CODEMINER: Budget-Aware Certification of Heterogeneous Evidence for Repository-Level Code Completion

Anonymous ACL submission

Abstract

Repository-level code completion (RLCC) requires retrieving evidence from an entire codebase, yet such evidence is inherently heterogeneous and unevenly reliable, spanning file-level structure, dataflow dependencies, usage patterns, and local context. Existing retrieval-augmented methods often treat retrieved fragments as directly usable prompt context, which may introduce weakly supported or redundant code. We propose CODEMINER, a heterogeneous evidence certification framework for RLCC. CODEMINER builds a candidate evidence graph from complementary repository signals and certifies it through a two-stage graph pruning mechanism. Hard Pruning extracts a dependency-backed evidence backbone via reliable semantic paths to the completion target, while Soft Pruning preserves useful residual local evidence compatible with this backbone under the prompt budget. The resulting certified support context is compact, dependency-centered, and completion-oriented. Experiments on ReccEval and CCEval with multiple code LLMs across different programming languages show that CODEMINER consistently outperforms strong retrieval-augmented baselines.

1 Introduction

Repository-level code completion (RLCC) (Zhang et al., 2025) aims to generate unfinished code by exploiting knowledge distributed across an entire codebase. Unlike function-level completion (Yu et al., 2024), the correct continuation may depend on cross-file definitions, project-specific APIs, type constraints, dataflow relations, and recurring invocation (Li et al., 2025b) patterns that are not visible in the local prefix. Retrieval-augmented generation (RAG) has therefore become a common paradigm for RLCC: repository fragments are retrieved and inserted into the prompt to compensate for the limited context window of large language models (LLMs) (Gao et al., 2023).

However, the evidence required for RLCC is intrinsically multi-granular and heterogeneous (Yang et al., 2024). It is not merely a set of snippets that can be uniformly ranked by similarity, but a collection of complementary signals that serve different support roles for the target completion. File-level structural priors suggest where useful information may reside; definition-side dataflow dependencies explain how symbols and types are derived; usage-side invocation examples reveal project-specific calling conventions (Wang et al., 2025a); and local residual context preserves nearby control-flow or class-level semantics. Because these evidence types differ in granularity, reliability, and utility, none of them alone can fully justify the final prompt context (Liu et al., 2024a). A structurally related file may lack target-supporting code, a dataflow-related (Cheng et al., 2024) symbol may not determine the correct invocation form, and a usage example may provide calling cues without an explicit dependency path to the current prefix.

This heterogeneity exposes a key limitation of existing RAG-based RLCC methods (Zhang et al., 2025). Most prior approaches focus primarily on candidate retrieval: they rank snippets by textual similarity, retrieve dependency-related entities, or combine multiple retrieval signals to enlarge the candidate context (Deng et al., 2024). Although these strategies improve recall, they often treat retrieved fragments as immediately usable prompt content. As a result, the final prompt may mix reliable dependency evidence with superficially similar snippets, redundant local code, weakly supported cross-file fragments, and auxiliary context that consumes the limited prompt budget. In other words, retrieval can propose possible evidence, but it does not by itself certify whether each fragment truly supports the current completion target (Li et al., 2025a).

We argue that the central challenge of RLCC context construction is therefore not only hetero-

geneous evidence retrieval, but heterogeneous evidence certification (Li et al., 2024). Retrieved fragments should be regarded as unverified evidence candidates rather than directly usable context. Before being inserted into the prompt, they should be structurally or locally justified with respect to the completion target (Bi et al., 2024). This view shifts RLCC context construction from flat snippet selection to certified support construction: *the system should determine which heterogeneous candidates form a compact, dependency-centered, and budget-aware support context for generation (Bairi et al., 2024).*

To this end, we propose CODEMINER, a heterogeneous evidence certification framework for RLCC. CODEMINER first constructs a candidate evidence graph from complementary repository signals, including file-level structural priors, definition-side dataflow dependencies, and usage-side invocation examples. Rather than directly inserting these heterogeneous candidates into the prompt, CODEMINER certifies them through a two-stage graph pruning mechanism. Hard Pruning extracts a reliable dependency backbone by retaining candidates supported by explicit semantic dependency paths to the completion target. Soft Pruning further preserves useful residual local evidence that is compatible with this backbone under the prompt budget. Together, they transform noisy heterogeneous retrieval results into a compact, dependency-centered support context for code generation (Wang et al., 2025b).

In summary, our contributions are as follows:

- We formulate RLCC context construction as heterogeneous evidence certification, where retrieved file-level, definition-side, usage-side, and local residual signals are treated as unverified candidates that must be justified before entering the prompt.
- We propose CODEMINER, a support-certified context construction framework that builds a heterogeneous candidate evidence graph from file-level structural priors, dataflow dependencies, and project-specific usage examples.
- We design a two-stage graph pruning mechanism for certified support construction: Hard Pruning derives a dependency-backed evidence backbone, while Soft Pruning retains useful residual local evidence under the prompt budget.
- Experiments on benchmarks ReccEval and CCEval with multiple code LLMs across different programming languages show that CODEMINER consistently improves over strong retrieval-augmented baselines, demonstrating the value of certifying heterogeneous repository evidence before generation.

2 Related Work

2.1 Repository-Level Code Completion

Recent research has shifted from single-file generation (Liu et al., 2024b) to repository-level context modeling, recognizing that code files are intrinsically interdependent. RepoFusion (Shrivastava et al., 2023) fuses multiple file contexts into the prompt to extend the model's receptive field, while CoCoMIC (Ding et al., 2024) models the joint probability of intra-file and cross-file contexts to improve locality awareness. Beyond static context fusion, RepoFormer (Wu et al., 2024) learns when to retrieve external context, ProCC (Tan et al., 2026) uses contextual bandits to select multi-perspective prompts, and RLCoder (Wang et al., 2025c) optimizes prompt selection through reinforcement learning. These methods improve repository-level context utilization, but they mainly focus on how to retrieve or select candidate contexts. In contrast, CODEMINER emphasizes how heterogeneous retrieved evidence should be certified before prompt construction, using file-level structural priors and dependency-centered pruning to build a compact support context for completion.

2.2 Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) has been widely adopted to provide repository context for code completion. RepoCoder (Zhang et al., 2023) iteratively refines retrieval with intermediate generations, RepoFuse (Liang et al., 2024) fuses analogy- and rationale-based snippets, and ReACC (Lu et al., 2022) combines dense and sparse retrieval to select relevant contexts. To incorporate program structure, GraphCoder (Liu et al., 2024c) builds code knowledge graphs over data and control dependencies, while DraCo (Cheng et al., 2024) uses dataflow analysis to improve variable-level consistency. Although these methods improve candidate retrieval, the retrieved evidence is still not necessarily prompt-ready: text-based retrievers may return semantically similar but completion-irrelevant snippets, and graph- or DFG-based retrievers may

preserve excessive dependency candidates without assessing their actual utility. In contrast, CODEMINER treats retrieved fragments as heterogeneous evidence candidates and certifies them through dependency-centered graph pruning before prompt construction.

3 Our Proposed CODEMINER

3.1 Overview

As shown in Fig. 1, CODEMINER constructs repository context through heterogeneous evidence certification rather than directly inserting retrieved snippets into the prompt. Given an incomplete query x_q, repository signals from different granularities first propose a heterogeneous candidate evidence graph. These signals include file-level structural priors, definition-side dataflow dependencies, and usage-side invocation examples. Since such candidates differ in reliability and utility, CODEMINER further certifies them through a two-stage graph pruning mechanism before prompt construction.

Overall, CODEMINER transforms repository context construction from flat snippet retrieval into certified support graph construction:

$$x_q \rightarrow \mathcal{G}_{\text{cand}} \xrightarrow{\text{Hard}} \mathcal{G}_{\text{backbone}} \xrightarrow{\text{Soft}} \mathcal{G}_{\text{sup}} \rightarrow \mathcal{C}^*, \quad (1)$$

where $\mathcal{G}_{\text{cand}}$ denotes the heterogeneous candidate evidence graph, $\mathcal{G}_{\text{backbone}}$ is the dependency-certified evidence backbone obtained by Hard Pruning, $\mathcal{G}_{\text{sup}}$ is the final certified support graph after Soft Pruning, and $\mathcal{C}^*$ is the prompt context serialized from $\mathcal{G}_{\text{sup}}$.

3.2 File-Level Retrieval

This module serves as the coarse-grained first stage of CODEMINER, providing a file-level structural prior for the following retrieval and pruning steps. Instead of searching over all repository snippets, CODEMINER models structural relations among files and uses Personalized PageRank to rank candidate files conditioned on the current prefix. This reduces text-only retrieval noise and yields a compact file scope, from which semantically related snippets are extracted for later dataflow-guided retrieval and DFG pruning.

3.2.1 Prior-Regularized File Graph Construction

We model repository-level navigation with a directed file graph:

$$G_f = (V_f, E_f), \quad (2)$$

where each node $v_i \in V_f$ corresponds to a file f_i, and each directed edge $(v_i, v_j) \in E_f$ indicates that f_i may require definitions, types, or implementation details from f_j. This graph provides a file-level abstraction of cross-file information flow.

We construct E_f as a set of weighted directed edges from static repository relations, including explicit imports, inheritance, export shortcuts, IDF-filtered use-to-definition links, co-reference signals, and lightweight locality priors, with detailed construction rules provided in Appendix A.

To reduce noisy links induced by frequent identifiers, symbol-based relations are weighted by file-level inverse frequency. Specifically, for a symbol-supported relation from f_i to f_j, we compute:

$$w_{ij}^{\text{sym}} = \sum_{s \in \mathcal{S}_{ij}} \log \frac{|\mathcal{R}| + 1}{|\{f_k : s \in f_k\}| + 1}, \quad (3)$$

where $\mathcal{S}_{ij}$ denotes the symbols supporting the cross-file relation. Thus, rare symbols contribute more to cross-file edges, while generic names have limited impact.

We further introduce a repository-adaptive and bounded file prior $\pi(v_i)$ to encode stable repository-level signals, while deferring its implementation details to Appendix B.

The prior regularizes transition strengths without overriding explicit structural relations:

$$\tilde{\mathbf{A}}_{ij} = \mathbf{A}_{ij}\pi(v_j), \quad \mathbf{P}_{ij} = \frac{\tilde{\mathbf{A}}_{ij}}{\sum_k \tilde{\mathbf{A}}_{ik} + \epsilon}, \quad (4)$$

where $\mathbf{P}$ is the row-normalized transition matrix used for reachability propagation.

3.2.2 File Scope Selection via Graph Propagation

Given the incomplete prefix x_q, CODEMINER extracts lightweight query cues from imported modules, visible identifiers, object prefixes, partial API calls, and the current file path. These cues are used to initialize a personalization vector $\mathbf{q}$ over the file graph. To avoid repeatedly selecting the unfinished local file f_q, transitions to f_q are downweighted, yielding $\hat{\mathbf{P}}$.

We then compute prefix-conditioned file scores with Personalized PageRank:

$$\mathbf{r}^{(t+1)} = \alpha\mathbf{q} + (1-\alpha)\hat{\mathbf{P}}^\top \mathbf{r}^{(t)}, \quad (5)$$

where α is the restart probability. After convergence, r_i measures the structural relevance of file

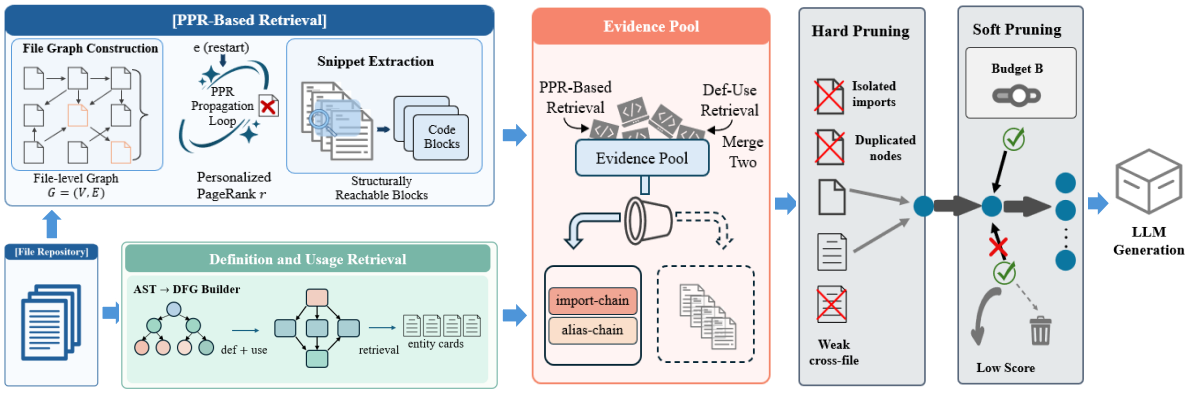

Figure 1: **Overview of CODEMINER.**

f_i to the current prefix. The file scope is selected as:

$$\mathcal{F}_{\text{scope}} = \text{TopK}_{f_i \in \mathcal{R}}\, r_i. \tag{6}$$

The selected scope provides a compact search region for the subsequent dependency-constrained evidence mining stage.

After convergence, the top-ranked files define the candidate scope. To avoid using whole files as coarse context, CODEMINER uses BM25 to extract relevant semantic snippets from these files and scores each snippet by combining its BM25 relevance to the prefix with the PageRank score of its source file.

3.3 Definition and Usage Retrieval

Although file-level retrieval localizes a structurally relevant scope, the resulting snippets are not necessarily dependency-relevant to the completion target. CODEMINER therefore follows DraCo to construct repository-level dataflow graphs and mine dependency-grounded context.

DFG-based retrieval is effective for capturing definition-side dependencies, but it is less capable of revealing how a function or method is actually invoked within the project. To complement such definition-oriented evidence, CODEMINER further performs callee usage retrieval. Given the incomplete query x_q, we extract the target callee symbol τ_q from the unfinished statement and collect repository snippets that invoke the same callee:

$$\mathcal{U}(\tau_q) = \{s_i \in \mathcal{S} \mid \text{callee}(s_i) = \tau_q\}. \tag{7}$$

Each usage snippet is ranked by its semantic compatibility with the local query context:

$$\mathcal{U}^* = \text{TopK}_{s_i \in \mathcal{U}(\tau_q)}\, \text{sim}\left(\phi(x_q), \phi(s_i)\right), \tag{8}$$

where $\mathcal{S}$ is the repository snippet set, $\phi(\cdot)$ denotes the code encoder, and $\text{sim}(\cdot,\cdot)$ is cosine similarity.

Algorithm 1: Hard Pruning for Dependency-Backbone Certification

Input: Candidate evidence graph $G = (V, E)$, completion target nodes T, reliable edge types $\mathcal{E}_r$, hop limit K.
Output: Dependency-certified backbone $G_h = (V_h, E_h)$.

```
V_h ← T, E_h ← ∅, F ← T, Visited ← T;
for i ← 1 to K do
    F' ← ∅;
    foreach v ∈ F do
        foreach incoming edge (u, v, e) ∈ E do
            if type(e) ∈ E_r then
                Add (u, v, e) to E_h;
                if u ∉ Visited then
                    Add u to V_h and F';
                    Add u to Visited;
    F ← F';
Remove duplicated nodes, isolated imports, and
  weakly connected cross-file nodes from V_h;
Remove edges in E_h whose endpoints are not both
  retained in V_h;
Compress non-critical function or class bodies into
  lightweight declarations when possible;
return G_h = (V_h, E_h);
```

The selected usage snippets provide concrete invocation cues, including argument order, keyword arguments, receiver objects, and common calling patterns. Thus, CODEMINER constructs a bidirectional evidence view: definition retrieval explains what the target symbol is, while usage retrieval reveals how it is typically consumed in the repository.

3.4 Support-Certified Graph Pruning

After file-level retrieval and dataflow-guided retrieval, the collected repository evidence still cannot be directly used as prompt context. Although these retrieval stages improve evidence recall, they may also introduce redundant snippets, isolated imports, weakly supported cross-file symbols, and lo-

cally relevant but target-irrelevant code. Such noisy candidates can dilute truly useful dependency information and consume the limited prompt budget.

To address this issue, CODEMINER performs support-certified graph pruning, treating retrieved snippets as unverified evidence candidates rather than immediately usable context. The goal is to construct a compact and completion-oriented support graph by deciding which candidates are sufficiently justified with respect to the completion target. Specifically, Hard Pruning certifies high-confidence evidence supported by reliable semantic dependency paths, while Soft Pruning preserves locally compatible residual evidence whose support outweighs its noise and serialization cost under the prompt budget. In this way, CODEMINER transforms heterogeneous retrieval results into a dependency-centered and budget-aware support context for repository-level code completion.

Algorithm 2: Soft Pruning for Residual Evidence Certification

Input: Candidate evidence graph $G = (V, E)$, hard-pruned dependency backbone $G_h = (V_h, E_h)$, completion target nodes T, local scope Ω, context budget B.
Output: Final certified support graph $G_p = (V_p, E_p)$.

```
V_p ← V_h, E_p ← E_h;
C ← {u ∈ V \ V_h | u ∈ Ω};
foreach u ∈ C do
    Evaluate positive certificates P and negative
      certificates N for u;
    Compute certificate strength Γ(u) by Eq. 9;
Sort C in descending order by Γ(u);
foreach u ∈ C do
    if Cert_S(u) = 1 then
        Add u to V_p;
        Add edges (u, v, e) or (v, u, e) from E to
          E_p where v ∈ V_p;
return G_p = (V_p, E_p);
```

3.4.1 Hard Pruning: Dependency-Backbone Certification

Hard Pruning is the first certification stage, aiming to convert a noisy candidate evidence graph into a reliable dependency backbone. Instead of admitting nodes based on surface relevance, it follows a conservative support criterion: a candidate is hard-certified only if it can justify its relevance to the completion target through reliable semantic dependency paths. Starting from the target nodes, CODEMINER performs bounded reverse propagation along trusted relations, including assignment, aliasing, type association, member-source links, and inheritance. This makes the certified backbone dependency-grounded rather than lexically matched, as detailed in Algorithm 1.

After propagation, CODEMINER further removes weakly supported evidence, such as duplicated nodes, isolated imports, and loosely connected cross-file fragments, and compresses non-critical function or class bodies into lightweight declarations when possible. Thus, Hard Pruning serves as a reliability-oriented certification step: it admits the most defensible dependency evidence, suppresses retrieval-induced noise, and produces a compact backbone for subsequent soft refinement.

3.4.2 Soft Pruning: Budget-Aware Residual Evidence Certification

Soft Pruning complements Hard Pruning by certifying residual evidence that is locally useful but not covered by strict dependency reachability. The hard-pruned backbone provides a high-confidence dependency core, but its conservative nature may exclude nearby control-flow cues, class-level constraints, and project-specific usage patterns that still support the completion target. Therefore, CODEMINER introduces a residual certification criterion to determine whether such candidates are qualified to enter the prompt under the context budget.

For each residual candidate u, CODEMINER evaluates two groups of checkable evidence certificates. The positive certificate set $\mathcal{P}$ captures support signals indicating that u is compatible with the dependency backbone, including dataflow compatibility with retained nodes, structural proximity to the completion target or backbone nodes, and naming consistency with retained symbols or the enclosing scope. The negative certificate set $\mathcal{N}$ captures counter-evidence indicating that u may reduce context quality, including duplicated content, weak or non-semantic connections to the backbone, and high serialization cost relative to its support. We define the residual certificate strength of u as:

$$\Gamma(u) = \sum_{p \in \mathcal{P}} \alpha_p \mathbf{1}[p(u)] - \sum_{n \in \mathcal{N}} \beta_n \mathbf{1}[n(u)] - \lambda \frac{\mathrm{Cost}(u)}{B}, \tag{9}$$

where $\mathbf{1}[\cdot]$ indicates whether a certificate predicate is activated, α_p and β_n denote certificate weights, and λ controls the budget penalty.

A residual candidate enters the final support

graph only if it passes the soft certification gate:

$$\mathrm{Cert}_S(u) = \mathbf{1}\left[\Gamma(u) > 0 \ \wedge \ \mathrm{Cost}(V_p \cup \{u\}) \leq B\right]. \tag{10}$$

This allows CODEMINER to recover useful residual evidence while preventing weak, redundant, or overly expensive fragments from diluting the dependency-centered prompt context.

3.5 Completion Prompt Generation

After support-certified pruning, CODEMINER converts the retained evidence into the final prompt context under the given token budget. Evidence from the dependency backbone is placed first because it provides the most reliable semantic support, while locally useful residual evidence is appended only when it remains compatible with the backbone and does not exceed the budget. The constructed context is then combined with the local prefix and used as input to the code LLM for completion.

4 Experiment

4.1 Experimental Setup

4.1.1 Datasets

We evaluate our framework on two benchmarks: ReccEval (Cheng et al., 2024) and CCEval (Ding et al., 2023). ReccEval is a Python-centric benchmark built from real-world PyPI projects, containing 6,461 test cases for evaluating modern-library code completion. CCEval is a repository-level benchmark emphasizing cross-file context, covering multiple programming languages such as Java, C#, and TypeScript. In this work, we use its Python subset with 2,665 test cases to evaluate dependency-aware code completion, while results on the other languages are reported in the appendix.

4.1.2 Metrics

We adopt four complementary metrics to assess different aspects of repository-level code completion: Exact Match (EM), Edit Similarity (ES), Identifier-Level Exact Match (ID.EM), and token-level F1. EM evaluates exact textual correctness, ES reflects partial similarity under edit distance, ID.EM measures whether key identifiers are predicted correctly, and F1 quantifies token-level overlap with the reference completion.

4.1.3 Baselines

For a fair comparison, all experiments are conducted under a unified context window of 2048 tokens. We compare our method with several representative RLCC baselines: Zero-Shot, CCFinder (Ding et al., 2024), RG-1 (Zhang et al., 2023), RepoCoder (Zhang et al., 2023), DraCo (Cheng et al., 2024), RepoFuse (Liang et al., 2024), RepoFormer-3B (Wu et al., 2024), and CodeRAG (Zhang et al., 2025).

4.1.4 Code LLM

The evaluation is conducted using four representative code LLMs with varying sizes as the code generator: SantaCoder-1.1B (Allal et al., 2023), StarCoder2-3B (Lozhkov et al., 2024), Qwen2.5-Coder-7B (Hui et al., 2024), Qwen3-Coder-30B (Yang et al., 2025).

4.2 Overall Performance

Tab. 1 and Tab. 2 present the overall results on ReccEval and CCEval, respectively. Across both benchmarks and all evaluated code LLMs, CODEMINER consistently achieves the best performance, demonstrating the effectiveness of certifying heterogeneous repository evidence before prompt construction. Notably, the improvement remains consistent even against CodeRAG, a strong retrieval-augmented baseline, indicating that merely retrieving relevant fragments is insufficient; certifying dependency-centered support context is crucial for RLCC.

On ReccEval, CODEMINER ranks first under every generator and across all metrics. With Qwen3-Coder-30B, it achieves 52.45 EM and 73.94 ES, outperforming CodeRAG, the strongest existing baseline, and further confirming the effectiveness of the proposed method. This advantage is not restricted to large generators: smaller models such as SantaCoder-1.1B and StarCoder2-3B show the same trend.

A similar trend is observed on CCEval. With Qwen3-Coder-30B, CODEMINER reaches 39.93 EM and 71.96 ES, showing consistent improvements over CodeRAG on both EM and ES. These results suggest that the proposed certification mechanism can produce more reliable prompt context across different repository-level completion settings, rather than overfitting to a specific benchmark or generator.

The consistent gains over CodeRAG are particularly informative. CodeRAG already retrieves relevant repository knowledge, but CODEMINER further verifies which heterogeneous evidence candidates are structurally or locally justified for the cur-

Methods	SantaCoder-1.1B				StarCoder2-3B				Qwen2.5-Coder-7B				Qwen3-Coder-30B			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	6.27	42.22	12.89	30.08	7.86	45.04	14.44	33.34	11.48	47.72	18.37	36.43	14.72	50.00	21.50	38.43
CCFinder	19.08	50.99	26.67	43.31	28.12	58.93	36.44	53.42	28.43	58.95	36.76	53.09	31.62	61.12	39.50	54.89
RG-1	24.07	54.72	31.26	46.29	29.35	59.43	36.76	52.27	33.01	61.55	40.15	54.68	34.69	63.12	41.90	56.14
RepoCoder	26.78	56.59	34.31	49.07	34.27	63.09	42.30	57.39	34.99	62.71	42.38	56.20	38.80	65.52	46.11	59.57
RepoFuse	28.73	57.89	36.50	51.74	33.88	61.96	41.43	56.52	38.24	65.11	45.88	60.11	40.98	67.07	48.88	62.50
DraCo	30.27	59.38	38.97	55.50	36.57	64.31	45.61	61.42	39.99	66.26	48.55	63.41	44.31	69.44	53.29	66.96
CodeRAG	36.17	63.00	44.17	59.64	42.69	68.07	51.34	65.73	47.48	70.82	55.47	68.68	49.53	72.61	57.95	70.60
CODEMINER	**37.18**	**63.17**	**45.16**	**60.32**	**43.86**	**69.11**	**52.48**	**66.94**	**49.54**	**72.19**	**57.52**	**70.57**	**52.45**	**73.94**	**60.38**	**72.91**

Table 1: Results on ReccEval. Bold/underlined numbers indicate the best/second-best results.

	SantaCoder-1.1B				StarCoder2-3B				Qwen2.5-Coder-7B				Qwen3-Coder-30B			
Methods	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	4.35	46.52	10.58	41.88	6.53	48.56	12.91	44.52	11.11	52.19	18.09	48.35	13.55	54.50	20.45	50.78
CCFinder	14.63	43.44	23.08	51.90	21.08	58.11	29.42	57.46	24.80	59.52	33.47	62.61	29.31	64.48	38.84	64.88
RG-1	12.83	53.78	21.99	51.76	17.78	57.74	27.35	56.55	22.51	61.84	32.91	60.68	24.88	63.29	34.93	62.37
RepoCoder	15.12	55.66	24.62	53.78	21.24	60.90	31.56	60.00	25.89	63.65	36.21	63.09	28.71	65.63	39.29	65.21
RepoFuse	17.64	56.52	27.02	55.65	23.34	61.28	33.43	61.16	27.73	64.76	38.39	64.82	30.81	66.67	41.28	66.40
DraCo	19.70	57.17	29.04	56.82	26.68	62.11	36.29	62.75	30.69	65.46	40.64	66.26	34.67	67.69	45.11	68.99
CodeRAG	22.89	59.92	32.42	60.24	30.66	65.46	41.13	66.62	35.20	68.93	45.97	70.47	38.09	70.12	48.67	71.68
CODEMINER	**23.44**	**60.65**	**32.99**	**60.85**	**31.32**	**66.25**	**42.05**	**67.32**	**36.94**	**70.18**	**47.35**	**71.82**	**39.93**	**71.96**	**50.33**	**73.48**

Table 2: Results on CCEval. Bold/underlined numbers indicate the best/second-best results.

Method	EM	ES	ID.EM	F1
RepoFormer-3B (ReccEval$_l$)	12.88	48.21	19.81	36.93
RepoFormer-3B (CCEval$_l$)	8.18	50.19	15.53	46.25
RepoFormer-3B (CCEval$_{lr}$)	25.29	63.45	33.77	61.48
CODEMINER (StarCoder2-3B, CCEval)	**31.32**	**66.25**	**42.05**	**67.32**
CODEMINER (StarCoder2-3B, ReccEval)	**43.86**	**69.11**	**52.48**	**66.94**

Table 3: Performance comparison under a comparable model scale (3B). RepoFormer-3B is evaluated with left-only ("l") and bidirectional ("lr") context inputs, while CODEMINER uses left-only context under the standard RLCC setting.

rent completion target. By extracting a dependency-backed evidence backbone and preserving useful residual local context, CODEMINER provides code LLMs with a more compact and completion-oriented support context. This helps the generator better capture cross-file dependencies, identifier usage, and project-specific invocation patterns.

To further verify that the improvement comes from the context construction framework rather than larger model capacity, we compare CODEMINER with StarCoder2-3B against RepoFormer-3B under a comparable 3B scale. RepoFormer-3B is originally designed to exploit both left and right contexts around the cursor, whereas CODEMINER follows the standard RLCC setting with only left-side context. For a fairer comparison, we additionally evaluate RepoFormer-3B under the left-only setting. As shown in Tab. 3, left-only RepoFormer-3B performs substantially worse than CODEMINER on both ReccEval and CCEval. Even when RepoFormer-3B uses bidirectional context on CCEval, it still lags behind CODEMINER with the same-size StarCoder2-3B generator. These results indicate that the advantage of CODEMINER mainly stems from support-certified repository evidence construction, rather than stronger input assumptions or larger generator capacity.

4.3 Cross-Language Generalizability

We further evaluate CODEMINER on other languages to examine its cross-language generalizability. The detailed experimental results are reported in Appendix C.

4.4 Ablation Study

Tab. 4 presents the ablation results of CODEMINER on ReccEval with Qwen2.5-Coder-7B. Removing either retrieval component consistently degrades performance, confirming the necessity of retrieval-level evidence collection. In particular, removing DFG retrieval causes a larger drop than removing PPR-based retrieval, with EM decreasing from 49.54 to 39.32 and F1 from 70.57 to 61.67. This shows that dataflow-guided retrieval provides crucial dependency evidence, while PPR-based retrieval adds file-level structural guidance.

Disabling support-certified pruning components also leads to consistent performance drops, showing that heterogeneous retrieval alone is insufficient for constructing effective prompt context. Removing the entire support-certified pruning stage causes the largest degradation, indicating that unverified candidates may dilute dependency-critical evidence and waste the prompt budget. Removing Hard Pruning weakens the dependency backbone,

Variation	EM	ES	ID.EM	F1
Full Model	**49.54**	**72.19**	**57.52**	**70.57**
Retrieval-level				
w/o PPR-based Retrieval	44.23	67.27	52.55	65.78
w/o DFG Retrieval	39.32	65.49	47.96	61.67
Pruning-level				
w/o Certified Pruning	38.37	64.95	46.28	60.78
w/o Hard Pruning	42.69	66.60	50.65	63.94
w/o Soft Pruning	46.57	68.75	52.48	64.78

Table 4: Ablation study of CODEMINER on ReccEval using Qwen2.5-Coder-7B.

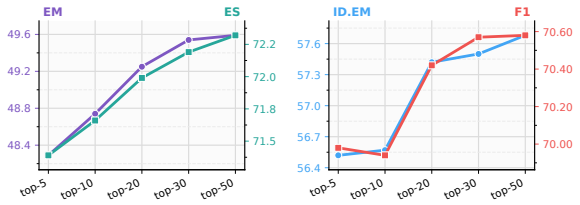

Figure 2: The impact of setting the number of files on ReccEval.

while removing Soft Pruning loses useful residual cues such as local control flow, class-level constraints, and usage patterns. Overall, these results demonstrate that CODEMINER benefits from the joint design of heterogeneous evidence retrieval and support-certified graph pruning.

4.5 Parameter Sensitivity Analysis

Fig. 2 shows the effect of the accessible file scope size in the first-stage Top-Down retrieval on ReccEval. As top-k increases from 5 to 20, all four metrics (EM, ES, ID.EM, and F1) improve consistently, indicating that a larger file scope helps CODEMINER capture more cross-file dependencies and useful contextual evidence. However, after $k = 20$, the performance gains become much slower and gradually plateau around $k = 30$. Therefore, setting the file scope between $k = 20$ and $k = 30$ provides the best balance between dependency recall and context density, while further expansion may introduce irrelevant noise.

We also evaluate CODEMINER under different context window sizes to examine its performance across varying prompt budgets, and the corresponding results are provided in Appendix D.

4.6 Efficiency Analysis

Preprocessing and Prompt Construction Cost. Fig. 3 shows that CODEMINER achieves a favorable efficiency–accuracy trade-off. It takes 0.17s per prompt, faster than CodeRAG (0.23s) while substantially outperforming the fastest baseline DraCo (0.04s) by 9.53 EM points. Offline preprocessing is lightweight and amortizable: on ReccEval, DFG construction takes 46 ms per repository, comparable to DraCo's 42 ms, and file graph construction adds only 40 ms.

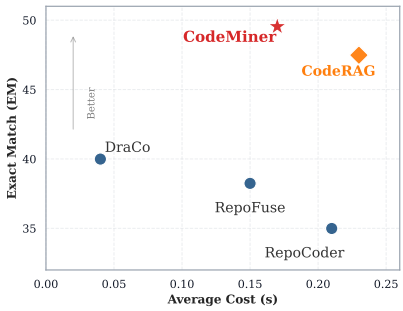

Figure 3: The trade-off between prompt construction efficiency and completion quality.

Memory Cost. CODEMINER incurs only minor storage overhead. The DFG requires about 200 KB per repository, and the file-level PPR component stores a normalized transition matrix of about 10 KB. Both costs scale linearly and can be amortized across repeated completion queries. This lightweight structure preserves dependency-aware evidence and file-level structural priors, enabling denser prompt context with a favorable memory–performance trade-off.

5 Conclusion

In this paper, we presented CODEMINER, a retrieval-augmented RLCC framework built around budget-aware heterogeneous evidence certification. Rather than directly using retrieved fragments as prompt context, CODEMINER constructs a candidate evidence graph from file-level structure, dataflow dependencies, usage patterns, and local context, and then certifies them through dependency-centered pruning. Experiments show that CODEMINER improves repository-level code completion across different code LLMs and benchmarks while maintaining competitive prompt construction cost.

Future work may explore combining static repository evidence with dynamic signals, such as execution feedback, type checking, and runtime traces, to further improve evidence certification for complex and evolving codebases.

Limitations

CODEMINER mainly certifies evidence based on static repository information, such as file structure, dataflow dependencies, and usage patterns. While these signals are effective for RLCC, they may not fully capture dynamic behaviors in some codebases. Exploring runtime-aware evidence certification is left for future work.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, and 22 others. 2023. Santacoder: don't reach for the stars! *arXiv Preprint*. https://arxiv.org/abs/2301.03988.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proc. ACM Softw. Eng.*, 1(FSE):675–698.

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. In *ACL (Findings)*, Findings of ACL, pages 2336–2353.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. In *ACL*, pages 7957–7977.

Ken Deng, Jiaheng Liu, He Zhu, Congnan Liu, Jingxin Li, Jiakai Wang, Peng Zhao, Chenchen Zhang, Yanan Wu, Xueqiao Yin, Yuanxing Zhang, Wenbo Su, Bangyu Xiang, Tiezheng Ge, and Bo Zheng. 2024. R2c2-coder: Enhancing and benchmarking real-world repository-level code completion abilities of code large language models. *arXiv Preprint*. https://arxiv.org/abs/2406.01359.

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 *NeurIPS*, volume 36, pages 46701–46723.

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 *LREC/COLING*, pages 3433–3445.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv Preprint*. https://arxiv.org/abs/2312.10997.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *arXiv Preprint*. https://arxiv.org/abs/2409.12186.

Yanzhou Li, Shangqing Liu, Kangjie Chen, Tianwei Zhang, and Yang Liu. 2025a. Impact-driven context filtering for cross-file code completion. *arXiv Preprint*. https://arxiv.org/abs/2508.05970.

Yifan Li, Ensheng Shi, Dewu Zheng, Kefeng Duan, Jiachi Chen, and Yanlin Wang. 2024. Repomincoder: Improving repository-level code generation based on information loss screening. In *Internetware*.

Zhihao Li, Chuanyi Li, Changan Niu, Ying Yan, Jidong Ge, and Bin Luo. 2025b. Api-repo: Api-centric repository-level code completion. In *Internetware*, pages 449–460.

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. *arXiv Preprint*. https://arxiv.org/abs/2402.14323.

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*. https://arxiv.org/abs/2406.10018.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024b. Repobench: Benchmarking repository-level code auto-completion systems. In *ICLR*.

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*. https://arxiv.org/abs/2406.07003.

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. *arXiv Preprint*. https://arxiv.org/abs/2402.19173.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. In *ACL*, volume 1, pages 6227–6240.

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

Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2026. Prompt-based code completion via multi-retrieval augmented generation. *ACM Trans. Softw. Eng. Methodol.*, 35(1):9:1–9:28.

Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2025a. Teaching code llms to use autocompletion tools in repository-level code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(7):191:1–191:27.

Xingliang Wang, Baoyi Wang, Chen Zhi, Junxiao Han, Xinkui Zhao, Jianwei Yin, and Shuiguang Deng. 2025b. GRACE: graph-guided repository-aware code completion through hierarchical code fusion. *arXiv Preprint*. https://arxiv.org/abs/2509.05980.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025c. Rlcoder: Reinforcement learning for repository-level code completion. In *ICSE*, pages 1140–1152.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. In *ICML*, volume 2183, pages 53270–53290.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 40 others. 2025. Qwen3 technical report. *arXiv Preprint*. https://arxiv.org/abs/2505.09388.

Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiyao Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, Binyuan Hui, and Junyang Lin. 2024. Execrepobench: Multi-level executable code completion evaluation. *arXiv Preprint*. https://arxiv.org/abs/2412.11990.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *ICSE*, pages 37:1–37:12.

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 *EMNLP*, pages 2471–2484.

Sheng Zhang, Yifan Ding, Shuquan Lian, Shun Song, and Hui Li. 2025. Coderag: Finding relevant and necessary knowledge for retrieval-augmented repository-level code completion. In *EMNLP*, pages 23278–23288.

A Construction of the file-level repository graph

This appendix details how we instantiate the weighted file-level edge set E_f used in Sec. 3.2.1. We construct E_f from the static repository relations introduced in the main text: explicit imports, inheritance, export shortcuts, IDF-filtered use-to-definition links, co-reference signals, and lightweight locality priors. For each ordered file pair (v_i, v_j), let $\mathcal{R}_{ij}$ denote the set of detected relations indicating that file f_i may require definitions, types, or implementation details from file f_j. The raw edge weight is computed as

$$\mathbf{A}_{ij} = \sum_{r \in \mathcal{R}_{ij}} \omega(r) \cdot \phi(r, i, j), \tag{11}$$

where $\omega(r)$ is a relation-level reliability weight, and $\phi(r, i, j)$ is the relation-specific evidence strength. Importantly, $\omega(r)$ only reflects the reliability of the relation type and does not include symbol specificity.

We organize relation weights by reliability rather than assigning uniform strength to all static signals:

$$\omega_{\text{explicit}} > \omega_{\text{implicit}} > \omega_{\text{aux}}. \tag{12}$$

Explicit imports, inheritance, and export shortcuts use ω_{explicit} because they directly encode cross-file program dependencies, connecting the using or importing file to the file that provides the referenced definition or type. IDF-filtered use-to-definition links use ω_{implicit} and follow the same direction, from the file containing a repository-specific symbol usage to the file defining that symbol. Co-reference signals and lightweight locality priors use ω_{aux} and serve as weak smoothing evidence when direct dependency evidence is sparse. The concrete values of these weights are selected by searching over a predefined validation range for each weight and choosing the setting that yields the best validation performance.

For symbol-supported relations, we use the same inverse-frequency signal defined in Sec. 3.2.1 to compute the evidence score. To preserve the effect of multiple supporting symbols while avoiding unbounded growth for large files, we use a saturated

Relation type	$\omega(r)$	$\phi(r,i,j)$
Inheritance	25	Eq. 13
Symbol import	15	Eq. 13
Export shortcut	15	1
Module import	4	1
Use-to-definition	10	Eq. 13
Co-reference	5	$\eta(s)/\sqrt{m}$
Same-directory locality	3	$1/\sqrt{m}$

Table 5: Edge-weight settings for file-level graph construction. $\omega(r)$ denotes relation reliability, while $\phi(r,i,j)$ denotes relation-specific evidence strength. $\eta(s)$ is the inverse-frequency score defined in Sec. 3.2.1, and m is the size of the corresponding co-reference or locality group.

aggregation:

$$\phi(r,i,j) = \log(1+|\mathcal{S}_{ij}^r|) \cdot \frac{1}{|\mathcal{S}_{ij}^r|} \sum_{s \in \mathcal{S}_{ij}^r} \eta(s), \quad (13)$$

where $\mathcal{S}_{ij}^r$ is the set of symbols supporting relation r, and $\eta(s)$ is the repository-level inverse-frequency score of symbol s. Thus, multiple project-specific symbols provide stronger evidence than a single symbol, while common identifiers have limited influence. For non-symbol relations, such as directory locality, $\phi(r,i,j)$ is determined by the corresponding structural evidence.

Auxiliary relations are additionally normalized by group size to avoid over-amplifying broad local clusters. If a co-reference or locality group contains m files, each pairwise auxiliary edge is discounted by $1/\sqrt{m}$. This discount allows local signals to improve graph connectivity without overwhelming explicit dependency paths.

The concrete weight settings used in our implementation are summarized in Tab. 5. The table separates relation reliability $\omega(r)$ from evidence strength $\phi(r,i,j)$ to avoid double-counting symbol specificity. These weights are fixed across all experiments and are not tuned on the test set. They encode a coarse reliability ordering: explicit dependency relations receive the largest weights, IDF-filtered implicit relations receive moderate weights, and auxiliary smoothing relations receive lower or group-discounted weights.

When multiple relations connect the same file pair, their contributions are accumulated through Eq. 11. The resulting raw adjacency matrix is then combined with the adaptive file prior and normalized into the transition matrix used for file-level reachability propagation.

B Impact of the adaptive file-prior module $\pi(v_i)$

The adaptive file prior $\pi(v_i)$ is designed to capture stable repository-level usefulness of a file, independent of the current completion query. Instead of treating all files as equally likely transition targets, we compute a lightweight prior from normalized repository-level signals. Specifically, each file v_i is represented by a small feature vector:

$$\mathbf{x}_i = [\bar{h}_i^{\text{def}}, \bar{h}_i^{\text{cent}}], \quad (14)$$

where $\bar{h}_i^{\text{def}}$ denotes the normalized definition signal, and $\bar{h}_i^{\text{cent}}$ denotes the normalized structural centrality signal.

The definition signal is computed from the number of class and function definitions in the file, using logarithmic scaling to reduce file-size bias. The structural centrality signal is computed from the file's incoming structural links relative to the repository average, so the prior adapts to each repository rather than using an absolute global threshold. Both signals are normalized within the repository before being combined.

We then compute a file score with a simple linear scorer:

$$g(v_i) = \mathbf{w}^\top \mathbf{x}_i, \quad (15)$$

where $\mathbf{w}$ assigns small fixed weights to the normalized signals. The score is mapped to a bounded multiplicative prior around 1:

$$\pi(v_i) = 1 + \delta\big(2\sigma(g(v_i)) - 1\big), \quad (16)$$

where $\sigma(\cdot)$ is the sigmoid function and δ controls a small bounded deviation from 1. During graph normalization, the prior is applied as a target-side reweighting:

$$\tilde{\mathbf{A}}_{ij} = \mathbf{A}_{ij}\pi(v_j). \quad (17)$$

Thus, the prior only redistributes probability mass among existing outgoing neighbors and does not introduce new edges. Its bounded form prevents the prior from dominating explicit dependency relations.

To evaluate the contribution of this module, we remove $\pi(v_i)$ from transition construction and compare it with the full model. As shown in Tab. 6, disabling the adaptive file prior consistently degrades all metrics, indicating that repository-level file usefulness provides complementary guidance to explicit dependency evidence.

Setting	EM	ES	ID.EM	F1
with $\pi(v_i)$	49.54	72.19	57.52	70.57
w/o $\pi(v_i)$	47.98	69.35	54.49	68.07

Table 6: Impact of the adaptive file-prior module $\pi(v_i)$ on ReccEval using Qwen2.5-Coder-7B.

C Cross-Language Generalizability

To examine whether the proposed retrieval strategy generalizes beyond Python, we further conduct cross-language experiments on the Java, C#, and TypeScript subsets of CCEval. This setting is particularly important because repository-level code completion may involve different syntax, type systems, dependency conventions, and project organizations across programming languages. CODEMINER is designed to be largely language-agnostic: instead of relying on Python-specific lexical patterns or hand-crafted retrieval rules, it represents repository context through general structural signals, including file-level relations, dependency/dataflow links, and graph-based pruning. These signals can be instantiated across different languages under the same retrieval framework, which allows CODEMINER to be directly applied to multi-language repositories.

For a fair comparison, we report Zero-Shot and DraCo as cross-language baselines, since both methods can be applied without requiring Python-specific preprocessing. Other baselines used in the Python setting depend on Python-specific parsing procedures, repository analysis rules, or implementation assumptions. Directly applying them to Java, C#, and TypeScript would require non-trivial reimplementation and could introduce additional confounding factors; therefore, we exclude them from this cross-language comparison.

As shown in Tab. 7, CODEMINER consistently achieves the best performance on all three languages. Compared with DraCo, CODEMINER improves EM from 26.43 to 29.28 on Java, from 11.68 to 16.68 on C#, and from 20.05 to 23.85 on TypeScript. The corresponding F1 gains are 2.45, 5.56, and 2.17 points, respectively. These improvements are consistent across languages with different programming paradigms and repository organizations, suggesting that the proposed retrieval strategy does not merely exploit Python-specific code patterns. Instead, it can identify useful repository context by leveraging language-independent structural dependencies.

Overall, the cross-language results provide additional evidence for the generality of CODEMINER. The proposed retrieval and pruning strategy is not limited to Python, but can construct effective repository-level context for Java, C#, and TypeScript as well. This indicates that CODEMINER can serve as a robust context construction framework for repository-level code completion in multi-language scenarios.

D Sensitivity to Context Window Length

We further examine the impact of context window length on the performance of CODEMINER on the ReccEval benchmark. Specifically, we use Qwen2.5-Coder-7B as the base model and evaluate it under three context budgets: 2,048, 4,096, and 8,192 tokens. The corresponding analysis is presented in Fig. 4.

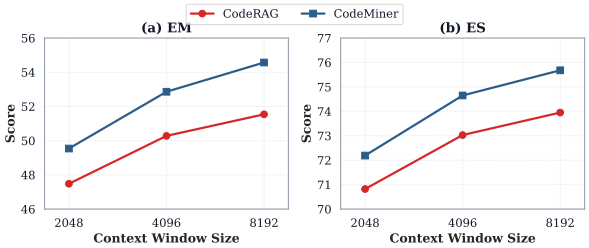

Figure 4: Performance on ReccEval (Python subset) with varying context window size.

The performance of CODEMINER on the ReccEval Python subset consistently improves as the context window size increases from 2,048 to 8,192 tokens. Across both EM and ES metrics, CODEMINER steadily outperforms CodeRAG under all context budgets, demonstrating its stronger ability to utilize extended contextual information. Meanwhile, the improvement from 4,096 to 8,192 tokens becomes relatively more gradual compared with the gain from 2,048 to 4,096 tokens, suggesting that the benefit of increasing the context window tends to saturate at larger budgets.

Overall, CODEMINER is an effective and context-aware method that can better exploit long-context information while maintaining a clear advantage over the strongest baseline. More importantly, as the token budget increases, the advantage of CODEMINER becomes more meaningful because its retrieval and pruning strategy can provide context with higher information density. Instead of simply adding more code snippets into the prompt, CODEMINER prioritizes repository context that is more structurally and semantically relevant to the completion target. Therefore, when more tokens

Methods	Java				C#				TypeScript			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	15.99	60.83	22.49	57.92	5.99	48.00	7.91	36.87	13.75	55.82	17.93	56.96
DraCo	26.43	63.89	34.21	63.38	11.68	52.35	15.83	43.25	20.05	59.78	24.57	60.28
CODEMINER	**29.28**	**65.88**	**35.69**	**65.83**	**16.68**	**56.47**	**22.74**	**48.81**	**23.85**	**62.84**	**28.12**	**62.45**

Table 7: Performance comparison on different language subsets of CCEval using Qwen2.5-Coder-7B.

are available, CODEMINER can include additional useful dependencies and related files while avoiding excessive redundant or noisy content, leading to larger and more reliable improvements.