the retrieved information as part of the context of LMs (Ram et al., 2023; Shi et al., 2023). Following these studies, we directly concatenate the in-file context with CC to provide it to the model (Figure 1). To prompt CodeGen-Mono, we use the following input ordering:

```
[Right Context] [Cross-file Context] [Left Context]
```

To prompt StarCoder, we use the following fill-in-the-middle-prompt:

```
<fim_prefix> [Left Context] <fim_suffix> [Right Context] [Cross-file Context] <fim_middle>
```

For the cross-file contexts, we add a # symbol to present them as comments and add the following line before each cc_i:

```
# the below code fragment can be found in: [file path]
```

After concatenating the verbalized cc_i together, we add another line to the start of the CC:

```
# Here are some relevant code fragments from other files of the repo:
```

For the in-file completion baselines such as in Section 5.1 and Appendix B, our prompts are exactly the previous prompts with the `[Cross-file Context]` part removed.

Decoding and Post-processing   For all the experiments, we follow previous work and use greedy search (Zhang et al., 2023; Ding et al., 2023). We left-truncate the left context to 1024 tokens, right-truncate the right context to 512 tokens, and right-truncate the cross-file context to 512 tokens. The max generation length is set to 50 tokens for line, API, and chunk completion, and 256 tokens for function completion. We perform task-specific post-processing on the model's raw predictions. For line, API, and chunk completion, we truncate the prediction to having the same number of lines as in Y. For function completion, we first add a placeholder `pass` function body and use tree-sitter[10] to determine the position of the function in the file. Then, we concatenate the X_l and $\hat{Y}$, parse the string again with tree-sitter, and extract the function body as the final $\hat{Y}$ if the string can be parsed. Otherwise, we directly return the raw $\hat{Y}$ without post-processing.

B   Why infilling?

As part of the in-file context, X_r contains rich information about how the future execution relies on the code to complete. Right contexts are also shown useful for tasks such as function call argument completion (Pei et al., 2023). However, previous literature such as Zhang et al. (2023) suggests splitting X_r and retrieving code chunks from it. With code LMs trained on fill-in-the-middle such as StarCoder, we argue that directly providing X_r in the prompt is more preferable.

To illustrate, we investigate the effect of directly providing X_r in the prompt for CodeGen-Mono 16B and StarCoder on current-file code completion and retrieval-augmented code completion. Figure 7 presents the performance on RepoEval

---

[10]https://tree-sitter.github.io/tree-sitter/

with different types of contexts provided in the prompt. Whether cross-file contexts are present or not, providing right contexts can greatly improve the code completion performance. The gain is consistent for both API and function completion. Compared to CodeGen, StarCoder can better leverage the right context to generate more accurate code. Overall, we observe that leveraging the entire right context to perform infilling represents a much stronger baseline. Therefore, in this paper we have exclusively focused on the infilling setting with the StarCoder family.
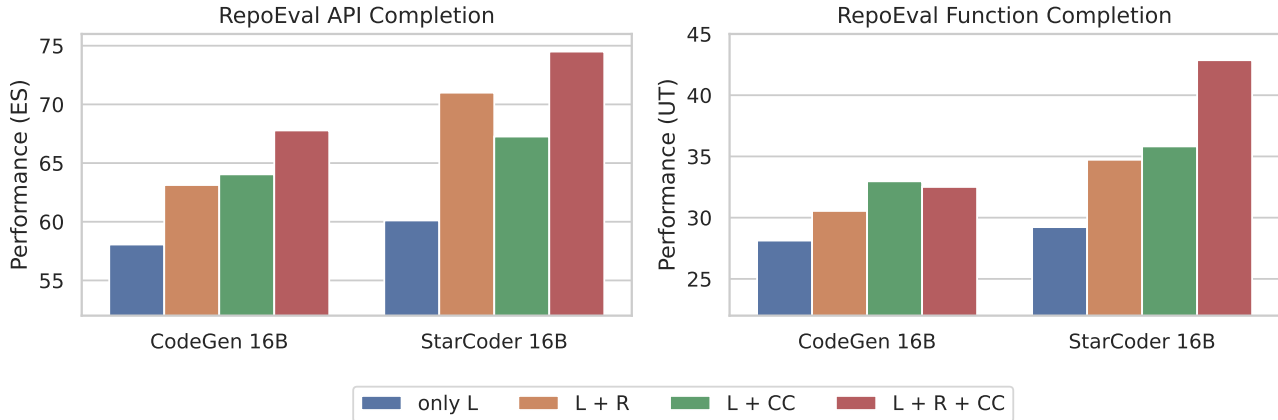

Figure 7. A comparison between four prompting strategies for RepoEval by combining left context (L), right context (R), and cross-file contexts (CC). Leveraging right contexts to build infilling-style prompt generally improves the performance regardless whether CC is present or not. StarCoder exhibits larger gains from right contexts, potentially due to its fill-in-the-middle pre-training.

C   Trial Retrieval and Trial Generation

In this section, we present a detailed evaluation of two selective RAG strategies: trial retrieval and trial generation.

C.1   Trial Retrieval

To gauge the relevance of retrieved context, using the similarity scores from the retrievers is a natural option. In this section, we investigate *trial retrieval* as a baseline for informing the decisions for selective RAG. We apply three off-the-shelf retrievers on RepoEval. For each retriever, we score each of the instances with the similarity between the top-1 retrieved code chunk and the query. The score is compared to a threshold decide whether the prompt should feature CC or not. If score is higher than the threshold, we use top-10 code chunks retrieved by the same retriever as the cross-file context. We consider the following three retrievers:

- **jaccard**: the Jaccard index (Jaccard, 1912).

- **weighted_ngram**: the weighted n-gram matching term introduced in the CodeBLEU metric (Ren et al., 2020).

- **unixcoder**: the cosine similarity of UniXcoder embedding (Guo et al., 2022).

Figure 8 presents the selective RAG performance of StarCoder under different budgets. We observe that the retrievers' similarity scores serve as a promising signal for deciding whether the retrieved information can improve the RAG performance. For most retrievers and tasks, the performance of full retrieval could be reached with at most 60% retrieval budget. This trend also aligns with the remark in Zhang et al. (2023) on the correlation between in-repository duplication and the gain from CC. However, it is worth noting that this strategy brings no latency gain as it still implements always retrieving. In addition, the knowledge of whether the LM could be benefited by the retrieved context is not leveraged.

C.2   Trial Generation

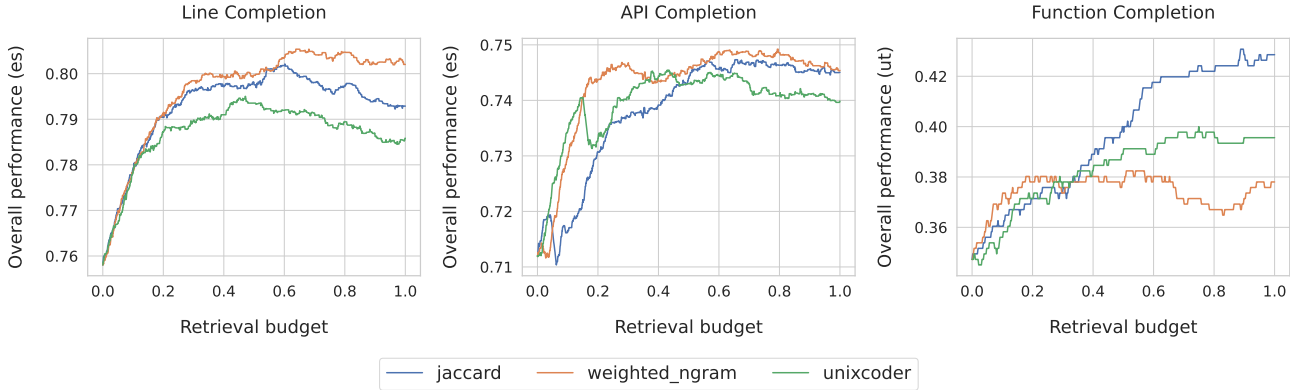Next, we evaluate two uncertainty-based selective RAG strategies that have been explored by previous works.

Figure 8. A comparison of the effectiveness of different similarity functions for selective RAG with StarCoder 16B. We plot the retrieval budget in the x-axis, which is the percentage of instances to perform retrieval. We report score on the entire testing dataset for each budget. Specifically, the retriever's similarity score is used select a subset to perform retrieval, and for the other instances in-file completion is performed without retrieval. In most of the cases, 40% retrieval can be saved without sacrificing the code completion performance.
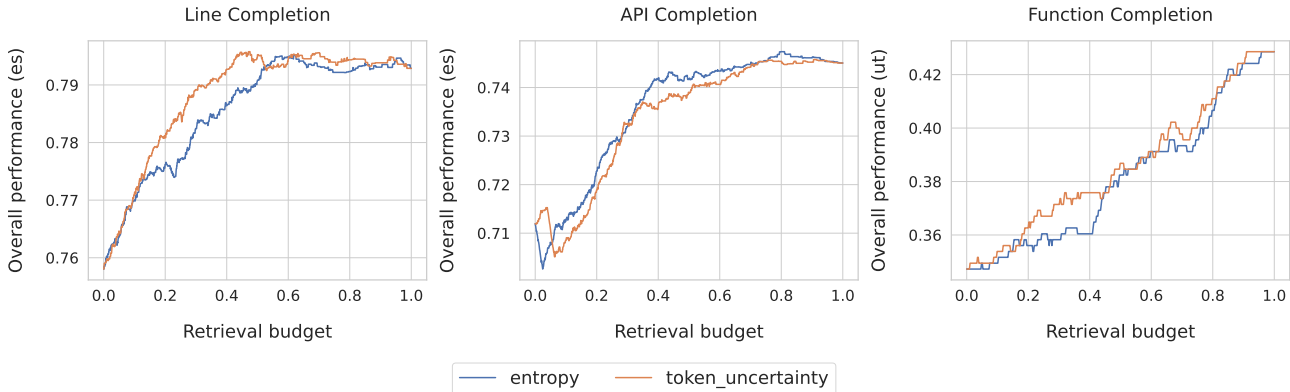

Figure 9. A comparison of the effectiveness of two uncertainty metrics for selective RAG with StarCoder 16B. We plot the retrieval budget in the x-axis and report score on the entire testing dataset for each budget. We observe that the uncertainty-based metrics fail for long sequence generation such as function completion. Token uncertainty outperforms entropy for line completion while entropy is slightly better for API completion. Overall, we find that uncertainty-based selective RAG is not as effective as retriever-based (Figure 8).

- **entropy**: the sequence-level entropy as used in Li et al. (2023a). We estimate the entropy by performing vanilla sampling for 20 times without any temperature scaling or distribution truncation.

- **token uncertainty**: the probability of the most unlikely token in the sequence decoded with greedy search, as used in Jiang et al. (2023). This metric can be seen as the lower bound of the per-token maximum probability.

Figure 9 presents the selective RAG performance of StarCoder under different budgets, similar to the previous evaluation setting. We find that the selective RAG performance of uncertainty-based metrics is inconsistent across sequence lengths. As the length of $\hat{Y}$ increases (from line to API, and form API to function), the effectiveness of uncertainty-based metrics drops significantly. In addition, the selective performance cannot outperform the methods based on trial retrieval.

D   Data Creation for REPOFORMER Training and CrossCodeLongEval

We present the full self-supervised data creation algorithm in Algorithm 1 (for chunk completion data) and Algorithm 2 (for function completion data). $R_{filtered}$ stands for the remaining repositories after applying the filtering criteria in Section 3.3. In the next section, we present further analyses on the training data distribution.

Algorithm 1 REPOFORMER Training Data Creation (Chunk Completion)

Input: Filtered set of repositories $R_{filtered}$, language model $\mathcal{M}$, label threshold T
Output: chunk completion training dataset $\mathcal{D}$
$\mathcal{D} \leftarrow \emptyset$
for each $r \in R_{filtered}$ **do**
    $\mathcal{D}_r \leftarrow \emptyset$
    $\mathcal{C}_{raw} \leftarrow$ Break r into non-overlapping chunks of 10 lines each
    $\mathcal{C}_r \leftarrow$ Cluster $\mathcal{C}_{raw}$ with KMeans using TF-IDF features, with the constraint $|\mathcal{C}_r| = 0.2|\mathcal{C}_{raw}|$
    for each $c \in \mathcal{C}_r$ **do**
        $k \sim \text{Poisson}(\lambda = 3)$
        $s \leftarrow$ Randomly sample a chunk from c
        $Y \leftarrow$ Randomly cut a sub-chunk from s that spans k consecutive lines
        $X_l, X_r \leftarrow$ Recover the in-file left context and right context corresponding to Y
        if $rand(0,1) > 0.5$ **then**
            $\mathcal{Q} \leftarrow$ Concatenate(last $5k$ lines of X_l, Y, first $5k$ lines of X_r)   // query formation
        else
            $\mathcal{Q} \leftarrow$ Concatenate(last $5k$ lines of X_l, first $5k$ lines of X_r)
        end if
        $CC \leftarrow$ Retrieve top-3 cross-file contexts from r using $\mathcal{Q}$ via jaccard similarity, each of length $10k$
        $\hat{Y}_{base} \leftarrow \mathcal{M}(X_l, X_r)$
        $\hat{Y}_{RAG} \leftarrow \mathcal{M}(X_l, X_r, CC)$
        $label \leftarrow ES(\hat{Y}_{RAG}, Y) - ES(\hat{Y}_{base}, Y) > T$ // boolean value
        Append $(X_l, X_r, Y, CC, label)$ to $\mathcal{D}_r$
    end for
    $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_r$
end for

Algorithm 2 REPOFORMER Training Data Creation (Function Completion)

Input: Filtered set of repositories $R_{filtered}$, language model $\mathcal{M}$, label threshold T
Output: function completion training dataset $\mathcal{D}$
$\mathcal{D} \leftarrow \emptyset$
for each $r \in R_{filtered}$ **do**
    $\mathcal{D}_r \leftarrow \emptyset$
    $\mathcal{C}_{raw} \leftarrow$ Gather all the functions between 3 and 30 lines
    $\mathcal{C}_r \leftarrow$ Cluster $\mathcal{C}_{raw}$ with KMeans using TF-IDF features, with the constraint $|\mathcal{C}_r| = 0.2|\mathcal{C}_{raw}|$
    for each $c \in \mathcal{C}_r$ **do**
        $s \leftarrow$ Randomly sample a function from c
        $Y \leftarrow$ Cut only the body part of the function
        $X_l, X_r \leftarrow$ Recover the in-file left context and right context corresponding to Y
        if $rand(0,1) > 0.5$ **then**
            $\mathcal{Q} \leftarrow$ Concatenate(last 20 lines of X_l, Y, first 20 lines of X_r)
        else
            $\mathcal{Q} \leftarrow$ Concatenate(last 20 lines of X_l, first 20 lines of X_r)
        end if
        $CC \leftarrow$ Retrieve top-3 cross-file contexts from r using $\mathcal{Q}$ via jaccard similarity, each of length $10k$
        $\hat{Y}_{base} \leftarrow \mathcal{M}(X_l, X_r)$
        $\hat{Y}_{RAG} \leftarrow \mathcal{M}(X_l, X_r, CC)$
        $label \leftarrow ES(\hat{Y}_{RAG}, Y) - ES(\hat{Y}_{base}, Y) > T$ // boolean value
        Append $(X_l, X_r, Y, CC, label)$ to $\mathcal{D}_r$
    end for
    $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_r$
end for

Training Data Analysis For the 240k chunk completion and 120k function completion instances, we plot the performance change after providing CC in Figure 10. In total, 30.18% chunk completion instances and 35.16% function completion instances are labeled with positive (i.e., retrieval should be triggered). The average length of Y is 3.53 lines for chunk completion and 11.77 lines for function completion.
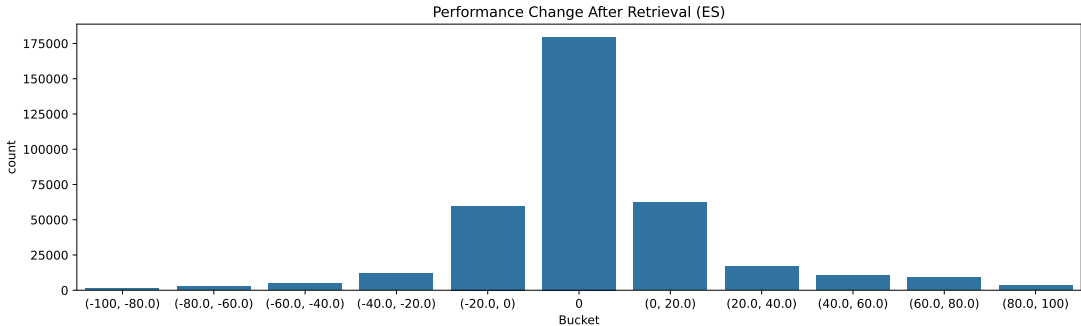

Figure 10. The performance gain on REPOFORMER training data exhibited by StarCoderBase-1B from retrieved cross-file context. The sign of the performance change is used to generate the label for REPOFORMER training. Each (start, end) bucket contains values ranging from start to end except for the central bucket, which corresponds to exactly 0.

CrossCodeLongEval Construction One drawback of RepoEval is its limited repository coverage. To verify the performance on diverse repositories, we collect and curate a new evaluation dataset for repository-level code completion.

- **Repository collection.** We first solicited 1744 raw Python repositories from the authors of CrossCodeEval (Ding et al., 2023). These repositories were created between 2023-03-05 to 2023-06-15 and collected on 2023-09-01. They have been ensured to not overlap with the Stack (Kocetkov et al., 2022).

- **Target line sampling.** We avoided using the CrossCodeEval benchmark as the original benchmark explicit removed the instances where StarCoderBase-1B can correctly answer without the retrieved context. To simulate a more natural distribution of code completion, we sample new blanks from the raw repositories. Specifically, we run Algorithm 1 and Algorithm 2 to gather chunk completion and function completion instances.

- **Data analysis** In Table 6, we present the basic statistics of RepoEval and CrossCodeLongEval.

	RepoEval			CrossCodeLongEval	
	Line	**API**	**Function**	**Chunk**	**Function**
# repositories	16	16	16	944	1460
# instances	1600	1600	455	5000	5000
$|X_l|_{line}$	30.7	30.8	31.1	24.7	31.7
$|X_l|_{token}$	796.3	890.7	761.1	661.9	672.1
$|X_r|_{line}$	15.1	13.9	16.2	12.9	14.4
$|X_r|_{token}$	449.9	430.4	412.4	404.2	371.3
$|Y|_{line}$	1.0	2.1	7.8	1.47	9.5
$|Y|_{token}$	12.0	25.4	97.8	19.2	111.2

Table 6. Descriptive statistics of RepoEval and CrossCodeLongEval. For $|Y|$, $|X_l|$, and $|X_r|$, we report both the number of lines as well as the number of tokens (using the StarCoder tokenizer) in the groundtruth, left context, and the right context.

E   Extended Analyses

E.1   Calibration of REPOFORMER's Selective Retrieval Prediction

We evaluate the calibration of REPOFORMER-1B's selective decisions. Figure 11 plots the probability of <cc> against the probability of the model's performance could be improved by the CC, measured by comparing the prediction with and without CC. When ES is used as the evaluation metric, REPOFORMER-1B generally makes near-calibrated predictions for Line and API Completion. However, when it comes to longer-formed function completion, especially when UT is employed as the metric, REPOFORMER-1B's predictions are not calibrated. One possible reason is the use of ES as the training signal. We encourage future work to devise methods for effectively labeling the correctness of function completion. In addition, future work should consider training REPOFORMER to perform multiple self-assessments for long-form generations.
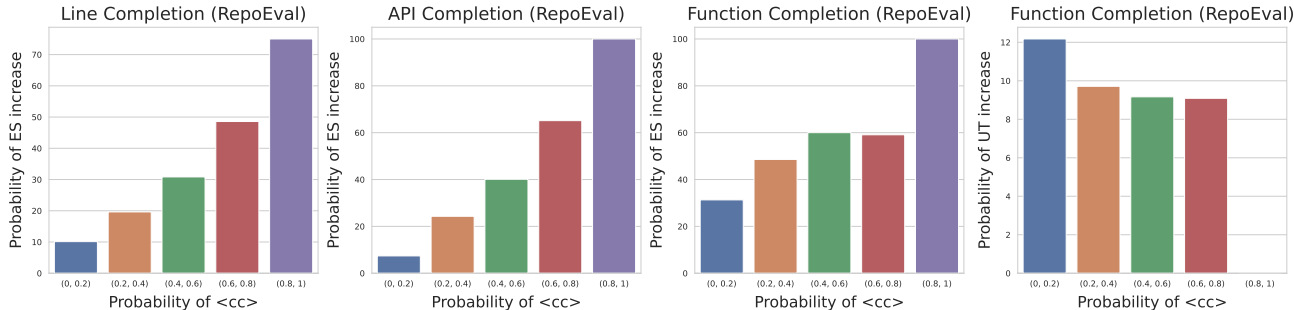

Figure 11. The calibration of selective retrieval predictions. REPOFORMER makes generally calibrated predictions when ES is used as the metric and the generation is of moderate lengths. The prediction is not calibrated for function completion when the metric is UT.

E.2   CrossCodeEval and Multilingual REPOFORMER

This section provides additional results on the 4-language original CrossCodeEval test set (Ding et al., 2023). We choose to not present the results in the main text as the data creation process of CrossCodeEval explicitly selected the instances where cross-file information is generally required, thus making the contributions from selective retrieval incomplete. On this dataset, we evaluate StarCoder, REPOFORMER-1B/3B/7B trained on Python and REPOFORMER-M trained on multilingual repository-level code completion. Despite the setup difference, we are still able to observe substantial performance gains.

Multilingual REPOFORMER   We experimented with applying the REPOFORMER training scheme to multiple languages. Specifically, we collect public Python, Java, C#, and TypeScript repositories from the Stack (Kocetkov et al., 2022) that contain at least 20 files and 20,000 lines of code. We do not apply the local import criteria due to implementation difficulties. Then, we follow the algorithm described in Appendix D to create 90k chunk completion and 30k function completion instances per language. Using this dataset, we fine-tune StarCoderBase following the setup described in Section 4.1 (same infrastructure and hyperparameters). We call this model REPOFORMER-M.

Evaluation Results   We present the results on CrossCodeEval in Table 7 and summarize the observations below:

- **Strong cross-lingual transfer.** REPOFORMER trained on Python data achieves strong performance across multiple languages, including three languages it is not fine-tuned on. The result highlights the generalizability of the learned self-evaluation and robust code completion abilities.

- **Multi-lingual REPOFORMER.** REPOFORMER-M outperforms the same-sized STARCODERBASE by a large margin. For the 1B, 7B, REPOFORMER-M outperforms REPOFORMER by a small margin. For 3B, the two models give similar performance. This is reasonable as the two models are learned on similar sized training data.

Model	RAG Policy	Python		Java		C#		TypeScript	
		Code ES	ID F1	Code ES	ID F1	Code ES	ID F1	Code ES	ID F1
STARCODERBASE-1B	No	68.83	58.18	73.60	63.69	79.30	66.40	67.09	60.15
	Always	71.57	62.42	74.54	65.83	79.04	66.82	67.66	60.60
REPOFORMER-1B	Selective$_T$	71.29	62.81	75.12	67.16	83.08	74.24	69.90	64.07
REPOFORMER-M-1B	Selective$_T$	71.55	62.89	75.92	67.86	84.44	76.00	70.07	64.41
STARCODERBASE-3B	No	71.07	61.63	76.10	67.56	81.46	69.95	70.56	64.83
	Always	73.65	65.93	77.52	70.15	81.75	71.26	70.91	65.09
REPOFORMER-3B	Selective$_T$	74.57	66.86	78.40	71.26	85.92	78.62	73.70	68.66
REPOFORMER-M-3B	Selective$_T$	73.80	66.72	77.68	71.01	85.31	77.70	72.51	67.06
STARCODERBASE-7B	No	72.47	63.76	77.21	68.97	83.06	72.06	72.34	67.06
	Always	75.02	67.69	77.70	70.57	83.64	74.39	73.01	67.56
REPOFORMER-7B	Selective$_T$	75.34	**68.27**	78.90	72.35	83.80	76.88	73.59	69.10
REPOFORMER-M-7B	Selective$_T$	**75.35**	67.88	**79.11**	**72.82**	**86.53**	**79.77**	**74.60**	**70.01**

Table 7. Evaluation results on CrossCodeEval. We report edit similarity for code matching as well as the F1 score for identifier matching. The best scores across all models are boldfaced.

E.3 REPOFORMER's Robustness to the Retriever Choice

In this section, we investigate the performance of REPOFORMER with the cosine similarity of UniXcoder embedding (Guo et al., 2022) as the retriever instead of Jaccard similarity. As shown in Table 8, we are able to observe similar patterns compared to Table 3: selective retrieval is able to improve both the accuracy and the latency of the entire RAG system. In addition, as retrieval consumes a larger proportion of latency than when sparse retriever is used, selective retrieval brings more substantial performance gains, with threshold selection bringing more than 70% speedup.

Model	RAG Policy	API Completion			Line Completion		
		ES	%RAG	SU	ES	%RAG	SU
REPOFORMER-1B	Always	71.69	100%	0%	75.25	100%	0%
	Selective$_G$	70.82	18%	71%	73.70	19%	71%
	Selective$_T$	72.39	61%	33%	75.65	62%	33%
REPOFORMER-3B	Always	74.48	100%	0%	78.24	100%	0%
	Selective$_G$	73.26	19%	65%	76.74	20%	66%
	Selective$_T$	74.69	78%	21%	78.63	74%	31%

Table 8. RAG performance of REPOFORMER with two self-selective RAG paradigms and dense retrieval used instead of Jaccard similarity. **%RAG** = ratio of instances where RAG is performed. **SU** = Speedup compared to always retrieving. Compared to the always retrieving baseline, the Selective$_T$ strategy consistently demonstrates gains in both accuracy and latency. The Selective$_G$ strategy shows much larger latency gains with a small performance degradation. Compared to sparse retrieval, we observe more substantial latency gains.

E.4 Full Latency-Accuracy Visualizations

In this section, we present the latency-accuracy trade-off plots for REPOFORMER-1B, REPOFORMER-3B, STARCODERBASE-7B, and STARCODER on the three tasks from RepoEval. We use self-selective RAG for the RE-POFORMER models and for STARCODER, we use REPOFORMER-1B to make the selective RAG decisions. The results are presented in Figure 12 to Figure 15. Overall, we observe that no matter for self-selective RAG or making selective predictions for a larger model, REPOFORMER is able to improve the accuracy and latency at the same time. The improvement is more apparent in the line and API completion tasks. For function completion, as discussed in the main text, RepoEval uses very small repositories to enable easy unit testing. As a result, the retrieval overhead is low in general and thus does not significantly affect the latency of the entire RAG system.
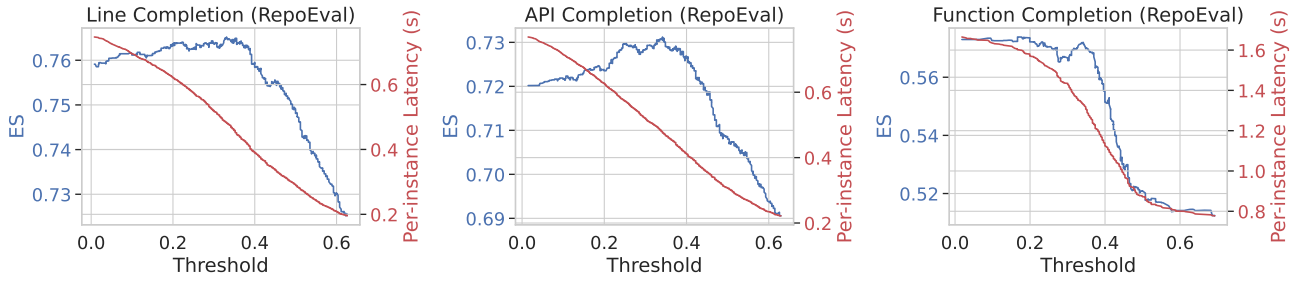
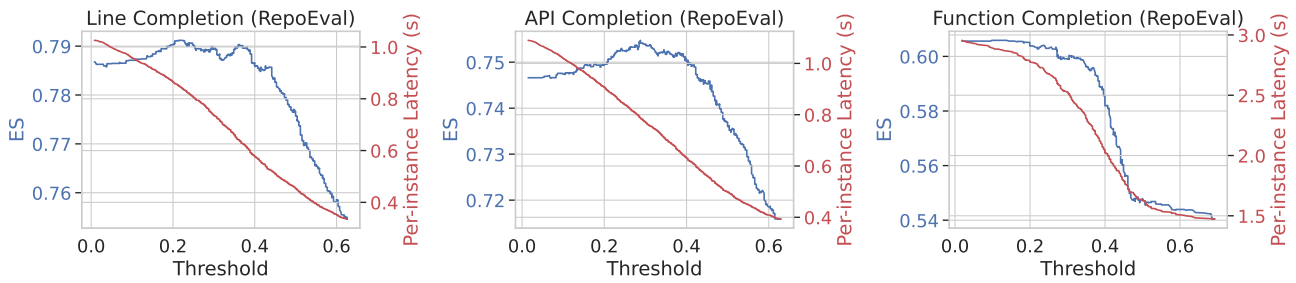*Figure 12.* Latency-accuracy trade-off of self-selective RAG for REPOFORMER-1B.

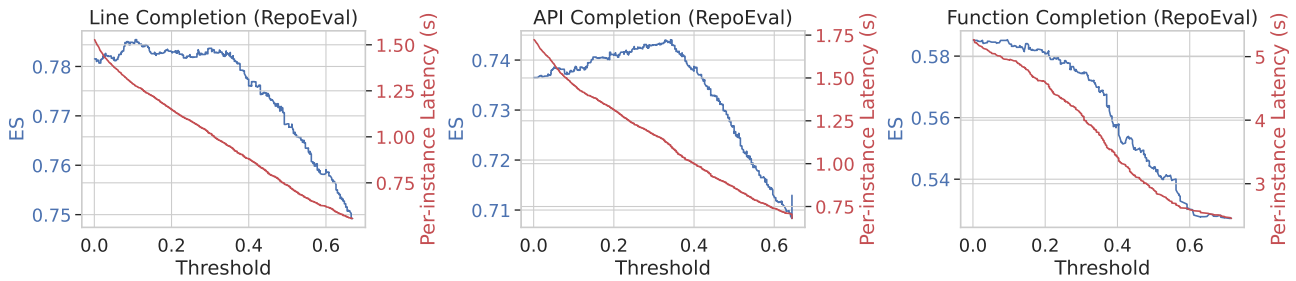*Figure 13.* Latency-accuracy trade-off of self-selective RAG for REPOFORMER-3B.

Figure 14. Latency-accuracy trade-off of selective RAG for STARCODERBASE-7B. REPOFORMER-1B is used for the selective decisions.
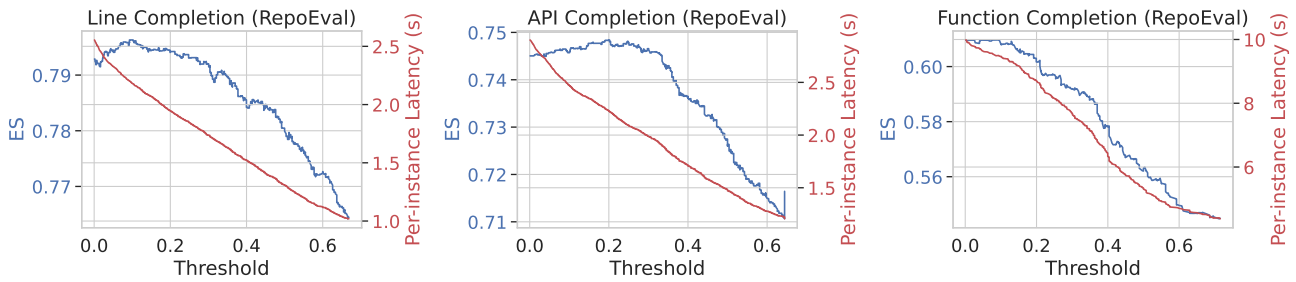

Figure 15. Latency-accuracy trade-off of selective RAG for STARCODER. REPOFORMER-1B is used for the selective decisions.