# AIRCoder: Adaptive Integration of Multi-dimensional Retrieval for Repository-level Code Completion

**Chuanqi Shi[1], Miao Gao[1], Zhiqiang Gao[1*],**
[1]School of Computer Science and Engineering, Southeast University,
{chuanqi_shi, miaogao, zqgao}@seu.edu.cn

## Abstract

Repository-level code completion relies on retrieval strategies to select relevant context from large codebases. Most existing methods employ single-dimensional retrieval based on textual similarity or dependency existence, leading to inconsistent performance across completion scenarios. Even task-adaptive and hybrid methods incur high computational costs while failing to explicitly consider structural hierarchy for retrieving functionally similar code. We introduce **AIRCoder**, a multi-dimensional retrieval framework that combines eight complementary metrics across three dimensions: textual similarity, dependency existence, and structural hierarchy. By proposing a structure-preserving chunking strategy and lightweight fusion module, AIRCoder learns context-dependent weights to adaptively integrate retrieval metrics for each query. Experiments on CrossCodeEval and RepoEval demonstrate that AIRCoder achieves an average improvement of **4.63%** in exact match over the best baseline, with **10.2**× higher efficiency and strong cross-language generalization across Python, Java, C#, and TypeScript.[1]

## 1 Introduction

Code large language models (LLMs) (Lozhkov et al., 2024; Guo et al., 2024) have been emerged as powerful tools in software development (Chen et al., 2021; Nguyen-Duc et al., 2025). However, repository-level code completion tasks require long context from entire codebases, which often contain hundreds of thousands to millions of tokens (Zhang et al., 2023). Retrieval-augmented code generation (RACG) retrieves relevant context rather than consuming entire repositories (Parvez et al., 2021; Zhou et al., 2022), accommodating LLMs' context window constraints while mitigating performance degradation from irrelevant information (Shi et al., 2023; Yoran et al., 2024).

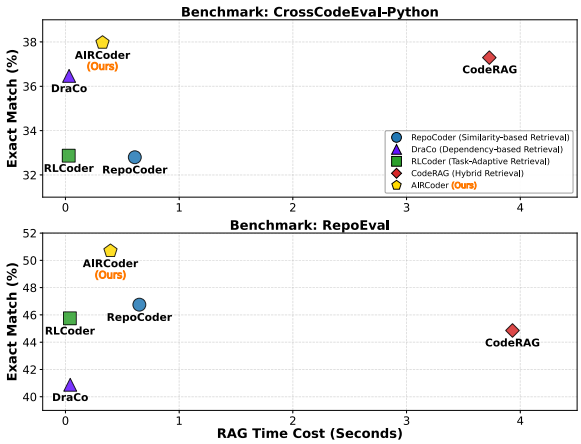


Figure 1: Performance-efficiency trade-off across baseline methods and AIRCoder on Qwen2.5-Coder-7B.

Existing methods retrieve relevant code from two single dimensions: similarity-based retrieval retrieves analogous implementations through textual similarity (Zhang et al., 2023), while dependency-based retrieval leverages dependency contexts such as API definitions (Cheng et al., 2024). As shown in Figure 1, **single-dimensional retrieval fails to address the complexity of diverse completion scenarios and leads to contextual misalignment** (Leung et al., 2025). This limitation manifests differently across scenarios. Dependency-based retrieval proves effective in CrossCodeEval (Ding et al., 2023) yet introduces noise in RepoEval (Zhang et al., 2023). Conversely, similarity-based retrieval exhibits the opposite pattern. Beyond performance degradation, this inconsistency reflects a disconnect with actual developer behavior, which demonstrate that developers naturally navigate code through multiple complementary dimensions, examining analogous implementations, data flows, and structural hierarchies (Ustalov et al., 2025). To address this inconsistency, task-adaptive

[1]Our code is publicly available at https://github.com/chuanky/AIRCoder.



*Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 25458–25470

retrieval (Wang et al., 2025) and hybrid retrieval (Zhang et al., 2025a) implicitly integrate multi-dimensional retrieval by introducing additional LLMs for retrieval or reranking. As shown in Figure 1, although these methods demonstrate moderate performance across scenarios, they **incur substantial computational overhead**. More fundamentally, all existing methods **fail to explicitly consider structural hierarchy constraints for retrieving functionally similar code** (Min and Li Ping, 2019). This oversight prevents LLMs from discovering reusable patterns and similar practices across different modules.

To address these challenges, we propose **AIRCoder** (**A**daptive **I**ntegration of multi-dimensional **R**etrieval). By proposing structure-preserving chunking, AIRCoder mitigates structural disruption caused by arbitrary code splitting (Zhang et al., 2023; Shi et al., 2025), preserving syntactic and semantic integrity. We then design eight complementary retrieval metrics covering textual similarity, dependency existence, and structural hierachy. A lightweight fusion module dynamically learns context-dependent weights for these metrics, enabling adaptive retrieval that mirrors developers' multi-dimensional navigation while ensuring computational efficiency. The main contributions of AIRCoder can be summarized as follows:

- We propose AIRCoder, a novel framework featuring adaptive integration of multi-dimensional retrieval for repository-level code completion. AIRCoder introduces a structure-preserving chunking strategy, combined with eight complementary retrieval metrics and a fusion module to achieve task-adaptive retrieval across multiple dimensions.

- We introduce both a training-free fusion approach based on Reciprocal Rank Fusion (RRF) and a trainable lightweight MLP-based fusion module that learns context-dependent weights to adaptively integrate retrieval metrics for each query, both substantially outperforming baseline methods.

- Extensive experiments demonstrate that AIRCoder outperforms the strongest baseline by an average improvement of **4.63%** in exact match, while achieving **10.2**× faster efficiency and robust generalization across Python, Java, C#, and TypeScript.

## 2 Related Work

### 2.1 Repository-level Code Completion

Code LLMs have demonstrated remarkable capabilities in code understanding and generation (Chen et al., 2021; Rozière et al., 2023; Lozhkov et al., 2024; Guo et al., 2024). While early approaches primarily focused on function-level generation (Chen et al., 2021; Austin et al., 2021), recent efforts have increasingly shifted toward repository-level completion to address real-world development scenarios. However, a significant misalignment exists between the vast size of repository contexts and the limited context windows of LLMs. For example, a standard repository often exceeds hundreds of thousands of tokens, substantially exceeding the input capacity of most LLMs (Nijkamp et al., 2023; Peng et al., 2025).

To address this, early research incorporated cross-file context through fine-tuning (Ding et al., 2024; Shrivastava et al., 2023a). Recent work has gravitated towards RACG, treating LLMs as black-box generators (Shi et al., 2024; Ram et al., 2023). In this framework, the repository is sliced into code snippets, and relevant snippets are retrieved based on the current cursor context to construct a prompt (Tan et al., 2024; Zhang et al., 2023). As the initial phase, repository slicing has a significant influence on the final prompt composition and generation quality. However, there is is no consensus on the optimal slicing method. While simple line-based partitioning remains common due to its minimal preprocessing overhead (Zhang et al., 2023), other approaches advocate for syntax-aware boundaries, partitioning the repository into coherent logical blocks such as classes, functions, or global variables to preserve semantic integrity (Cheng et al., 2024; Zhang et al., 2025a; Fehr et al., 2025; Liu et al., 2024). Additionally, in the retrieval phase, existing methods typically prioritize a single similarity metric, such as textual overlap, dense semantic similarity, or AST-based structural proximity (Liu et al., 2024). However, a single metric often fails to capture the nuanced and multi-faceted dependencies required for complex cross-file reasoning.

### 2.2 Retrieval Strategies in RACG

Existing retrieval strategies in RACG can be categorized into similarity-based, dependency-based, task-adaptive, and hybrid retrieval.

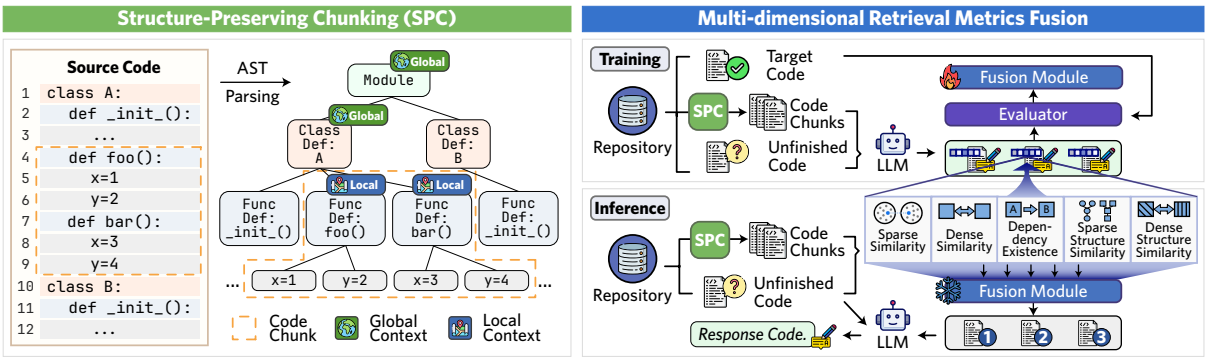


Figure 2: An overview of our proposed AIRCoder. The left panel illustrates the structure-preserving chunking (SPC) strategy for code chunking. The right panel depicts the fusion process of multi-dimensional retrieval metrics, including sparse/dense textual similarity, dependency existence, and sparse/dense structural similarity.

**Similarity-based Retrieval.** The most prevalent methods rely on textual or semantic similarity to identify analogous implementations. Sparse retrieval techniques, such as Jaccard Index (Jaccard, 1912) and BM25, match keywords between queries and candidate code snippets, effectively locating identifier overlaps (Zhang et al., 2023; Liu et al., 2024). Conversely, dense retrieval methods leverage pre-trained encoders to capture semantic proximity in vector space (Guo et al., 2022). However, these methods often fail to capture syntactically dependent code lacking textual overlap.

**Dependency-based Retrieval.** Source code is interconnected through intricate dependencies. Recognizing that standard similarity-based retrieval often overlooks these relationships, recent studies have shifted toward dependency-aware strategies. For instance, Shrivastava et al. (2023b) exploit repository inheritance relationships for prompt generation, while DraCo (Cheng et al., 2024) leverages dataflow analysis to retrieve dependency contexts such as function or class definitions.

**Task-Adaptive Retrieval.** Beyond static and single-dimensional metrics, several methods optimize retrievers through training. RLCoder (Wang et al., 2025) aligns retriever with generator using reinforcement learning, while Repoformer (Wu et al., 2024) constructs task-specific datasets for selective retrieval. However, these methods rely on implicit, data-driven adaptation and generally lack explicit structural constraints.

**Hybrid Retrieval and Our Proposition.** Recent trends integrate distinct retrieval sources. CodeRAG (Zhang et al., 2025a) aggregates multi-retriever's results with LLM-based reranking. RepoFuse (Liang et al., 2024) combines import analysis for background context with similarity search. Distinct from these methods, our approach explicitly incorporates the **structural hierarchy** of the repository alongside similarity and dependency information. Moreover, unlike methods that require expensive retriever training (Wang et al., 2025), we propose a **lightweight, adaptive** fusion module, enabling both effective and efficient integration of multi-dimensional retrieval.

## 3 AIRCoder

As shown in Figure 2, AIRCoder comprises two main components: structure-preserving chunking (Section 3.1) and multi-dimensional retrieval metrics fusion, with Section 3.2 detailing the retrieval metrics and Section 3.3 describing the training and inference pipeline of AIRCoder.

### 3.1 Structure-Preserving Chunking (SPC)

Unlike text splitting methods that fragment logical units (Zhang et al., 2023), we propose a structure-preserving chunking (SPC) strategy for code syntactic integrity. Specifically, we segment repository files with AST-based decomposition (Zhang et al., 2025b). By traversing the AST structure under the target LLM's token constraints, SPC ensures that each chunk maintains semantic coherence and respects functional boundaries, effectively balancing context retention with retrieval granularity.

Moreover, we explicitly attach two type of **structural metadata** to each chunk to preserve inherent hierarchical structure information: (1) **Global Context** captures external information invisible within the chunk, yet essential for locating its position in the repository, including the file path and the signatures of the enclosing class and function. This

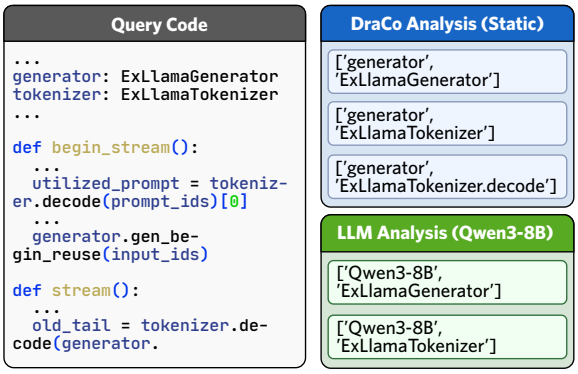


Figure 3: A comparison of code dependency extraction results obtained using DraCo versus our proposed LLM-based approach using Qwen3-8B.

ensures that even deeply nested code fragments retain their lineage. (2) **Local Context** serves as a structural summary of the chunk itself, comprising signatures of any classes or functions defined internally within the chunk. The appended global context maintains the repository's hierarchical skeleton, while the extracted local context provides a concise structural overview. Figure 2 (left) illustrates SPC and the extraction of global and local contexts for a code chunk.

### 3.2 Multi-dimensional Retrieval Metrics Fusion

We incorporate sparse and dense textual similarity along with dependency existence as retrieval metrics, and further introduce sparse and dense structural similarity for multi-dimensional fusion.

**Sparse Similarity.** This metric reflects scenarios where developers use full-text search to retrieve relevant code. It is calculated based on Jaccard Similarity (Jaccard, 1912):

$$Sim_{sparse}(S_q, S_c) = \frac{|T_q \cap T_c|}{|T_q \cup T_c|} \quad (1)$$

where $S_q$ and $S_c$ represent the query and candidate code snippets, respectively, and $T_q$ and $T_c$ represent their corresponding token sets.

**Dense Similarity.** This metric captures semantic similarity between code snippets, mimicking developers searching for semantically related code. It is calculated with cosine similarity (Guo et al., 2022):

$$Sim_{dense}(S_q, S_c) = \frac{\vec{q} \cdot \vec{c}}{\|\vec{q}\|\|\vec{c}\|} \quad (2)$$

where $\vec{q}$ and $\vec{c}$ are the vector representations of the query and candidate code snippet encoded by a pretrained model.

**Dependency Existence (DE).** This metric indicates whether the dependency context extracted from the query code appears in the candidate.

$$DE(S_q, S_c) = \begin{cases} 1 & \text{if } Dep(S_q) \in \text{MC}(S_c) \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where $\text{MC}(S_c)$ is the structural metadata (global and local context) to the candidate code, and $Dep(S_q)$ is the dependency context of the query code. In contrast to DraCo (Cheng et al., 2024), which relies on static analysis to extract type-specific identifiers, we propose **LLM-based dependency extraction**. As demonstrated in Figure 3, this approach captures the essential semantic constraints for prediction, even in scenarios where formal static analysis is infeasible. Detailed prompt can be found in Appendix A.

**Sparse Structure Similarity.** This metric leverages the repository's **structural metadata** defined in Section 3.1. Specifically, we measure the character-level similarity between their respective file paths, enclosing scopes (Global Context), and internal structural signatures (Local Context). For any two structural identifiers $t_1, t_2$, we employ a normalized **Edit Similarity** ($ES$) based on the Levenshtein distance (Levenshtein, 1966):

$$ES(t_1, t_2) = 1 - \frac{\text{Lev}(t_1, t_2)}{\max(\text{len}(t_1), \text{len}(t_2))} \quad (4)$$

This metric comprises **four distinct sub-metrics**: (1) **File Proximity** ($Sim_{dist}$), calculated as $1/d$ where $d$ is the directory distance, reflecting the spatial lineage within the file tree; (2) **Lexical Overlap** ($Sim_{file}, Sim_{cls}, Sim_{func}$) compare the filenames, class signatures, and function signatures extracted from both contexts, prioritizing lexicographically aligned candidates. Detailed formulations can be found in Appendix B.

**Dense Structure Similarity.** Sparse structure similarity is derived from explicit rules and symbolic representations. Inspired by information retrieval, where sparse and dense features demonstrate complementary strengths, we introduce a dense structural representation to capture latent structural semantics. Specifically, we concatenate the global and local contexts of each code snippet and encode them into dense embeddings. The

metric is formally defined as the cosine similarity between the two embeddings:

$$Sim_{struct}(\tilde{S}_q, \tilde{S}_c) = \frac{\vec{q_s} \cdot \vec{c_s}}{\|\vec{q_s}\|\|\vec{c_s}\|} \quad (5)$$

where $\tilde{S}_q$ and $\tilde{S}_c$ denote the concatenated context for the query and the candidate, respectively, while $\vec{q_s}$ and $\vec{c_s}$ are their corresponding embeddings.

Consequently, we obtain a comprehensive set of **eight multi-dimensional retrieval metrics**, including: (1) textual similarity ($Sim_{sparse}$ and $Sim_{dense}$), (2) dependency existence ($DE$), and (3) structural hierarchy ($Sim_{dist}$, $Sim_{file}$, $Sim_{cls}$, $Sim_{func}$, and $Sim_{struct}$). To effectively integrate these metrics, we employ Reciprocal Rank Fusion (RRF) (Cormack et al., 2009). This **RRF-based fusion**, denoted as **AIRCoder**$_r$, consolidates multiple ranking lists by prioritizing consistently high-ranked candidates and enables integration without requiring manual weight tuning:

$$RRFscore(S_c \in \mathbb{S}) = \sum_{m \in M} \frac{1}{k + m(S_c)} \quad (6)$$

where $\mathbb{S}$ is the set of all candidates, $M$ is the set of ranking lists generated by different retrieval metrics, $m(S_c)$ is the rank of a candidate $S_c$ in a specific ranking list, and $k$ is a smoothing constant.

### 3.3 Adaptive Integration Retrieval Framework

Although RRF-based fusion effectively integrates multi-dimensional retrieval metrics, different code completion tasks exhibit varying demands for retrieval metrics. Therefore, we extend this approach into an adaptive integration framework that ensures the retrieval process dynamically aligns with the specific contextual requirements of each task.

**Fusion Module.** To account for varying metric importance across coding contexts, we employ a Multi-Layer Perceptron (MLP) as the **adaptive fusion module**, denoted as **AIRCoder**$_a$. It takes the vector representation of the query code as input and outputs a weight vector $\mathbf{w} \in \mathbb{R}^8$. The final score $s$ for a candidate snippet is then calculated as the weighted sum of its eight metric values.

**Training.** For training data construction, we first pair the incomplete query code with each candidate snippet from the repository. The features of each training instance are the eight retrieval metrics calculated between the candidate snippet and the query, while the label represents code generation quality, measured by Edit Similarity between the model's prediction and the ground truth. To optimize the fusion module's ability to distinguish between snippets of varying quality, we utilize a Pairwise Mean Squared Error Loss:

$$\mathcal{L} = (\Delta s_{\text{pred}} - \Delta y_{\text{true}})^2 \quad (7)$$

where $\Delta s_{\text{pred}} = s_{\text{pos}} - s_{\text{neg}}$ is the difference in predicted scores between a positive and a negative sample, and $\Delta y_{\text{true}} = y_{\text{pos}} - y_{\text{neg}}$ denotes the difference in their respective ground-truth labels.

**Inference.** The inference process follows a multi-stage pipeline. First, the repository is chunked into snippets with SPC. For a given query, we calculate the multi-dimensional feature vector for each snippet. Concurrently, the MLP generates context-dependent weights based on the query representation. Each snippet is then ranked by its weighted relevance score. Finally, we select the snippets in descending order and prepend them to the query code as context until reaching the context length.

## 4 Experiments

### 4.1 Experimental Setup

**Datasets.** We utilize two widely used repository-level code completion benchmarks for evaluation: (1) **CrossCodeEval** (Ding et al., 2023), which focuses on intra-line completion across Python, C#, Java, and TypeScript; and (2) **RepoEval** (Zhang et al., 2023), targeting next-line completion in Python. For RepoEval, we employ the API subset, which is highly challenging as it requires resolving at least one cross-file API dependency.

**Evaluation Metrics.** Following Zhang et al., we use two categories of metrics: (1) **Code Match**, including Exact Match (EM) and Edit Similarity (ES); and (2) **Identifier Match**, which calculates identifier exact match (ID-EM) and F1-score (ID-F1) by comparing identifiers extracted from the predicted and ground-truth code.

**Baselines.** We compare AIRCoder against several baseline methods: (1) **Zero-Shot** relies solely on in-file context without retrieval. (2) **RAG** (Lewis et al., 2020) follows the standard RAG with a sparse retriever. (3) **RepoCoder** (Zhang et al., 2023) employs a three-iteration RAG pipeline to refine queries. (4) **DraCo** (Cheng et al., 2024) constructs and traverses a repository-specific context

| Method | DeepSeek-Coder-1.3B | | | | StarCoder2-3B | | | | Qwen2.5-Coder-3B | | | | Qwen2.5-Coder-7B | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Code Match | | Identifier Match | | Code Match | | Identifier Match | | Code Match | | Identifier Match | | Code Match | | Identifier Match | |
| | **EM** | **ES** | **EM** | **F1** | **EM** | **ES** | **EM** | **F1** | **EM** | **ES** | **EM** | **F1** | **EM** | **ES** | **EM** | **F1** |
| *Benchmark: CrossCodeEval-Python* | | | | | | | | | | | | | | | | |
| Zero-Shot | 5.93 | 45.76 | 12.16 | 40.59 | 6.60 | 47.25 | 12.87 | 41.95 | 8.44 | 48.14 | 14.26 | 42.81 | 10.69 | 50.69 | 17.30 | 45.35 |
| RAG | 25.40 | 61.85 | 34.48 | 60.39 | 26.75 | 63.85 | 36.89 | 62.23 | 27.69 | 63.72 | 37.30 | 61.84 | 31.56 | 66.81 | 41.54 | 65.43 |
| RepoCoder | 27.28 | 63.45 | 36.92 | 62.64 | 28.82 | 65.29 | 38.91 | 64.69 | 30.02 | 65.36 | 40.38 | 64.63 | 32.80 | 67.72 | 42.78 | 66.87 |
| DraCo | 30.28 | 64.60 | 40.04 | 65.21 | 31.48 | 65.92 | 41.50 | 66.42 | 33.21 | 66.42 | 42.89 | 66.43 | 36.47 | 69.61 | 46.57 | 69.59 |
| RLCoder | 26.90 | 62.99 | 36.14 | 61.33 | 28.26 | 64.29 | 37.56 | 62.97 | 29.31 | 64.54 | 39.02 | 63.25 | 32.87 | 67.55 | 42.66 | 66.10 |
| CodeRAG | 30.92 | 65.02 | 40.04 | 64.36 | 32.80 | 66.79 | 43.19 | 66.77 | 32.95 | 66.44 | 42.78 | 66.31 | 37.32 | 70.51 | 47.38 | 70.03 |
| **AIRCoder$_r$** | **31.26** | **66.63** | **41.84** | **66.63** | **33.77** | **68.12** | **43.75** | **68.06** | **34.45** | **68.83** | **45.74** | **69.05** | **37.97** | **71.35** | **48.67** | **71.28** |
| *Benchmark: RepoEval* | | | | | | | | | | | | | | | | |
| Zero-Shot | 25.87 | 56.10 | 29.50 | 57.76 | 26.06 | 53.85 | 29.12 | 54.27 | 33.75 | 61.77 | 36.94 | 62.94 | 36.81 | 64.02 | 39.62 | 65.70 |
| RAG | 40.38 | 67.05 | 44.56 | 69.42 | 42.88 | 68.74 | 46.81 | 71.03 | 45.50 | 70.32 | 49.25 | 71.95 | 46.62 | 72.10 | 50.56 | 74.21 |
| RepoCoder | 40.56 | 67.26 | 44.25 | 69.62 | 43.62 | 69.47 | 47.19 | 71.95 | 45.56 | 70.98 | 49.19 | 73.07 | 46.75 | 72.13 | 50.38 | 74.40 |
| DraCo | 30.06 | 59.88 | 33.44 | 61.74 | 30.00 | 57.13 | 33.12 | 59.27 | 37.38 | 66.05 | 41.25 | 67.83 | 40.88 | 69.42 | 44.25 | 70.72 |
| RLCoder | 39.56 | 66.50 | 43.38 | 68.83 | 40.75 | 67.85 | 44.81 | 70.45 | 44.00 | 70.09 | 47.50 | 71.97 | 45.75 | 71.95 | 49.69 | 73.87 |
| CodeRAG | 35.75 | 64.81 | 39.75 | 66.71 | 34.69 | 60.67 | 37.94 | 61.81 | 41.69 | 68.59 | 46.19 | 70.20 | 44.88 | 71.79 | 48.31 | 73.11 |
| **AIRCoder$_r$** | **43.94** | **70.41** | **48.25** | **72.70** | **46.38** | **72.14** | **50.88** | **74.48** | **49.62** | **74.03** | **54.00** | **75.84** | **50.69** | **74.91** | **54.69** | **76.85** |

Table 1: Performance comparison among AIRCoder$_r$ and other methods across four LLMs on full evaluation set.

graph to capture dataflow and entity relations. (5) **RLCoder** (Wang et al., 2025) optimizes a dense retriever with reinforcement learning. (6) **CodeRAG** (Zhang et al., 2025a) utilizes a hybrid retrieval strategy with LLM-based reranking.

**Implementation Details.** We employ SPC with an 800-token budget. Code is parsed and encoded with `tree-sitter`[2] and *UniXcoder* (Guo et al., 2022), respectively. For dependency context extraction, we adopt *Qwen3-8B*[3]. The RRF-based AIRCoder$_r$ is training-free, while the MLP-based AIRCoder$_a$ is trained using a 60%/10%/30% train/-validation/test split. Detailed hyperparameter settings, fusion module architecture and hardware configurations are provided in Appendix C.

## 5 Experimental Results and Analysis

We evaluate AIRCoder across five dimensions: performance comparison with state-of-the-art baselines (Section 5.1), investigation of the adaptive fusion module (Section 5.2) and component analysis (Section 5.3). We also assess AIRCoder's cross-language generalization (Section 5.5) and analyze its computational efficiency (Section 5.6).

### 5.1 Overall Performance

Table 1 presents the performance of AIRCoder$_r$, the training-free RRF-based version of our method, against various state-of-the-art baselines. Our method consistently achieves the best results across all LLMs and benchmarks. On CrossCodeEval, AIRCoder$_r$ significantly surpasses the strongest baselines, CodeRAG and DraCo. With *Qwen2.5-Coder-3B*, AIRCoder$_r$ achieves relative improvements of **3.73%** in Code-EM and **6.64%** in ID-EM over the best baseline, validating the effectiveness of our approach. These advantages become even more evident on RepoEval, where AIRCoder$_r$ achieves a substantial absolute improvement of **5.81%–16.38%** in Code-EM compared to DraCo and CodeRAG. This performance gap demonstrates that our structure-preserving chunking and multi-dimensional retrieval metrics fusion offer superior robustness when completion scenario changes.

Furthermore, the performance gains remain stable across various model scales from 1.3B to 7B parameters. Notably, AIRCoder$_r$ achieves a **8.43%** relative improvement in Code-EM over RepoCoder using *Qwen2.5-Coder-7B* on RepoEval. To further verify the model-agnostic effectiveness of our approach, we also report the results using *DeepSeek-V3.2* as the backbone in Table 2. DeepSeek-V3.2 is a large-scale Mixture-of-Experts (MoE) model featuring 671B total parameters, with approximately 37B parameters activated per token during inference. Consistent with previous observations, AIRCoder$r$ achieves the best performance across both benchmarks. On RepoEval, where AIRCoder$r$ achieves a Code-EM of 56.33%, rep-

[2] https://github.com/tree-sitter/tree-sitter
[3] https://huggingface.co/Qwen/Qwen3-8B

| Method | Code Match | | Identifier Match | |
|---|---|---|---|---|
| | EM | ES | EM | F1 |
| *Benchmark: CrossCodeEval-Python* | | | | |
| Infile | 22.33 | 59.28 | 28.33 | 56.14 |
| RAG | 39.00 | 72.10 | 50.33 | 73.21 |
| RepoCoder | 41.00 | 74.89 | 52.00 | 75.54 |
| DraCo | 44.67 | 74.18 | 52.67 | 74.64 |
| RLCoder | 41.33 | 73.94 | 50.00 | 72.44 |
| CodeRAG | 47.67 | 77.14 | 56.00 | 77.46 |
| **AIRCoder$_r$** | **48.67** | **77.31** | **58.00** | **77.77** |
| *Benchmark: RepoEval* | | | | |
| Infile | 48.33 | 74.69 | 51.33 | 77.76 |
| RAG | 49.67 | 75.94 | 52.33 | 79.30 |
| RepoCoder | 52.00 | 77.53 | 56.00 | 80.87 |
| DraCo | 47.33 | 75.30 | 50.67 | 79.59 |
| RLCoder | 47.00 | 75.79 | 50.00 | 78.93 |
| CodeRAG | 48.33 | 75.39 | 52.00 | 79.79 |
| **AIRCoder$_r$** | **56.33** | **80.93** | **60.33** | **84.01** |

Table 2: DeepSeek-V3.2 performance across different methods on CCEval and RepoEval.

| Method | Code Match | | Identifier Match | |
|---|---|---|---|---|
| | EM | ES | EM | F1 |
| *Benchmark: CrossCodeEval-Python* | | | | |
| Zero-Shot | 5.38 | 47.92 | 14.12 | 43.81 |
| RAG | 26.50 | 65.51 | 40.38 | 64.91 |
| RepoCoder | 29.25 | 67.21 | 44.00 | 68.04 |
| DraCo | 32.00 | 67.33 | 44.12 | 68.95 |
| RLCoder | 28.75 | 64.41 | 41.00 | 64.29 |
| CodeRAG | 34.00 | 68.04 | 48.38 | 69.67 |
| **AIRCoder$_r$** | 34.50 | 69.30 | 48.75 | 71.05 |
| **AIRCoder$_a$** | **36.98 ± 0.21** | **73.56 ± 0.64** | **51.68 ± 0.34** | **76.83 ± 0.30** |
| *Benchmark: RepoEval* | | | | |
| Zero-Shot | 25.21 | 53.61 | 28.33 | 55.27 |
| RAG | 43.75 | 69.17 | 47.71 | 72.19 |
| RepoCoder | 44.79 | 69.86 | 48.75 | 73.43 |
| DraCo | 29.17 | 56.39 | 32.08 | 59.45 |
| RLCoder | 39.79 | 66.87 | 43.54 | 70.39 |
| CodeRAG | 33.54 | 59.74 | 37.08 | 61.42 |
| **AIRCoder$_r$** | 48.33 | 72.39 | 52.29 | 75.29 |
| **AIRCoder$_a$** | **50.12 ± 0.46** | **75.38 ± 0.40** | **53.84 ± 0.32** | **77.83 ± 0.30** |

Table 3: Performance comparison of different methods with *StarCoder2-3B*, evaluated on test set. Results for AIRCoder$_a$ are reported as the mean ± standard error over 5 independent runs.

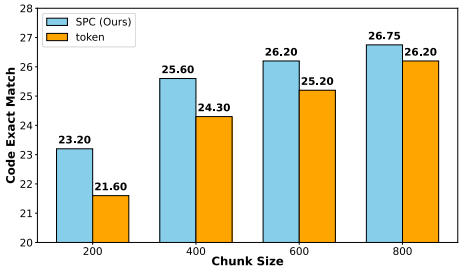


Figure 4: Performance comparison between our structure-preserving chunking (SPC) and token-based chunking methods across different chunk sizes on CrossCodeEval-Python under the RAG setting.

resenting a substantial absolute improvement of 4.33% over the most competitive baseline. This consistent scalability demonstrates that integrating multi-dimensional retrieval metrics effectively complements the generation capabilities of both small and large-scale language models.

## 5.2 Impact of Adaptive Fusion

Table 3 evaluates the performance of methods equipped with different retrieval strategies across two benchmarks with *StarCoder2-3B*, where AIRCoder$_a$ denotes our proposed adaptive fusion module. Our results show that both AIRCoder$_r$ and AIRCoder$_a$ consistently surpass all baseline methods. The effectiveness of the adaptive fusion module is particularly evident when comparing the two variants of our approach. On CrossCodeEval, AIRCoder$_a$ facilitates a significant performance gain over AIRCoder$_r$, improving Code Match EM by 2.48% and ID-EM by 2.93%. A similar trend is also observed on RepoEval. These results validate that a single or static integration of retrieval metrics is suboptimal. Our adaptive framework successfully captures the varying importance of different retrieval metrics across different completion contexts, leading to more precise code generation.

## 5.3 Component Analysis

We conduct component analysis from four perspectives: (1) impact of chunking strategies, (2) ablation studies of retrieval metrics, (3) comparison of dependency extraction methods, and (4) generalization analysis across different training data.

Figure 4 illustrates the impact of different chunking strategies. At smaller chunk sizes, token-based chunking is significantly inferior to the SPC approach. This performance gap is likely because that smaller token windows fragment atomic code structures and compromise semantic integrity. As chunk size increases, token-based performance gradually approaches but remains inferior to SPC's. Beyond superior performance, SPC serves as the structural foundation for our framework, enabling stable extraction of structural metadata. By preserving syntactic boundaries, it ensures each chunk retains its functional lineage and internal definitions, which

| Method | Code Match | | ID Match | |
|---|---|---|---|---|
| | EM | ES | EM | F1 |
| *Benchmark: CrossCodeEval-Python* | | | | |
| AIRCoder$_a$ | 36.68 | 73.18 | 51.50 | 76.99 |
| - sparse | 35.50 | 72.52 | 49.75 | 74.58 |
| - dense | 34.25 | 71.03 | 49.62 | 74.99 |
| - struct$_{sparse}$ | 35.25 | 71.44 | 48.12 | 73.89 |
| - struct$_{dense}$ | 35.00 | 71.64 | 48.32 | 73.49 |
| - DE | 32.25 | 67.93 | 46.88 | 69.70 |
| AIRCoder$_a$ (DraCo) | 36.88 | 73.85 | 52.83 | 77.39 |
| AIRCoder$_a$ (RepoEval) | 33.75 | 70.79 | 48.38 | 72.96 |
| *Benchmark: RepoEval* | | | | |
| AIRCoder$_a$ | 49.77 | 75.12 | 53.96 | 77.15 |
| - sparse | 47.23 | 73.62 | 51.29 | 76.21 |
| - dense | 47.92 | 72.56 | 51.08 | 75.97 |
| - struct$_{sparse}$ | 47.75 | 73.96 | 52.50 | 76.62 |
| - struct$_{dense}$ | 46.46 | 70.88 | 50.04 | 73.48 |
| - DE | 48.36 | 74.42 | 52.12 | 76.86 |
| AIRCoder$_a$ (DraCo) | 47.86 | 72.32 | 51.76 | 75.95 |
| AIRCoder$_a$ (CCEval) | 47.50 | 73.24 | 51.46 | 76.03 |

Table 4: Ablation study of retrieval metrics with *StarCoder2-3B*. "AIRCoder$_a$ (DraCo)" denotes replacing the LLM-based dependency extraction with the static analysis approach from DraCo. "AIRCoder$_a$ (RepoEval)" and "AIRCoder$_a$ (CCEval)" represent the adaptive fusion module trained with the training data in RepoEval and CrossCodeEval respectively.

| Method | Code Match | | Identifier Match | |
|---|---|---|---|---|
| | EM | ES | EM | F1 |
| *Benchmark: CrossCodeEval-Java* | | | | |
| Zero-Shot | 7.73 | 50.26 | 15.60 | 47.04 |
| RAG | 27.66 | 63.15 | 37.07 | 60.85 |
| RepoCoder | 29.26 | 64.15 | 37.88 | 62.18 |
| RLCoder | 29.40 | 65.03 | 38.60 | 62.80 |
| **AIRCoder$_r$** | 32.00 | 66.00 | 41.40 | 63.65 |
| **AIRCoder$_a$** | **34.00 ± 0.55** | **71.15 ± 0.49** | **44.18 ± 0.64** | **69.26 ± 0.50** |
| *Benchmark: CrossCodeEval-C#* | | | | |
| Zero-Shot | 3.87 | 45.21 | 6.67 | 33.00 |
| RAG | 24.45 | 61.17 | 28.26 | 53.84 |
| RepoCoder | 25.45 | 60.44 | 29.06 | 53.55 |
| RLCoder | 29.60 | 62.59 | 32.60 | 56.07 |
| **AIRCoder$_r$** | 27.20 | 61.79 | 30.40 | 54.78 |
| **AIRCoder$_a$** | **29.73 ± 0.27** | **68.58 ± 0.34** | **33.16 ± 0.26** | **60.07 ± 0.32** |
| *Benchmark: CrossCodeEval-TypeScript* | | | | |
| Zero-Shot | 7.07 | 42.80 | 11.73 | 42.63 |
| RAG | 22.65 | 59.42 | 28.86 | 60.66 |
| RepoCoder | 25.65 | 61.40 | 31.26 | 61.80 |
| RLCoder | 26.60 | 61.55 | 32.60 | 63.29 |
| **AIRCoder$_r$** | 28.00 | 62.56 | 34.00 | 64.16 |
| **AIRCoder$_a$** | **30.40 ± 0.24** | **66.47 ± 0.20** | **37.16 ± 0.26** | **67.69 ± 0.21** |

Table 5: Performance comparsion with *StarCoder2-3B* on the Java, C# and TypeScript subset of CrossCodeEval benchmark, evaluated on test set.

is a prerequisite for retrieval fusion.

The ablation study in Table 4 evaluates individual contributions of retrieval metrics within AIRCoder. Removing any single metric degrades performance, with the exclusion of struct$_{dense}$ causing substantial drop across both benchmarks. This demonstrates that our proposed structural metadata provides critical retrieval signals. Removing DE causes ID-EM drops significantly of 4.62% on CrossCodeEval, underscoring that dependency existence is highly effective for retrieving code snippets on how to utilize APIs.

We also compare our LLM-based dependency extraction versus DraCo's static analysis in Table 4. DraCo relies on target-line identifiers, while our LLM-based method infers dependencies from broader contextual intent. On CrossCodeEval, where target lines are partially available, DraCo achieves a marginal 0.3% improvement in CodeEM, demonstrating the precision of static analysis. However, its performance decreases on RepoEval due to **contextual misalignment**: when the target line is empty, DraCo retrieves dependencies for completed statements rather than forthcoming code. In contrast, our proposed LLM-based dependency extraction infers latent intent of the next line, maintaining robustness across benchmarks.

Notably, as shown in Table 4, AIRCoder$_a$ maintains performance superior to or competitive with the best baseline, even when transferring across training data. We attribute the performance degradation during cross-dataset transfer to the distinct feature preferences inherent in each benchmark. A detailed analysis of context-dependent weight distributions in Appendix D reveals significant shifts in retrieval metric priorities across benchmarks. These findings further validate the necessity of our adaptive fusion module for capturing diverse context-dependent requirements.

### 5.4 Reliability of LLM-based Dependency Extraction

To assess the reliability of LLM-based dependency extraction, we conduct a diagnostic experiment on the CrossCodeEval-Python dataset. We adopt the dependencies derived by DraCo (Cheng et al., 2024) as the reference ground truth for evaluating the identifiers extracted by the LLM. Specifically, we compare identifiers from three sources: those identified by DraCo, those predicted by the LLM, and the raw identifiers present in the input prompt.

| # DraCo | # LLM | # Halluc. | Recall | Halluc. Rate |
|---|---|---|---|---|
| 908 | 3,069 | 349 | 69.93% | 11.37% |

Table 6: Reliability of LLM-based Dependency Extraction on CrossCodeEval-Python. "#" denotes number of identifiers, and "Halluc." is short for "Hallucination".

Based on this, we define two key metrics:

- **Recall**: the proportion of identifiers extracted by DraCo that are successfully recovered by the LLM. This metric measures the LLM's coverage of statically verifiable dependencies.
- **Hallucination Rate**: the proportion of identifiers predicted by the LLM that appear in neither DraCo's output nor the original prompt. This metric reflects the risk of introducing non-existent entities.

As summarized in Table 6, the LLM predicts 3,069 identifiers in total, substantially exceeding the 908 identified by DraCo. This discrepancy suggests that the LLM captures implicit or cross-file dependencies that static tools may overlook due to strict parsing constraints. The LLM achieves a Recall of 69.93% against the DraCo reference. More importantly, the Hallucination Rate remains low at 11.37%, with only 349 predicted identifiers absent from both the prompt and DraCo's output. These results indicate that although LLM-based extraction introduces a marginal hallucination risk compared with rigid static analysis, the vast majority of extracted dependencies are well grounded in the actual codebase. This balance between comprehensive coverage and factual grounding supports the use of LLMs as a flexible alternative to traditional dependency parsers in repository-level tasks.

### 5.5 Cross-language Generalization

To evaluate the cross-language generalization of AIRCoder, we evaluate on the Java, C#, and TypeScript subsets of the CrossCodeEval benchmark. Baselines such as DraCo and CodeRAG are excluded from this comparison because they use Python-centric static analysis and lack flexibility for other programming languages. As presented in Table 5, AIRCoder$_a$ maintains a consistent performance lead across all languages, affirming its language-agnostic robustness. On Java, AIRCoder$_a$ surpasses the strongest baseline RLCoder by a significant margin of 4.6% in CodeEM. For C# and TypeScript, while RLCoder show narrowed gaps, AIRCoder$_a$ remains the best performance. These results underscore that the synergistic integration of multi-dimensional retrieval metrics and adaptive fusion provides a versatile solution for repository-level code completion without requiring language-specific engineering.

| Method | Time (s) | AIRCoder Stages | Time (s) |
|---|---|---|---|
| DraCo | 0.032 | Query-emb | 0.004 |
| RepoCoder | 0.610 | Struct-emb | 0.003 |
| CodeRAG | 3.732 | Dependency Extraction | 0.320 |
| **AIRCoder$_a$** | **0.366** | Multi-Dim Metrics | 0.039 |

Table 7: Efficiency comparison and detailed latency breakdown of AIRCoder stages. All values represent the average latency per query (in seconds).

### 5.6 Efficiency Analysis

Table 7 presents the computational overhead of AIRCoder$_a$ compared to baselines, alongside a breakdown of its detailed stages. While DraCo remains the fastest due to its specialized data structure for retrieval, AIRCoder shows significant advantage over other baselines, being approximately 1.7× faster than RepoCoder and 10.2× faster than CodeRAG. Notably, CodeRAG's execution time is mostly consumed by the LLM-based reranking, averaging 3.02s/query. The breakdown analysis reveals that the LLM-based dependency extraction takes 87.4% of the total time, whereas the embedding and multi-dimensional metrics stages contribute less than 0.05s combined. These results demonstrate that AIRCoder provides an efficient solution for repository-level code completion without the prohibitive latency costs associated with iterative or heavyweight reranking methods.

## 6 Conclusion

This paper presents AIRCoder, an adaptive framework for repository-level code completion. We demonstrate that multi-dimensional retrieval metrics integrating textual similarity, dependency existence and structural hierarchy is essential for different completion scenarios. Both our training-free RRF-based fusion AIRCoder$_r$ and trainable adaptive fusion AIRCoder$_a$ demonstrate superior performance and efficiency over baselines, along with strong cross-dataset and cross-language generalization across Python, Java, C#, and TypeScript.

## Limitations

Despite its performance and efficiency, our work has several limitations. First, while LLM-based dependency extraction is language-agnostic, it remains more computationally expensive than traditional rule-based static analysis, posing potential scalability challenges in high-concurrency environments. Second, sparse structure similarity relies on lexical identifiers and may fail to capture deep semantic dependencies when naming conventions are inconsistent. Finally, although the adaptive fusion module generalize well across the four languages represented in different benchmarks, they may require further adaptation for specialized domains or highly unconventional repository architectures.

## References


Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/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), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 7957–7977. Association for Computational Linguistics.

Gordon V. Cormack, Charles L A Clarke, and Stefan Buettcher. 2009. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, page 758–759, New York, NY, USA. Association for Computing Machinery.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Yangruibo Ding, Zijian Wang, Wasi Uddin 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, 20-25 May, 2024, Torino, Italy*, pages 3433–3445. ELRA and ICCL.

Fabio James Fehr, Prabhu Teja S, Luca Franceschi, and Giovanni Zappella. 2025. CoRet: Improved retriever for code editing. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 775–789, Vienna, Austria. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. 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), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

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

Kin Kwan Leung, Mouloud Belbahri, Yi Sui, Alex Labach, Xueying Zhang, Stephen Rose, and Jesse C. Cresswell. 2025. Classifying and addressing the diversity of errors in retrieval-augmented generation systems. *Preprint*, arXiv:2510.13975.

VI Levenshtein. 1966. Binary coors capable or 'correcting deletions, insertions, and reversals. In *Soviet physics-doklady*, volume 10.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Wei Jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. 2024. REPOFUSE: repository-level code completion with fused dual context. *CoRR*, abs/2402.14323.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *Proceedings of the 39th IEEE/ACM*

*International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 570–581. ACM.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 38 others. 2024. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173.

Hou Min and Zhang Li Ping. 2019. Survey on software clone detection research. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*, ICMSS 2019, page 9–16, New York, NY, USA. Association for Computing Machinery.

Anh Nguyen-Duc, Beatriz Cabrero-Daniel, Adam Przybylek, Chetan Arora, Dron Khanna, Tomas Herda, Usman Rafiq, Jorge Melegati, Eduardo Guerra, Kai-Kristian Kemell, and 1 others. 2025. Generative artificial intelligence for software engineering—a research agenda. *Software: Practice and Experience*, 55(11):1806–1843.

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, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Md. Rizwan Parvez, Wasi Uddin 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, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2719–2734. Association for Computational Linguistics.

YIBO Peng, Zora Zhiruo Wang, and Daniel Fried. 2025. Can long-context language models solve repository-level code generation? In *LTI Student Research Symposium 2025*.

Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-context retrieval-augmented language models. *Trans. Assoc. Comput. Linguistics*, 11:1316–1331.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Richard James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. 2024. REPLUG: retrieval-augmented black-box language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 8371–8384. Association for Computational Linguistics.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2025. Longcodezip: Compress long context for code language models. *CoRR*, abs/2510.00446.

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

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023b. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31693–31715. PMLR.

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. *CoRR*, abs/2405.07530.

Dmitry Ustalov, Egor Bogomolov, Alexander Bezzubov, Yaroslav Golubev, Evgeniy Glukhov, Georgii Levtsov, and Vladimir Kovalenko. 2025. Challenge on optimization of context collection for code completion. *CoRR*, abs/2510.04349.

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 *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 1140–1152. IEEE.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Ori Yoran, Tomer Wolfson, Ori Ram, and Jonathan Berant. 2024. Making retrieval-augmented language

models robust to irrelevant context. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

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, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics.

Sheng Zhang, Yifan Ding, Shuquan Lian, Shun Song, and Hui Li. 2025a. 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, Suzhou, China. Association for Computational Linguistics.

Yilin Zhang, Xinran Zhao, Zora Zhiruo Wang, Chenyang Yang, Jiayi Wei, and Tongshuang Wu. 2025b. cAST: Enhancing code retrieval-augmented generation with structural chunking via abstract syntax tree. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 8106–8116, Suzhou, China. Association for Computational Linguistics.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. 2022. Doccoder: Generating code by retrieving and reading docs. *CoRR*, abs/2207.05987.

# A LLM-based Code Dependency Extraction

The prompt we used for LLM-based code dependency extraction is shown in the colorbox below.

**Input Prompt**

**<|im_start|>system**
A conversation between User and Assistant. The user asks a question, and the Assistant solves it. The assistant provides the user with the answer enclosed within <answer> </answer> tags, i.e., <answer> answer here </answer>. Do not show thinking steps, answer directly.
**<|im_end|>**
**<|im_start|>user**
{Query Code} Infer the classes or functions most likely needed to complete the code above. Provide at most two and select the most important ones. Ensure that the selected content appear somewhere in the provided code.
**<|im_end|>**

# B Sparse Structure Similarity

In this section, we present the definitions and formulations of $Sim_{file}, Sim_{cls}, Sim_{func}$ for sparse structure similarity.

**File Name Similarity.** This metric suggests that files with similar names may contain relevant code.

$$Sim_{file} = ES(N_q, N_c) \tag{8}$$

where $N_q$ and $N_c$ are the filenames of the query and candidate code, respectively. $ES$ denotes Edit Similarity, defined as:

$$ES(t_1, t_2) = 1 - \frac{\text{Lev}(t_1, t_2)}{\max(\text{len}(t_1), \text{len}(t_2))} \tag{9}$$

where $\text{Lev}(t_1, t_2)$ is the Levenshtein distance between two strings.

**Class Name Similarity** This metric operates on the premise that code enclosed in similarly named classes is likely relevant.

$$Sim_{cls} = ES(Cls_q, Cls_c) \tag{10}$$

where $Cls_q$ and $Cls_c$ are the names of the enclosing classes for the query and candidate code, respectively.

**Function Name Similarity** Similarly, this metric assumes that code enclosed in similarly named functions may be relevant.

$$Sim_{func} = ES(Func_q, Func_c) \quad (11)$$

where $Func_q$ and $Func_c$ are the names of the enclosing functions for the query and candidate code.

## C Implementation Details

**Hyperparameters.** For code representation, we use the Sentence Transformers framework[4] and employ the embedding of the [CLS] token to represent the code. For Reciprocal Ranking Fusion (RRF), the smoothing constant is set to $k = 60$. The generation output is capped at 48 tokens. The adaptive fusion module is trained for 10 epochs with a learning rate of 1e-3 and a batch size of 32. Generation is performed using vLLM[5] with an 8,000-token input limit and a temperature of 0.

**Architecture of the Fusion Module.** The fusion module is implemented as a multilayer perceptron (MLP) with a hierarchical bottleneck architecture. The network maps the high-dimensional query embedding $\mathbf{q} \in \mathbb{R}^{768}$ to a weight vector $\mathbf{w} \in \mathbb{R}^8$ through a series of non-linear transformations. The network first projects the input to a 256-dimensional latent space. This is followed by two successive hidden layers with dimensions of 128 and 64, respectively. Within each of these three layers, we apply linear transformation, Layer Normalization (LN), Gaussian Error Linear Unit (GELU), and a Dropout layer with a probability of 0.1. Finally, the 64-dimensional intermediate representation is mapped to $\mathbf{w}$ via a linear transformation. A Softmax function is applied to the final output, ensuring that the generated weights $w_i \in [0, 1]$ and $\sum_{i=1}^{K} w_i = 1$. This formulation constrains the output to a probability simplex, which provides a normalized and interpretable representation of the relative contribution of each retrieval feature.

**Hardware and Software.** Experiments are conducted on an Ubuntu server equipped with an Intel Xeon Silver 4314 CPU and a single NVIDIA A6000 GPU.

## D Analysis of Context-dependent Weights

To investigate the behavior of the adaptive fusion module AIRCoder$_a$, we visualize the distribution of context-dependent weights across different benchmarks. Specifically, we apply K-means clustering to the weight vectors obtained from all test samples in CrossCodeEval and RepoEval, respectively. As illustrated in Figure 5, there is a distinct divergence in feature preferences between the two benchmarks. CrossCodeEval primarily allocates higher weights to $Sim_{sparse}$, $Sim_{dense}$, and $DE$, whereas RepoEval exhibits a heavier reliance on $Sim_{sparse}$, $Sim_{dense}$, and $Sim_{struct}$. These observations underscore the necessity of our proposed adaptive weighting and fusion mechanism, as a static or global weighting scheme would fail to capture these fine-grained, sample-specific requirements. Furthermore, this distributional discrepancy explains the limited cross-dataset generalization observed in our experiments: a modulle optimized for the feature importance of one dataset may struggle when the underlying retrieval signals shift significantly in another.

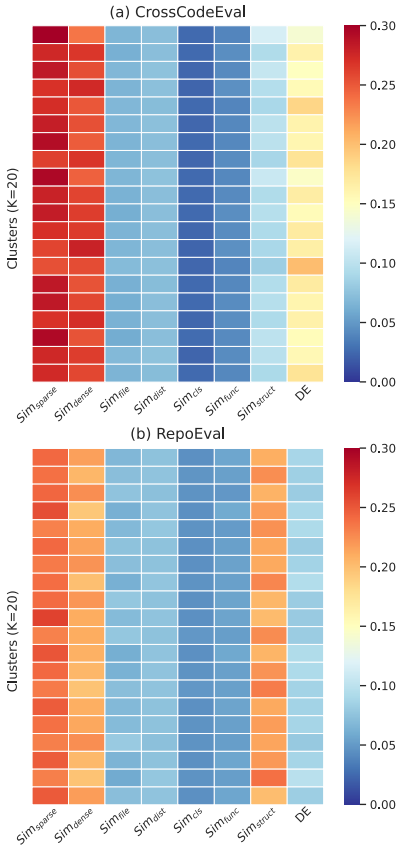


Figure 5: Visualization of clustered dynamic context-dependent weights for CrossCodeEval and RepoEval. Each row represents a cluster center. The distinct patterns across datasets highlight the necessity of context-dependent weights to adaptively integrate multi-dimensional retrieval metrics for each query.

[4] https://huggingface.co/sentence-transformers
[5] https://github.com/vllm-project/vllm