RepoAttention: Focusing on Relevant Context for Repository-Level Code Completion

Anonymous ACL submission

Abstract

Repository-level code completion, which leverages the entire codebase to generate suggestions, is crucial for enhancing developer productivity. While retrieval-augmented generation (RAG) with code large language models (code LLMs) has become the standard approach for this task, existing methods still suffer from several problems: they struggle with uninformative query based on incomplete code, overlook the impact of prompt structure and snippet ordering, and retrieve textually similar but functionally different code. These issues collectively introduce noise and lead to ineffective context for code LLMs. To address these problems, we introduce RepoAttention, a training-free RAG-based framework for repository-level code completion. It improves retrieval and completion performance through dual-path query refinement, relevance-aware reranking of retrieved snippets, and a dynamic relevance-guided attention mechanism. Experiments on CCEval and RepoEval show that RepoAttention surpasses state-of-the-art methods by 23.9% in Exact Match accuracy and generalizes well across multiple code LLMs and programming languages.

1 Introduction

With the rapid development of code large language models (code LLMs) (Chen et al., 2021; Guo et al., 2024; Li et al., 2023; Roziere et al., 2023), code completion has become a pivotal capability integrated into modern programming environments, aiming to enhance developer productivity (Izadi et al., 2024; Mastropaolo et al., 2023; Nam et al., 2024; Tang et al., 2023; Xu et al., 2022). While contemporary models excel at local code synthesis within a single file, repository-level code completion remains challenging, as it requires leveraging cross-file dependencies and structural patterns across an entire codebase. This difficulty stems from the limited context window of code LLMs and the fact that naively incorporating entire repository is both computationally inefficient and prone to introducing irrelevant noise that interferes with completion. Therefore, recent research has adopted retrieval-augmented generation (RAG) frameworks (Yu et al., 2024; Liang et al., 2024; Liu et al., 2024a; Wang et al., 2025; Zhang et al., 2023, 2025a,b), which leverage the incomplete code within the current file as a query to retrieve relevant code snippets from the whole repository, providing cross-file context. These retrieved code snippets are then concatenated with the incomplete code to form the prompt for code LLMs. While these approaches have showed encouraging results, several problems remain in both the retrieval and completion stages.

P1 Deficient Query Representation. In most prior methods (Wu et al., 2025; Zhang et al., 2025a; Phan et al., 2025), the query to retrieve code snippets is simply the incomplete code within the current file, and the prompt is constructed by directly concatenating the query and the retrieved code snippets. RepoCoder (Zhang et al., 2023) uses an iterative retrieval method that builds new queries by combining intermediate completions with incomplete code, but incorrect generations may propagate to later iterations and multiple retrieval rounds greatly reduce efficiency. The absence of the correct code completion and the presence of noise in the query reduce the likelihood of retrieving truly helpful code snippets, and interfere with both the retrieval and completion stages.

P2 Prompt Order Sensitivity. Prior works (Jin et al., 2025; Jiang et al., 2024; Liu et al., 2024b) in natural language processing tasks have demonstrated that position of retrieved context in prompt significantly impacts model performance. However, existing code completion research (Wang et al., 2025; Zhang et al., 2025a) often overlooks the influence of code snippet ordering within the prompt on generation outcomes. Even when retrieved snippets contain useful information, the ar-

bitrary or suboptimal arrangement of these snippets can mislead the code LLMs, resulting in unsatisfactory completion quality.

P3 Similar Snippet Confusion. Many prior retrieval approaches tend to retrieve multiple code snippets that are highly similar to each other in textual form but differ in functionality (Wang et al., 2025; Liang et al., 2024). These similar code snippets can introduce confusion for code LLMs, potentially leading to inaccurate or semantically incorrect completions. The inability to effectively mitigate the interference from superficial textual similarity while preserving true functional relevance remains a major challenge in guiding the completion phase towards correct solutions.

To address these problems, we introduce RepoAttention, a novel RAG-based framework for repository-level code completion. First, we adopt a dual-path query refinement strategy that combines the raw incomplete code with a distilled, semantically enriched version to capture both lexical patterns and completion intent. We further improve retrieval quality by using diverse candidate completions to expand the queries for dense retrieval (addressing **P1**). Second, we develop a relevance-aware reranking mechanism that mimics how programmers prioritize auxiliary code according to its relationship with the completion task (addressing **P2**). Finally, during completion, we employ a dynamic relevance-guided attention mechanism to focus on the most pertinent code snippets while reducing interference from similar but functionally different code snippets (addressing **P3**).

We evaluate RepoAttention through extensive experiments using several representative code LLMs on CCEval (Ding et al., 2023) and RepoEval (Zhang et al., 2023) benchmarks. Experiment results show that RepoAttention achieves 23.9% improvement of Exact Match metric compared with state-of-the-art (SOTA) methods. Furthermore, RepoAttention demonstrates high generalizability, showing effectiveness across various code LLMs and programming languages.

The main contributions of this paper are:

- We introduce the RepoAttention framework for repository-level code completion. By adopting a dual-path query refinement strategy that combines semantic enrichment with diverse hypothetical completions, RepoAttention effectively addresses the problem of deficient query representation and bridges the semantic gap between incomplete code and the completion target.
- We develop a relevance-aware reranking mechanism that orders retrieved code snippets according to their functional relevance to the completion target. This approach reduces prompt order sensitivity and leads to more accurate completions.
- We propose a dynamic relevance-guided attention mechanism for the completion stage. This mechanism enables RepoAttention to robustly focus on the most pertinent code context, while mitigating the impact of textually similar but functionally different code snippets.
- A comprehensive evaluation shows that RepoAttention outperforms the SOTA approaches and consistently generalizes across different code LLMs and programming languages. To facilitate further research, we have made our source code publicly available[1].

2 Related Works

Code Large Language Models. LLMs have significantly advanced software engineering by automating tasks like code completion (Wang et al., 2023a). To better capture the properties of programming languages, recent work has focused on code LLMs. These models are either fine-tuned from general LLMs (Touvron et al., 2023) on code datasets (Wang et al., 2024c; Luo et al., 2024) or trained from scratch on vast amounts of source code (Nijkamp et al., 2023; Wang et al., 2023b). Recent mainstream code LLMs, such as Code Llama series (Roziere et al., 2023), Code-Qwen series (Hui et al., 2024), and DeepSeek-Coder series (Guo et al., 2024), are typically trained on massive, diverse code corpora spanning multiple programming languages, primarily utilizing self-supervised objectives like next-token prediction, sometimes augmented with fill-in-the-middle objectives (Bavarian et al., 2022) or repository-level context modeling to improve code completion performance.

Repository-level Code Completion. Repository-level code completion has attracted growing attention in recent years. RLCoder (Wang et al., 2025) introduces reinforcement learning to optimize RAG, training retrievers without labeled data. STALL++ (Liu et al., 2024a), A^3-CodGen (Liao et al., 2024), RepoFuse (Liang et al., 2024), and CodePlan (Bairi et al., 2024) leverage static analysis to obtain relevant retrieved code. DRACO (Cheng et al., 2024), RepoHyper (Phan et al., 2025),

[1] https://anonymous.4open.science/r/RepoAttention-38EE/

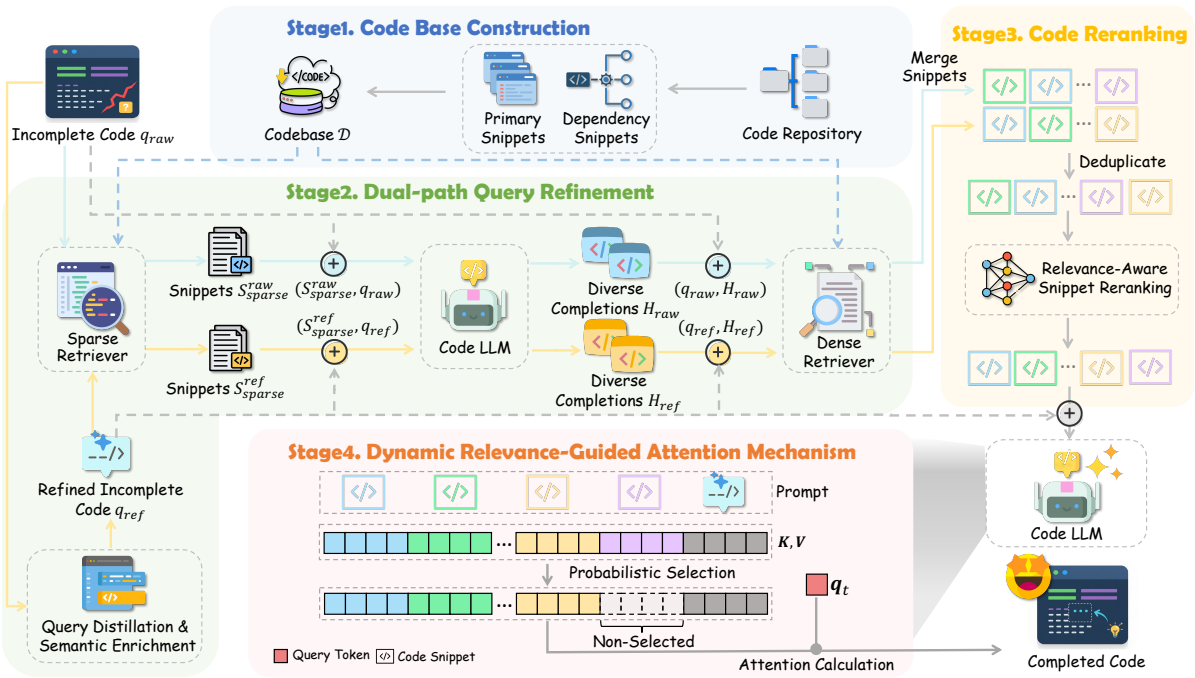


Figure 1: Overview of RepoAttention.

and CoCoMIC (Ding et al., 2024) utilize dependency analysis to retrieve relevant code snippets. RRG (Gao et al., 2024) and HCP (Zhang et al., 2025a) refine the raw retrieved code through transformation or pruning. CodeRAG (Zhang et al., 2025b) optimizes code selection by integrating multi-path retrieval and preference-aligned reranking. RepoGenReflex (Wang et al., 2024b), De-Hallucinator (Eghbali and Pradel, 2024), and RepoCoder (Zhang et al., 2023) employ iterative retrieval and generation using feedback. Repoformer (Wu et al., 2024) improves efficiency through selective retrieval, while ProCC (Tan et al., 2024) combines multiple retrieval results to better utilize context. ToolGen (Wang et al., 2024a) and CodeAgent (Zhang et al., 2024) leverage external tools like agents to tackle code completion tasks. GraphCoder (Liu et al., 2024c) and ContextModule (Guan et al., 2024) construct graphs to assist retrieval but incur high construction costs and face challenges in formulating accurate graph queries. Despite their effectiveness, these approaches do not fully solve the problems described in Section 1.

3 Preliminary Studies

To further investigate the challenges of repository-level code completion, we conduct a series of preliminary experiments under the setup detailed in Section 5.1. The results highlight two key factors that motivate our proposed approach: (1) the content of the query is essential for identifying useful information, given that enriching the incomplete code with predicted sequences can help bridge the semantic gap and facilitate the retrieval of more relevant cross-file context; (2) the ordering of retrieved code snippets in the prompt plays a vital role in model performance, as inconsistent or random placement leads to a noticeable decrease in completion accuracy. These findings highlight the importance of both query refinement and prompt structuring. Detailed experimental results and comprehensive analysis are provided in Appendix A.

4 Methodology

As shown in Figure 1, RepoAttention involves four parts: codebase construction, dual-path query refinement, relevance-aware snippet reranking and dynamic relevance-guided attention mechanism.

4.1 Codebase Construction

Constructing the repository-level codebase involves transforming raw source code into structured snippets to facilitate effective retrieval. Traditional fixed-window slicing approaches (Zhang et al., 2023; Guan et al., 2024) often compromise code continuity and structural integrity, while dependency-parsing methods (Liu et al., 2024a,c) capture limited context and struggle with complex repository dependencies. To address these limitations, we construct a retrieval codebase composed of primary and dependency code snippets.

For primary code snippets, we align with typical

Figure 2: An example of the hybrid aggregation process for dependency code snippet construction.

programming practices by initially segmenting the repository code based on blank lines (Wang et al., 2025). Subsequently, we merge consecutive short code fragments into primary code snippets, ensuring each snippet maintains an appropriate length and improved semantic coherence.

For dependency code snippets, we utilize tree-sitter[2] to parse import statements from in-file context to identify intra-repository dependencies. We then access the referenced files to extract only the signatures of classes, methods, and functions. This process captures critical semantic details regarding class inheritance and API interplay (Zhang et al., 2025a). Distinct from simply listing these signatures, we adopt a hybrid aggregation strategy: the dependency context for each imported class is encapsulated as a standalone snippet, whereas the signatures of all imported functions and methods are consolidated into a single unified snippet. Figure 2 provides an example of this strategy. The final codebase is constructed by integrating these primary and dependency code snippets.

4.2 Dual-Path Query Refinement

Repository-level code completion relies on retrieving snippets that are semantically aligned with the intended ground truth. However, directly using incomplete code as the retrieval query introduces two issues. First, incomplete code often contains noise, which refers to code elements that are weakly related to the functional semantics required for completion. Such noise can weaken semantic signals and mislead both the retriever and the code LLM. Second, the retrieval process is inherently biased toward the query rather than the actual completion target, resulting in snippets similar to the query but irrelevant to the completion target. To overcome these issues, we introduce a dual-path query refinement strategy, which consists of two steps: (1) query distillation and semantic enrichment, and (2) retrieval with hypothetical completions.

Query Distillation and Semantic Enrichment. As shown in Appendix E, for the first path, we retain the original incomplete code c_{in} as the raw query q_{raw} to preserve exact lexical patterns. For the second path, we construct a refined query q_{ref} by applying query distillation and semantic enrichment to c_{in}. Specifically, we remove the following elements from c_{in} that are likely to introduce noise:

- **Import statements of unused modules** within the context near the completion point. These statements increase prompt length without contributing useful information, potentially distracting the retriever or model.
- **Print and logging statements**, which typically serve debugging purposes and are unrelated to the functional semantics required for completion.
- **Irrelevant variable definitions and statements** that are not semantically or data-flow related to the completion point. Only statements with strong relevance should be preserved.

To replicate the semantic clarity provided by programmer comments, we generate descriptive comment when it is missing. We utilize a lightweight LLM to analyze surrounding identifiers (e.g., function names) and generate a concise natural language comment describing the intended behavior. This comment is inserted before the completion point. By applying these modifications, q_{ref} becomes structurally concise while highlighting information vital for code completion. We use Qwen3-0.6B (Yang et al., 2025) for comment generation.

Retrieval with Hypothetical Completions. To capture both explicit keyword matches and implicit semantic relationships, we first use BM25 as the sparse retriever for both queries to independently retrieve code snippets from the codebase $\mathcal{D}$. This yields two initial sets of code snippets, S^{raw}_{sparse} and S^{ref}_{sparse}. These snippets may include both primary

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

and dependency code snippets, offering information that ranges from basic code attributes to semantic details of class structures and API interplay. We construct two distinct prompts by prepending the sparse retrieval results to their respective queries and input them into the code LLM $\mathcal{M}$ to generate hypothetical completions. For each path, we sample m diverse completions:

$$H_{raw} = \{h_1, h_2, \ldots, h_m\} \sim P_{\mathcal{M}}(\cdot|S^{raw}_{sparse} \oplus q_{raw}), \quad (1)$$

$$H_{ref} = \{h'_1, h'_2, \ldots, h'_m\} \sim P_{\mathcal{M}}(\cdot|S^{ref}_{sparse} \oplus q_{ref}), \quad (2)$$

where $\oplus$ denotes concatenation. Although these completions may not be correct, they can serve as candidate completions for understanding the likely functionality and form of the correct completion.

By appending the generated completions to their corresponding queries, we form expanded queries that explicitly represent the potential completion target. Both expanded queries are then input into a dense retriever to independently retrieve the top-k most relevant code snippets from the codebase $\mathcal{D}$. This process can be formalized as:

$$S^{raw}_{dense} = \text{DenseRetrieve}(q_{raw} \oplus H_{raw}, \mathcal{D}) = \{s_1, s_2, \ldots, s_k\}, \quad (3)$$

$$S^{ref}_{dense} = \text{DenseRetrieve}(q_{ref} \oplus H_{ref}, \mathcal{D}) = \{s'_1, s'_2, \ldots, s'_k\}. \quad (4)$$

4.3 Relevance-Aware Snippet Reranking

Although the previous stage attempts to include all potentially relevant snippets, it does not assess the degree of relevance between each snippet and the query. Meanwhile, as shown in Section A.4, the position of retrieved code snippets within the prompt notably impacts completion quality. For most code LLMs, placing more relevant cross-file snippets either earlier or later in the prompt tends to improve completion accuracy (Jiang et al., 2024; Liu et al., 2024b). Therefore, we perform a reranking process to evaluate snippet relevance and arrange them in an optimal order within the prompt.

Our relevance-aware reranking strategy is inspired by the typical workflow of human programmers during code completion. When completing code, programmers primarily rely on the local context near the completion point to infer the intended functionality. Code snippets that are more similar to this local context in terms of functionality and semantics are typically more helpful.

Figure 3: An example where similar code snippets within the retrieved valuable context introduce noise and result in incorrect code completion.

This intuition suggests that focusing on code lines near the completion point is better than treating the entire query uniformly. Concretely, we extract the last t lines before the completion point of q_{ref} as the *key code* segment c_k and use a lightweight reranker to compute a relevance score r_i between c_k and each retrieved code snippet. Higher-scored snippets are placed earlier in the prompt. This process is formalized as:

$$C_v = \text{Rerank}(c_k, S^{raw}_{dense} \cup S^{ref}_{dense}), \quad (5)$$

where C_v denotes the final retrieved valuable context, and r_i denotes the relevance score of the i-th snippet. We use Qwen3-Reranker-0.6B (Zhang et al., 2025c) as the reranker.

4.4 Dynamic Relevance-Guided Attention Mechanism

With the integration of the methods introduced above, the retrieved valuable context C_v is generally sufficient to guide the code LLM toward generating the correct completion. Nevertheless, in a non-trivial number of cases, C_v contains code snippets that are textually similar yet functionally

different, which can introduce noise and lead to incorrect completions, as examplified in Figure 3.

To mitigate this issue, it is insufficient to simply feed the retrieved results directly into the code LLM. Instead, we propose that the model dynamically adjust its reliance on different parts of the retrieved context during the generation phase.

To this end, we introduce Dynamic Relevance-Guided Attention (DRGA), which extends the standard Transformer model by dynamically selecting relevant code snippets for attention computation. By replacing the traditional attention mechanism (Vaswani et al., 2017) with DRGA, the code LLM can focus more on the relevant snippets and ignore distractors, leading to more efficient and effective reasoning during code generation.

Unlike traditional attention mechanisms, where each query token attends to the entire context, DRGA restricts each query token to attend only to a subset of selected keys and values:

$$\mathrm{DRGA}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{Softmax}\left(\boldsymbol{q}\boldsymbol{K}[E]^\top\right)\boldsymbol{V}[E], \tag{6}$$

where $\boldsymbol{q} \in \mathbb{R}^{1\times d}$ denotes a single query token, and $\boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N\times d}$ represent the keys and values over N tokens, respectively. $E \subseteq [N]$ is the set of selected key and values. The dimension d corresponds to the size of a single attention head. Multi-head attention extends this formulation by computing multiple such heads in parallel and concatenating their outputs.

In our code completion task, the prompt provided to the code LLM represents the entire context, consisting of the refined query q_{ref} and the retrieved valuable context C_v, which includes multiple code snippets. We enable each query token to attend only to a subset of tokens from q_{ref} and the most relevant code snippets in C_v as follows:

$$E = \left(\bigcup_{r_i \geq I} \{T_i, T_q\}\right) \cup \left(\bigcup_{r_i < I} \mathrm{Select}(T_i)\right), \tag{7}$$

where T_i denotes the tokens of the i-th code snippet in C_v, T_q denotes the tokens of q_{ref}, and r_i is the relevance score of the i-th code snippet and c_k (defined in Section 4.3), which quantifies the relevance between q_{ref} and the i-th code snippet. I denotes the relevance threshold. The operation $\mathrm{Select}(T_i)$ refers to the process of proportional mapping of r_i to a corresponding probability, and then decides whether to retain T_i based on that probability.

5 Experiment

5.1 Experimental Setup

Benchmarks. We utilize two widely adopted benchmarks for evaluation: CCEval (Ding et al., 2023) and RepoEval (Zhang et al., 2023). CCEval is a benchmark for repository-level code completion across multiple programming languages, reflecting real-world development scenarios. RepoEval evaluates line-level, API-level, and function-level completion on full-scale repositories. In our experiments, we use the Python and Java subsets of CCEval, which contain 2,665 and 2,139 test cases respectively, as these languages are widely used and provide sufficient diversity. For RepoEval, we focus on line-level and API-level tasks, each with 1,600 test cases, since our work targets code completion rather than function-level code generation.

Metrics. The completion results were evaluated using two widely adopted metrics: ***Exact Match (EM)*** and ***Edit Similarity (ES)*** (Levenshtein et al., 1966). Specifically, EM measures whether the predicted code completion exactly matches the correct code without any differences. ES evaluates the textual similarity between the predicted and correct code completions based on the edit distance.

Baselines. To evaluate the effectiveness of RepoAttention, we compared it with the simple RAG method (Parvez et al., 2021) and the SOTA repository-level code completion approaches:

- **Simple RAG**. In repository-level code completion, simple RAG is a basic retrieval and generation baseline. It uses the left context of incomplete code as the query, retrieves relevant snippets from the repository with BM25, and concatenates them with the in-file context to form the prompt.
- **HCP** (Zhang et al., 2025a). HCP constructs high-quality code completion prompts by modeling the code repository at the function level, retaining topological dependencies between files while pruning irrelevant content from the retrieved code, without requiring model training.
- **RLCoder** (Wang et al., 2025). RLCoder employs a reinforcement learning framework to train a retriever without labeled data, iteratively evaluating and retrieving useful code snippets based on the perplexity of the target code.
- **CodeRAG** (Zhang et al., 2025b). CodeRAG utilizes a multi-path retrieval strategy followed by

Table 1: Repository-level code completion performance comparison.

Code LLM	Method	CCEval (Python)		CCEval (Java)		RepoEval (Line)		RepoEval (API)	
		EM	ES	EM	ES	EM	ES	EM	ES
StarCoderBase-7B	Simple RAG	14.78	66.30	16.36	66.41	44.19	65.48	32.69	55.91
	HCP	19.87	68.75	/	/	/	/	/	/
	CodeRAG	24.01	60.89	/	/	/	/	/	/
	RLCoder	25.89	72.12	25.81	68.91	47.75	68.48	35.06	58.09
	RepoAttention	**31.52**$^{\uparrow 21.7\%}$	**75.53**$^{\uparrow 4.7\%}$	**28.63**$^{\uparrow 10.9\%}$	**70.21**$^{\uparrow 1.9\%}$	**49.11**$^{\uparrow 2.8\%}$	**68.89**$^{\uparrow 0.6\%}$	**36.68**$^{\uparrow 4.6\%}$	**60.24**$^{\uparrow 3.7\%}$
CodeLlama-7B	Simple RAG	15.76	66.30	16.78	65.37	44.25	65.65	37.00	63.43
	HCP	20.06	68.18	/	/	/	/	/	/
	CodeRAG	23.11	60.44	/	/	/	/	/	/
	RLCoder	26.57	71.43	26.23	66.37	46.62	67.88	37.94	64.32
	RepoAttention	**32.13**$^{\uparrow 20.9\%}$	**75.38**$^{\uparrow 5.5\%}$	**29.56**$^{\uparrow 12.7\%}$	**68.79**$^{\uparrow 3.6\%}$	**48.08**$^{\uparrow 3.1\%}$	**68.25**$^{\uparrow 0.5\%}$	**40.12**$^{\uparrow 5.7\%}$	**66.17**$^{\uparrow 2.9\%}$
DeepSeekCoder-1.3B	Simple RAG	14.18	65.12	13.70	61.63	41.31	63.60	35.31	62.07
	HCP	17.95	66.77	/	/	/	/	/	/
	CodeRAG	21.24	58.78	/	/	/	/	/	/
	RLCoder	23.98	70.44	20.80	63.39	44.12	66.46	36.06	62.72
	RepoAttention	**29.71**$^{\uparrow 23.9\%}$	**74.24**$^{\uparrow 5.4\%}$	**24.28**$^{\uparrow 16.7\%}$	**65.13**$^{\uparrow 2.7\%}$	**45.78**$^{\uparrow 3.8\%}$	**67.04**$^{\uparrow 0.9\%}$	**37.87**$^{\uparrow 5.0\%}$	**64.81**$^{\uparrow 3.3\%}$
DeepSeekCoder-6.7B	Simple RAG	18.31	68.38	17.48	65.23	45.94	66.68	38.25	64.99
	HCP	22.62	70.49	/	/	/	/	/	/
	CodeRAG	25.85	62.70	/	/	/	/	/	/
	RLCoder	30.24	73.58	26.13	66.11	48.75	69.42	39.88	66.24
	RepoAttention	**36.78**$^{\uparrow 21.6\%}$	**78.19**$^{\uparrow 6.3\%}$	**29.09**$^{\uparrow 11.3\%}$	**68.78**$^{\uparrow 4.0\%}$	**50.63**$^{\uparrow 3.9\%}$	**70.48**$^{\uparrow 1.5\%}$	**42.64**$^{\uparrow 6.9\%}$	**68.34**$^{\uparrow 3.2\%}$

Table 2: Ablation study results on CCEval and RepoEval.

Method	CCEval (Python)		CCEval (Java)		RepoEval (Line)		RepoEval (API)	
	EM	ES	EM	ES	EM	ES	EM	ES
RepoAttention	36.78	78.19	29.09	68.78	50.63	70.48	42.64	68.34
w/o QR	35.13$_{\downarrow 4.5\%}$	77.42$_{\downarrow 1.0\%}$	28.31$_{\downarrow 2.7\%}$	67.43$_{\downarrow 2.0\%}$	50.26$_{\downarrow 0.7\%}$	70.28$_{\downarrow 0.3\%}$	41.62$_{\downarrow 2.4\%}$	67.57$_{\downarrow 1.1\%}$
w/o RO	35.88$_{\downarrow 2.4\%}$	77.76$_{\downarrow 0.5\%}$	28.91$_{\downarrow 0.6\%}$	68.48$_{\downarrow 0.4\%}$	50.18$_{\downarrow 0.9\%}$	70.25$_{\downarrow 0.3\%}$	41.39$_{\downarrow 2.9\%}$	67.41$_{\downarrow 1.4\%}$
w/o DA	34.56$_{\downarrow 6.0\%}$	77.16$_{\downarrow 1.3\%}$	28.14$_{\downarrow 3.3\%}$	67.19$_{\downarrow 2.3\%}$	50.06$_{\downarrow 1.1\%}$	70.19$_{\downarrow 0.4\%}$	41.02$_{\downarrow 3.8\%}$	67.13$_{\downarrow 1.8\%}$

a preference-aligned reranking mechanism to select necessary code snippets.

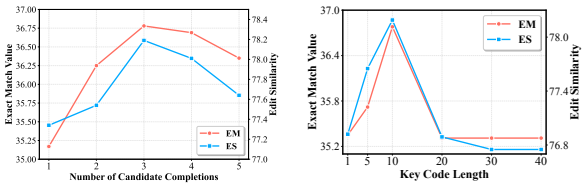
Experimental Details. To prevent potential data leakage, we select four popular open-source code LLMs that were released prior to the collection of benchmarks for our experiments: StarCoderBase-7B (Li et al., 2023), CodeLlama-7B (Roziere et al., 2023), DeepSeekCoder-1.3B (Guo et al., 2024), and DeepSeekCoder-6.7B. We employ the popular dense retriever RLRetriever (Wang et al., 2025), and set the `max_new_tokens` to 64, `max_context_tokens` to 2048 for all models. We set m to 3 in Section 4.2 and t to 10 in Section 4.3. Following the vLLM (Kwon et al., 2023) documentation, we set the temperature to 0.8 and top-p to 0.95 by default during sampling. All experiments were conducted on NVIDIA A100 GPUs.

5.2 Evaluation Results

Effectiveness of RepoAttention. Table 1 presents the repository-level code completion performance of RepoAttention in comparison to the baselines. Note that both HCP and CodeRAG are primarily designed for Python programs on CCEval, with no available adaptation for Java or evaluation on RepoEval. It can be observed that RepoAttention consistently outperforms all baseline methods across different code LLMs and benchmarks, demonstrating the robustness of RepoAttention for repository-level code completion. The largest improvement is observed with DeepSeekCoder-1.3B, where RepoAttention achieves a 23.9% gain in EM on CCEval Python compared to RLCoder.

Compared with HCP, which is also a training-free method, RepoAttention shows substantially stronger performance. With DeepSeekCoder-6.7B, RepoAttention improves EM by 62.6% on CCEval Python and yields consistent ES improvements across all benchmarks. These results indicate that RepoAttention can more effectively focus on relevant code snippets and enhance completion quality without additional training.

Ablation Study. To understand the contributions of each component in RepoAttention, we conduct ablation studies with DeepSeekCoder-6.7B. The results are shown in Table 2. "w/o QR" means



(a) Performance on different candidate completion counts.

(b) Performance on different key code lengths.

Figure 4: Performance under different hyperparameter settings in query refinement and reranking.

adopting a single-path retrieval setting that uses the incomplete code as the query, without query refinement or candidate completion generation. "w/o RO" means not performing reranking operations on the retrieved code snippets. "w/o DA" means using the traditional attention mechanism instead of our dynamic relevance-guided attention mechanism. We observe that removing any component leads to performance degradation, showing that each component contributes to the effectiveness of RepoAttention. Notably, the performance drops the most significantly for "w/o DA", with EM decreasing by up to 6.0% on CCEval Python, indicating that the dynamic relevance-guided attention mechanism plays a critical role in RepoAttention. This highlights the importance of prioritizing relevant code snippets while discarding irrelevant or noisy information that may interfere with completion quality. Furthermore, the performance decrease in "w/o QR" and "w/o RO" suggests that both query refinement and snippet reranking consistently improve retrieval quality. These findings collectively confirm that this multi-stage approach is vital for effective repository-level code completion.

Impact of Candidate Completion Count. We evaluate RepoAttention on CCEval Python using DeepSeekCoder-6.7B. Figure 4(a) shows how performance varies with different numbers of candidate completions. As the number of candidate completions increases, RepoAttention initially achieves consistent improvements across all metrics. However, when this number exceeds three, performance starts to degrade. This suggests that when the number of candidate completions becomes excessive, there is an increased likelihood that some candidate completions are irrelevant to the correct completion, thereby introducing noise into the retrieval process and consequently degrading the quality of the final completion. Therefore, we adopt three candidate completions as the default setting.

Impact of Key Code Length. We further study the sensitivity of the key code length t. Figure 4(b) shows that both EM and ES peak at $t = 10$, which correlates precisely with the average size of immediate logical context in the incomplete code. When $t = 1$, the key code is too short to provide enough clues about the intended completion. As t increases beyond 10, performance drops and then becomes stable. This suggests that a longer key code starts to include more useless lines and weakens the focus of reranking. Therefore, we use $t = 10$ as the default setting. Additional sensitivity analyses on context lengths are provided in Appendix D.

Computational Cost. Table 7 reports the average cost per case on CCEval Python. RepoAttention achieves the lowest latency, outperforming other baselines. Although CodeRAG incurs higher latency due to its multi-path retrieval and probing process, RepoAttention remains efficient despite using a dual-path refinement strategy. These results indicate that RepoAttention achieves a good balance between retrieval quality and computational cost, making it suitable for practical code completion. More details are provided in Appendix B.

Case Study. We present a case study to illustrate the effectiveness of RepoAttention. Figure 5 shows an example where the model is required to complete the function `pred_topk_with_label` in the `CleanSpeechDetector` class. RepoAttention first produces a refined query that supports useful candidate sampling. It then reranks the retrieved snippets, placing the most relevant one containing the target function at the top of the context, which enables correct completion. In contrast, baselines such as CodeRAG, HCP, and RLCoder retrieve similar but irrelevant snippets, resulting in incorrect outputs. More details are provided in Appendix C.

6 Conclusion

In this paper, we presented RepoAttention, a robust framework for repository-level code completion. Our approach addresses the limitations of existing methods through three key innovations: a dual-path query refinement strategy to bridge the semantic gap between incomplete code and retrieval targets, a relevance-aware reranking mechanism to optimize context arrangement, and a dynamic relevance-guided attention mechanism that allows the model to focus on pertinent information while ignoring noise. Comprehensive experiments show that RepoAttention significantly outperforms SOTA methods and generalizes strongly across different programming languages and code LLMs.

Limitations

In this section, we discuss the current limitations in RepoAttention.

Hyperparameter Sensitivity. As indicated in our experimental setup and analysis, our framework involves several hyperparameters that require tuning, such as the sampling temperature and the number of retrieved code snippets. Identifying the optimal combination of these parameters for different programming languages or base models can be laborious and computationally expensive.

Evaluation Metrics and Human Utility. Our evaluation mainly relies on automatic metrics such as EM and ES. While these metrics are widely used, they may not fully reflect developers' subjective judgments of code correctness or usefulness in real scenarios. Future work could include human evaluation, user studies, or downstream task-based evaluations to better assess the practical effectiveness of RepoAttention.

References

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7957–7977.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and 1 others. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. 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)*, pages 3433–3445.

Aryaz Eghbali and Michael Pradel. 2024. De-hallucinator: Mitigating llm hallucinations in code generation tasks via iterative grounding. *arXiv preprint arXiv:2401.01701*.

Xinyu Gao, Yun Xiong, Deze Wang, Zhenhan Guan, Zejian Shi, Haofen Wang, and Shanshan Li. 2024. Preference-guided refactored tuning for retrieval augmented code generation. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 65–77. IEEE.

Zhanming Guan, Junlin Liu, Jierui Liu, Chao Peng, Dexin Liu, Ningyuan Sun, Bo Jiang, Wenchao Li, Jie Liu, and Hang Zhu. 2024. Contextmodule: Improving code completion via repository-level contextual information. *arXiv preprint arXiv:2412.08063*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–13.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1658–1677, Bangkok, Thailand. Association for Computational Linguistics.

Bowen Jin, Jinsung Yoon, Jiawei Han, and Sercan O Arik. 2025. Long-context llms meet rag: Overcoming challenges for long inputs in rag. In *The Thirteenth International Conference on Learning Representations*.

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

Vladimir I Levenshtein and 1 others. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Hongwei Chen, Chengpeng Wang, Gang Fan, and 1 others. 2024. Repofuse: Repository-level code completion with fused dual context. *arXiv preprint arXiv:2402.14323*.

Dianshu Liao, Shidong Pan, Xiaoyu Sun, Xiaoxue Ren, Qing Huang, Zhenchang Xing, Huan Jin, and Qinying Li. 2024. A 3-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware. *IEEE Transactions on Software Engineering*.

Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. 2024a. Stall+: Boosting llm-based repository-level code completion with static analysis. *arXiv preprint arXiv:2406.10018*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024b. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

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

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, volume 2024, pages 27168–27188.

Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on github copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2149–2160.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–13.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734.

Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. 2025. Repohyper: Search-expand-refine on semantic graphs for repository-level code completion. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 14–25. IEEE.

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Stephen E Robertson and Steve Walker. 1994. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR'94: Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval, organised by Dublin City University*, pages 232–241. Springer.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *ACM Transactions on Software Engineering and Methodology*.

Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain adaptive code completion via language models and decoupled domain databases. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 421–433. IEEE.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, and 1 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.

Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023a. Practitioners' expectations on code completion. *arXiv preprint arXiv:2301.03846*.

Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024a. Teaching code llms to use autocompletion tools in repository-level code generation. *ACM Transactions on Software Engineering and Methodology*.

Jicheng Wang, Yifeng He, and Hao Chen. 2024b. Repogenreflex: Enhancing repository-level code completion with verbal reinforcement and retrieval-augmented generation. *arXiv preprint arXiv:2409.13122*.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025. Rlcoder: Reinforcement learning for repository-level code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1140–1152. IEEE.

Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Weiran Xu, Jingang Wang, Mengdi Zhang, and Xunliang Cai. 2024c. Dolphcoder: Echo-locating code large language models with diverse and multi-objective instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4706–4721.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: selective retrieval for repository-level code completion. In *Proceedings of the 41st International Conference on Machine Learning*, pages 53270–53290.

Peiyang Wu, Nan Guo, Junliang Lv, Xiao Xiao, and Xiaochun Ye. 2025. Rtlrepocoder: Repository-level rtl code completion through the combination of fine-tuning and retrieval augmentation. *arXiv preprint arXiv:2504.08862*.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–47.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Xinran Yu, Chun Li, Minxue Pan, and Xuandong Li. 2024. Droidcoder: Enhanced android code completion with context-enriched retrieval-augmented generation. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 681–693. IEEE.

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

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658.

Lei Zhang, Yunshui Li, Jiaming Li, Xiaobo Xia, Jiaxi Yang, Run Luo, Minzheng Wang, Longze Chen, Junhao Liu, Qiang Qu, and 1 others. 2025a. Hierarchical context pruning: Optimizing real-world code completion with repository-level pretrained code llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25886–25894.

Sheng Zhang, Yifan Ding, Shuquan Lian, Shun Song, and Hui Li. 2025b. Coderag: Finding relevant and necessary knowledge for retrieval-augmented repository-level code completion. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 23289–23299.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, and 1 others. 2025c. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*.

A Preliminary Studies

A.1 Setup

To prevent potential data leakage, we selected four popular open-source code LLMs that were released prior to the collection of benchmark datasets for our experiments: StarCoderBase-7B (Li et al., 2023), CodeLlama-7B (Roziere et al., 2023), DeepSeekCoder-1.3B (Guo et al., 2024), and DeepSeekCoder-6.7B. We adopt the default greedy decoding strategy without sampling, and set the `max_new_tokens` to 64, `max_context_tokens` to 2048 for all models. All experiments were conducted on NVIDIA A100 GPUs.

A.2 Baseline Evaluation

A.2.1 In-file Context Only

We first provided code LLMs with in-file context, specifically the left context of the incomplete code without any cross-file context, so that the models could rely solely on their own prior knowledge to perform the completion task. The results are presented in Table 3. The overall performance of the completion was unsatisfactory. Even the top-performing model demonstrated a limited accuracy, reaching just 9.46% on CCEval Python. This suggests that, without access to broader cross-file context, code LLMs struggle to accurately complete code that depends on definitions or information residing outside the current file.

A.2.2 Simple RAG

RAG is a widely adopted strategy for enhancing LLM-based repository-level code completion. In the preliminary study, we employ a simple RAG approach in which the left context of the incomplete code serves as a query to retrieve relevant code snippets from the repository. These retrieved snippets are then concatenated with the in-file context to construct the prompt. Following the setup of RLCoder (Wang et al., 2025) and reflecting typical programming practices, we segment the repository code into code snippets based on blank lines, and merge consecutive short code fragments to ensure each snippet has an appropriate length and improved semantic coherence. We utilize BM25 (Robertson et al., 2009; Robertson and Walker, 1994) as a sparse similarity metric and also employ the popular dense retriever RLRetriever (Wang et al., 2025) to retrieve relevant code snippets. The results are shown in Table 3. It can be observed that code snippets retrieved from other files within the repository contribute to more accurate code completion, as they provide relevant contextual information for the completion task.

A.3 Query Content Analysis

In repository-level code completion tasks, the query provided to the retriever is typically an incomplete code segment. Due to the absence of the correct completion or similar code, the retrieved snippets are often limited in utility, frequently containing redundant or irrelevant content and lacking critical information associated with the desired completion. To address this problem, the query should be modified in a specific way. For example, by inputting the original query into a code LLM to generate multiple lines of completed code and appending these lines to the original query, a refined query can be constructed. In this preliminary analysis, we appended only the first generated line to the original query, as the initial line typically represents the most probable and locally relevant continuation. The results in Table 4 show that the content of the query is essential for identifying useful information, given that enriching the incomplete code with predicted sequences can help bridge the semantic gap and facilitate the retrieval of more relevant cross-file context.

A.4 Retrieved Code Snippets Position Analysis

Additionally, we investigate the impact of the position of retrieved code snippets within the prompt on code completion performance by varying their order before inputting the prompt to the code LLMs. In previous section, the default arrangement places code snippets in descending order of relevance to the query. In this experiment, we also consider reversing this order as well as randomizing the placement of the snippets, in order to examine how the position of retrieved code snippets within the prompt affects the final completion results. The results are presented in Table 5. We observe that placing relevant code snippets at the beginning or end of the prompt has a comparable impact on completion performance. However, when retrieved code snippets are randomly positioned within the prompt, the completion accuracy drops significantly. This indicates that the placement of retrieved code snippets within the prompt affects the completion results, which is consistent with findings from previous studies on natural language tasks (Liu et al., 2024b).

Table 3: The completion results of the baseline methods. **EM** denotes Exact Match, and **ES** denotes Edit Similarity.

Baseline Evaluation									
Code LLM	**Method**	**CCEval (Python)**		**CCEval (Java)**		**RepoEval (Line)**		**RepoEval (API)**	
		EM	ES	EM	ES	EM	ES	EM	ES
StarCoderBase-7B	In-file Only	6.79	60.69	9.44	63.83	36.25	59.95	26.19	51.83
	Simple RAG (BM25)	14.78	66.30	16.36	66.41	44.19	65.48	32.69	55.91
	Simple RAG (RLRetriever)	25.89	72.12	25.81	68.91	47.75	68.48	35.06	58.09
CodeLlama-7B	In-file Only	7.39	60.78	9.35	63.83	37.00	60.47	29.19	57.43
	Simple RAG (BM25)	15.76	66.30	16.78	65.37	44.25	65.65	37.00	63.43
	Simple RAG (RLRetriever)	26.57	71.43	26.23	66.37	46.63	67.88	37.94	64.32
DeepSeekCoder-1.3B	In-file Only	5.85	58.21	6.92	58.37	33.88	58.46	26.94	55.86
	Simple RAG (BM25)	14.18	65.12	13.70	61.63	41.31	63.60	35.31	62.07
	Simple RAG (RLRetriever)	24.05	70.46	20.85	63.43	44.12	66.46	36.06	62.69
DeepSeekCoder-6.7B	In-file Only	9.46	61.60	11.41	62.54	39.63	61.95	30.44	59.40
	Simple RAG (BM25)	18.31	68.38	17.48	65.23	45.94	66.68	38.25	64.99
	Simple RAG (RLRetriever)	30.24	73.58	26.13	66.11	48.75	69.42	39.88	66.24

Table 4: Comparison of completion results using different methods of changing query.

Query Content Analysis									
Code LLM	**Method**	**CCEval (Python)**		**CCEval (Java)**		**RepoEval (Line)**		**RepoEval (API)**	
		EM	ES	EM	ES	EM	ES	EM	ES
StarCoderBase-7B	Origin	25.89	72.12	25.81	68.91	47.75	68.48	35.06	58.09
	Append	26.34	72.40	26.65	69.44	47.56	68.34	35.13	58.19
CodeLlama-7B	Origin	26.57	71.43	26.23	66.37	46.63	67.88	37.94	64.32
	Append	27.24	72.75	26.79	68.05	46.69	67.92	37.81	64.22
DeepSeekCoder-1.3B	Origin	24.05	70.46	20.85	63.43	44.12	66.46	36.06	62.69
	Append	24.84	70.28	21.37	62.73	44.13	66.48	35.88	62.61
DeepSeekCoder-6.7B	Origin	30.24	73.58	26.13	66.11	48.75	69.42	39.88	66.24
	Append	30.66	74.79	26.37	67.28	49.63	69.62	40.25	66.46

Table 5: Comparison of completion results using different sort order of retrieved code snippets in prompt.

Relevant Information Position Analysis									
Code LLM	**Method**	**CCEval (Python)**		**CCEval (Java)**		**RepoEval (Line)**		**RepoEval (API)**	
		EM	ES	EM	ES	EM	ES	EM	ES
StarCoderBase-7B	Default	25.89	72.12	25.81	68.91	47.75	68.48	35.06	58.09
	Reverse	26.68	72.24	25.90	69.71	47.81	68.22	34.56	58.07
	Random	24.47	71.31	24.87	68.74	45.94	66.83	33.38	57.10
CodeLlama-7B	Default	26.57	71.43	26.23	66.37	46.63	67.88	37.94	64.32
	Reverse	26.15	72.12	26.41	68.83	46.94	68.20	38.63	64.54
	Random	24.20	70.98	23.94	66.83	44.88	66.79	36.38	63.09
DeepSeekCoder-1.3B	Default	24.05	70.46	20.85	63.43	44.12	66.46	36.06	62.69
	Reverse	24.92	70.60	21.46	64.06	44.75	66.42	36.56	62.94
	Random	21.88	68.27	19.92	61.92	42.00	65.18	34.44	61.46
DeepSeekCoder-6.7B	Default	30.24	73.58	26.13	66.11	48.75	69.42	39.88	66.24
	Reverse	29.91	74.28	26.88	67.72	48.75	69.52	40.56	66.76
	Random	26.75	73.07	24.68	66.62	46.56	68.35	38.06	65.01

Table 6: Average cost (seconds) for each step of RepoAttention on CCEval Python.

Dual-Path Query Refinement				Rerank	Completion
Distillation & Enrichment	Sparse Retrieval	Sample	Dense Retrieval		
0.045	0.022	0.549	0.662	0.159	0.185

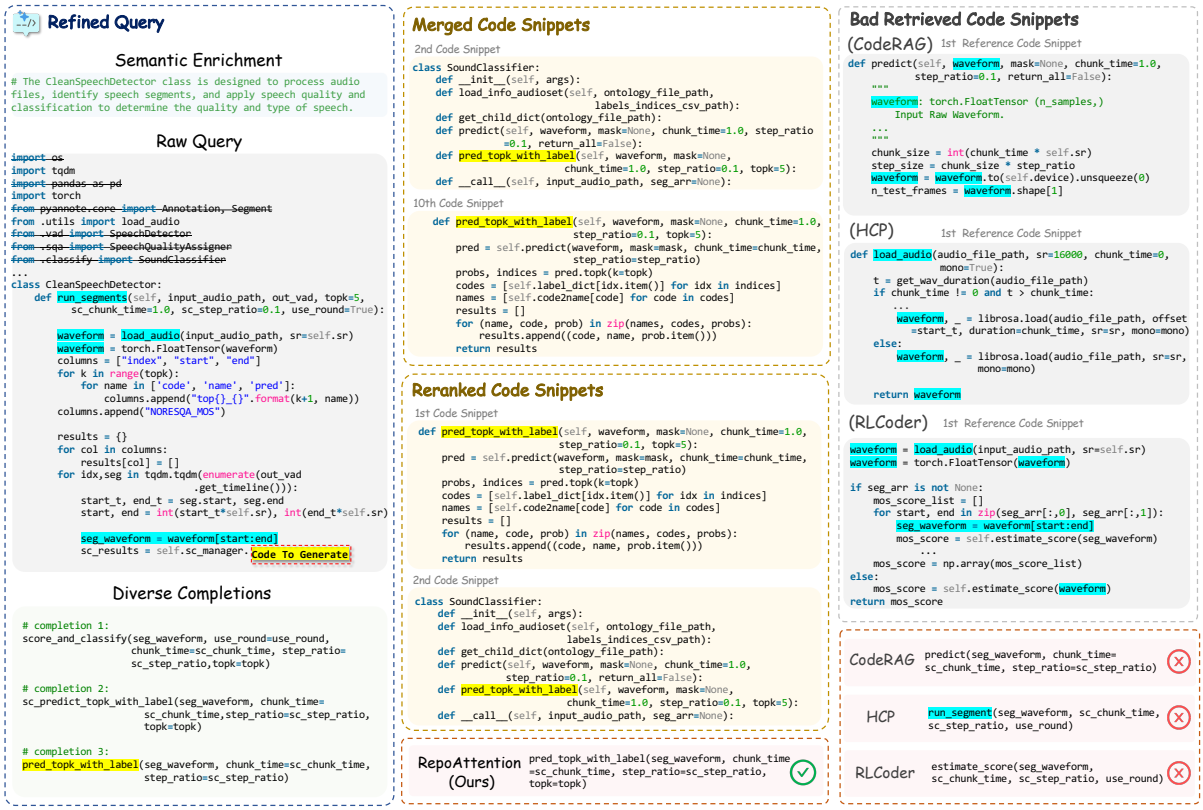


Figure 5: A case *project_cc_python/1474* from CCEval Python. The highlighted words indicate overlaps between the query and the code snippet. Strikethrough text marks code that is removed by the query distillation step.

Table 7: Average cost (seconds) per case on CCEval Python.

HCP	CodeRAG	RLCoder	RepoAttention
1.95	2.53	1.67	1.62

B Computational Cost Details

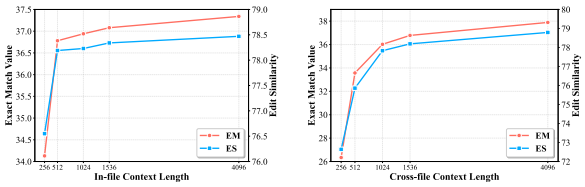
Table 6 reports the average time consumption for each step of RepoAttention. We observe that the primary computational costs stem from dense retrieval and sampling. This is consistent with expectations. The sampling step involves LLM inference which is naturally resource-intensive, while dense retrieval requires similarity searches in a high-dimensional vector space.

It is worth noting that our current implementation of RepoAttention prioritizes accuracy and has not been heavily optimized for speed. The latency could be further reduced through engineering improvements. For instance, model quantization could be applied to reduce memory bandwidth pressure. Additionally, employing accelerated vector search libraries (such as Faiss[3]) would likely reduce the overhead of the dominant stages, further enhancing the practical deployment of RepoAttention.

C Case Study

We present a case study to demonstrate the effectiveness of RepoAttention. Figure 5 shows an example where the model is required to complete the function `pred_topk_with_label` in the `CleanSpeechDetector` class. As illustrated by the raw query, the query distillation step removes

[3]https://github.com/facebookresearch/faiss



(a) Performance on different in-file context lengths.

(b) Performance on different cross-file context lengths.

Figure 6: Performance under different context lengths.

irrelevant code, indicated by the strikethrough lines. The resulting refined query enables the model to sample useful candidate completions. The reranking component then prioritizes the most relevant snippet containing the definition of `pred_topk_with_label` and places it at the top of the context. This precise retrieval enables RepoAttention to correctly complete the target function. In contrast, baselines like CodeRAG, HCP and RLCoder retrieve irrelevant snippets that appear textually similar to the incomplete code, leading to incorrect completions.

D Impact of Context Length

We conduct experiments on CCEval Python with DeepSeekCoder-6.7B. The prompt for code LLMs includes both in-file and cross-file context, with default lengths of 512 and 1536 tokens. We vary the length of one type of context while keeping the other fixed at its default value. Figures 6(a) and 6(b) show RepoAttention's performance as the context length increases from 256 to 4096 tokens.

Increasing either type of context consistently improves performance across all metrics, but the gains diminish as the context grows. For in-file context, expanding the length from 256 to 512 tokens yields a clear improvement, with EM increasing by 7.8% and ES by 2.1%, while further increases bring smaller gains. Cross-file context follows a similar trend, where early expansion is highly beneficial but additional tokens are less likely to introduce new relevant information, leading to a performance plateau. Overall, longer cross-file context is more effective than longer in-file context, as it provides more diverse and completion-relevant snippets. Based on these results, we recommend using around 512 tokens for in-file context and 1536 tokens for cross-file context in practice.

E Algorithm of RepoAttention

Algorithm 1: RepoAttention

Input: Incomplete code c_{in}, Repository $\mathcal{R}$, Code LLM $\mathcal{M}$, Reranker $\mathcal{R}_{rank}$, Relevance threshold I, Number of samples m, Key code lines t, Top-k retrieval

Output: Code completion result $\hat{c}$

Step 1: Codebase Construction

Parse $\mathcal{R}$ into primary snippets via blank-line split;

Extract dependency snippets via tree-sitter parsing;

$\mathcal{D} \leftarrow \{\text{primary snippets}\} \cup \{\text{dependency snippets}\}$;

Step 2: Dual-Path Query Refinement

$q_{raw} \leftarrow c_{in}$;

$q_{ref} \leftarrow \textsc{DistillAndEnrich}(q_{raw})$

`// Sparse retrieval via BM25`

$S^{raw}_{sparse} \leftarrow \text{BM25}(q_{raw}, \mathcal{D})$;

$S^{ref}_{sparse} \leftarrow \text{BM25}(q_{ref}, \mathcal{D})$;

`// Generate hypothetical completions`

$H_{raw} \leftarrow \{h_1, \ldots, h_m\} \sim P_{\mathcal{M}}(\cdot|S^{raw}_{sparse} \oplus q_{raw})$;

$H_{ref} \leftarrow \{h'_1, \ldots, h'_m\} \sim P_{\mathcal{M}}(\cdot|S^{ref}_{sparse} \oplus q_{ref})$;

`// Dense retrieval with expanded queries`

$S^{raw}_{dense} \leftarrow \text{DenseRetrieve}(q_{raw} \oplus H_{raw}, \mathcal{D})$;

$S^{ref}_{dense} \leftarrow \text{DenseRetrieve}(q_{ref} \oplus H_{ref}, \mathcal{D})$;

Step 3: Relevance-Aware Snippet Reranking

$c_k \leftarrow$ Extract last t lines from q_{ref} as key code;

`// Compute relevance score`

foreach *snippet* $s_i \in S^{raw}_{dense} \cup S^{ref}_{dense}$ **do**

  $r_i \leftarrow \mathcal{R}_{rank}(c_k, s_i)$

$C_v \leftarrow$ Sort snippets by r_i in descending order;

Step 4: Dynamic Relevance-Guided Attention

$E \leftarrow \emptyset$;

foreach *snippet* $s_i \in C_v$ *with tokens* T_i **do**

  if $r_i \geq I$ **then**

    $E \leftarrow E \cup T_i$;

  else

    $p_i \leftarrow \textsc{MapToProbability}(r_i)$;

    if $\textsc{Bernoulli}(p_i) = 1$ **then**

      $E \leftarrow E \cup T_i$;

$E \leftarrow E \cup T_q$; `// Always include query tokens`

`// Generate completion with DRGA`

$\hat{c} \leftarrow \mathcal{M}.\text{Complete}(C_v \oplus q_{ref})$ with attention restricted to E;

return $\hat{c}$;