REPOFORMER: Selective Retrieval for Repository-level Code Completion

Anonymous Authors[1]

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 frameworks often leads to inefficiencies and inaccuracies, with up to 80% of the retrieved contexts proving unhelpful for strong code language models (code LMs) such as CodeGen and StarCoder. To tackle the challenges, we propose a selective RAG framework where the retriever is invoked only when necessary. At the core of this framework is REPOFORMER, a novel code LM that self-determines the necessity of cross-file context based on the current file's information. To enhance the model's self-assessment and code completion capabilities, we design a multi-task learning approach that leverages self-supervision from public repositories with a contrastive data labeling paradigm. Extensive evaluations on diverse benchmarks reveal that REPOFORMER not only markedly outperforms existing retrieval-enhanced code LMs, but also reduces the inference latency by as much as 60%. These advancements position REPOFORMER as a significant 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 inside a file from user repositories. The major difficulty comes from the modular design of software (Parnas, 1972), which introduces local API usages and inter-module dependencies that require the understanding of information beyond the current file. As a result, retrieving and incorporating cross-file repository-level knowledge have been the key research focus (Tu et al., 2014; Hellendoorn & Devanbu, 2017; Svyatkovskiy et al., 2020).

Recently, this trend is reinforced by a paradigm shift towards *retrieval-augmented generation* (RAG) approaches. RAG systems use an in-repository retriever to gather cross-file contexts such as relevant code, documentation, or APIs. Such knowledge is provided as an extension to the current file's context to enable code language models (code LMs) to generate more accurate code completion. To design effective RAG-based approaches, existing works either improve the retrieval mechanism for prompting black-box code LMs (Lu et al., 2022; Shrivastava et al., 2023b; Zhang et al., 2023) or consider fine-tuning the LM to better leverage the retrieved context (Ding et al., 2022; Zan et al., 2022).

Despite their promising performance, existing RAG-based code completion approaches share a critical design choice: *retrieval is performed invariably*, for each and every instance. This assumption exposes several weaknesses. First, retrieval is unnecessary when the information from the current file is sufficient for the LM to make a correct prediction. Further, the LM's performance can be harmed by noisy retrieved contexts. In fact, as many as 80% of the retrieved contexts fail to improve the code completion accuracy of RAG systems using CodeGen (Nijkamp et al., 2022b) or StarCoder (Li et al., 2023b) (Section 3.1). Finally, retrieval is costly, especially in large repositories. The overhead is even more concerning for systems that perform frequent or iterative retrieval (Zhang et al., 2023). Following the invariable retrieval assumption leads to major latency costs with only sparse improvements in accuracy.

In this paper, we argue that *selective retrieval augmentation* is the key to overcoming the aforementioned challenges: given the current file, the RAG system decides whether it is necessary to retrieve the cross-file contexts and provide them to the code LM. We show that this decision could be informed by the retriever's similarity output from a trial retrieval or the LM's uncertainty from a trial generation. Howevre, these methods incur a large latency cost and accuracy concerns (Section 3.3). In response, we develop REPOFORMER, a code LM capable of *code infilling with*

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

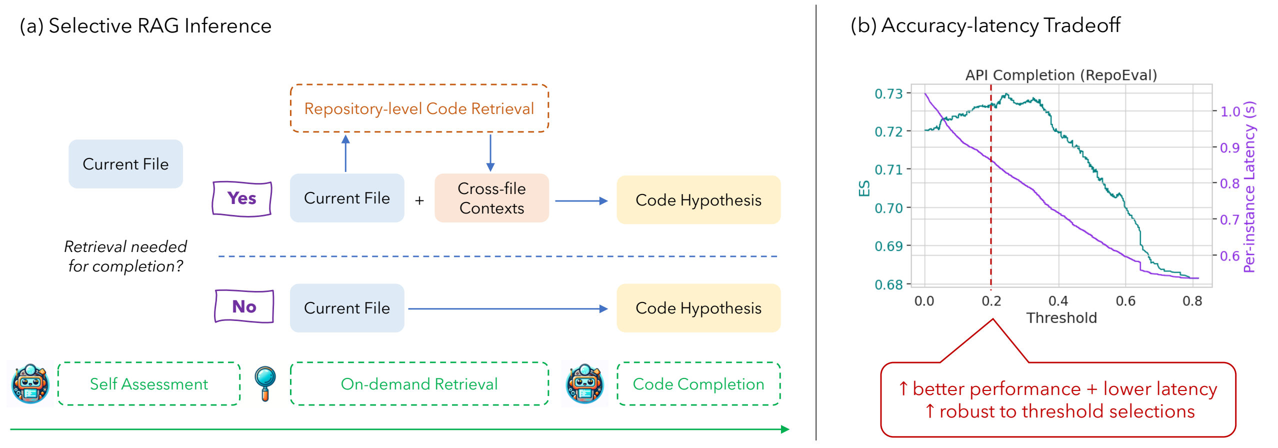

Figure 1. (a) An overview of the proposed selective RAG framework. Given the current file context, REPOFORMER first self-assesses whether retrieval is required and triggers the retriever correspondingly. Then, it makes a hypothesis with the optional retrieved context. (b) Using self-selective retrieval, REPOFORMER achieves better accuracy and better latency than performing retrieval invariably.

self-triggered retrieval (Figure 1). After observing the current file, REPOFORMER self-assesses its need for additional cross-file information to complete the code and expresses its confidence by predicting a special token that triggers the in-repository retriever. With the cross-file contexts retrieved only upon request, REPOFORMER then completes the code in a fill-in-the-middle manner.

Two abilities are crucial to the success of REPOFORMER: accurate self-evaluation and effective usage of possibly noisy retrieved contexts. We mine self-supervision for these abilities using the public repositories from the Stack (Kocetkov et al., 2022) with a contrastive data labeling paradigm: the LM's predictions with and without the retrieved repository contexts are compared to determine the necessity of retrieval for a given blank to fill in (Section 4.2). REPOFORMER is then obtained by fine-tuning a code LM on a novel multi-task objective for self-assessment and code completion with optional retrieved cross-file contexts.

We conduct comprehensive evaluations of REPOFORMER on repository-level code completion with a variety of use cases from RepoEval (Zhang et al., 2023) and a newly created large-scale benchmark based on CrossCodeEval (Ding et al., 2023). Results suggest that REPOFORMER achieves strong performance, outperforming the same-sized StarCoderBase model by more than 3 absolute points for edit similarity across multiple tasks. The 3B REPOFORMER model even performs on par with invariable retrieval using the 16B StarCoder model. In addition, the selective retrieval strategy used by REPOFORMER improves the inference latency of RAG-based code completion systems by as much as 60%. REPOFORMER is also effective at making selective decisions for accelerating RAG systems that use a larger code LM as the generator. Finally, we analyze REPOFORMER's threshold robustness, ability to leverage the retrieved contexts, and calibration behavior. We will release our code, model, and the evaluation dataset to facilitate future studies.

2. Preliminaries

2.1. Problem Formulation

We denote each instance for *repository-level code completion* as (X_l, X_r, Y, C). Y is the ground truth that needs to be completed. In this paper, Y 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 refer to them as the left context (X_l) and the right context (X_r). C is the set of all the other files in the repository. A code completion system utilizes X_l, X_r, and C to generate a hypothesis $\hat{Y}$.

2.2. Retrieval-Augmented Generation

In this paper, we consider RAG-based code completion frameworks with two components:

- An **in-repository retriever** $\mathcal{R}$ that queries from C 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 from a single file.
- A **code LM** $\mathcal{M}$ that leverages X_l, X_r, and CC to output a hypothesis $\hat{Y}$ for the hole. The inclusion of X_r and CC is optional.

The execution of this framework consists four stages: *query formation*, *indexing*, *retrieval*, and *generation*. We further detail each of the steps in Section 5.1. It is worth-noting that for generation, we use an *infilling* setup, where both X_l and X_r are directly provided in the prompt (Shrivastava et al.,

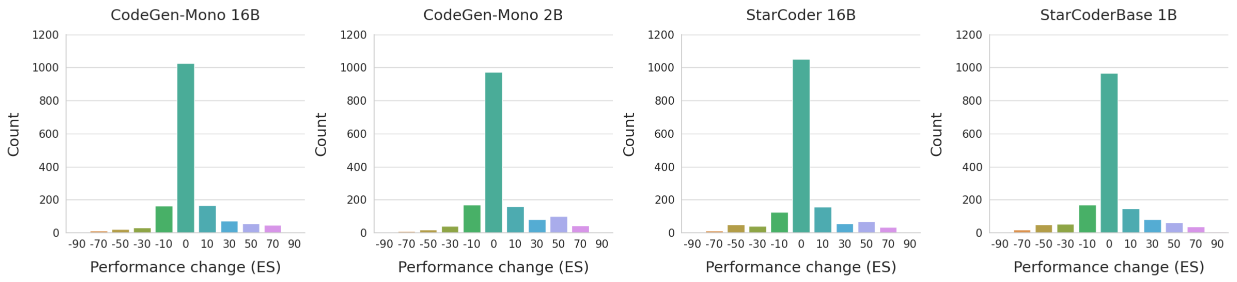

Figure 2. The performance gain on RepoEval API completion exhibited by four models from retrieved cross-file context. Each bucket contains values ranging from label-10 to label+10 except for the central bucket, which corresponds to exactly 0. The retrieved context only improves the performance in around 20% of instances. The trend is consistent across all the evaluated LMs.

2023b; Pei et al., 2023). We provide detailed discussions and empirical supports for this design in Appendix B. Also note that we do not consider iterative RAG (Zhang et al., 2023) because (1) single retrieval accounts for the major gains in accuracy and (2) its latency overhead is unrealistic.

2.3. 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

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

where Lev is the Levenshtein distance (Levenshtein et al., 1966). 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.

Datasets We use RepoEval (Zhang et al., 2023), which consists of line completion, API completion, and function completion tasks created from 32 recent Python repositories. To improve the repository coverage, we additionally leverage 1500 raw Python repositories from CrossCodeEval (Ding et al., 2023) to create chunk completion and function completion instances. We call this dataset CCEval. More details regarding CCEval are presented in Appendix F.

Models **CodeGen-Mono** (Nijkamp et al., 2022a) is pre-trained sequentially on 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 with an additional Python corpus. In this paper, we experiment with the 2B and 16B versions of CodeGen-Mono, the 1B, 3B, and 7B versions of StarCoderBase, as well as the 16B StarCoder.

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 context. For the majority of the instances, the retrieved context does not improve the performance. "↑", "=", "↓" denotes performance increase, no performance change, and performance drop.

3. Towards Selective Retrieval Augmentation

3.1. Is retrieval always required?

As discussed in Section 1, current studies invariably augment the code LM with the retrieved contexts. Indeed, under our evaluation setting, this strategy brings significant dataset-level performance gain in terms of both ES and UT (Appendix B). However, to fully justify the invariable retrieval, it is important that the retrieved context are generally beneficial at the *instance-level*. Surprisingly, out evaluations in this section suggest the opposite.

In Figure 2 and Table 1, we evaluate four code LMs of various sizes on API completion and function completion from RepoEval. For each model, we compute the instance-level performance change from current file code completion (X_l and X_r) to retrieval-augmented code completion (X_l, X_r, and CC). Detailed prompts are discussed in Appendix A.

The results reveal an intriguing *80-20 rule*: retrieval improves LMs' performance on only 20% instances. For most of the instances (60% for API completion and 85% for function completion), providing the retrieved contexts to the code LMs neither improves nor harms their performance. Further, for many instances, retrieval augmentation harms the performance. The observed trends are consistent for both API and function completion and for both small-sized LMs ($\leq 2B$) and moderate-to-large LMs ($16B$). Together, these findings reveal that for repository-level code completion, retrieval augmentation is often unnecessary.

3.2. Selective RAG: Formulation

How to improve the accuracy and efficiency of RAG if retrievals are unnecessary or even harmful? We argue that *selective retrieval-augmented generation* (selective RAG) is the essential path. In selective RAG, the system always explicitly decides whether the LM's generation could benefit from retrieved contexts, and avoids the retrieval augmentation when it is deemed unnecessary. Figure 1 (a) depicts the typical flow of selective RAG inference.

In the rest of this section, we argue that the commonly adopted heuristics are inadequate for this task. In response, we propose an LM-centered solution in Section 4.

3.3. Trial Retrieval or Trial Generation are inadequate

Two common techniques are relevant to selective RAG: (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 argue that both approaches could be informative to the selective decision, but are inherently suboptimal.

Trial Retrieval We consider a simple approach: only providing the LM with CC if the similarity of top-1 retrieved code chunk exceeds a given threshold. On RepoEval, this strategy achieves the performance of full retrieval with at most 60% retrieval budget across multiple tasks (Appendix C). However, trial retrieval cannot reduce the latency overhead. Further, it ignores whether the LM can truly leverage the retrieved context to improve its generation quality.

Trial Generation An alternative approach is to retrieve only when the LM has a high predictive entropy (Li et al., 2023a) or predicts uncertain tokens (Jiang et al., 2023). Our investigation on RepoEval shows that both techniques work well for line completion, while their predictive accuracy degrades for longer sequences such as API or function completion (Appendix C). Beyond their computational cost, the inherent drawback of uncertainty-based methods is the failure to directly reflect whether the current code completion problem warrants cross-file retrieval or not. For instance, retrieval could greatly benefit the completion of local API usages but may not facilitate completing simple code constructs or logic clearly defined by the in-file context.

4. REPOFORMER: code completion with self-triggered retrieval

To enable the RAG system to make intelligent and efficient selective decisions that consider both the question context and the knowledge of the code LM, we propose a novel *self-triggered retrieval* mechanism. This paradigm models *self-assessment* and *code completion* as a sequential process

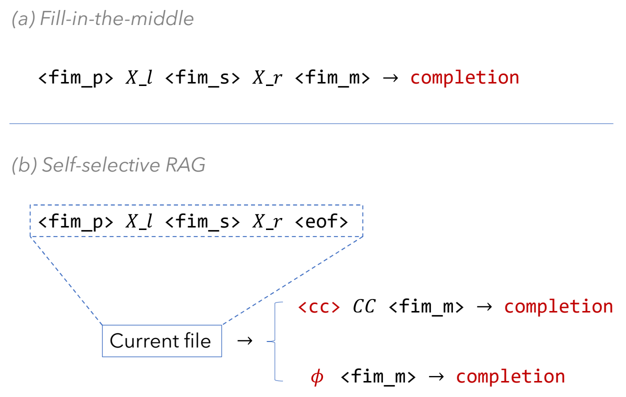
(a) Fill-in-the-middle
<fim_p> X_l <fim_s> X_r <fim_m> → completion
(b) Self-selective RAG
<fim_p> X_l <fim_s> X_r <eof>
Current file →
<cc> CC <fim_m> → completion
ϕ <fim_m> → completion

Figure 3. A comparison between the prompting scheme of fill-in-the-middle and self-selective RAG. $\rightarrow$ denotes the invocation of the LM. LM-generated parts are colored in red. `fim_p`, `fim_s`, and `fim_m` are short for the special tokens for fill-in-the-middle prompting: `fim_prefix`, `fim_suffix`, and `fim_middle`.

embedded in autoregressive generation, enabling a code LM to self-evaluate (Kadavath et al., 2022) whether its prediction could be improved by additional retrieved contexts. We provide further details in the following subsections.

4.1. Self-selective RAG inference

We model self-selective RAG inference as an extension to the fill-in-the-middle generation (Figure 3). After observing X_l and X_r, the LM triggers cross-file retrieval by generating a special token `<cc>` or abstains from retrieval via an empty token ϕ[1]. Then, the LM proceeds the code completion with X_l, X_r, and the optional CC. To avoid changing the semantics of `<fim_middle>`, we mark the end of the in-file context with a new token named `<eof>`.

We highlight the advantages of self-selective RAG:

1. **Accuracy**. The LM is able to explicitly express its confidence case-by-case, based on X_l and X_r.
2. **Generality**. The LM can seamlessly self-switch between RAG and normal fill-in-the-middle generation.
3. **Efficiency**. Self-selective RAG does not induce extra latency if retrieval is not triggered.

4.2. Multi-task training with self-supervision

Two abilities are essential to self-selective RAG: accurate self-assessment and successful incorporation of the retrieved context. We introduce a contrastive data labeling scheme to mine the supervision from public repositories, followed by fine-tuning with a novel multi-task objective.

[1]In practice, instead of greedily decoding `<cc>`, we check whether its probability is above a certain threshold (Section 5.2).

Data construction We randomly sample 18k Python repositories from the Stack (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. Then, we follow a three-step procedure to create the fine-tuning data:

1. Sample target lines Y that are either (1) random chunks of varied lengths or (2) function bodies.
2. Retrieve CC using the current file, with or without Y.
3. Label whether CC can improve a code LM $\mathcal{M}$'s code completion quality (evaluated by ES against Y) by more than a pre-determined threshold T.

The detailed algorithm is presented in Appendix D. We use $\mathcal{M}$ = StarCoderBase-1B and T = 0 to obtain 240k chunk completion and 120k function completion instances, each in the form $(X_l, X_r, Y, CC, label)$. We reserve 500 repositories for validation and use the rest for training.

Verbalization Based on $label$, each instance is verbalized into a sequence for fine-tuning. If $label$ is true, we provide CC after the special token `<cc>`. Otherwise, we use the template with only X_l and X_r. The two verbalizations correspond to the two branches in Figure 3 (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-encropy loss on predicting `<cc>` immediately following `<eof>`.

$$\mathcal{L}_{eval} = -\log p_{\mathcal{M}}(\text{<cc>}|X_l, X_r) \tag{2}$$

2. $\mathcal{L}_{gen}$: a cross-encropy 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.

$$\mathcal{L}_{gen} = \begin{cases} -\log p_{\mathcal{M}}(Y|X_l, X_r, CC), & \text{if } label \\ -\log p_{\mathcal{M}}(Y|X_l, X_r), & \text{otherwise} \end{cases} \tag{3}$$

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 including 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.

4.3. Implementation details

We fine-tune StarCoderBase-1B and StarCoderBase-3B on the created dataset. We use $\lambda = 1.0$, learning rate 2e-5, batch size 512, 50 warmup steps, and linear learning rate decay. The models are trained for 2 epochs, which takes 6 hours for the 1B model and 9 hours for the 3B model with 8 Nvidia A100 GPUs (40G memory). Our implementation is based on Jain et al. (2023)[2]. We call our models REPOFORMER-1B/3B as they are designed to actively leverage the repository context.

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 ES on the validation dataset.

5. Experimental Results

5.1. Detailed RAG Setup

Below, we describe the four steps we follow for executing RAG. We further detail the hyperparameters for retrieval, generation, and post-processing in Appendix A.

1. **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.
2. **Indexing.** All files in C are divided into code chunks of the same size as the query. We present details regarding window sizes and stride sizes in Appendix A.
3. **Retrieval.** A similarity function f is used to compare the query with every chunk and identify k most similar code chunks. We use 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.
4. **Generation.** CC is concatenated with the in-file context as a prompt for $\mathcal{M}$. We verbalize CC as comments to the program (see prompts in Appendix A).

5.2. REPOFORMER achieves strong selective RAG performance for code completion

We conduct a comprehensive evaluation of REPOFORMER on RepoEval and CCEval, including three settings:

1. **No Retrieval**. We benchmark the StarCoder family with only X_l and X_r in the prompt.
2. **Invariable Retrieval**. We benchmark StarCoder with CC always provided in addition to X_l and X_r.

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

Model	Size	Selection Policy	RepoEval						CCEval		
			Line		API		Function		Chunk		Function
			EM	ES	EM	ES	UT	ES	EM	ES	ES
No Retrieval											
	1B	-	43.44	67.77	37.81	66.54	22.20	47.65	31.08	60.09	47.49
StarCoderBase	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											
	1B	-	51.19	72.30	43.94	69.17	25.71	55.64	37.22	63.73	50.50
StarCoderBase	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	-	$\underline{61.25}$	$\underline{79.24}$	$\underline{51.12}$	74.50	$\underline{42.86}$	$\underline{60.96}$	$\underline{47.90}$	71.90	$\underline{58.06}$
Selective Retrieval											
Repoformer	1B	self selection	51.90	74.50	43.50	71.00	24.00	53.10	38.52	68.08	52.09
		<cc> prob	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
		<cc> prob	**59.63**	**79.02**	49.31	**$\underline{74.96}$**	**32.96**	**60.56**	**46.66**	**$\underline{72.23}$**	**56.24**

Table 2. Experiment results on RepoEval and CCEval. The best performance is underlined. The best performance among models below 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 works with lower retrieval budget. Among the two selective policies, <cc> prob enables the best selective RAG performance.

3. **Selective Retrieval**. We benchmark Repoformer by providing X_l and X_r in the prompt, optionally augmented with CC according to two selective policies:
    - **Self-selection**. If `<cfc>` is the most likely token following `<eof>`, retrieval is performed.
    - **`<cc>` prob**. If probability of `<cc>` following `<eof>` is greater than a threshold T, retrieval is performed on this instance[3].

Table 2 suggests that compared to no retrieval and invariable retrieval using StarCoderBase of the same size, Repoformer's selective retrieval strategy exhibits strong performance improvements. Via the `<cc>` prob strategy, Repoformer-3B is able to outperform StarCoderBase-7B on all the tasks and metrics except EM for API completion, even outperforming the 5x larger StarCoder model in terms of ES for API and Chunk completion. The threshold-controlled `<cc>` prob strategy outperforms the self-selection strategy on all the tasks. In the next section, we show that the two strategies represent different trade-off between accuracy and inference latency.

5.3. Repoformer improves inference efficiency

In this section, we illustrate the benefits of Repoformer for saving the inference latency during online serving.

Latency Model We assume that the repository is pre-split into code chunks. Given a code completion instance, the system issues three processes at the same time:

- P1: make a retrieval decision using Repoformer.
- P2: using $\mathcal{M}$, generate $\hat{Y}$ without CC.
- P3: retrieve CC and generate $\hat{Y}$ with CC using $\mathcal{M}$.

Depending on the result of P1, the system waits for either P2 or P3 and halts the other process. If Repoformer is used as $\mathcal{M}$, P1 and P2 are represented by the same process.

We consider three latency terms: (1) T_s, time required for the selective decision, (2) T_r, the retrieval latency, and (3) T_g, the generation latency. Therefore, the latency for P1, P2, and P3 are T_s, T_g, and $T_r + T_g$. When $\mathcal{M}$ is Repoformer or a model larger than Repoformer, it is safe to assume $T_s < T_g < T_r + T_g$. Therefore, the latency for the entire system is either T_g or $T_r + T_g$ depending on P1.

Table 3 presents the accuracy-latency trade-off with $\mathcal{M}$ = Repoformer. Line and API Completion are presented to cover generation of short and moderate lengths[4]. We observe that both selective strategies can improve the latency significantly, with a different trade-off: using a fixed threshold for `<cc>` prob results in improvements for both accuracy and latency compared to invariable retrieval, while using self selection results in a much larger latency gain with a small performance degradation.

[3]We find that $T = 0.15$ for function completion and $T = 0.2$ for the other tasks generally works well. In this paper, we always follow these two thresholds unless otherwise stated.

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

	Selective Strategy	API Completion		Line Completion	
		ES	Speedup	ES	Speedup
1B	invariable retrieval	72.02	0%	75.91	0%
	self selection	71.04	62% ↑	74.50	41% ↑
	<cc> prob	72.72	22% ↑	76.00	20% ↑
3B	invariable retrieval	74.66	0%	78.68	0%
	self selection	73.60	42% ↑	77.60	33% ↑
	<cc> prob	74.96	10% ↑	79.02	9% ↑

Table 3. The accuracy-latency tradeoff of REPOFORMER with two self-selective RAG paradigms. Compare to the invariable retrieval baseline, the <cc> prob strategy consistency demonstrates gains in both accuracy and latency. The self selection strategy shows much larger latency gains with a small performance degradation.

	Selective Strategy	API Completion		Line Completion	
		ES	Speedup	ES	Speedup
7B	invariable retrieval	73.65	0%	78.15	0%
	REPOFORMER-1B	74.10	12% ↑	78.31	7% ↑
16B	invariable retrieval	74.50	0%	79.24	0%
	REPOFORMER-1B	74.84	7% ↑	79.48	5% ↑

Table 4. The accuracy-latency tradeoff of STARCODER with REPOFORMER-1B as the policy model for selective RAG. Compare to the invariable retrieval baseline, the selective RAG strategy consistency improves both accuracy and inference latency.

Can REPOFORMER still improve the inference latency with *a larger model as* $\mathcal{M}$? We use <cc> prob on REPOFORMER-1B as the selective policy and use the 7x larger STARCODERBASE-7B and the 16x larger STARCODER as $\mathcal{M}$. As shown in Table 4, the selective predictions from REPOFORMER can be used to improve the performance and the latency in the RAG serving pipeline with these larger models. We further include the results on more tasks and more threshold settings in Appendix G.

6. Analysis

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 for the <cc> prob strategy. As the threshold increases, the model's overall code completion first increases due to avoidance of uninformative and misleading retrieved context. At threshold 0.4, the model still maintains the same level of performance compared to full retrieval augmentation, with latency reduced by 30%. This result demonstrates that REPOFORMER is able to accommodate various settings of the threshold and provide good accuracy-latency trade-off. We provide the visualization for other tasks in Appendix G.

Can REPOFORMER better leverage CC? We compare REPOFORMER-1B and STARCODERBASE-1B for their ability to leverage the retrieved CC. In Figure 4, we show the performance change caused by CC on the instances where REPOFORMER-1B requests for retrieval. The results

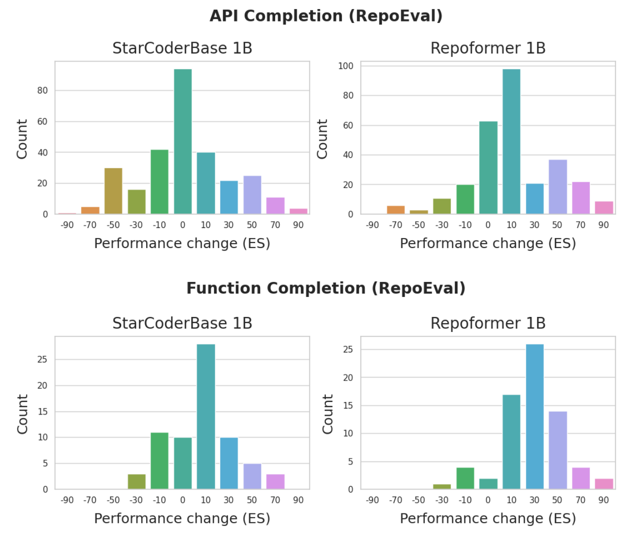

Figure 4. The performance change on RepoEval from retrieved cross-file context. Compared to StarCoder, REPOFORMER is better at leveraging CC for the self-selected instances.

clearly shows the advantage of our model, which exhibits more and greater performance gains upon observing CC. The instances with performance drop is also significantly reduced for REPOFORMER.

Does REPOFORMER make calibrated decisions? We further evaluate the calibration of REPOFORMER-1B's selective decisions. Figure 5 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.

Ablation Study We perform ablation studies of REPOFORMER-1B on several alternative design choices:

- (**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 the model on infilling.

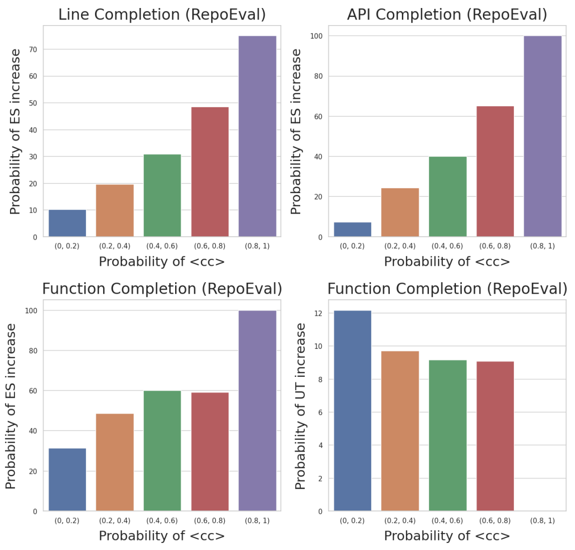

Figure 5. The calibration behavior of selective retrieval predictions. REPOFORMER makes generally calibrated predictions when ES is used as the metric and the generation is of moderate lengths.

- (**A4**) Placing <cc> and CC after <fim_middle> and marking its end with a new token <cc_end>. This item mainly studies whether training the model to recognize CC as part of the infilling is more beneficial.

We fine-tune StarCoderBase-1B with the same setup as REPOFORMER and present the results on CCEval in Table 5. Since A1 and A4 are trained on self-evaluation loss, we also test them on self-selective RAG. From Table 5, we observe that although A1 has slightly better RAG performance, it fails to make meaningful selective predictions: the conditional probability of <cc> is nearly 1 for all the instances. For A2, the results suggests that learning the self-evaluation ability only slightly affects its best-achievable RAG performance. For A3, it has the same performance for in-file completion as A2, but not as good as RAG performance, indicating the necessity of fine-tuning with CC. Finally, we observe that A4 achieves reasonable performance for chunk completion but performs much worse for function completion. We hypothesize that placing CC within the infilling section breaks the fill-in-the-middle semantics that StarCoder was pre-trained on.

Model	Selective Policy	Chunk Completion			Function Completion		
		T	%RAG	ES	T	%RAG	ES
SC	-	-	0%	60.09	-	0%	47.49
	-	-	100%	63.73	-	100%	50.50
RF	-	-	0%	66.22	-	0%	49.77
	<cc> prob	0.20	75%	69.97	0.15	76%	53.71
	-	-	100%	69.95	-	100%	53.56
A1	-	-	0%	66.14	-	0%	49.25
	<cc> prob	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%	66.49	-	0%	49.01
	-	-	100%	69.60	-	100%	53.58
A4	-	-	0%	64.96	-	0%	25.44
	<cc> prob	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 find T = 0.10 works better for A4 and thus applied it for all the A4 results. **%RAG** = ratio of instances where RAG is performed.

7. 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 (Tu et al., 2014). To flexibly generate completions with various granularity, language models (LMs) have been seen as a promising solution. Early works have proposed application-specific training methods for n-gram LMs (Tu et al., 2014), RNNs (Hellendoorn & Devanbu, 2017; Wang et al., 2021a), and Transformers (Svyatkovskiy et al., 2020) to leverage structured knowledge beyond current file's context. With the advent of more powerful pre-trained Transformer-based code LMs (Chen et al., 2021; Wang et al., 2021b; Ahmad et al., 2021; Nijkamp et al., 2022b; Li et al., 2023b) that is able to generate code with flexible granularity, recent studies investigate fine-tuning code LMs to leverage retrieved knowledge provided in context, including free-formed code and documentation snippets (Zan et al., 2022; Ding et al., 2022; Shrivastava et al., 2023a). Another series of studies reveal that black-box LMs can already leverage the in-context knowledge, depending 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 saves the effort of constructing the training data and promises to generate better to arbitrary repositories. Orthogonal to these studies, this paper points out that the vanilla way of always augmenting LMs with retrieved contexts is insufficient and studies training code LMs to self-assess the current file's context and self-trigger retrieval selectively.

Adaptive RAG This paper resonates with the recent trend of making the RAG paradigm adaptive. The core question is finding an effective policy for deciding when to retrieve. He et al. (2021) propose to learn the policy of combining the parametric and non-parametric knowledge from history. Li et al. (2023a) and Jiang et al. (2023) suggest that LMs should retrieve information only when they have high predictive uncertainty. Mallen et al. (2023) discover that retrieval can be avoided for high frequency facts. Concurrent to this work, two new studies approach the adaptive RAG problem from a learning perspective. SKR (Wang et al., 2023) annotates instances where proprietary LMs fails to answer a question 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 for evaluating whether retrieval a question can be benefitted from retrieval and whether the retrieved contexts are useful. In this paper, we formulate selective retrieval as a sequential decision process and fine-tune the LM for self-evaluation (Kadavath et al., 2022) without extra modules (SKR), knowledge store (SKR), or learning from the labels generated by a larger oracle LM (Self-RAG). We also note that SKR and Self-RAG mainly experiment in the question answering domain, while this work identifies and tackles the challenges specific to repository-level code completion.

Tool Use There is a growing interest in augmenting LMs with the ability to invoke tools via natural language API calls (Schick et al., 2023; Lu et al., 2023) or specialized tool tokens (Hao et al., 2023). The in-repository retriever can be viewed as a tool specialized for returning relevant and helpful code. This paper shares the same spirit as Hao et al. (2023) in abstracting tools as special tokens to enable on-the-fly planning during inference. Compared to these previous work, the output of retrievers is more free-formed and semantically rich, making it more difficult for the LM to leverage. Therefore, we take a data-oriented perspective and train the model to both self-evaluate and be more robust to the tool's outputs. Our objective also does not require human annotations to generate the training data.

8. Conclusion

In this paper, we challenge the common assumption of always performing retrieval for repository-level code completion by showing that the retrieved contexts often do not contribute to performance improvements. In response, we argue for selective retrieval augmentation and propose REPOFORMER, a code LM that intelligently identifies when cross-file context is necessary. Our evaluations demonstrate REPOFORMER's superiority 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 for the development of more intelligent retrieval-augmented LMs.

Discussion We now discuss several limitations and future directions of this work.

- **Further speeding up large LMs.** We have shown the effectiveness of REPOFORMER for making selective predictions that transfer to large models. Future work could consider further exploiting its utility for large models in a speculative decoding setting (Chen et al., 2023; Leviathan et al., 2023). As REPOFORMER is optimized for RAG, using it as the draft model could result in fewer rejections compared to the other similar-sized small LMs.

- **More languages and tasks.** Although we mainly experiment on Python, selective RAG is a generic paradigm and could be extended to a multilingual setting. More granularity beyond line, API, or functions could also be considered, such as those defined with language-specific constructs.

- **Improving long-form completion.** For function completion, we explored a simple one-time retrieval strategy. However, the knowledge required could vary at different positions in the generation and thus the LM could be benefited by retrieval in the middle of generation. Considering multiple retrievals and adapting REPOFORMER to better perform long-formed generation is thus an important research topic.

- **Personalized retrieval policy.** This paper applies 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 could be an important research direction.

References

Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.211. URL `https://aclanthology.org/2021.naacl-main.211`.

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.

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., et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *arXiv preprint arXiv:2310.11248*, 2023.

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.

Hao, S., Liu, T., Wang, Z., and Hu, Z. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *arXiv preprint arXiv:2305.11554*, 2023.

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.

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. 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.

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.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

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, P., Peng, B., Cheng, H., Galley, M., Chang, K.-W., Wu, Y. N., Zhu, S.-C., and Gao, J. Chameleon: Plug-and-play compositional reasoning with large language models. In *The 37th Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

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.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

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*, 2021a.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, November 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.

Wang, Y., Li, P., Sun, M., and Liu, Y. Self-knowledge guided retrieval augmentation for large language models. *arXiv preprint arXiv:2310.05002*, 2023.

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., Liu, J., Zan, D., Mao, Y., Lou, J.-G., and Chen, W. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.

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 Setup

In this section, we describe the details of retrieval and generation adopted in this paper.

Retrieval We set the query size for line, API, and chunk completion and set 50 for function completion. The indexing granularity is changed to always align with the task types. For indexing, we use half of the query size as the stride size. Despite the duplication caused by the overlap between adjacent chunks, we argue that 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. We always find $k = 10$ top-similarity results and follow the segment alignment strategy to return the code chunks immediately following the top-similarity chunks. Unless otherwise specified, Jaccard similarity (Jaccard, 1912) is used as the similarity function f. We explored other choices mentioned in Section 3.3 but find them failing to outperform Jaccard similarity.

Generation 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 3.1 and Appendix B, our prompts are exactly the previous prompts with the `[Cross-file Context]` part removed.

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 instances, and 256 for function completion instances. We perform different post-processing operations for different tasks. 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` statement to X_l and use tree-sitter[5] to determine the index 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 in the hole. Right contexts are also shown useful for closely relevant 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 the availability of 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 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.

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

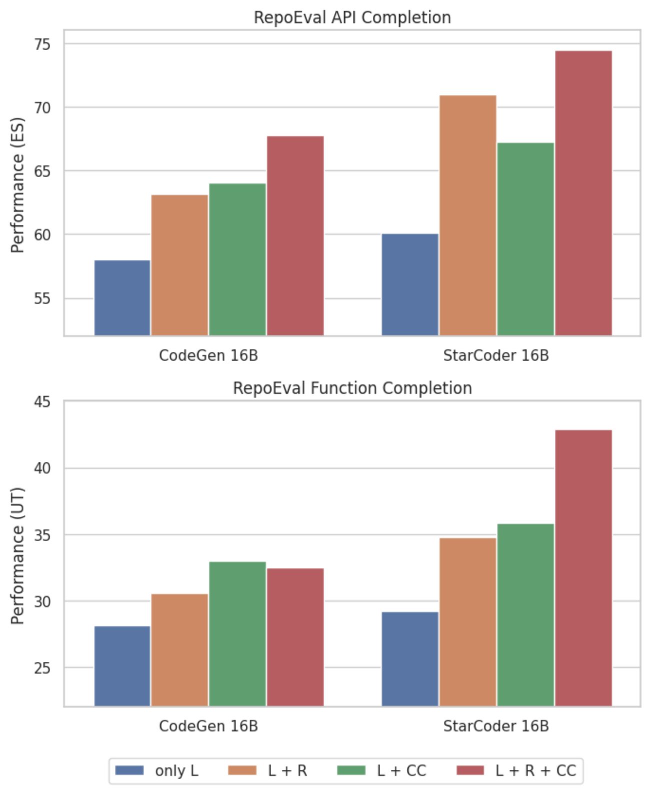

Figure 6. A comparison between four prompting strategies for RepoEval by combining left context (L), right context (R), and cross-file contexts (CC). Leveraging right contexts to build infilling-style prompt generally improves the performance regardless whether CC is present or not. StarCoder exhibits 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.

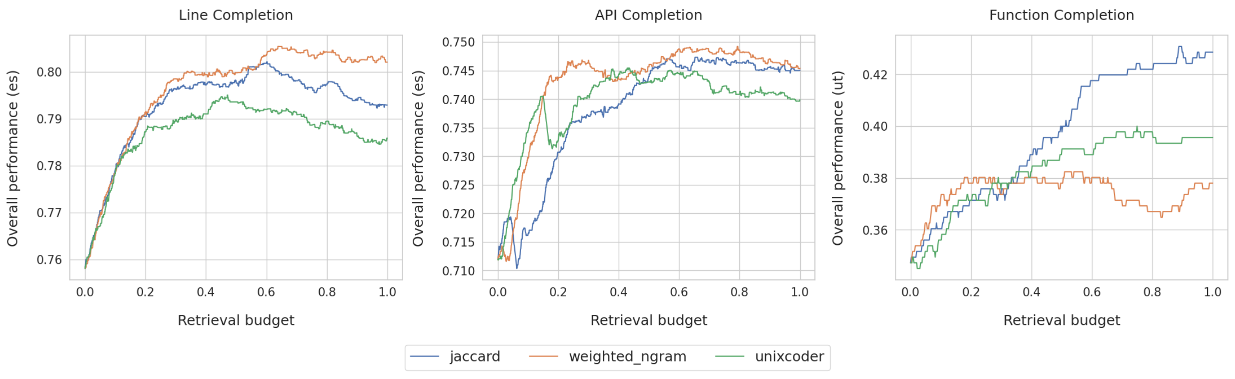

Figure 7. A comparison of the effectiveness of different similarity functions for selective RAG with StarCoder 16B. We plot the retrieval budget in the x-axis, which is the percentage of instances to perform retrieval. We report score on the entire testing dataset for each budget. Specifically, the retriever's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

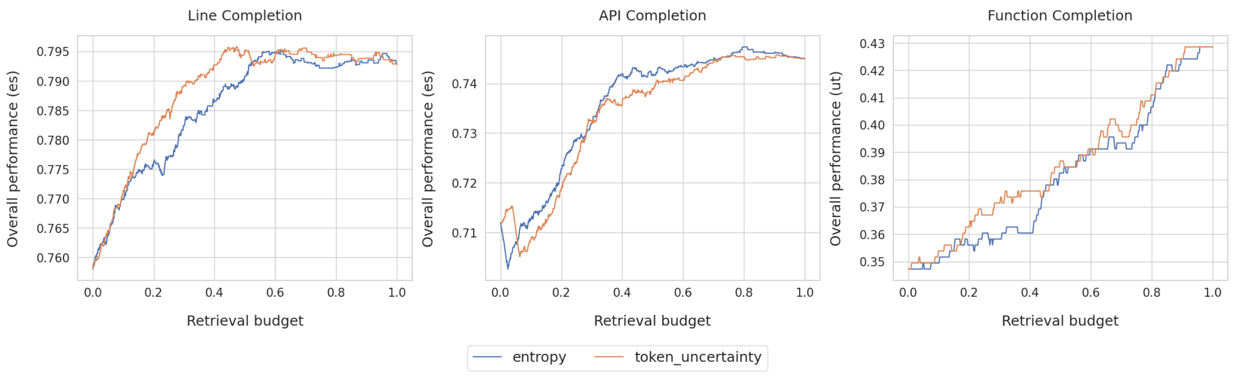

Figure 8. A comparison of the effectiveness of two uncertainty metrics for selective RAG with StarCoder 16B. We plot the retrieval budget in the x-axis and report score on the entire testing dataset for each budget. We observe that the uncertainty-based metrics fail for long sequence generation such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

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 4.2. In the next section, we present further analyses on the training data distribution.

Algorithm 1 Repoformer 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 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**
```

E. REPOFORMER Training Data Analysis

For the 240k chunk completion and 120k function completion instances, we plot the performance change with 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.

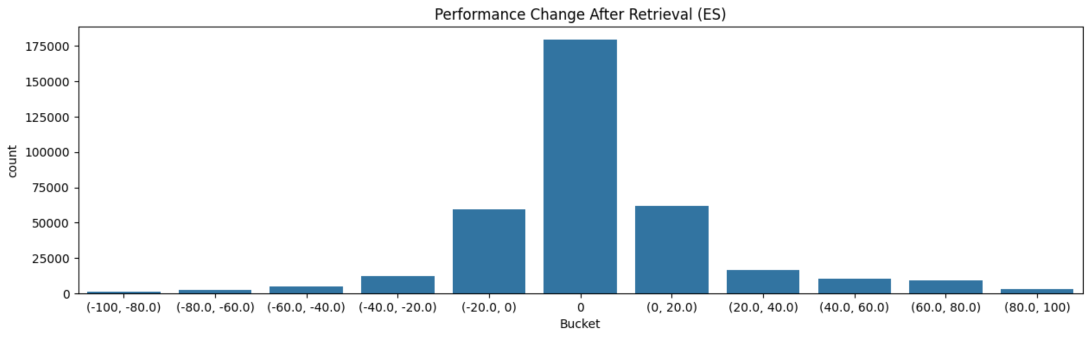

Figure 9. The performance gain on REPOFORMER training data exhibited by StarCoderBase-1B from retrieved cross-file context. The sign of the performance change is used to generate the label for REPOFORMER training. Each (start, end) bucket contains values ranging from start to end except for the central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

F. CCEval Benchmarking Dataset Creation

One drawback of RepoEval is its limited repository coverage. To verify the performance on a more diverse set of 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).

Blank 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.

G. 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 10 to Figure 13. 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.

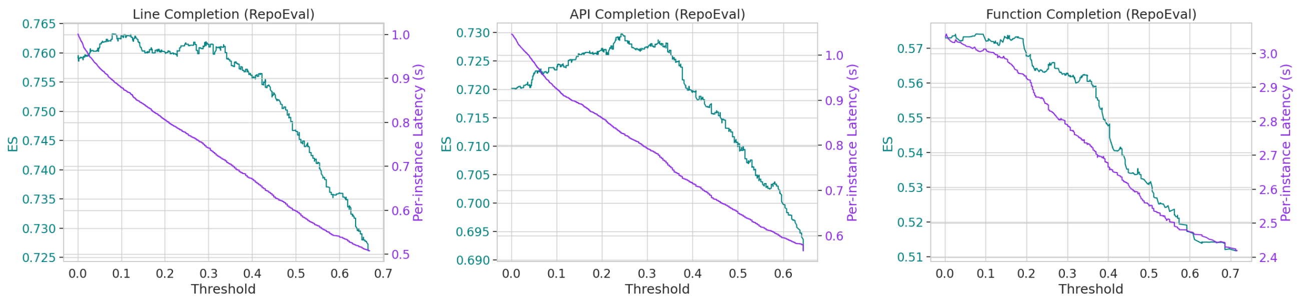

Figure 10. Latency-accuracy trade-off of self-selective RAG for REPOFORMER-1B.

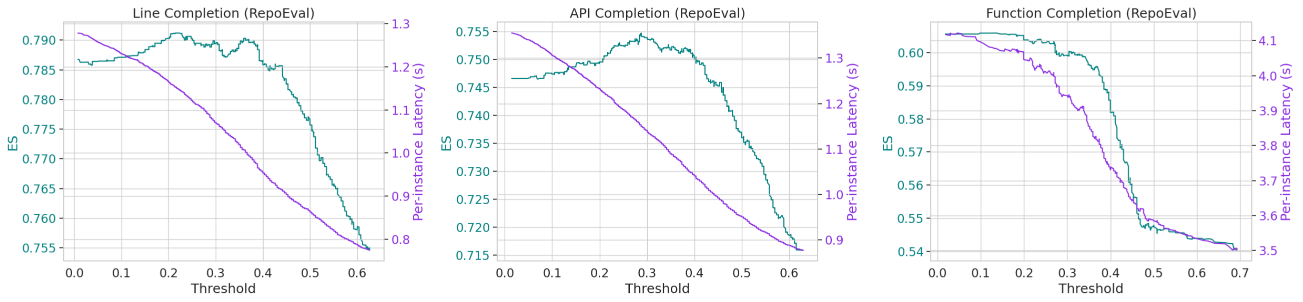

Figure 11. Latency-accuracy trade-off of self-selective RAG for REPOFORMER-3B.

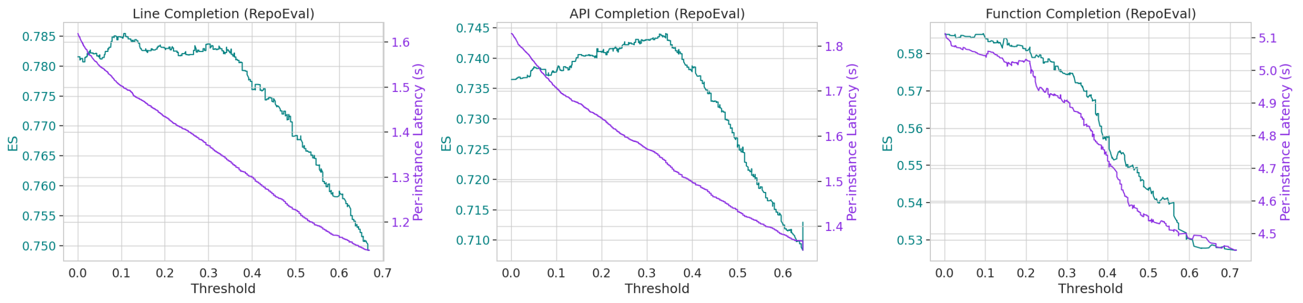

Figure 12. Latency-accuracy trade-off of selective RAG for STARCODERBASE-7B. REPOFORMER-1B is used for the selective decisions.

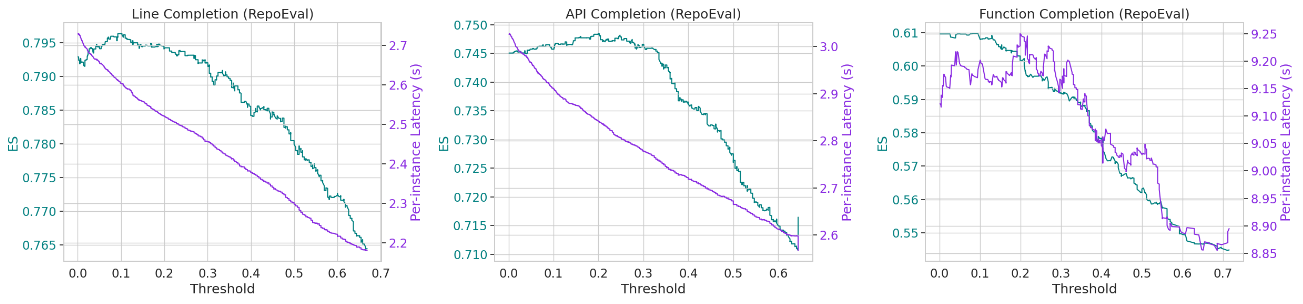

Figure 13. Latency-accuracy trade-off of selective RAG for STARCODER. REPOFORMER-1B is used for the selective decisions.