REPOFILTER: ADAPTIVE RETRIEVAL CONTEXT TRIMMING FOR REPOSITORY-LEVEL CODE COMPLETION

Anonymous authors
Paper under double-blind review

ABSTRACT

Retrieval-Augmented Generation (RAG) has recently emerged as a promising approach for repository-level code completion by integrating cross-file knowledge with in-file preceding code to provide comprehensive contexts for generation. To better understand the contribution of the retrieved cross-file contexts, we introduce a *likelihood*-based metric to evaluate the impact of each retrieved code chunk on the completion. Our analysis reveals that, despite retrieving numerous chunks, only a small subset positively contributes to the target completion, while some chunks even degrade performance. To address this issue, we leverage this metric to construct a repository-level dataset where each retrieved chunk is labeled as *positive*, *neutral*, or *negative* based on its relevance to the target completion. We then propose an adaptive retrieval context trimming framework, `REPOFILTER`, trained on this dataset to mitigate the harmful effects of negative retrieved contexts in RAG-based code completion. Extensive evaluation on the RepoEval and CrossCodeLongEval benchmarks demonstrates that `REPOFILTER` consistently improves completion accuracy compared to approaches without filtering operations across various tasks. Additionally, `REPOFILTER` significantly reduces the length of the input prompt, enhancing computational efficiency while exhibiting strong generalizability across different models. These results underscore the potential of `REPOFILTER` to enhance the accuracy, efficiency, and attributability of RAG-based repository-level code completion.

1 INTRODUCTION

Automatic code completion, particularly at the repository level, has gained significant attention due to its alignment with real-world coding scenarios. Repository-level code completion requires the model to understand the repository's domain knowledge, including cross-file contexts, to provide accurate recommendations (Zhang et al., 2023; Ding et al., 2024a). Retrieval-augmented generation (RAG) has emerged as an effective technique for integrating cross-file knowledge into the completion process. RAG-based framework first retrieves the most relevant code chunks from other files in the repository—such as user-defined APIs and inter-module dependencies—and incorporates these retrieved contexts into the prompt, which is then fed into large language models (LLMs) to enhance the completion of the current file. RAG-based methods for repository-level code completion have been extensively researched and have demonstrated substantial progress in recent years(Lu et al., 2022; Zhang et al., 2023; Liu et al.; Ding et al., 2024a).

In repository-level code completion, RAG-based methods typically rely on the preceding code snippet as a query to retrieve cross-file contexts. However, unlike natural language tasks such as question answering, where the query and relevant documents share a direct semantic relationship, the connection between the preceding code and the completed code segment is often indirect or implicit. This results in the retrieval of contexts that, despite exhibiting high semantic or token-level similarity, may not meaningfully contribute to the completion and may even degrade performance by introducing irrelevant information. Therefore, understanding the influence of each retrieved cross-file chunk is essential for optimizing the use of contextual information in code completion. Motivated by this, we systematically investigate which retrieved snippets truly support the completion process and evaluate the extent to which the retrieved context is necessary for effective code generation.

To answer this question, we conduct a preliminary experiment on the popular code completion benchmark RepoEval (Zhang et al., 2023). Specifically, we define a likelihood-based metric to evaluate the impact of each cross-file chunk on the target completion. This metric measures the difference in the model's likelihood of generating the ground-truth code with and without the inclusion of a particular context (i.e., chunk) in the prompt. Applying this metric to the retrieved top-10 cross-file contexts in the RepoEval-API dataset, we find that only 15% of the retrieved chunks genuinely support the completion, while 5.6% of the chunks degrade the performance, affecting **19.81**% of the instances in the benchmark. The remaining chunks are irrelevant. These experimental results highlight that most retrieved chunks (**85**%) either do not contribute to or even hinder code completion, underscoring the need for effective filtering strategies to identify the most beneficial contexts.

In this paper, we propose an adaptive retrieval context trimming framework, `REPOFILTER`[1], to effectively select relevant retrieved contexts for repository-level code completion. The framework is trained on our constructed dataset, where each retrieved cross-file chunk is annotated with its polarity to guide the model determine whether it is beneficial for completion. Specifically, we sample 43k instances from nearly 6k diverse Python repositories, each containing consecutive lines of code for LLMs to complete. These instances are associated with over 400k cross-file context chunks, each labeled as *positive*, *neutral*, or *negative* using our proposed *likelihood*-based metric computed against the ground-truth completion. This dataset is used to train LLMs to evaluate the polarity of retrieved code chunks and retain only the positive ones as supplementary context prior to code generation. Additionally, the model is trained to adaptively determine whether the available context is sufficient for the intended completion, thereby reducing unnecessary retrieval and computation. `REPOFILTER` redefines the generation process with a "filtering-then-generation" paradigm, enabling the model to perform on-demand retrieval and focus only on positive retrieved contexts, which mitigates the impact of noisy or irrelevant snippets and enhances overall code completion performance.

We conducted comprehensive experiments using different LLMs, including StarCoderBase-3B/7B (Li et al., 2023c) and CodeLlama-7B/13B (Roziere et al., 2023), on different repository-level benchmarks, including RepoEval and CrossCodeLongEval (Zhang et al., 2023; Ding et al., 2024a; Wu et al., 2024). Results show that `REPOFILTER` effectively filters out irrelevant retrieved content in both left-to-right and infilling code completion settings, achieving an average improvement of 3% in exact match over the baseline RAG frameworks. Moreover, `REPOFILTER` significantly reduces the length of cross-file contexts, shortening the original cross-file portion of the prompt by over 80% in token count. Notably, for those cases that contain negative-impact retrieved contexts, `REPOFILTER` successfully filters the negative contexts out, resulting in a substantial improvement of over 10% in exact match performance. Furthermore, we also establish that `REPOFILTER` can serve as a plug-and-play component, functioning as a retrieval context selection policy for larger models such as GPT-3.5 and improving their performance in code completion. Our contributions can be summarized as follows:

- We propose a likelihood-based metric to evaluate the impact of cross-file chunks on code completion and construct a code completion dataset with polarity-annotated contexts.
- We introduce `REPOFILTER`, an adaptive retrieval context trimming framework, which applies adaptive-retrieval and evaluates the polarity of retrieved code chunks and retains only beneficial contexts for repository-level code completion.
- Comprehensive experiments across multiple LLMs and benchmarks demonstrate that `REPOFILTER` consistently improves completion performance, reduces context length, and effectively mitigates the negative impact of harmful retrievals.

2 Related Work

Retrieval-Augmented Genration Despite the remarkable performance of large language models (LLMs) in text and code generation, hallucination remains a significant challenge. To address this issue, retrieval-augmented generation (RAG) has emerged as a key research area, significantly enhancing generation by providing LLMs with additional accurate knowledge (Guu et al., 2020; Lewis et al., 2020), particularly in knowledge-intensive tasks such as question answering (Izacard & Grave, 2020; Ram et al., 2023; Shi et al., 2023; Borgeaud et al., 2022). Recent studies have extended RAG to programming languages by incorporating external documents or code snippets to improve code generation (Gu et al., 2016; Zhou et al., 2022; Lu et al., 2022; Zan et al., 2022). To enhance RAG efficiency and the relevance of retrieved passages, adaptive methods have been proposed to dynamically determine when additional context should be retrieved (He et al., 2021; Mallen et al., 2022; Li et al., 2023b; Jiang et al., 2023; Wang et al., 2023a; Wu et al., 2024). Other works have focused on

[1] https://anonymous.4open.science/r/RepoFilter-5AC5

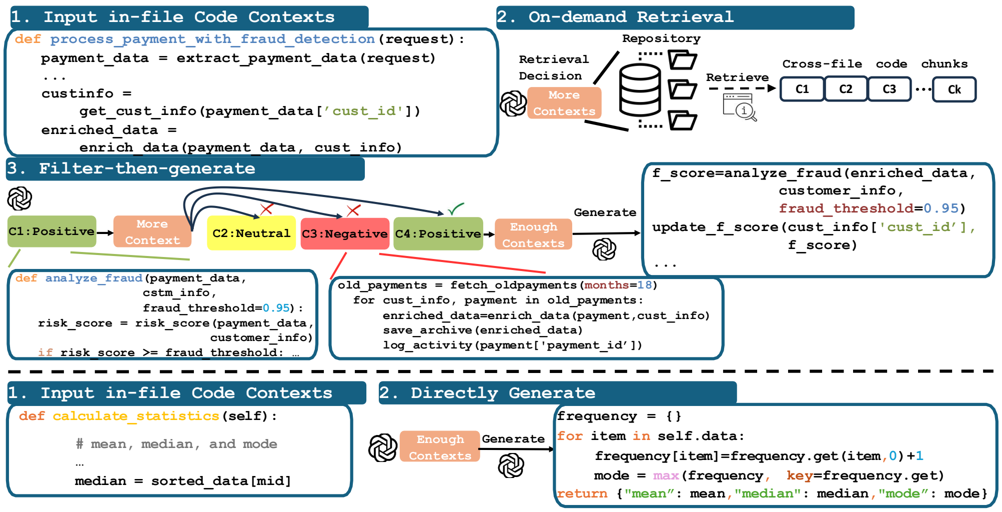


Figure 1: The overview of REPOFILTER, which initiates on-demand retrieval when the in-file context is insufficient for the intended completion; otherwise, it generates code directly. After retrieval, REPOFILTER sequentially predicts the impact of each cross-file chunk—categorized as positive, negative, or neutral—on the target completion, retaining only positive chunks. The process stops once the context is deemed sufficient, avoiding unnecessary computations.

dynamically selecting or weighting each retrieved context to improve supportiveness (Wang et al., 2023b; Asai et al.; Pan et al., 2024). Moreover, RAG has been shown to be effective in addressing various code-related tasks, such as code generation (Li et al., 2023a; Gou et al., 2024), summarization (Shi et al., 2022; Yu et al., 2022; Choi et al., 2023), and repair (Jin et al., 2023; Joshi et al., 2023). Our work builds on these advancements by introducing a dynamic context filtering approach from a sample-level perspective, specifically tailored for code completion.

Repository-level Code Completion Repository-level code completion aims to enhance developer productivity by providing context-aware code suggestions. Its practical benefits and challenges in integrating comprehensive project information have garnered significant attention. Recent research has introduced benchmarks for various completion targets, including line, API invocation, and entire function block completions, to evaluate the accuracy and functionality of completed code (Lu et al., 2022; Zhang et al., 2023; Ding et al., 2024a; Liu et al.; Li et al., 2024). While long-context LLMs are being explored to manage massive repository contexts (Guo et al., 2023), leveraging RAG to incorporate crucial cross-file contexts shows promise (Wu et al., 2024). Previous work primarily focused on how to format context to improve the accuracy of retrieval (Cheng et al., 2024; Liu et al., 2024) and enable models to better utilize these contexts (Ding et al., 2024b; Liang et al., 2024), or on incorporating information from different modalities, such as third-party libraries and similar code examples (Shrivastava et al., 2023; Liao et al., 2023; Phan et al., 2024). Apart from them, our approach emphasizes understanding the impact of each code snippet and filtering retrieved contexts based on completion intent to get the model to attend to genuinely supportive information.

3 Repository-level Retrieval-Augmented Code Completion

3.1 Problem Definition

We define the components of repository-level code completion as C_{out}, C_{in}, Y, where Y represents the target lines of code to be completed. C_{in} denotes the in-file context within the target file, while C_{out} refers to cross-file code from other files within the repository. To accommodate different completion scenarios, we introduce two distinct settings for C_{in}: (1) Infilling, where C_{in} includes both the preceding and subsequent code snippets, denoted as (C_p, C_s), and the model generates the missing code segment in between; and (2) Left-to-right, where C_{in} consists only of the preceding code snippet C_p, and the model sequentially generates the subsequent code based on this context alone. The RAG-based completion framework consists of a retrieval module that uses a retriever R and a generation module that leverages a generator G. Following previous work (Zhang et al., 2023; Ding et al., 2024a; Wu et al., 2024), we truncate cross-file contexts into chunks with a specified number of lines $C_{out} = (c_1, c_2, \ldots, c_n)$. The retriever R then queries these cross-file contexts using the chunk of the preceding code of C_{in} and retrieves the top-k candidate chunks with the highest similarity scores, denoted as $C_{cc} = R(C_{in}, C_{out}) = (c'_1, \ldots, c'_k)$. Given a CodeLLM as the generator G, the code is completed by formatting the in-file context C_{in} and the retrieved context C_{cc} into a single prompt, i.e., $\hat{Y} = G(C_{in}, C_{cc})$.

3.2 IDENTIFYING POLARITIES OF RETRIEVED CONTEXTS

We first aim to investigate the effect of context on code completion and propose a method to identify the polarity of each code chunk as positive, neutral, or negative. Our hypothesis is as follows: *A retrieved code chunk that contains critical information for the current completion will significantly increase the LLM's likelihood over the ground truth. Conversely, irrelevant or noisy chunks may have no effect or even decrease the likelihood.*

Based on this hypothesis, we define the contribution score S of a context chunk c_i to the target Y as the difference in log-likelihood between a prompt containing only the in-file context C_{in} and a prompt containing both C_{in} and the specific code chunk c_i. This is expressed as:

$$S(c_i|C_{in}, Y) = \frac{L(Y \mid C_{in}, c_i) - L(Y \mid C_{in})}{L(Y \mid C_{in})}$$

Here, $L(Y \mid C)$ represents the model's log-likelihood of the target sequence $Y = (y_1, \ldots, y_T)$ given the context C, which is formulated as $L(Y \mid C) = \sum_{t=1}^{T} \log P(y_t \mid y_1, y_2, \ldots, y_{t-1}, C; G)$. Therefore, the polarity of c_i with respect to Y can be defined as:

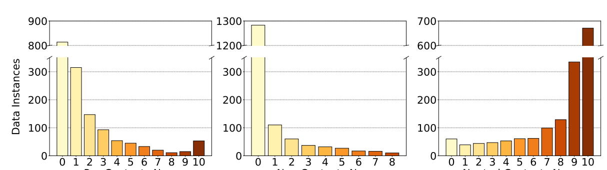
$$P(c_i|C_{in}, Y) = \begin{cases} \textit{Positive} & \text{if } S(c_i|C_{in}, Y) > T_p, \\ \textit{Negative} & \text{if } S(c_i|C_{in}, Y) < T_n, \\ \textit{Neutral} & \text{otherwise.} \end{cases}$$

where T_p and T_n represent the threshold values for determining Positive and Negative labels, respectively. In this paper, we set $T_p = 10.0\%$ and $T_n = -5.0\%$.

We evaluate the polarities of the top-10 retrieved code chunks based on the Jaccard similarity of each instance in the RepoEval dataset (Zhang et al., 2023). We compare the performance of StarCoderBase-3B in code completion using four different strategies for incorporating cross-file contexts into the prompt: (1) Full Retrieve, where all top-10 retrieved chunks are included in the prompt; (2) Positive-only, which retains only the chunks labeled as positive; (3) w/o Negative, which excludes negative chunks from the retrieved contexts; (4) w/o Neutral, which excludes chunks labeled as neutral. Results in Table (a) reveal three key findings: (1) The model with prompts containing only positive chunks outperforms the one including all candidate cross-file chunks; (2) Eliminating neutral chunks does not significantly affect the model's completion performance; (3) Removing negative chunks in the prompt improves code completion performance. These findings align with our expectations of how positive, neutral, and negative chunks impact completion, further validating the effectiveness of our likelihood-based metric by demonstrating that the model's likelihood scores can reliably indicate which retrieved contexts contribute meaningfully to the completion task.

Strategies	Exact Match (%)
Full Retrieve	47.27
Positive-only	**49.47**
w/o Neutral	47.02
w/o Negative	**49.96**

(a) The impact of different context selection strategies for completion on RepoEval-API.



(b) Distribution of the number of positive/negative/neutral cross-file chunks for each data instance on RepoEval-API.

Furthermore, Figure (b) illustrates the distribution of positive, negative, and neutral chunks within the cross-file contexts retrieved for each instance. The x-axis represents the number of positive, negative, or neutral chunks in each instance, while the y-axis indicates the number of data instances. The data reveals that only about half of the instances contain any positive-impact chunks within their retrieved contexts, and among these, most contain only 1-2 positive chunks out of the 10 retrieved. In contrast, nearly 20% of instances include negatively impactful chunks, while the majority of retrieved chunks are neutral and irrelevant to the target completion. This distribution indicates that only a small subset of the retrieved contexts contributes meaningfully to the completion, while the rest introduce noise or even hinder performance. To address this issue, we propose REPOFILTER, a framework that enhances efficiency by enabling on-demand retrieval and filtering out irrelevant or harmful chunks. By focusing on positive contexts, REPOFILTER improves both the performance and efficiency of code completion.

4 REPOFILTER

We introduce REPOFILTER, a repository-level code completion framework designed to achieve two key objectives: (1) selectively retrieving and incrementally adding code chunks as needed, and (2) filtering out irrelevant chunks to prevent the model from attending to noisy information, thereby making the code completion process more precise and interpretable. To achieve this, we predefine a set of special signal tokens, $\mathcal{T}$, categorized into two types. The first type consists of adaptive-retrieval tokens (<EC>, <MC>). <EC> indicates that there is sufficient context to proceed with code completion, eliminating the need for further retrieval or additional chunks. In contrast, the <MC> token signals that the model requires more cross-file chunks for intended completion. The second type includes polarity tokens (<pos>, <neg>, <neu>), which denote whether a cross-file chunk has a positive, negative, or neutral impact on code completion. During the generation process, the model is trained to autonomously evaluate and generate these signal tokens at various stages to perform their respective functions.

4.1 Training

Dataset Construction. We followed the approach outlined in (Wu et al., 2024) to construct a fine-tuning dataset using the licensed repository-level dataset, Stack (Kocetkov et al., 2022). First, we randomly sampled the target Y from the raw repository data, which could be a random line, a consecutive code chunk, or an entire function body. We then retrieved the top 10 cross-file code chunks using Jaccard Similarity (Jaccard, 1912) and labeled the polarity of each chunk based on the likelihood-based metric. The detailed data construction process is provided in Appendix A.

We verbalize the training data in a fill-in-the-middle format using two strategies. The first strategy can be sequentially expressed as:

<PREFIX>[Left Context]<SUFFIX>[Right Context]<MC>[C_1]<pos><MC>[C_2]<neu>..<MC>[C_n]<pos><EC><MIDDLE>

where <PREFIX><SUFFIX><MIDDLE> are special tokens defined by the code LLM for the fill-in-the-middle format. Additionally, the sub-sequence <MC>[C_1]<pos><MC>[C_2]<neu>..<MC>[C_n]<pos><EC> represents the verbalized cross-file contexts augmented with both adaptive-retrieval tokens and polarity tokens. Moreover, the order of the candidate chunks is randomly shuffled, but the sequence includes all positive chunks, with the final chunk always labeled as positive to ensure it provides the last critical piece of information for code completion. This training data format is designed to guide the model accurately labeling the polarity of cross-file code chunks. The second format is denoted as:

<PREFIX>[Left Context]<SUFFIX>[Right Context]<MC>[C_1]<pos><MC>[C_4]<pos><MC>[C_n]<pos><EC><MIDDLE>[Target]

Here, the sub-sequence of cross-file chunks includes only the positive chunks. This format is designed to help the model determine whether additional information is required for completion and to complete the code based on positive cross-file code chunks. If there are no positive-labeled candidate chunks, the cross-file chunk sequence will consist solely of the token <EC>, indicating that the in-file context is sufficient and no further retrieval is necessary.

Training objectives. Using the verbalized training dataset, we optimize the model with a standard teacher-forcing approach. This optimization is achieved by minimizing a weighted sum of the cross-entropy loss over both the signal tokens and the target tokens for code completion:

$$\mathcal{L} = -\log P_G(Y|C_{in}, C_{cc}) + \lambda(-\log P_G(\mathcal{T}|C_{in}, C_{cc}))$$

To prevent the model from memorizing irrelevant local contexts, we mask the in-file and cross-file contexts during loss calculation.

4.2 Inference

The inference process of REPOFILTER is designed to dynamically balance retrieval and generation for code completion. As outlined in Algorithm 1, the process can be divided into four key phases:
Analyzing In-File Context: The model first evaluates the in-file context to determine whether additional cross-file retrieval is needed. This decision is based on generating an adaptive-retrieval token, choosing between <EC> (enough context) and <MC> (more context needed), guided by a predefined threshold applied to the softmax probability of these tokens.

Algorithm 1: REPOFILTER Inference Process

Input: Generator G, Retriever R, Cross-file contexts C_{out}, In-file contexts $C_{in} = (C_p, C_s)$, Adaptive-retrieval token set $\mathcal{T}_A$, Polarity token set $\mathcal{T}_P$,threshold for choosing polarity tokens t_p, t_n, threshold for choosing adaptive retrieval token t_c

Output: Completed code lines $\hat{Y}$

$X \leftarrow$ (PREFIX, C_p, SUFFIX, C_s) /* Initialize input sequence */

$m \leftarrow$ Select(Softmax$_{\mathcal{T}_A}(G(m|X)), t_c$) /* Generate adaptive-retrieval token */

if $m =< EC >$ **then**

  $X \leftarrow append(X,$ [MIDDLE])

else if $m =< MC >$ **then**

  $C_{cc} \leftarrow R(C_{in}, C_{out})$ /* Retrieve Top-K cross-file chunks */

  foreach *chunk* $c_i \in C_{cc}$ **do**

    $p \leftarrow$ Select(Softmax$_{\mathcal{T}_P}(G(p|X)), t_p, t_n$) /* Generate polarity token */

    if $p =< pos >$ **then**

      $X \leftarrow append(X, c_i)$

      $m \leftarrow$ Select(Softmax$_{\mathcal{T}_A}(G(m|X)), t_c$) /* Reassess context sufficiency */

      if $m =< EC >$ **then**

        $X \leftarrow append(X,$ [MIDDLE])

        break /* Context is sufficient */

return $\hat{Y} \leftarrow G(X)$ /* Generate final completed code */

Initiating Retrieval (if needed): If <MC> is selected, the retriever R is triggered to fetch the top-K cross-file code chunks relevant to the input. These retrieved chunks are then sequentially appended to the input sequence for further evaluation.
Chunk Evaluation and Filtering: Each retrieved chunk is assessed by the model for relevance using polarity tokens (<pos>, <neu>, and <neg>). Positive chunks (<pos>) are retained and appended to the sequence to enrich the context. Neutral or negative chunks (<neu> or <neg>) are filtered out to avoid irrelevant or misleading information.
Reassessing Context and Generating Code: After adding a positive chunk, the model reevaluates whether the context is now sufficient for code generation. If <EC> is selected at this stage, the model switches to code generation using a fill-in-the-middle format. Otherwise, the process continues, iterating through the remaining retrieved chunks.
This iterative process ensures that only the most relevant cross-file contexts are used, improving the model's ability to generate accurate and efficient code completions. Moreover, we present a generation case utilizing our framework in Appendix F for illustrating the inference process.

5 Experimental Setup

5.1 Training & Inference

Dataset. Following Wu et al. (2024), we sampled 6k Python repositories from Stack (Kocetkov et al., 2022). For each data instance, we retrieved 10 candidate cross-file code chunks with the highest Jaccard Similarity scores. Each chunk was labeled by computing its contribution score S. Several post-processing steps were implemented to filter out low-quality data based on three criteria: (1) the target file contains at least three local import statements; (2) the target lines do not include comments or import statements, and the target sequence consists of at least six tokens; and (3) the in-file context and the 10 candidate cross-file chunks are expected to provide sufficient context for completion. To evaluate this, we set a threshold of 0.5 for edit similarity, assuming that when a positive chunk achieves an edit similarity score above this threshold, the contexts can be considered sufficiently informative for completion. After applying these criteria, we obtained 43k instances containing 400k labeled cross-file chunks. We verbalized these instances based on the two strategies mentioned in section 4.1 to construct the final dataset, consisting of 130k instances. We allocated 95% of the data for training and the remaining 5% for validation. More details on the implementation and statistics can be found in Appendix A.

Train. We train LLMs using different variants from two model families: StarCoderBase-3B/7B (Li et al., 2023c) and CodeLlama-7B/13B (Roziere et al., 2023). The models are optimized over 2 epochs, utilizing an initial learning rate of 2e-5, 5% warm-up steps, and linear decay. Additionally,

Table 1: Code completion in Infilling completion setting.

Model	RAG Strategies	Repoeval-Line		Repoeval-API		Repoeval-Func		Cclongeval-Chunk		Cclongeval-Func
		EM	ES	EM	ES	UT	ES	EM	ES	ES
StarCoderBase-3B	No-Retrieve	46.56	68.93	39.09	65.19	22.42	39.43	33.57	62.15	48.62
	Full-Retrieve	56.25	74.72	47.27	72.69	27.25	48.34	38.21	64.05	46.33
	RepoFormer	57.13	75.47	49.22	74.06	27.91	48.70	39.69	67.67	48.75
	REPOFILTER	**60.50**	**79.07**	**50.59**	**77.28**	**29.67**	**51.35**	**41.55**	**68.63**	**52.61**
StarCoderBase-7B	No Retrieve	50.50	71.75	40.71	66.78	24.18	43.26	36.93	64.16	51.11
	Full-Retrieve	58.56	76.86	48.16	74.62	29.23	51.77	43.23	68.31	46.40
	RepoFormer	59.25	78.06	49.47	77.00	31.21	50.43	44.64	70.40	45.84
	REPOFILTER	**61.44**	**80.12**	**51.09**	**78.53**	**33.41**	**53.69**	**45.97**	**71.28**	**55.37**
CodeLlama-7B	No Retrieve	50.69	72.22	40.34	65.80	23.74	43.32	36.17	64.05	49.23
	Retrieve	59.06	77.89	47.59	72.21	28.79	51.37	44.29	68.11	51.92
	RepoFormer	59.19	78.18	48.34	74.91	**32.09**	51.50	45.74	68.39	50.92
	REPOFILTER	**62.56**	**81.24**	**51.53**	**77.46**	31.65	**53.23**	**49.53**	**73.78**	**53.45**
CodeLlama-13B	No Retrieve	52.69	73.63	41.03	66.89	25.05	46.08	40.88	66.22	51.65
	Full-Retrieve	60.31	77.15	48.66	73.39	30.55	53.29	46.17	69.45	54.18
	RepoFormer	61.00	80.38	49.28	**78.02**	33.19	53.28	47.74	70.09	54.17
	REPOFILTER	**62.94**	**81.56**	**51.84**	77.74	**34.29**	**56.70**	**50.22**	**73.03**	**57.69**

we set $\lambda = 2.0$, a batch size of 512, and a maximum sequence length of 4096. Training is conducted on 4 NVIDIA A100 GPUs, each with 80GB of memory.

Retrieval. In line with previous studies (Zhang et al., 2023; Ding et al., 2024a), we divide cross-file code into chunks using a window size of 10 lines and a stride size of 5 lines. The preceding 10 lines of in-file code are then used as a query to retrieve the top-10 cross-file chunks, ranked by their Jaccard similarity scores (Jaccard, 1912). Our main experiments focus on sparse retrieval, as prior research (Ding et al., 2024a) has demonstrated that dense retrieval methods do not improve completion performance. This limitation occurs because cross-file chunks with high semantic similarity to the preceding code do not necessarily capture the code's underlying intent, and thus may not meaningfully contribute to completion. Additionally, we evaluate the performance of `REPOFILTER` when using a dense retriever with UniXcoder as the encoder (Guo et al., 2022), as detailed in Appendix C.

Inference. In our experiments, we use greedy decoding for code completion. For special signal tokens, the probability threshold for `<MC>` is set to 0.3, and `<EC>` is generated otherwise. For polarity tokens, we apply a threshold of 0.3 for both `<pos>` and `<neg>`, prioritizing `<pos>` if it meets the threshold first, and defaulting to `<neu>` if neither does. Detailed ablation studies on threshold settings are provided in Section 6.5. Additionally, we set the maximum token length of the prompt to 4096, with 1024 tokens allocated for the in-file context and 3072 for the cross-file chunks. We utilize vLLM (Kwon et al., 2023) to accelerate the inference process.

5.2 Evaluation

Datasets. We evaluate our model on two benchmarks: RepoEval (Zhang et al., 2023), which includes line, API, and function completion tasks derived from 14 high-quality Python repositories; and CrossCodeLongEval (Wu et al., 2024), which extends the repositories from CrossCodeEval (Ding et al., 2024a) to include chunk-level and function-level code completion tasks. We consider two completion settings in our experiments: (1) Infilling, where the model completes the middle part of the code based on both the preceding and subsequent context; and (2) Left-to-right, where the model generates code sequentially using only the preceding context. We evaluate both settings on the RepoEval and CrossCodeLongEval datasets using these benchmarks.

Model & Baselines. The baseline models are consistent with the CodeLlama and StarCoderBase variants used to train `REPOFILTER`. We compare our model against three baseline retrieval settings: (1) No Retrieve—the model completes the code using only in-file contexts; (2) Full Retrieve (Zhang et al., 2023; Ding et al., 2024a)—the model completes the code using in-file contexts along with the top 10 candidate cross-file code chunks; and (3) RepoFormer (Wu et al., 2024)—the model determines whether retrieval is necessary for completion be for initiate retrieval. Detailed implementations of these baselines are provided in Appendix B.

Metrics. Following (Zhang et al., 2023; Ding et al., 2024a; Wu et al., 2024), we use the execution-based metric pass rate of Unit Tests (UT) to evaluate function-level completion in the RepoEval dataset. For all other data, we use the reference-based metrics Exact Match (EM) and Edit Similarity (ES) for evaluation.

Table 2: Code completion in Left-to-right completion setting.

Model	RAG Strategies	Repoeval-Line		Repoeval-API		Repoeval-Func		Cclongeval-Chunk		Cclongeval-Func
		EM	ES	EM	ES	UT	ES	EM	ES	ES
StarCoderBase-3B	No Retrieve	33.37	57.94	27.33	56.11	17.80	36.63	23.08	51.09	42.44
	Full Retrieve	48.00	68.44	38.21	65.37	23.96	46.39	33.82	57.37	43.32
	RepoFormer	47.38	69.67	38.59	67.32	25.05	47.40	34.20	**59.38**	44.89
	REPOFILTER	**50.50**	**71.23**	**40.84**	**70.76**	**25.49**	**48.81**	**35.61**	59.02	**46.40**
StarCoderBase-7B	No Retrieve	35.69	59.64	28.96	57.51	19.56	37.54	27.03	56.16	51.11
	Full Retrieve	48.94	69.05	39.96	65.97	25.93	48.11	39.24	62.40	46.40
	RepoFormer	48.44	68.09	38.40	**70.22**	25.71	46.16	38.68	62.27	45.84
	REPOFILTER	**51.32**	**71.90**	**42.15**	69.70	**26.59**	**49.87**	**39.65**	**64.28**	**55.37**
CodeLlama-7B	No Retrieve	37.25	61.61	28.52	57.76	20.00	40.04	27.80	55.74	43.04
	Full Retrieve	50.00	68.47	40.90	66.33	24.62	47.64	38.95	61.47	50.16
	RepoFormer	48.63	68.97	38.34	68.29	26.37	47.52	37.06	60.49	48.12
	REPOFILTER	**51.12**	**70.63**	**41.46**	**71.04**	**27.47**	**48.33**	**39.40**	**63.05**	**51.03**
CodeLlama-13B	No Retrieve	39.25	62.55	28.89	58.14	21.76	41.06	29.07	55.19	43.62
	Full Retrieve	51.81	71.92	42.28	69.42	26.59	49.00	40.95	**65.67**	47.82
	RepoFormer	50.06	69.03	41.59	69.16	26.25	48.73	41.10	65.38	49.96
	REPOFILTER	**52.94**	**72.76**	**42.71**	**72.59**	**27.69**	**49.99**	**41.80**	65.34	**54.69**

Table 3: Code completion in data instances containing negative cross-file chunks.

Model	RAG-strategy	Left-to-right				Infilling			
		RepoEval-Line		RepoEval-API		RepoEval-Line		RepoEval-API	
		EM	ES	EM	ES	EM	ES	EM	ES
Starcoderbase-7B	No Retrieve	7.82	34.92	3.36	35.98	17.13	42.92	11.02	44.04
	Full Retrieve	7.23	37.39	5.46	39.46	16.39	42.15	10.63	47.45
	REPOFILTER	**28.92**	**57.84**	**16.03**	**55.79**	**29.28**	**62.36**	**22.83**	**63.90**
CodeLlama-7b	No Retrieve	7.83	38.99	3.36	34.47	17.13	46.36	10.24	42.44
	Full Retrieve	6.62	38.01	6.30	43.24	12.15	44.23	9.84	43.14
	REPOFILTER	**22.89**	**55.31**	**17.64**	**54.71**	**34.25**	**65.25**	**20.47**	**63.21**

6 RESULTS & ANALYSIS

6.1 MAIN RESULTS

We evaluate the code completion performance of REPOFILTER in two settings across different models and compare it with several baseline RAG strategies. The results, presented in Tables 1 and 2, demonstrate that incorporating retrieved cross-file chunks significantly improves performance over models that rely solely on preceding code for generation. This improvement is evident across both reference-based and execution-based evaluation metrics. When compared to full and adaptive retrieval methods, REPOFILTER consistently achieves notable enhancements across various tasks. For example, in the infilling setting, the performance of StarCoderBase-3B under REPOFILTER framework is comparable to, or even surpasses, that of the StarCoderBase-7B model using full retrieval. Furthermore, the gains achieved by REPOFILTER in this setting mirror those observed when negative chunks are removed from the prompt, as shown in Table (a) in Section 3.2. This indicates that our method effectively filters out noise in cross-file chunks, retaining only those that positively contribute to code generation.

We also evaluate code completion performance across two settings: Infilling and Left-to-right completion. The results in Tables 1 and 2 show that incorporating subsequent code snippets significantly enhances the model's completion capabilities. For line-level and chunk-level tasks, including subsequent code results in nearly a 10% improvement compared to the Left-to-right setting under the same RAG strategy. However, for function-level completion, the benefit of incorporating in-file subsequent code is less pronounced. This may be due to the function body serving as an independent module, making subsequent code (i.e., code outside the function) less relevant to the function's content. Additionally, RepoFormer underperforms compared to the full-retrieval strategy in some tasks under the Left-to-right setting, and the improvements achieved by REPOFILTER in this setting are less substantial than those observed in the Infilling setting. We hypothesize that the model's ability to assess the utility of retrieved chunks for completion depends on its understanding of the code's intention. However, with only the preceding code available, identifying the code's intention becomes more challenging, leading to a decline in performance compared to the Infilling setting.

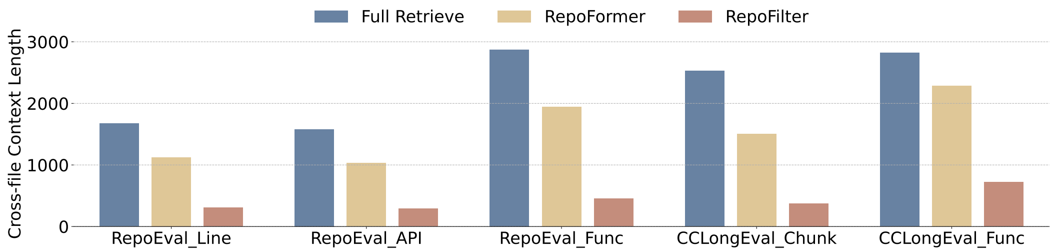


Figure 3: Cross-file Length of different benchmarks when applying different RAG Strategies.

Table 4: Model performance on RepoEval-API when provided with cross-file contexts filtered by REPOFILTER-3B.

Setting	RAG Strategy	Starcoder-7B		Starcoder-15B		Starcoder2-7B		CodeLlama-7B		CodeLlama-13B		DeepSeek-16B		QwenCoder-7B		GPT-3.5-turbo	
		EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
Infilling	Full Retrieve	48.16	74.62	50.66	75.73	40.90	67.14	47.59	72.21	48.66	73.39	49.78	73.60	41.84	69.19	34.21	56.13
	REPOFILTER	**50.28**	**74.81**	**51.72**	**77.08**	**42.21**	**70.33**	**49.22**	**73.34**	**49.47**	**73.78**	**51.59**	**76.05**	**45.97**	**71.41**	**36.27**	**58.75**
Left-to-right	Full Retrieve	39.96	65.97	42.46	**70.34**	37.34	62.12	40.90	66.33	42.28	69.42	42.03	70.14	42.56	69.02	31.14	56.35
	REPOFILTER	**41.90**	**66.70**	**42.90**	69.92	**38.84**	**63.77**	**41.65**	**69.06**	**43.46**	**70.85**	**42.96**	**71.47**	**42.63**	**69.10**	**32.02**	**57.03**

6.2 PERFORMANCE ON INSTANCES RETRIEVED WITH NEGATIVE CONTEXTS

REPOFILTER is designed to filter out irrelevant and noisy retrieved chunks that may negatively impact the model's completion. Although REPOFILTER demonstrates improvements over baseline RAG strategies, it remains unclear how the model performs when provided with negative chunks. To address this, we use the method proposed in Section 3.2 to identify instances containing negative chunks among the top-10 retrieved cross-file code chunks in the RepoEval-API and RepoEval-Line tasks. Out of the 1,600 test instances, we identified 285 and 166 instances containing negative chunks for API and line-level completion, respectively. We then evaluate REPOFILTER on these subsets in both the Infilling and Left-to-right settings to investigate whether the model can effectively filter out noisy information. The results, summarized in Table 3, show that full retrieval exhibits poor performance on these samples, often performing worse than directly generating code based solely on in-file context in most scenarios. This finding validates that these identified negative chunks directly degrade the model's completion performance. In contrast, REPOFILTER outperforms full retrieval by a significant margin across both tasks and settings, confirming that our model can effectively filter chunks based on their supportiveness for completion, thereby mitigating the impact of potential noise.

6.3 LENGTH OF CROSS-FILE CONTEXTS

In code completion, the lack of explicit information about the code's intent often results in inaccurate retrievals. To address this, a common practice is to provide the generator with up to 10 candidate chunks. However, this approach results in overly lengthy contexts, and as analyzed in Section 3.2, only a small portion of these chunks are truly relevant to the completion task. We use the length of cross-file context tokens provided to the generator as an indicator of both efficiency and attributability. Figure 3 illustrates the final cross-file context lengths under three strategies: full retrieval, RepoFormer, and REPOFILTER, evaluated across multiple benchmarks in the infilling setting. Notably, REPOFILTER refers to its StarCoderBase-3B variant, and similar context lengths were observed with other model variants of REPOFILTER after filtering. We observe that full retrieval with 10 cross-file chunks leads to excessively lengthy contexts, all exceeding 1,500 tokens. RepoFormer, which selectively determines the necessity of retrieving cross-file contexts, reduces context length by approximately 30%. Building on this, REPOFILTER further filters out irrelevant chunks, reducing context length by nearly **80**% compared to full retrieval. This substantial reduction in context length not only improves efficiency but also increases information density, thereby enhancing the attributability of the model's completions without compromising performance.

6.4 REPOFILTER AS FILTERING POLICY

We investigate whether the filtering decisions made by a smaller model can effectively generalize to larger models from different families and sizes. In our experiments, the StarCoderBase-3B variant of REPOFILTER is used to perform adaptive retrieval and context filtering, with the filtered

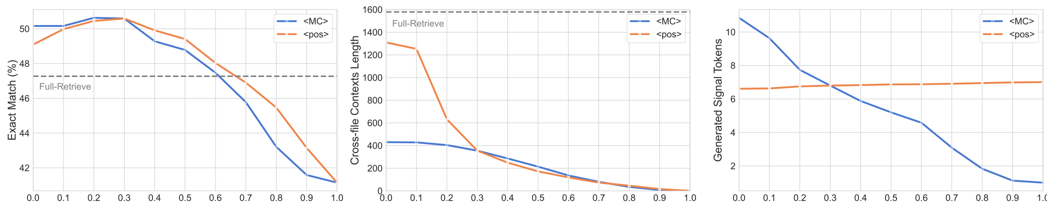


Figure 4: Ablations of threshold setting of signal tokens

prompts subsequently provided to larger models for code completion. The evaluation is conducted on the RepoEval-API task under both Infilling and Left-to-right settings. Larger models, including StarCoderBase-7B/16B, StarCoder2-7B, CodeLlama-7B/13B, DeepSeek-16B, QWen2.5-Coder-7B, and GPT-3.5-turbo (Hui et al., 2024), are tested to assess their performance using the filtered contexts. The results, shown in Table 4, demonstrate consistent improvements in both EM and ES scores, along with significantly shorter prompt lengths when larger models generate code using the filtered contexts compared to full retrieval. These findings suggest that the filtering decisions made by the smaller model generalize well across diverse architectures, highlighting the potential of our method to serve as a plug-and-play module for enhancing both the performance and efficiency of larger models in code completion tasks.

6.5 Ablations

We ablate the impact of different thresholds for `<MC>` and `<pos>` on code completion performance and resource efficiency. The experiments are conducted using the StarCoderBase-3B model on the RepoEval-API task under the infilling setting, while ablations on other tasks are presented in Appendix E. Figure 4 presents the results across three key metrics: Exact Match (EM), cross-file context length, and the number of generated signal tokens.

Thresholds vs. Exact Match: The left figure of 4 shows that EM values are highly sensitive to thresholds for `<MC>` and `<pos>` tokens. For `<MC>`, EM remains stable and relatively high (50.16% to 50.59%) between thresholds 0.0 and 0.3 but drops sharply beyond this range, reaching 41.15% at a threshold of 1.0 due to disabled retrieval excluding relevant information. A similar pattern is observed for `<pos>`: a higher threshold prevents identifying positive chunks, while a lower threshold risks including noise, reducing accuracy.

Thresholds vs. Cross-file Context Lengths: The middle figure of 4 shows that increasing thresholds for `<MC>` and `<pos>` reduces cross-file context length, improving efficiency. This effect is more pronounced for `<pos>`, with a sharp reduction as its threshold rises. At a threshold of 0.3 for both tokens, the context length is reduced to 355 tokens—less than 30% of the original—balancing minimal context length and high EM scores.

Thresholds vs. Generated Signal Tokens: While REPOFILTER improves completion performance and reduces prompt length, filtering incurs additional computational costs compared to direct generation. The number of generated signal tokens indicates this cost. As shown in the right figure of 4, the `<pos>` threshold minimally affects this metric, while the `<MC>` threshold significantly impacts it. A lower `<MC>` threshold causes more chunks to be evaluated, increasing signal token counts, which drop sharply from 10.85 tokens at a threshold of 0.0 to 1.0 token at a threshold of 1.0.

Model performance is highly sensitive to the signal token thresholds. A low `<MC>` threshold increases resource demand without improving performance, while a high threshold enhances efficiency but reduces completion quality. A threshold of 0.3 achieves the best trade-off.

7 Conclusion

In this paper, we introduced a metric to evaluate the influence of retrieved cross-file chunks on code completion and constructed a labeled dataset to categorize these chunks by their impact. We developed REPOFILTER, a framework that adaptively retrieves and filters relevant contexts, improving both accuracy and efficiency in code generation. Our results show that REPOFILTER significantly enhances performance, particularly by mitigating the effects of misleading contexts, while reducing the computational load. Additionally, the framework generalizes well across various models, demonstrating its versatility and effectiveness in repository-level code completion. Further discussion regarding the limitations and potential impact of our work can be found in Appendix D.

REFERENCES

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR, 2022.

Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level code completion. *arXiv preprint arXiv:2405.19782*, 2024.

Yunseok Choi, Cheolwon Na, Hyojun Kim, and Jee-Hyong Lee. Readsum: Retrieval-augmented adaptive transformer for source code summarization. *IEEE Access*, 11:51155–51165, 2023.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36, 2024a.

Yangruibo Ding, Zijian Wang, Wasi U Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Cocomic: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 3433–3445, 2024b.

Qianwen Gou, Yunwei Dong, Yujiao Wu, and Qiao Ke. Rrgcode: Deep hierarchical search-based code generation. *Journal of Systems and Software*, 211:111982, 2024.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pp. 631–642, 2016.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 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, 2022.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*, pp. 12098–12107. PMLR, 2023.

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

Junxian He, Graham Neubig, and Taylor Berg-Kirkpatrick. Efficient nearest neighbor language models. *arXiv preprint arXiv:2109.04212*, 2021.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

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

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

Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983*, 2023.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1646–1656, 2023.

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 5131–5140, 2023.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33: 9459–9474, 2020.

Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. Skcoder: A sketch-based approach for automatic code generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2124–2135. IEEE, 2023a.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2404.00599*, 2024.

Junyi Li, Tianyi Tang, Wayne Xin Zhao, Jingyuan Wang, Jian-Yun Nie, and Ji-Rong Wen. The web can be your oyster for improving language models. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 728–746, 2023b.

R Li, LB Allal, Y Zi, N Muennighoff, D Kocetkov, C Mou, M Marone, C Akiki, J Li, J Chim, et al. Starcoder: May the source be with you! *Transactions on machine learning research*, 2023c.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Hongwei Chen, Chengpeng Wang, Gang Fan, et al. Repofuse: Repository-level code completion with fused dual context. *arXiv preprint arXiv:2402.14323*, 2024.

Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *arXiv preprint arXiv:2312.05772*, 2023.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003*, 2024.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 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, 2022.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. *arXiv preprint arXiv:2212.10511*, 2022.

Ruotong Pan, Boxi Cao, Hongyu Lin, Xianpei Han, Jia Zheng, Sirui Wang, Xunliang Cai, and Le Sun. Not all contexts are equal: Teaching llms credibility-aware generation. *arXiv preprint arXiv:2404.06809*, 2024.

Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. Repohyper: Better context retrieval is all you need for repository-level code completion. *arXiv preprint arXiv:2403.06095*, 2024.

Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics*, 11:1316–1331, 2023.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. Race: Retrieval-augmented commit message generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 5520–5530, 2022.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. Replug: Retrieval-augmented black-box language models. *arXiv preprint arXiv:2301.12652*, 2023.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*, 2023.

Yile Wang, Peng Li, Maosong Sun, and Yang Liu. Self-knowledge guided retrieval augmentation for large language models. *arXiv preprint arXiv:2310.05002*, 2023a.

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

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. *arXiv preprint arXiv:2403.10059*, 2024.

Chi Yu, Guang Yang, Xiang Chen, Ke Liu, and Yanlin Zhou. Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 82–93. IEEE, 2022.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. When language model meets private library. *arXiv preprint arXiv:2210.17236*, 2022.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, 2023.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao JIang, and Graham Neubig. Doccoder: Generating code by retrieving and reading docs. *arXiv preprint arXiv:2207.05987*, 2022.

A Details of Dataset Construction

Data Collection We collected a dataset comprising 5,824 distinct projects from Stack (Kocetkov et al., 2022), each with a minimum of 10 stars, ensuring no overlap with the RepoEval and CrossCodeEval datasets used as test sets in our main experiments. To ensure interaction with other modules within the repository, target code lines were randomly sampled from files containing at least three local import statements. These target code segments include a variety of formats to promote model generalization, ranging from single lines and chunks of 2-20 lines to full functions with fewer than 50 lines. Additionally, target lines were filtered to exclude comments and include at least six tokens. The sampled data was divided equally into two completion settings: the left-to-right setting, where only the preceding code is provided, and the infilling setting, which includes both preceding and subsequent code as in-file context.

Data Labeling We then chunked the cross-file contexts and constructed queries based on the in-file context for each target Y. For each query, we retrieved the top 10 cross-file chunks and labeled them with a polarity—positive, neutral, or negative—using our likelihood-based metric, as detailed in Pseudo-code 2. To determine the thresholds for classifying positive and negative chunks, we experimented with various threshold settings to generate different sets of positive and negative chunks. The StarCoder-3B model's completion performance was then evaluated on the validation set using only positive chunks or using top-10 retrieved contexts excluding negative chunks, allowing us to identify the optimal threshold. The adaptive retrieval token in our model is designed to indicate whether the current in-file and cross-file contexts provide sufficient information for code completion. To ensure this condition is met when constructing the training data, we evaluated the Edit Similarity (ES) score between the in-file context and the retrieved positive cross-file chunks. Only instances where the ES score exceeded 0.5 were included in the dataset, ensuring that the retrieved contexts contributed meaningfully to the completion task. This filtering process guarantees that the final training instances contain sufficient and relevant information. After the filtering process, we finally got **43k** instances of labelling with more than **400k** cross-file chunks. The instance-level statistics are presented in Table 5, and the chunk-level polarity distributions are shown in Figure 5. We observe that positive chunks make up 20-30% of the retrieved chunks, while 10-20% of instances contain negative chunks. The remaining chunks are generally irrelevant to the completion task, reflecting a distribution similar to the RepoEval-API test set.

Algorithm 2: Cross-file chunks labeling

Input: Generator G, Retriever R, In-file contexts $C_{in} = (C_p, C_s)$, Cross-file code C_{out}, Window size w, stride size s, Polarity thresholds T_p, T, n

Output: Labeled Cross-file Chunk set $\hat{C}_{cc}$

$\hat{C}_{cc} \leftarrow \emptyset$

$Q \leftarrow C_p[-w:]$ `/* Query */`

$C_{out} \leftarrow$ chunklize cross-file contexts with w and s

$C_{cc} \leftarrow R(Q, C_{out})$ `/* Retrieve the top-10 cross-file contexts */`

$L(Y|c_{in}) = \sum_{t=1}^{T} logP(y_t|y_1, .., y_{t-1}, C_{in}; G)$ `/* Compute likelihood */`

foreach *chunk* $c_i \in C_{cc}$ **do**

 $L(Y|C_{in}, c_i) = \sum_{t=1}^{T} logP(y_t|y_1, .., y_{t-1}, C_{in}, c_i; G)$

 $S(c_i|C_{in}, Y) = \frac{L(Y|C_{in},c_i) - L(Y|C_{in})}{L(Y|C_{in})}$

 $Polarity(c_i|C_{in}, Y) \leftarrow (S(c_i|C_{in}, Y), T_p, T_n)$ `/* Polarity of` c_i `*/`

 $\hat{C}_{cc} \leftarrow Append(c_i, Polarity(c_i|C_{in}, Y))$

return $\hat{C}_{cc}$

B Implementation Details

Since our methods primarily focus on how the generator utilizes the retrieved contexts, we maintain the same setup for the retrieval process across all experiments. The details of truncation, query formation, and the retrieval procedure have been introduced in previous sections. In this section, we

Table 5: Training data statistics.

Completion Setting	Line		Chunk		Function		Total
	Left-to-right	Infilling	Left-to-right	Infilling	Left-to-right	Infilling	
Instances	6982	7056	7850	7746	7415	6561	43610

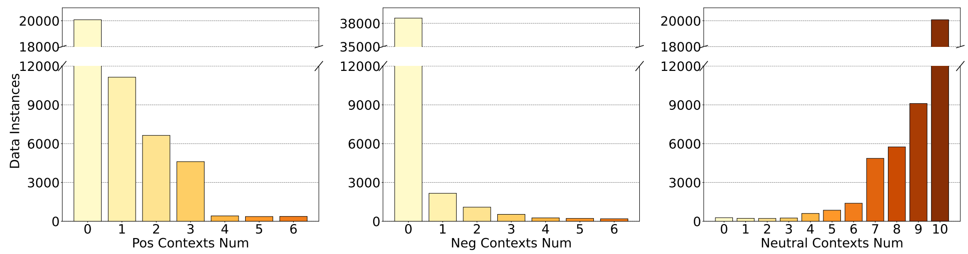


Figure 5: The statistics of chunk polarities of training data.

will outline the basic prompt structure used for code generation, followed by the implementation details of both the baseline models and our proposed methods.

Prompt Format. We adopt the same prompt format for both the baselines and our proposed methods to ensure consistency. Since our experiments are conducted in both the left-to-right and infilling settings, we standardize the prompt format using the fill-in-the-middle structure for fair comparison. Additionally, following the approach in (Zhang et al., 2023; Wu et al., 2024), we represent the cross-file contexts using both natural language descriptions and the token # to denote code snippets. To provide structural information, the original file path of each code chunk is also included. A typical chunk is verbalized as:

```
#Here are relevant code fragments from other files of the repo:
# -----
# the below code fragment can be found in:
# huggingface_diffusers/examples/dreambooth/train_dreambooth_lora.py
# -----
#    pipeline = DiffusionPipeline.from_pretrained(
#        args.pretrained_model_name_or_path, revision=args.revision,
#        torch_dtype=weight_dtype
#    )
#    pipeline.scheduler = ...
#    ...
```

Baseline Implementation. To ensure consistency across different baselines, we formatted prompts for the no-retrieval and full-retrieval settings based on the same retrieval setup and prompt structure. In the full-retrieval baseline, cross-file contexts are ordered by their Jaccard similarity score. For the adaptive retrieval baseline i.e., RepoFormer, due to the absence of an open-source dataset as well as its model, we trained RepoFormer on the same dataset used for REPOFILTER to ensure a fair comparison. RepoFormer is designed to determine whether retrieval is necessary for code completion. To achieve this, we adapted our dataset, labeling instances with no positive cross-file samples as *no retrieval needed*, while those with relevant cross-file contexts were labeled as *retrieval needed*. This resulted in 20,076 instances labeled as the no retrieval needed and 23,534 instances labeled as the retrieval needed. Additionally, we introduced the special token <MC> to indicate whether retrieval is required in the prompt for model training. RepoFormer was trained using the same data construction process and hyperparameter settings as REPOFILTER, ensuring that the comparison between the two models remains fair and consistent.

Table 6: Code completion performance when adopting UniXcoder as a retriever.

Completion Setting	RAG Strategy	Line		API		Function
		EM	ES	EM	ES	ES
Infilling	Full Retrieve	57.81	77.45	46.40	72.83	49.86
	RepoCoder	60.50	78.79	48.16	75.33	55.03
	REPOFILTER	60.88	79.53	49.09	77.62	53.62
	REPOFILTER + RepoCoder	**61.38**	**79.80**	**50.72**	**77.80**	**55.17**
Left-to-right	Full Retrieve	48.06	68.75	38.15	63.70	48.28
	RepoCoder	51.44	70.16	39.90	65.91	49.07
	REPOFILTER	51.00	69.52	40.34	67.48	48.90
	REPOFILTER + RepoCoder	**52.44**	**71.64**	**40.84**	**69.17**	**50.07**

C GENERALIZATION OF REPOFILTER ON OTHER RETRIEVERS

We employ sparse retrieval in our main experiments, as previous work has demonstrated that sparse retrievers perform well in repository-level code completion, offering comparable results to dense retrievers. However, since our methods primarily focus on filtering the retrieved contexts, they should, in principle, be generalized to different retrieval methods, regardless of whether they are sparse or dense.

C.1 UNIXCODER

To explore the generalizability of our approach to dense retrievers, which may produce different retrieved contexts, we experimented with UniXcoder (Guo et al., 2022) as the dense retriever. We leverage UniXCoder to embed each cross-file chunk as well as the query from the in-file preceding code chunk. Then the candidate chunks are retrieved by the cosine similarity between the embedded vector of cross-file chunks and the query chunk. We maintained all indexing and query settings unchanged to ensure consistency in evaluation. Our model was then tested on the RepoEval dataset under the left-to-right and infilling settings. The results in terms of reference-based metrics are shown in Table 6.

The results show that REPOFILTER consistently improves performance compared to full-retrieve, which aligns with our findings from the main experiments. This suggests that REPOFILTER can generalize to other retrieval methods. Furthermore, when comparing the results with those from the main experiments, we observe that using UniXcoder does not outperform sparse retrieval and slightly reduces efficiency. This outcome is consistent with previous research (Zhang et al., 2023; Ding et al., 2024a), and we hypothesize that in repository-level code completion, the preceding code may not always convey the necessary intent for completion. As a result, semantically similar chunks retrieved by dense methods may not always contribute meaningfully to the task. While sparse retrieval also does not specifically capture the intent behind the code, its token-level similarity can help retrieve useful chunks, particularly when similar API or function names are involved. However, this approach is still suboptimal. Future research should explore methods that extract the underlying intent of the incomplete code and retrieve truly relevant chunks based on that intent.

C.2 REPOCODER

In addition to conventional dense or sparse retrievers, some research has specifically designed retrieval frameworks tailored for repository-level code completion, aiming to retrieve more accurate and helpful cross-file contexts. A representative work in this area is RepoCoder (Zhang et al., 2023), which employs an iterative retrieval approach, incorporating the model's generated content into subsequent retrieval rounds. In our implementation, we kept all retrieval settings consistent with our main experiment and conducted two iterations of retrieval. The results for RepoCoder are also presented in the table 6, demonstrating consistent improvements over the one-time retrieval baseline. Building on this foundation, we further applied REPOFILTER to their framework to filter the final round of retrieval results. From the table, we observe that our method further enhances RepoCoder's performance, illustrating that our approach can generalize to other sophisticatedly designed retrieval frameworks.

D Discussions

This work focuses on analyzing and filtering retrieved cross-file contexts in the scenario of repository-level code completion. However, there are several limitations in our current study. For instance, our analysis and experiments are confined to Python and Python repositories, without exploring whether our labeling method and `REPOFILTER` can be extended to other programming languages. Future work will address this limitation by investigating and expanding to a broader range of languages. Additionally, most of our evaluations rely on reference-based metrics. In the context of code completion and generation, especially for longer completion targets such as chunks or functions, execution-based metrics, such as unit tests, provide a more accurate assessment of completion quality. Developing such benchmarks would establish a stronger foundation for further research in this domain. Beyond repository-level code completion, other tasks in the code domain, such as code generation and code repair, could potentially benefit from retrieval-augmented generation (RAG) to enhance model performance. Our methods can be extended to these tasks to analyze how retrieved code snippets influence the generated outputs. Notably, our likelihood-based metric for identifying the polarity of retrieved contexts is model-agnostic and does not impose strict requirements on data formats, such as being specific to repository-level completion or code-related tasks. This makes it promising for application in knowledge-intensive natural language tasks like question answering. Exploring whether this metric remains effective for more diverse natural languages is an intriguing direction for future research.

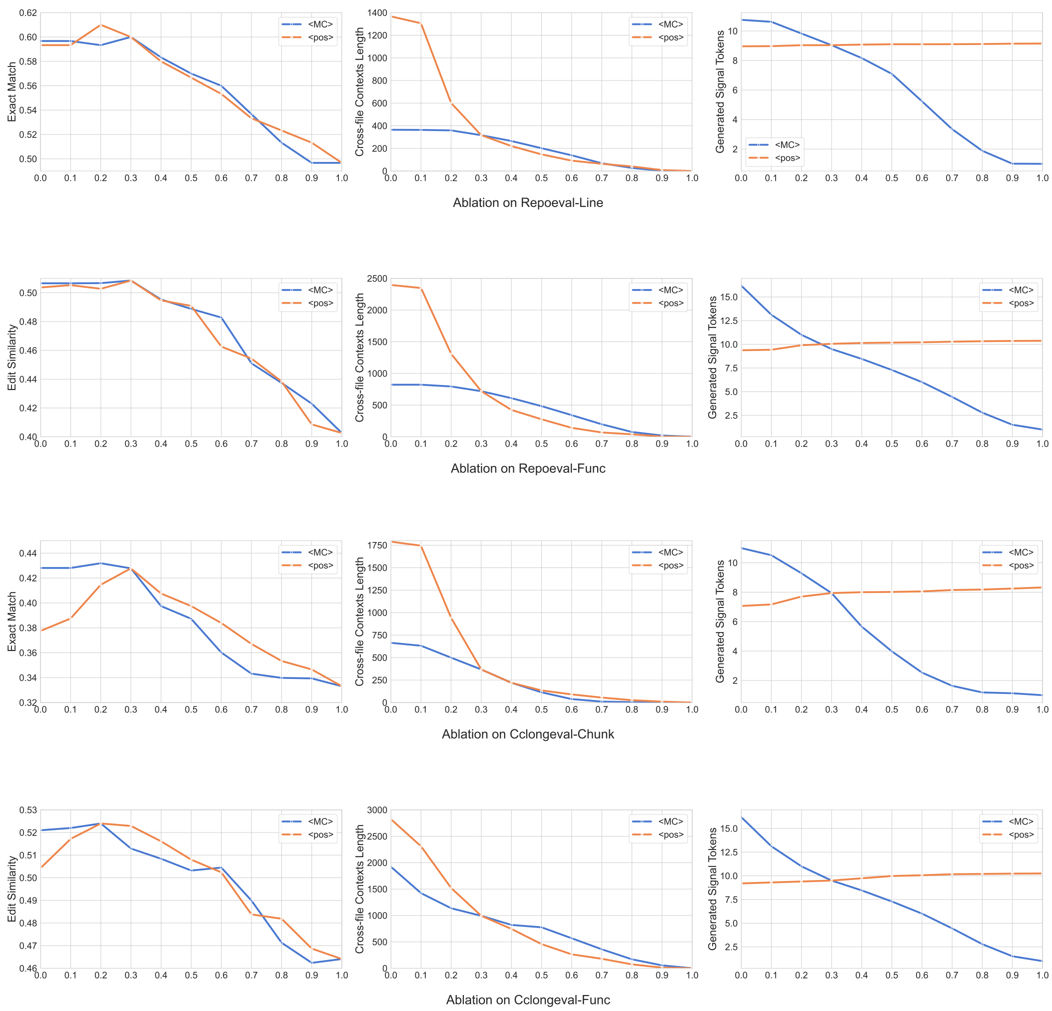


Figure 6: Ablations of threshold setting of signal tokens on extended tasks

E EXTENDED ABLATION STUDIES

In Section 6.5, we analyzed the impact of the threshold setting for signal tokens during the inference process on the model's completion performance, cross-file context length, and the number of additional tokens generated in the ReproEval-API completion task. To further explore whether this threshold setting affects these three aspects consistently across other datasets and assess its generalizability, we retained the same ablation setting and conducted ablation studies on the other four tasks from the main experiment. The results, shown in the figure 6, indicate a similar pattern across these tasks. Specifically, setting the threshold for the two signal tokens to 0.2–0.3 achieves a relative balance between efficiency and performance, yielding the optimal trade-off.

F Case Study

In this section, we present a case study to illustrate how our model determines the polarity of each chunk and utilizes filtering to facilitate correct code generation. In the example depicted below, the `<prefix>` contains a portion of the preceding code, while the `<suffix>` section shows the model initiating the retrieving process by first outputting `<MC>`. Subsequently, the model sequentially evaluates each chunk, outputting the corresponding polarity token at the end of each chunk (denoted by a special end-of-chunk token).

When a chunk is identified with `<pos>`, the model further evaluates the sufficiency of the context. Once the context is deemed sufficient, the model outputs `<EC>` and directly proceeds with code generation.

In this case study, we compare the generation results of the baseline with full-retrieval and our framework. Notably, our framework's output exactly matches the ground truth, whereas the baseline generates incorrect results. We attribute the baseline's failure to being misled by the first negative chunk, which involves interval computations and symbolic mappings. These suggest testing WildFunction in a more complex applied context, leading to incorrect guidance.

In contrast, the two chunks identified as ¡pos¿ provide critical support for the correct completion. One chunk shows how objects like Function are instantiated, directly informing the appropriate completion for WildFunction. Another chunk clearly explains the purpose and usage of WildFunction through its docstring, further reinforcing the correct completion.

This demonstrates how our framework effectively filters and prioritizes relevant contexts, enabling precise code generation. `<PREFIX>`

```python
...

def test_sympy__core__function__Application():
    from sympy.core.function import Application
    assert _test_args(Application(1, 2, 3))


def test_sympy__core__function__AppliedUndef():
    from sympy.core.function import AppliedUndef
    assert _test_args(AppliedUndef(1, 2, 3))


def test_sympy__core__function__Derivative():
    from sympy.core.function import Derivative
    assert _test_args(Derivative(2, x, y, 3))


@SKIP("abstract class")
def test_sympy__core__function__Function():
    pass


def test_sympy__core__function__Lambda():
    assert _test_args(Lambda((x, y), x + y + z))

```

```

def test_sympy__core__function__Subs():
    from sympy.core.function import Subs
    assert _test_args(Subs(x + y, x, 2))


def test_sympy__core__function__WildFunction():
```

<SUFFIX>

```
<mc>
# Here are some relevant code fragments from other files of the repo:
# -----
# The below code fragment can be found in Shardul555/sympy/sympy/solvers/ode/tests/test_single.py
_ode_solver_test(_get_examples_ode_sol_nth_linear_undetermined_coefficients)


def test_nth_order_reducible():
    from sympy.solvers.ode.ode import _nth_order_reducible_match

    F = lambda eq: _nth_order_reducible_match(eq, f(x))
    D = Derivative
    assert F(D(y*f(x), x, y) + D(f(x), x)) is None
    assert F(D(y*f(y), y, y) + D(f(y), y)) is None
    <End_of_Chunk><neu>

#####################################################
#####################################################
# -----
# The below code fragment can be found in Shardul555/sympy/sympy/solvers/ode/tests/test_single.py
@_add_example_keys
def _get_examples_ode_sol_almost_linear():
    from sympy import Ei
    A = Symbol('A', positive=True)
    f = Function('f')
    d = f(x).diff(x)

    return {
    <End_of_Chunk><pos><mc>
#####################################################
#####################################################
# -----
# The below code fragment can be found in Shardul555/sympy/sympy/sets/tests/test_sets.py
x = Symbol('x', real=True)
assert Interval(0, x).free_symbols == {x}


def test_image_interval():
    from sympy.core.numbers import Rational
    x = Symbol('x', real=True)
    a = Symbol('a', real=True)
    assert imageset(x, 2*x, Interval(-2, 1)) == Interval(-4, 2)
    assert imageset(x, 2*x, Interval(-2, 1, True, False)) == \
    <End_of_Chunk><neg>
#####################################################
#####################################################
# -----
# The below code fragment can be found in Shardul555/sympy/sympy/solvers/ode/tests/test_ode.py
def test_issue_4785():
    from sympy.abc import A
```

```
    eq = x + A*(x + diff(f(x), x) + f(x)) + diff
    (f(x), x) + f(x) + 2
    assert classify_ode(eq, f(x)) == ('1st_linear', 'almost_linear',
        '1st_power_series', 'lie_group',
        'nth_linear_constant_coeff_undetermined_coefficients',
        'nth_linear_constant_coeff_variation_of_parameters',
        '1st_linear_Integral', 'almost_linear_Integral',
        'nth_linear_constant_coeff_variation_of_parameters_Integral')
    # issue 4864
    <End_of_Chunk><neu>
#####################################################
#####################################################
# -----
# The below code fragment can be found in sympy/sympy/core/function.py
class WildFunction(Function, AtomicExpr):  # type: ignore
    """
    A WildFunction function matches any function (with its arguments).

    Examples
    ========

    >>> from sympy import WildFunction, Function, cos
    >>> from sympy.abc import x, y
    >>> F = WildFunction('F')
    >>> f = Function('f')
    """
    <End_of_Chunk><pos><ec>
```

REPOFILTER COMPLETION

```
def test_sympy__core__function__WildFunction():
    from sympy.core.function import WildFunction
    assert _test_args(WildFunction('f'))
```

BASELINE COMPLETION

```
def test_sympy__core__function__WildFunction():
    from sympy.core.function import WildFunction
    from sympy import Symbol, imageset, Interval
    x = Symbol('x', real=True)
    f = WildFunction('f')
    assert imageset(x, f(x), Interval(-2, 2)) == imageset(x, f(x),
        Interval(-2, 2))
```