REPOFORMER: Selective Retrieval for Repository-Level Code Completion

Anonymous Authors

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). To tackle the challenges, this paper proposes a selective RAG framework where retrieval is avoided when unnecessary. To power this framework, we design a self-supervised learning approach that enables 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 retrieval policy and the generation model, our framework consistently outperforms the state-of-the-art prompting with an invariable retrieval approach on diverse benchmarks including RepoEval, CrossCodeEval, and a new benchmark. Meanwhile, our selective retrieval strategy results in strong efficiency improvements by as much as 70% inference speedup without harming the performance. We demonstrate that our framework effectively accommodates 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 current file's context to generate coherent code of flexible granularity (Tu et al., 2014; Svyatkovskiy et al., 2020; Chen et al., 2021). However, their performance is limited by the lack of repository-specific knowledge such as user-defined APIs and inter-module dependencies that require information beyond the current file to fully comprehend (Zan et al., 2022; Zhang et al., 2023; Ding et al., 2023). To bridge this gap, recent approaches adopt the *retrieval-augmented generation* (RAG) paradigm: cross-file contexts including code snippets or documentations are retrieved from the same repository and provided to code LMs as augmentations to the current file. Under this paradigm, existing works have advanced the retrieval mechanism for prompting black-box code LMs (Lu et al., 2022; Shrivastava et al., 2023b; Zhang et al., 2023) as well as adapted the LM to better leverage structured retrieved contexts such as classes, functions, or APIs (Ding et al., 2022; 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, primarily for two reasons. First, the benefit from retrieval is often sparse. On various code completion tasks, we discover that up to 80% of retrievals made by a standard RAG method do not enhance the performance of strong code LMs such as StarCoder (Li et al., 2023b), with many degrading performances by introducing irrelevant information (Section 5.1). Second, always retrieving introduces notable inefficiencies. The size of retrieval index grows linearly with the number of lines in the repository. For moderately sized repositories, sparse retrieval is already as time consuming as completion with a 3B LM (Section 5.3 and 6). This inefficiency is more pronounced with dense retrieval, enterprise-scale repositories, and iterative retrieval methods (Zhang et al., 2023). Hence, there is a critical need to reevaluate the"invariable retrieval" assumption. Such reevaluation is not only pivotal for enhancing code completion but also offers valuable insights for a broad range of similar applications that use domain-specific structured retrieval.

To tackle these challenges, we propose a novel repository-level code completion framework underpinned by a *selective retrieval* mechanism: the system uses a policy to proactively abstain from unnecessary or potentially detrimental retrievals (Figure 1 (a)). At the core of our framework is REPOFORMER, an intelligent code LM fine-tuned for *robust*

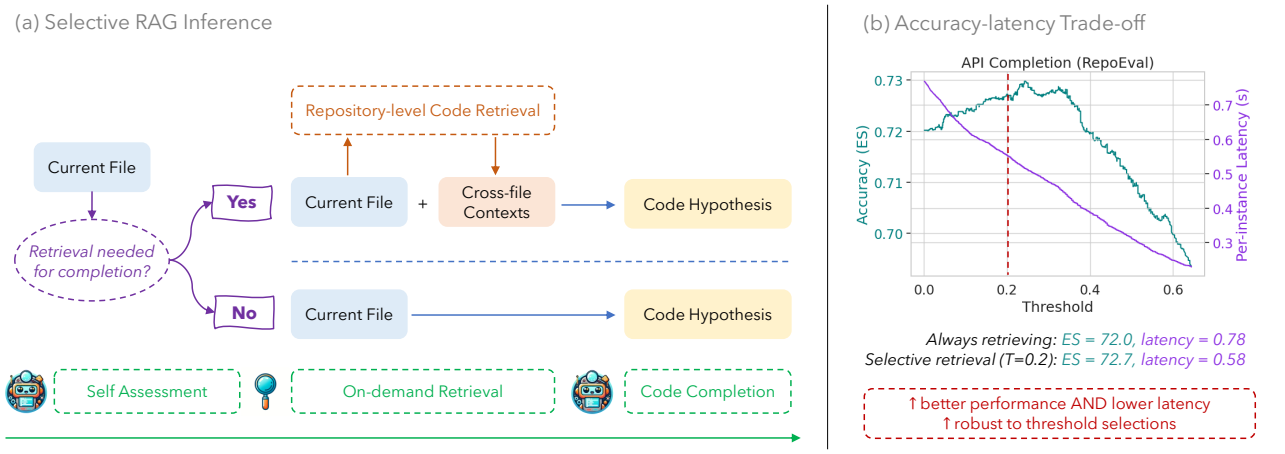

Figure 1. (a) An overview of the proposed selective RAG framework. Given the current file context, a policy first assesses whether retrieval is required and triggers the retriever selectively. Then, the code LM makes a hypothesis with the optional retrieved context. With REPOFORMER, the two stages are streamlined via self-assessment. (b) With REPOFORMER acting as both the selective retrieval policy and the generator LM, our framework achieves better accuracy and better latency than performing retrieval invariably.

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 would be improved by additional retrieval. Our training strategy enables the model to combine two factors for abstention: the *LM* already knowing the answer without retrieval (Kadavath et al., 2022) and the code completion *question* not depending on cross-file information to answer 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, language, and retriever. In addition, as a selective retrieval policy, REPOFORMER should be able to function as a plug-and-play policy for other black-box code LMs.

Correspondingly, we design a simple self-supervised RAG simulation procedure to create the training data. Diverse code chunks and function bodies are sampled and paired with the retrieved cross-file contexts. Then, the ground-truth label for selective retrieval can be obtained by contrasting the quality of a code LM's outputs with and without retrieval augmentation. With this dataset, a multi-task objective is designed to jointly teach the code LM the ability to self-evaluate and robustly completing the code with optional retrieval augmentation (Section 3.3).

We perform comprehensive evaluations on a variety of repository-level code completion tasks from RepoEval (Zhang et al., 2023), CrossCodeEval (Ding et al., 2023), and a newly created large-scale benchmark. Results show that REPOFORMER achieves strong performance, outperforming invariable retrieval with the same-sized StarCoderBase by more than 3 absolute points for edit similarity across multiple tasks. The 3B REPOFORMER even performs on par with invariable retrieval using the 16B StarCoder (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 code 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, we analyze REPOFORMER's threshold sensitivity, the precision and calibration of its abstention decisions, its robustness to retrieved contexts, and its generalization to other languages or retrievers (Section 6). We will release our code, model, and the new benchmark to facilitate future studies.

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., 2022b; 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., 2022; 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. 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., 2023) 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 good abstention 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. It is worth noting that for generation, 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 is 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 (a)).

For this selective decision, two traditional heuristics are relevant: (1) performing a *trial retrieval* and only augmenting the high-relevance contexts or (2) performing a *trial generation* and conducting RAG only when the model's uncertainty is high. We find that these two strategies are informative to some extent: in line completion and API completion from RepoEval, both types of heuristics can achieve the performance of invariable retrieval with only 50% retrieval budget. However, they incur a high latency cost and do not generalize well to all tasks (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 crucial formulation enables us to train a strong classifier that considers both the question's characteristics (i.e., whether it requires cross-file information or not) and the LM's self-knowledge (i.e., whether the LM can already correctly answer without retrieval, as explored by Kadavath et al. (2022)). After this optional retrieval step, the LM proceeds with the code completion with X_l, X_r, combined with CC if retrieval is triggered.

As shown in Figure 2, the inference procedure of self-

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

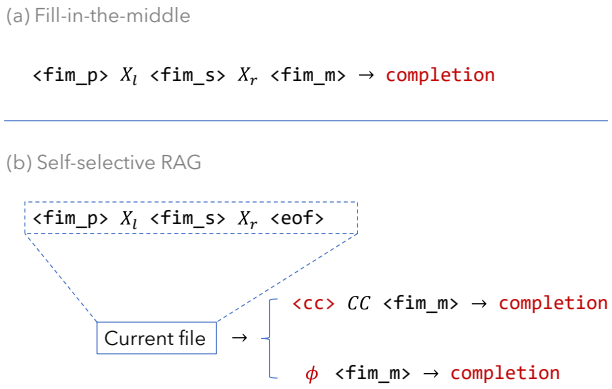

Figure 2. A comparison between fill-in-the-middle and self-selective RAG. We mark the end of the current file with a new token <eof>, which triggers the LM's self-evaluation. → denotes the invocation of the LM. LM-generated parts are colored in red. fim_p, fim_s, and fim_m refer to the special tokens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

selective RAG is conveniently modeled as an extension to fill-in-the-middle (Bavarian et al., 2022). 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 trial generation (such as Jiang et al. (2023)) 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, with or without Y.
3. **Label** whether extending the current file with CC can 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 detailed algorithm is 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>.
$$\mathcal{L}_{eval} = -\log p_{\mathcal{M}}(\texttt{<cc>}|X_l, X_r) \quad (1)$$
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.
$$\mathcal{L}_{gen} = \begin{cases} -\log p_{\mathcal{M}}(Y|X_l, X_r, CC), & \text{if } label \\ -\log p_{\mathcal{M}}(Y|X_l, X_r), & \text{otherwise} \end{cases} \quad (2)$$

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 18k Python repositories from the Stack[2] (Kocetkov et al., 2022) that have (1) at least three imports per file, (2) at least two local imports per file, and (3) at least five Python files. These criteria ensure the existence of local dependencies where RAG could be meaningful. 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 and 3B 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 takes 8 hours for the 1B model and 12 hours for the 3B model with 8 Nvidia A100 GPUs (40G memory). We will call our models REPOFORMER-1B/3B.

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

[2]We have also trained a version of REPOFORMER on a multilingual dataset. The results are reported in Appendix E.2.

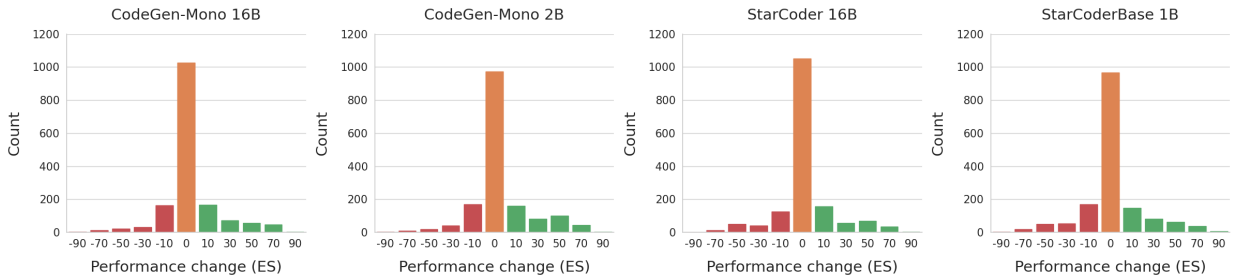

Figure 3. The performance gain on RepoEval API completion from retrieved cross-file contexts. Each bucket contains values ranging from label-10 to label+10 except for the central bucket, which corresponds to exactly 0. The retrieved contexts only improve the performance in about 20% of instances. The trend is consistent across all the evaluated LM families and sizes.

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 improve repository coverage, we additionally leverage 1500 raw Python repositories from CrossCodeEval (Ding et al., 2023) to create a new chunk and function completion benchmark, which we call CCEval (details in Appendix D). To investigate the generalization to other languages, we also evaluated the original CrossCodeEval, which covers four languages: Python, Java, C#, and TypeScript (Appendix E.2).

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

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

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.

Models We experiment on two families of strong code LMs. **CodeGen-Mono** (Nijkamp et al., 2022a) 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.

Model	Size	Performance (UT)		UT Change		
		$X_l + X_r$	$X_l + X_r + CC$	$\downarrow$	$=$	$\uparrow$
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.

5 Results

In this section, we first show that retrieval augmentation is often unhelpful for RAG with black-box code LMs. Then we demonstrate that our framework allows the model to avoid unnecessary retrievals and be more robust to the retrieved contexts, resulting in better accuracy and latency.

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 Figure 3 and Table 1, we evaluate four code LMs on API completion and function 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: retrieval improves LMs' performance on only 20% or fewer instances. For the rest 80%, retrieval augmentation mostly does not affect the performance, but also actually harms the performance almost as often as it helps. The observed trends are consistent for both API and function completion and for both small-sized LMs ($\leq 2B$) and moderate-to-large LMs ($16B$). The generality of this observation is further confirmed by an analysis of REPOFORMER's training data (Appendix D). Together, these findings highlight the suboptimality of the "invariable retrieval" approach and justify our selective retrieval proposal.

Model	Size	Selective Policy	RepoEval Line EM	RepoEval Line ES	RepoEval API EM	RepoEval API ES	RepoEval Function UT	RepoEval Function ES	CCEval Chunk EM	CCEval Chunk ES	CCEval Function ES
No Retrieval											
STARCODERBASE	1B	-	43.44	67.77	37.81	66.54	22.20	47.65	31.08	60.09	47.49
	3B	-	49.00	72.12	40.44	69.02	24.84	51.22	36.14	64.65	49.88
	7B	-	51.88	74.03	43.31	70.79	25.49	52.28	38.88	66.61	52.45
STARCODER	16B	-	55.25	76.07	44.50	71.00	34.73	53.60	42.58	69.40	54.20
Invariable Retrieval											
STARCODERBASE	1B	-	51.19	72.30	43.94	69.17	25.71	55.64	37.22	63.73	50.50
	3B	-	56.69	76.68	47.00	72.62	29.67	57.68	42.26	67.74	53.39
	7B	-	59.44	78.15	**49.56**	73.65	31.43	58.51	44.44	69.53	55.41
STARCODER	16B	-	61.25	79.24	51.12	74.50	42.86	60.96	47.90	71.90	58.06
Selective Retrieval											
REPOFORMER	1B	self-selection	51.90	74.50	43.50	71.00	24.00	53.10	38.52	68.08	52.09
		$P(<\texttt{cc}>)$	54.40	76.00	46.10	72.70	28.79	57.30	41.92	69.97	53.71
	3B	self-selection	56.30	77.60	46.10	73.60	28.57	54.70	42.06	70.70	54.47
		$P(<\texttt{cc}>)$	**59.63**	**79.02**	49.31	**74.96**	**32.96**	**60.56**	**46.66**	**72.23**	**56.24**

Table 2. Experiment results on RepoEval and CCEval. The best performance is underlined, and the best among models under 10B is boldfaced. Compared to STARCODERBASE of the same size, REPOFORMER exhibits a strong performance, with REPOFORMER-3B outperforming other invariable retrieval models under 10B in most of the tasks. More importantly, REPOFORMER consumes a lower retrieval budget via selective retrieval. Among the two selective policies, $P(<\texttt{cc}>)$ enables the best selective RAG performance.

5.2 REPOFORMER achieves strong code completion performance via selective RAG

Next, we evaluate the code completion performance of REPOFORMER. We compare the following three settings[3]. 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 compared to the CodeGen family.

1. **No Retrieval**. This baseline only provides X_l and X_r to the model in the prompt.
2. **Invariable Retrieval**. 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:
   - **Self-selection**. Retrieval is performed if <cc> is the most likely token following <eof>.
   - $P(\texttt{<cc>})$. If the probability of <cc> following <eof> is greater than a threshold T, retrieval augmentation is performed for this instance[4].

The results are summarized in Table 2. Compared to no retrieval and invariable retrieval with StarCoderBase of the same size, REPOFORMER's selective retrieval strategy exhibits strong performance improvements. Via the $P(<\texttt{cc}>)$ 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. We also show that the performance improvement from our paradigm can generalize to three *languages beyond Python* (Appendix E.2), the REPOFORMER-7B (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 an organic combination of the learned abstention ability and the improved robustness to retrieval.

In terms of code completion accuracy, the threshold-controlled $P(<\texttt{cc}>)$ strategy outperforms the self-selection strategy on all the tasks. In the next section, we show that the two strategies represent different ways to trade-off 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" scenario.

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:

- 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 deci-

[3] We do not consider iterative retrieval because we found the first iteration contributes the majority of the performance gains.

[4] We find that $T = 0.15$ for function completion and $T = 0.2$ for the other tasks generally work well for $P(<\texttt{cc}>)$. These two thresholds are always used unless otherwise stated.

	Selective Policy	API Completion ES	%RAG	SU	Line Completion ES	%RAG	SU
1B	invariable retrieval	72.02	100%	0%	75.91	100%	0%
	self-selection	71.04	18%	69%	74.50	19%	61%
	$P(\texttt{<cc>})$	72.72	61%	28%	76.00	62%	27%
3B	invariable retrieval	74.66	100%	0%	78.68	100%	0%
	self-selection	73.60	19%	46%	77.60	20%	43%
	$P(\texttt{<cc>})$	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 invariable retrieval (the higher, the better). Compared to the invariable retrieval baseline, the $P(\texttt{<cc>})$ strategy consistently demonstrates gains in both accuracy and latency. The self-selection strategy shows much larger latency gains with a small performance degradation.

	Selective Policy	API Completion ES	SU	Line Completion ES	SU
7B	invariable retrieval	73.65	0%	78.15	0%
	REPOFORMER-1B	74.10	24%	78.31	25%
16B	invariable retrieval	74.50	0%	79.24	0%
	REPOFORMER-1B	74.84	24%	79.48	24%

Table 4. Accuracy and latency of STARCODERBASE-7B and STARCODER as the generation model and with REPOFORMER-1B as the policy model for selective RAG. **SU** = Speedup compared to invariable retrieval (the higher, the better). Compared to the invariable retrieval baseline, REPOFORMER's selective decisions improve both the accuracy and latency of these larger LMs.

sion. 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[5]. Both selective strategies significantly improve the latency, with a different trade-off: using a fixed threshold for $P(\texttt{<cc>})$ results in *improvements for both accuracy and latency* compared to invariable retrieval, while using self-selection results in a larger latency gain with minor performance degradation (around 1.0 ES). It's worth mentioning that the speed improvement from selective RAG could be further enhanced with a more advanced retrieval setup. For instance, dense retrieval on large repositories often consumes more than 80% of the entire RAG pipeline's latency. In that case, a 20% RAG policy translates into more than 70% speedup.

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

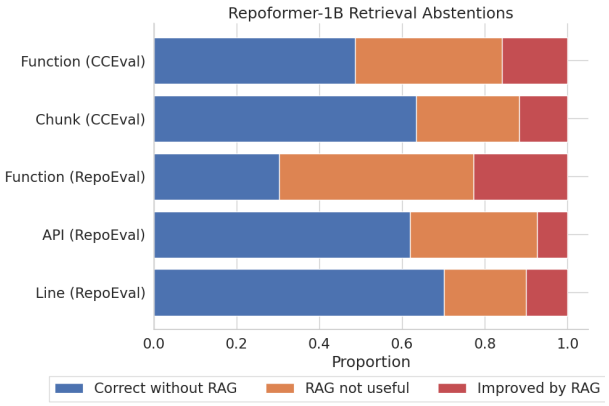

Figure 4. An analysis of the instances where REPOFORMER-1B abstains from retrieval. We divide the instances into (1) the model answering correctly without retrieval (blue), the model making a mistake that cannot be improved by retrieval (yellow), and the model achieving better performance when retrieval 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.

We empirically verify this statement in Appendix E.3.

Next, we consider using larger LMs as $\mathcal{M}$ in the framework and using $P(\texttt{<cc>})$ with REPOFORMER-1B as a *plug-and-play* selective RAG policy. As shown in Table 4, the selective predictions from REPOFORMER successfully reduce the latency of RAG with these 7x and 16x larger LMs by approximately 25% while improving their accuracy. The findings collectively indicate that REPOFORMER has acquired robust selective retrieval capabilities, which could partially transfer to other larger code LMs.

6 Analysis

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

Does REPOFORMER make accurate and calibrated selective retrieval decisions? In Figure 4, we evaluate the precision of retrieval abstention decisions made by REPOFORMER's $P(\texttt{<cc>})$ strategy. We find that the abstentions are accurate for over 80% instances across all the tasks. 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 5, we show the performance change caused by CC on the instances where REPOFORMER requests for retrieval. Com-

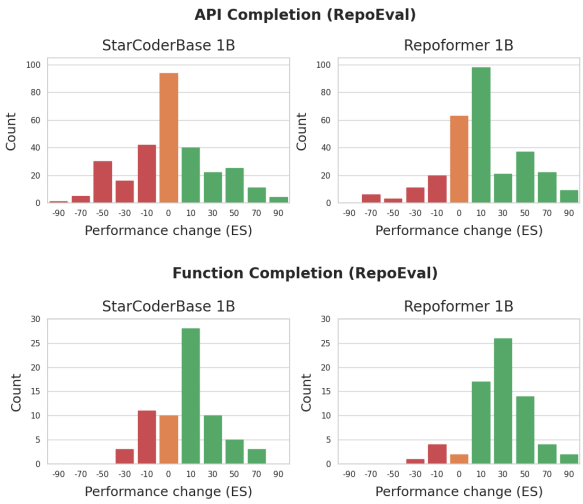

Figure 5. The performance change on RepoEval from retrieved cross-file context for the instances where REPOFORMER self-selects retrieval. Compared to StarCoderBase, REPOFORMER is better at leveraging CC to improve the generation quality.

pared 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.

Is REPOFORMER sensitive to threshold settings? In Figure 1 (b), 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 invariable retrieval, 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.

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 outweighted by $\mathcal{L}_{gen}$ in long sequences: $P(\texttt{<cc>})$ is almost always 1. For **A2**, we find it only slightly outperforms

Model	Selective Policy	Chunk Completion T	Chunk Completion %RAG	Chunk Completion ES	Function Completion T	Function Completion %RAG	Function Completion ES
SC	-	-	0%	60.09	-	0%	47.49
	-	-	100%	63.73	-	100%	50.50
RF	-	-	0%	66.22	-	0%	49.77
	$P(\texttt{<cc>})$	0.20	75%	69.97	0.15	76%	53.71
	-	-	100%	69.95	-	100%	53.56
A1	-	-	0%	66.14	-	0%	49.25
	$P(\texttt{<cc>})$	0.99	100%	70.21	0.99	100%	53.93
	-	-	100%	70.21	-	100%	53.93
A2	-	-	0%	66.49	-	0%	49.02
	-	-	100%	70.45	-	100%	53.90
A3	-	-	0%	66.25	-	0%	49.01
	-	-	100%	68.85	-	100%	52.12
A4	-	-	0%	64.96	-	0%	25.44
	$P(\texttt{<cc>})$	0.10	86%	69.35	0.10	83%	26.50
	-	-	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 retrieval 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.

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. We hypothesize that placing CC within the infilling part breaks 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. This work opens up new avenues for RAG-based repository-level code completion and motivates their development.

Discussion Building on our study, future research may consider further leveraging REPOFORMER for speeding up large LMs in settings such as *speculative decoding* (Chen et al., 2023). Besides, the current function completion performance could be further improved by adapting selective RAG to *dynamically invoking multiple retrievals* during the generation. Moreover, tailoring retrieval policies to individual repositories by leveraging their unique characteristics could improve the effectiveness of our selective retrieval framework, aligning with our goal of developing smarter, more efficient RAG-based code completion solutions.

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. While 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 maximizing their societal benefits while mitigating risks. 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. *arXiv preprint arXiv:2310.11511*, 2023.

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 arXiv:2207.14255*, 2022.

Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

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

Ding, Y., Wang, Z., Ahmad, W. U., Ramanathan, M. K., Nallapati, R., Bhatia, P., Roth, D., and Xiang, B. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*, 2022.

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 *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/pdf/2310.11248.pdf.

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. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*, pp. 763–773, 2017.

Hill, R. and Rideout, J. Automatic method completion. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pp. 228–235. IEEE, 2004.

Jaccard, P. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

Jiang, Z., Xu, F. F., Gao, L., Sun, Z., Liu, Q., Dwivedi-Yu, J., Yang, Y., Callan, J., and Neubig, G. Active retrieval augmented generation, 2023.

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 arXiv:2207.05221*, 2022.

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 arXiv:2211.15533*, 2022.

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 *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

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.

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.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., 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.

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., 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 *International Conference on Learning Representations*, 2022a. URL https://api.semanticscholar.org/CorpusID:252668917.

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*, 2022b.

OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.

Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15 (12):1053–1058, 1972.

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*, 2023. URL https://arxiv.org/abs/2302.00083.

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 arXiv:2009.10297*, 2020.

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

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

Shrivastava, D., Larochelle, H., and Tarlow, D. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pp. 31693–31715. PMLR, 2023b.

Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

Tu, Z., Su, Z., and Devanbu, P. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 269–280, 2014.

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 arXiv:2107.01933*, 2021.

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, December 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 *Proceedings of the 24th international conference on Software engineering*, pp. 513–523, 2002.

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, December 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., Wang, Z., Jiang, Z., and Neubig, G. Docprompting: Generating code by retrieving the docs. In *International Conference on Learning Representations (ICLR)*, Kigali, Rwanda, May 2023. URL https://arxiv.org/abs/2207.05987.

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 7 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[6] 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 6 presents the performance on RepoEval

[6] 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 StarCoder.

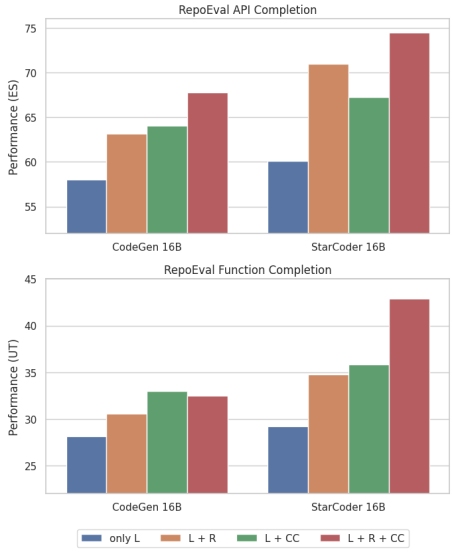

Figure 6. A comparison between four prompting strategies for RepoEval by combining left context (L), right context (R), and cross-file contexts (CC). Leveraging right contexts to build infilling-style prompt generally improves the performance regardless whether CC is present or not. StarCoder exhibits 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 7 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 invariable retrieval. In addition, the knowledge of whether the LM could be benefited by the retrieved context is not leveraged.

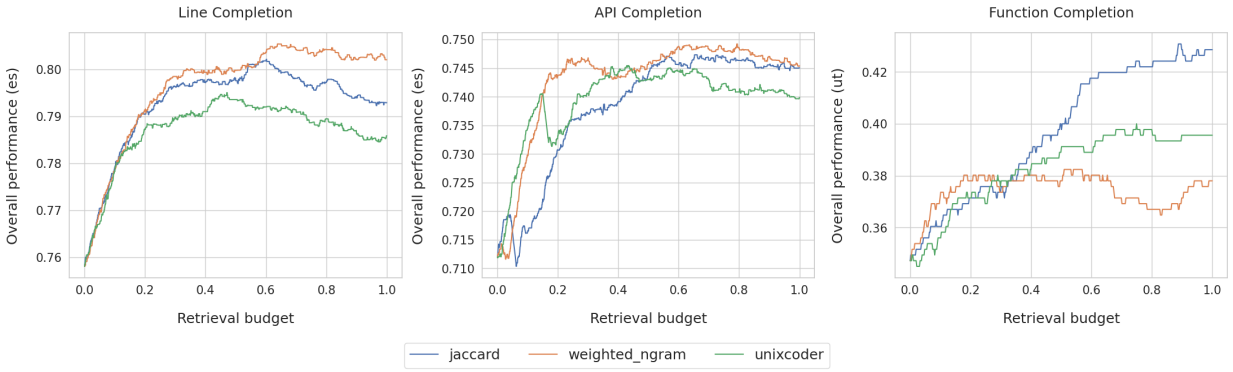

Figure 7. A comparison of the effectiveness of different similarity functions for selective RAG with StarCoder 16B. We plot the retrieval budget in the x-axis, which is the percentage of instances to perform retrieval. We report score on the entire testing dataset for each budget. Specifically, the retriever's 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.

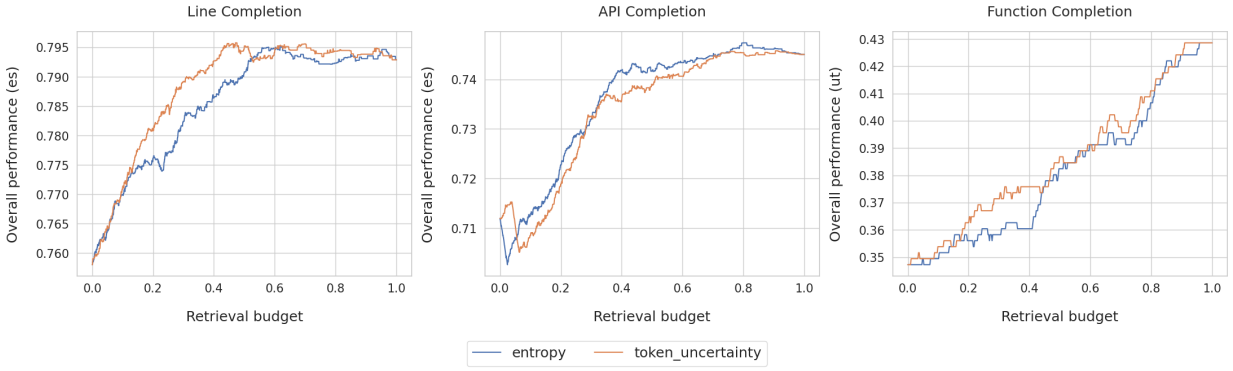

Figure 8. A comparison of the effectiveness of two uncertainty metrics for selective RAG with StarCoder 16B. We plot the retrieval budget in the x-axis and report score on the entire testing dataset for each budget. We observe that the uncertainty-based metrics fail for long sequence generation such 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 7).

C.2 Trial Generation

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

- **entropy**: the sequence-level entropy as used in Li et al. (2023a). We estimate the entropy by performing vanilla sampling for 20 times.
- **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 8 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 CCEval

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**
    $k \sim$ Poisson($\lambda = 3$)
    $s \leftarrow$ Randomly sample a chunk from c
    $Y \leftarrow$ 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)
    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 9. In total, 30.18% chunk completion instances and 35.16% function completion instances are labeled with positive. The average length of Y is 3.53 lines for chunk completion and 11.77 lines for function completion.

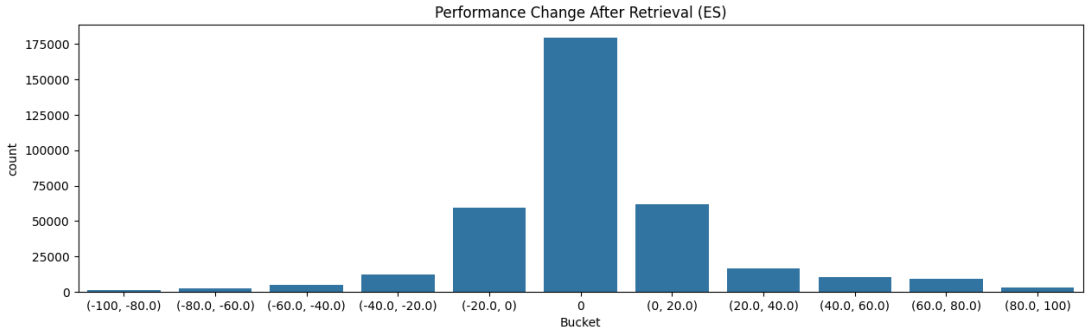

Figure 9. The performance gain on REPOFORMER training data exhibited by StarCoderBase-1B from retrieved cross-file context. The sign of the performance change is used to generate the label for REPOFORMER training. Each (start, end) bucket contains values ranging from start to end except for the central bucket, which corresponds to exactly 0.

CCEval Benchmarking Dataset Creation 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 CCEval.

	RepoEval			CCEval	
	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 CCEval. 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.

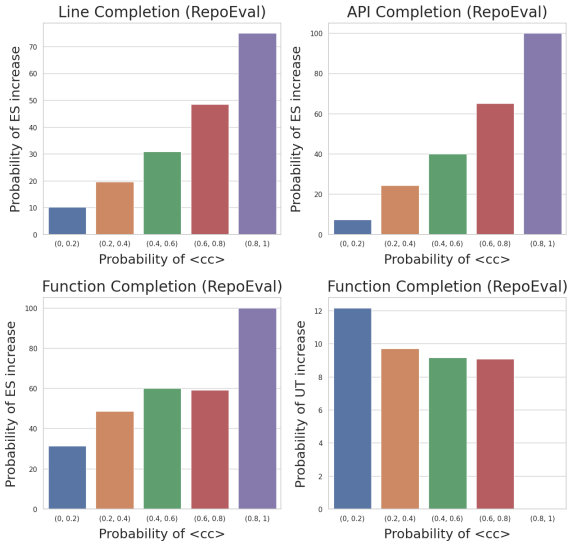

Figure 10. The calibration of selective retrieval predictions. REPOFORMER makes generally calibrated predictions when ES is used as the metric and the generation is of moderate lengths. The prediction is not calibrated for function completion when the metric is UT.

E Extended Analyses

E.1 Calibration of REPOFORMER's Selective Retrieval Prediction

We evaluate the calibration of REPOFORMER-1B's selective decisions. Figure 10 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.

E.2 Results on CrossCodeEval: REPOFORMER-7B 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.

REPOFORMER-7B We apply the REPOFORMER training scheme on REPOFORMER-7B with the dataset created for REPOFORMER-1B/3B. We keep the training infrastructure and hyperparameters the same.

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.

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
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
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-1B	$P(<\texttt{cc}>)$	71.29	62.81	75.12	67.16	83.08	74.24	69.90	64.07
REPOFORMER-3B	$P(<\texttt{cc}>)$	74.57	66.86	78.40	71.26	85.92	78.62	73.70	68.66
REPOFORMER-7B	$P(<\texttt{cc}>)$	75.34	**68.27**	78.90	72.35	83.80	76.88	73.59	69.10
REPOFORMER-M-1B	$P(<\texttt{cc}>)$	71.55	62.89	75.92	67.86	84.44	76.00	70.07	64.41
REPOFORMER-M-3B	$P(<\texttt{cc}>)$	73.80	66.72	77.68	71.01	85.31	77.70	72.51	67.06
REPOFORMER-M-7B	$P(<\texttt{cc}>)$	**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 similar for code matching as well as the F1 score for identifier matching.

- **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.
- **REPOFORMER-7B.** We observe the same trends as we have observed on the 1B and 3B model. Selective retrieval with REPOFORMER is able to outperform invariable retrieval with the original StarCoder model.

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 $P(<\texttt{cc}>)$ bringing more than 70% speedup.

	Selective Policy	API Completion			Line Completion		
		ES	%RAG	SU	ES	%RAG	SU
1B	invariable retrieval	71.69	100%	0%	75.25	100%	0%
	self-selection	70.82	18%	71%	73.70	19%	71%
	$P(<\texttt{cc}>)$	72.39	61%	33%	75.65	62%	33%
3B	invariable retrieval	74.48	100%	0%	78.24	100%	0%
	self-selection	73.26	19%	65%	76.74	20%	66%
	$P(<\texttt{cc}>)$	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 invariable retrieval. Compared to the invariable retrieval baseline, the $P(<\texttt{cc}>)$ strategy consistently demonstrates gains in both accuracy and latency. The self-selection 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 REPOFORMER models and for STARCODER, we use REPOFORMER-1B to make the selective RAG decisions. The results are presented in Figure 11 to Figure 14. 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.

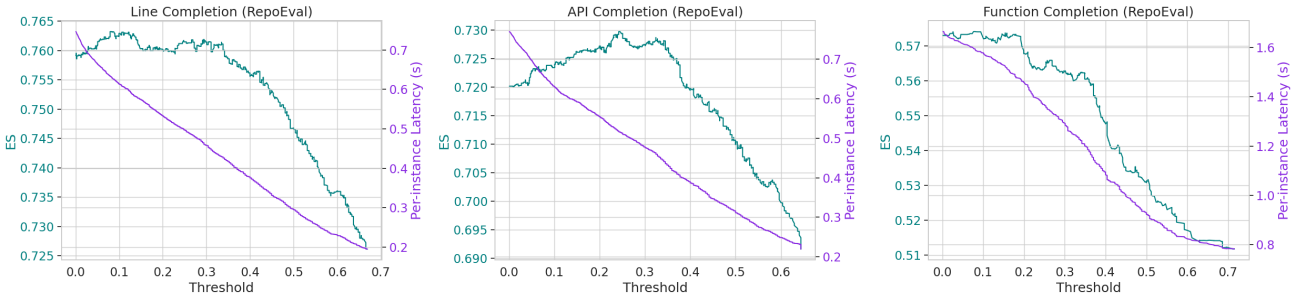

Figure 11. Latency-accuracy trade-off of self-selective RAG for REPOFORMER-1B.

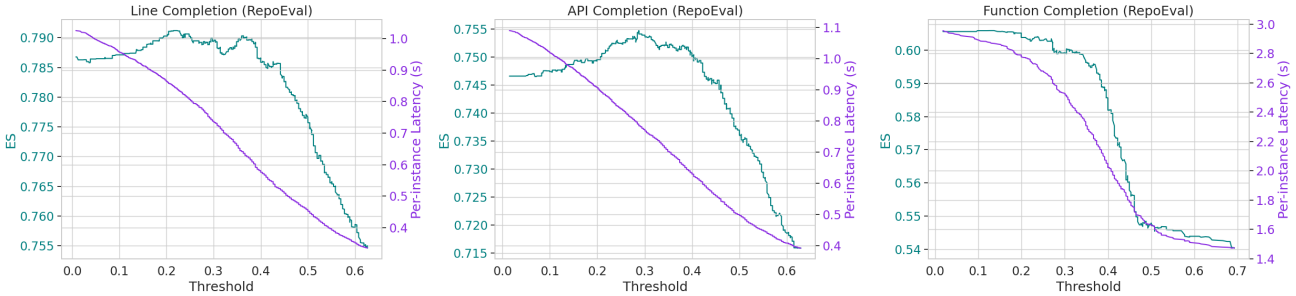

Figure 12. Latency-accuracy trade-off of self-selective RAG for REPOFORMER-3B.

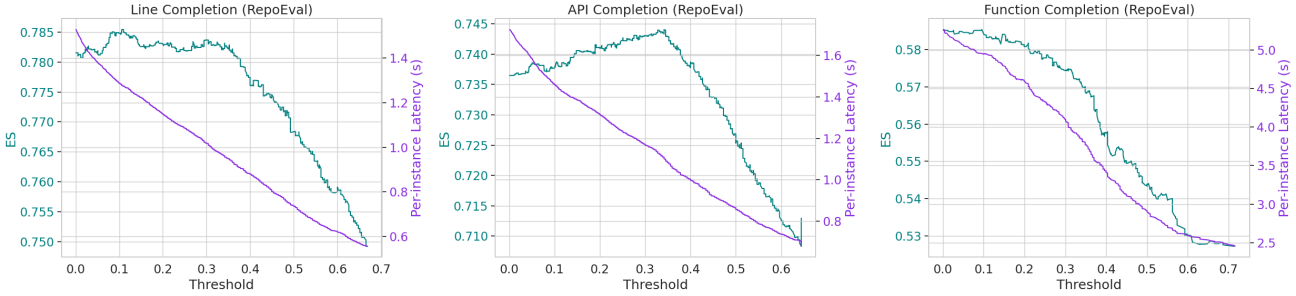

Figure 13. Latency-accuracy trade-off of selective RAG for STARCODERBASE-7B. REPOFORMER-1B is used for the selective decisions.

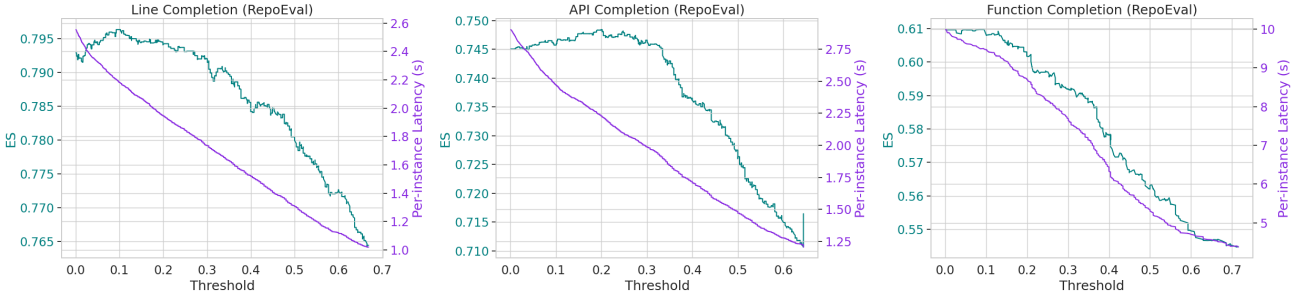

Figure 14. Latency-accuracy trade-off of selective RAG for STARCODER. REPOFORMER-1B is used for the selective decisions.