

Fast and Focused: Interactive Repository-Level Fill-in-the-Middle Code Completion with Hybrid Structural Retrieval

Long S. T. Nguyen^{1,2,3}, Dat T. Huynh^{1,2}, Quynh T. N. Vo^{1,2,3}, Dang Le⁴,
Cong-Duy Nguyen³, Anh Tuan Luu^{5,3}, Anh Nguyen-Duc⁶, Tho T. Quan^{1,2*}

¹URA Research Group, Ho Chi Minh City University of Technology (HCMUT), Vietnam

²Vietnam National University Ho Chi Minh City, Vietnam

³Center for AI Research (CAIR), VinUniversity, Vietnam

⁴Japan Advanced Institute of Science and Technology, Japan

⁵Nanyang Technological University, Singapore

⁶University of South-Eastern Norway, Norway

*Correspondence: qttho@hcmut.edu.vn

Abstract

Large language models are effective at code completion when the required evidence is local, but repository-level fill-in-the-middle completion remains challenging when it depends on code and structural relationships across files. We present ACAR, a framework that combines lexical and AST/LSP-based structural retrieval through line-level reciprocal rank fusion, then applies syntax-aware post-processing to align generated completions with the surrounding code. We evaluate ACAR on RepoEval, ReccEval, and CrossCodeEval across Python, Java, TypeScript, and C#, covering more than 3,000 real-world repositories. In the main Qwen2.5-Coder experiments, ACAR outperforms strong retrieval baselines in most settings, reaching 42.98 EM on ReccEval and 32.86 on CrossCodeEval at 3B, versus 36.25 and 28.40 for the strongest retrieval baseline, respectively. With the largest tested Qwen and DeepSeek backbones, ACAR improves EM over their corresponding no-retrieval baselines across all five evaluation settings. Ablations show that lexical and structural evidence are complementary, while removing syntax-aware refinement causes the largest drop in edit similarity. These gains come with 87–95 ms of non-decoding overhead under warm-cache conditions across the three benchmarks, supporting latency-sensitive code completion. To support reproducibility, we make our implementation and evaluation scripts publicly available at <https://github.com/longstnguyen/ACAR>.

1 Introduction

Large Language Models (LLMs) have become an important component of modern software development, supporting tasks such as code completion, bug fixing, and automated documentation. Developers now work alongside these assistants through

out everyday programming (Barke et al., 2023). Code completion is especially useful in practice: it reduces cognitive overhead and helps developers navigate unfamiliar or evolving codebases. Early LLM-based systems such as CODEX (Chen et al., 2021) and CODEGEN (Nijkamp et al., 2023) perform well when the prediction depends on local context, but they condition only on the current file. In realistic repositories, functionality is distributed across files through shared abstractions, imported modules, and cross-file symbol dependencies, so the evidence a completion needs often lies beyond the file boundary.

Recent work addresses this limitation by retrieving repository context before generation (Zhang et al., 2023, 2025). Approaches range from selective retrieval in REPOFORMER (Wu et al., 2024) to dataflow-guided retrieval in DRACO (Cheng et al., 2024) and graph-based retrieval in GRAPHCODER (Liu et al., 2024), yet two difficulties persist. Lexical similarity alone can surface textually related snippets while missing relevant structural dependencies. Structural retrieval returns spans of varying granularity that are difficult to fit within prompt budgets. More broadly, retrieved context is not always beneficial: irrelevant or noisy evidence can degrade generation quality (Wu et al., 2024).

These challenges are particularly important in *Fill-In-the-Middle* (FIM) completion, where the model infers a missing segment conditioned jointly on prefix and suffix. The inserted code must respect control flow and symbol usage on *both* sides of the insertion point, making the structural relevance of retrieved context especially important. Even with relevant context, generated completions may extend beyond the intended insertion and duplicate existing suffix code. To address these challenges, we introduce ACAR, a novel framework that uni-

fies lexical and structural evidence through line-level fusion for repository-level FIM completion.

ACAR combines fixed-width lexical windows with variable-span structural evidence derived from *Abstract Syntax Tree* (AST) analysis and symbol relations resolved through the *Language Server Protocol* (LSP). Coupling this fused context with syntax-aware refinement addresses both repository evidence selection and insertion consistency in a single FIM pipeline. The design targets *interactive* completion by keeping retrieval *fast* through lightweight operations and context *focused* on evidence relevant to the insertion point. We use locally deployable backbones in the 0.5B–7B range and evaluate retrieval, fusion, and refinement overhead separately from decoding.

Our contributions are summarized as follows.

- We introduce a line-level integration design for lexical and structural retrieval. Snippet- and node-level approaches treat retrieved units as indivisible; ACAR re-ranks fixed-width lexical windows and variable-span structural evidence in a shared line-level space through *Reciprocal Rank Fusion* (RRF), then reconstructs contiguous spans. It can select relevant subspans across symbols, and the fused system outperforms both single-source variants on every evaluated dataset.
- We show that syntax-aware post-processing improves ACAR’s completion quality, with benefits that vary across generation pipelines. Removing it causes the largest drop in *Edit Similarity* (ES) in our ablations, yet applying the same step to REPOFORMER and GRAPH-CODER in the tested DeepSeek settings yields mixed or negative changes.
- We show that hybrid structural retrieval remains lightweight. Under warm-cache conditions, ACAR adds 87–95 ms of non-decoding overhead across the three benchmarks, while structural and learned-retrieval baselines incur roughly 3.5–26 times as much. On aggregate stress tests of 12k–117k files, measured retrieval overhead is 0.77–1.83 s.

We evaluate ACAR on RepoEval (Zhang et al., 2023), ReccEval (Cheng et al., 2024), and Cross-CodeEval (Ding et al., 2023), using four languages and five backbones and comparing against seven lexical, structural, and cross-file baselines. On ReccEval with Qwen2.5-Coder-3B, ACAR reaches an

Exact Match (EM) score of 42.98 and an ES score of 72.46, versus 36.25 and 69.20 for the strongest retrieval baseline.

2 Related Work

We review code completion methods by how they obtain context, from *in-file modeling* to *cross-file retrieval* and *repository-level retrieval*.

In-file context. Pretrained Transformer models (Feng et al., 2020; Ahmad et al., 2021; Ding et al., 2022) underpin modern code intelligence. For completion, autoregressive models (Black et al., 2022) use left context, whereas INCODER (Fried et al., 2023) conditions on both sides of the insertion point through causal-masked infilling. These approaches, however, remain limited to evidence from the active file.

Early cross-file retrieval. Cross-file methods retrieve evidence from neighboring files through similarity matching (Henninger, 1991) or learning over near-duplicate code segments (Hill and Rideout, 2004). This broadens coverage, but scalability and the resolution of distant dependencies remain challenging in large repositories.

Repository-level retrieval. REACC (Lu et al., 2022) established the value of repository retrieval, while REPOCODER (Zhang et al., 2023), CO-COMIC (Ding et al., 2024), and REPOFORMER (Wu et al., 2024) couple retrieval with generation or selective filtering. Structural approaches range from dataflow-guided retrieval in DRACO (Cheng et al., 2024) and code context graphs in GRAPH-CODER (Liu et al., 2024) to repository navigation in REPOGRAPH (Ouyang et al., 2025) and multi-semantic graphs in GRACE (Wang et al., 2025a). CODERAG (Zhang et al., 2025) and RLCODER (Wang et al., 2025b) instead use learned or fused retrieval, adding per-query neural computation that ACAR avoids.

ACAR’s positioning. Broader context improves coverage but sharpens the trade-off between structural expressiveness and prompt efficiency: lightweight retrieval can miss distant dependencies, whereas graph-based methods incur maintenance cost and often retrieve units too coarse for a limited FIM prompt. ACAR’s contribution is its task-specific integration of established retrieval and syntax-analysis tools: line-level fusion reconciles retrieval granularities, while insertion-aware refinement aligns completions with surrounding code. Appendix C compares related methods.

3 The ACAR Framework

The ACAR pipeline comprises (i) repository pre-processing, (ii) context retrieval, (iii) prompt construction and generation, and (iv) *post-processing*. Figure 1 separates an *indexing phase*, which builds lexical and structural retrieval spaces, from an *inference phase*, which fuses evidence before generation and refines the output afterward. Lightweight adaptation to backbone-specific FIM templates keeps the framework model-agnostic.

3.1 Repository Pre-Processing

We formalize repository-level code completion as predicting Y from an incomplete snippet X , whose metadata includes the *prefix*, *suffix*, file identity, and insertion position. ACAR constructs two retrieval spaces: a lexical corpus $C_{\text{repo}} = \{c_1, c_2, \dots, c_N\}$, which stores reusable *lexical context* obtained through *sliding-window* segmentation, and a *repository context graph* $G_{\text{repo}} = (V, E)$. V contains AST-backed entities and source spans; E contains syntax-containment relations and LSP-resolved definition, reference, and cross-file dependency links. Tree-sitter supplies the syntax hierarchy, while LSP provides a lightweight *symbol table* for cross-file relations. The LSP server is initialized once per repository and cached across queries, reducing the cost of repeated inference-time lookups.

3.2 Context Retrieval and Fusion

Given the incomplete snippet X , ACAR queries the lexical corpus and repository context graph independently before fusion.

Text-based retrieval. To identify lexically similar repository snippets, ACAR applies a sliding-window retriever over the lexical corpus using a *Jaccard index*. Let S_t denote the query window from the tail of the current prefix, and S_m a candidate window from a repository file. Lexical similarity is computed as

$$J(S_t, S_m) = \frac{|S_t \cap S_m|}{|S_t \cup S_m|},$$

where both windows are token sets after tokenization. After lightweight normalization, candidate windows are ranked through a retrieval operator $R_{\text{text}}(\cdot)$, producing $C_{\text{text}} = R_{\text{text}}(C_{\text{repo}}, X)$. This branch works well for recovering near-duplicate code, recurring implementation motifs, and local lexical regularities.

Graph-based retrieval. Lexical similarity alone is often insufficient when completion depends on symbol definitions or cross-file dependencies. ACAR therefore adds graph-based retrieval over the repository context graph defined in Section 3.1. During inference, ACAR identifies an anchor node v_0 , defined as the smallest AST node whose span covers the insertion line, and performs bounded *Depth-First Search* (DFS):

$$C_{\text{graph}} = \text{DFS}(v_0, s),$$

where s denotes traversal depth. We use a small bound ($s = 2$ in all experiments) to control retrieval latency. Dependencies beyond depth 2 are not reached by the structural branch, but may still be surfaced by the lexical branch through token overlap and incorporated by RRF fusion.

Line-level fusion. Because lexical retrieval returns fixed-size windows while graph retrieval returns structural spans, ACAR first normalizes both into line-level evidence before fusion. Each retrieved snippet is expanded into line identities of the form `file_path:line_number`. ACAR then applies RRF:

$$\text{RRF}(L) = \sum_{r \in \mathcal{R}(L)} \frac{1}{\text{rank}_r(L) + c},$$

where L is a normalized line identity, $\mathcal{R}$ is the set of ranked outputs from the text and graph retrievers, $\mathcal{R}(L) \subseteq \mathcal{R}$ contains the outputs in which L appears, and c is a damping constant. For $r \in \mathcal{R}(L)$, $\text{rank}_r(L)$ is the best (lowest) rank of a retrieved snippet containing L . Lines retrieved by both branches receive contributions from both rankings, favoring regions supported by lexical and structural evidence. After fusion, ranked lines are reconstructed into spans by grouping consecutive lines from the same file: lines with identical `file_path` and adjacent `line_number` values are merged into contiguous code blocks. Fragmentation is limited in practice because both branches produce multi-line outputs: the Jaccard retriever operates over 50-line windows, and LSP returns whole-symbol bodies. Appendix G reports span statistics.

3.3 Prompt Construction and Generation

The fused *retrieved code snippets* are serialized in ranked order and inserted into a FIM prompt:

$$\text{Prompt} = C_{\text{fused}} + \text{filename} + \text{FIM}(\text{prefix}, \text{suffix}).$$

Here, $+$ denotes prompt concatenation, C_{fused} is the fused repository context, `filename` is the ac-

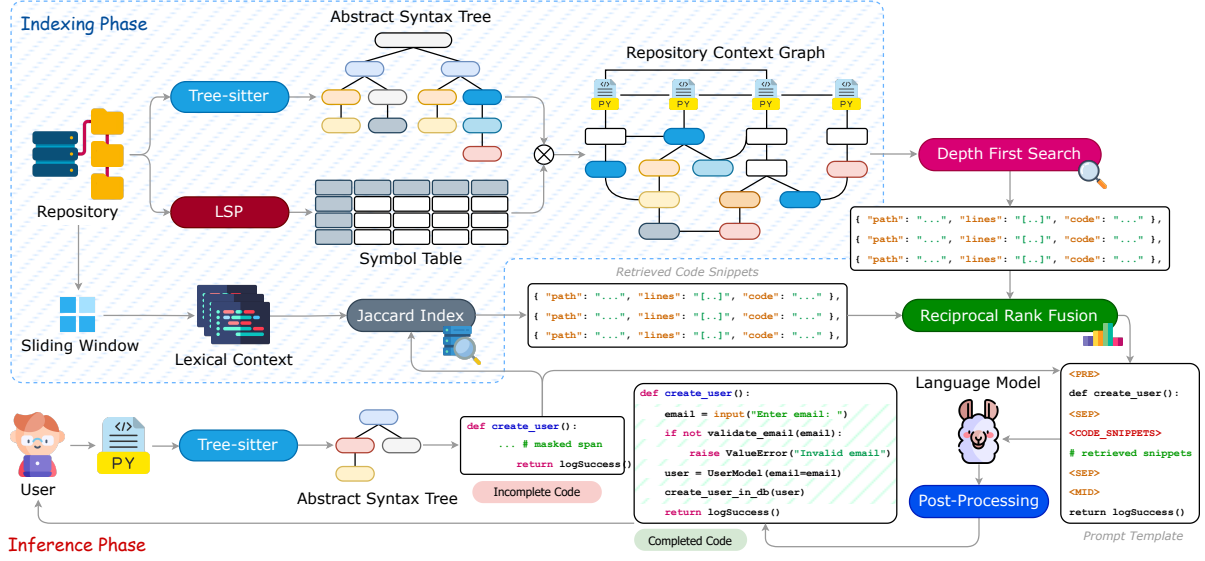


Figure 1: Overview of ACAR’s repository-level FIM pipeline, comprising indexing and inference phases.

tive file identifier, and $\text{FIM}(\cdot, \cdot)$ serializes the local prefix and suffix using the backbone’s native FIM tokens. Retrieved snippets are prepended to this local context within the retrieved-context and total input budgets. The resulting prompt is passed to the *language model* M , producing an initial completion $\hat{Y}_{\text{raw}} = M(\text{Prompt})$.

3.4 Syntax-Aware Post-Processing

A FIM model may generate the correct insertion but also reproduce suffix code, causing the output to fail exact match against the reference completion. After generation, ACAR applies AST-guided truncation using line-aligned ancestor selection, as summarized in Algorithm 1. Starting from the insertion node n_0 , the algorithm traverses upward in the user-side AST to identify the highest ancestor whose span remains aligned with the insertion line, or an adjacent line under open-delimiter conditions represented by $o(\cdot)$. ACAR repairs missing closing brackets before truncation and applies overlap-aware trimming between the candidate insertion and existing code so that the final completion $\hat{Y}$ remains consistent with surrounding syntax.

4 Experiments

We evaluate ACAR through four *Research Questions* (RQs) covering completion quality, efficiency and scalability, generalization, and the contribution of individual components:

RQ1. How does ACAR perform on representative repository-level code completion benchmarks?

Algorithm 1 Line-Aligned Ancestor Selection

Input: Initial node n_0 ; functions $\text{parent}(\cdot)$ and $\text{line}(\cdot)$; predicate $o(\cdot)$

Output: Highest ancestor n satisfying line alignment

```

1:  $n \leftarrow n_0$ 
2: while  $n \neq \text{nil}$  and  $\text{parent}(n) \neq \text{nil}$  do
3:    $p \leftarrow \text{parent}(n)$ 
4:   if  $\text{line}(p) = \text{line}(n_0)$  then
5:      $n \leftarrow p$ 
6:   else if  $\text{line}(p) = \text{line}(n_0) - 1$  and  $o(p)$  then
7:      $n \leftarrow p$ 
8:   else
9:     break
10:  end if
11: end while
12: return  $n$ 

```

RQ2. How efficient and scalable is ACAR under practical development constraints?

RQ3. How well does ACAR generalize across model backbones and programming languages, and where do its gains hold?

RQ4. How do individual components of ACAR contribute to overall performance?

4.1 Experimental Setup

Datasets. We evaluate ACAR on three representative benchmarks covering large-scale, repository-level, and multilingual code completion: ReccEval (Cheng et al., 2024), RepoEval (Zhang et al.,

2023), and CrossCodeEval (Ding et al., 2023). ReccEval consists of real-world Python repositories collected from the Python Package Index; RepoEval evaluates repository-level completion at three granularities: line completion, API invocation, and function-body generation; CrossCodeEval extends evaluation to Python, Java, TypeScript, and C#. Appendix B gives dataset statistics.

Baselines. We compare ACAR against seven repository-level baselines: REPOCODER (Zhang et al., 2023) (iterative lexical retrieval-generation, using one iteration following the original paper), REPOFORMER (Wu et al., 2024) (selective retrieval-augmented generation), RLCODER (Wang et al., 2025b), CODERAG (Zhang et al., 2025), DRACO (Cheng et al., 2024) (dataflow-guided structural retrieval), GRAPHCODER (Liu et al., 2024) (graph-based structural retrieval), and CoCoMIC (Ding et al., 2024) (joint in-file and cross-file modeling). Appendix C details these methods and their evaluation protocols. For multilingual evaluation, we added Tree-sitter-based parsing adapters for DRACO on Java, TypeScript, and C#, and for GRAPHCODER on TypeScript and C#, while preserving their retrieval procedures. For parser-dependent baselines, end-to-end parsing failures are excluded from numerator and denominator rather than scored as incorrect. Their scores therefore use method-specific subsets; Appendix D reports the coverage.

Evaluation metrics. We report EM and ES on a 0–100 scale. EM measures exact agreement with the reference, while ES captures near-correct outputs through normalized Levenshtein similarity. Appendix I gives the formal definitions and aggregation conventions.

Implementation details. Methods use the same backbone family and decoding configuration wherever their released interfaces permit, while retaining each baseline’s retrieval pipeline. Our main experiments use Qwen-based code models (Hui et al., 2024). Appendix A reports the hyperparameters, machine configuration, and backbone checkpoints; Appendices C and D describe method-specific protocols and multilingual adaptations. Tables 1–3 report the main comparisons on RepoEval, ReccEval, and CrossCodeEval.

4.2 Completion Quality (RQ1)

Across the three benchmarks, ACAR achieves the strongest overall results among the evaluated

retrieval-augmented methods. Table 1 shows that ACAR leads the RepoEval retrieval baselines in 5 of 6 EM and 5 of 6 ES settings. For function-level completion, ACAR reaches 5.27 and 8.35 EM at 0.5B and 3B, versus 3.83 and 6.14 for REPOFORMER. On ReccEval, Table 2 reports 30.21 and 42.98 EM for ACAR, versus 27.97 and 36.25 for the strongest retrieval baseline. ACAR also leads aggregate CrossCodeEval EM and ES at both scales in Table 3. Section 4.4 examines when retrieval helps relative to standalone generation.

4.3 Efficiency and Scalability (RQ2)

Runtime overhead. We measure the time spent on tokenization, lexical retrieval, LSP queries, graph traversal, RRF, prompt serialization, and post-processing in ACAR under warm-cache conditions. We exclude model decoding to isolate the retrieval-and-refinement pipeline; end-to-end latency also depends on the backbone, prompt length, and deployment hardware. Indexing is a one-time cost reported separately in Figure 2. Figure 4 shows that ACAR adds 87–95 ms of non-decoding overhead across the three benchmarks. Relative to ACAR, the overhead is roughly 3.5 times as high for DRACO, 5.7–6.0 times for GRAPHCODER, 9–26 times for RLCODER and CODERAG, and over 40 times for CoCoMIC. All methods were timed on the machine described in Appendix A, using the same 2,048-token retrieved-context budget.

Repository scalability. Figure 2 reports a separate profiling study on three selected Python repositories of 179–450 files and aggregate corpora of 12k–117k files. These repository-specific measurements complement the benchmark-level overhead reported in Figure 4. The three largest corpora are aggregate stress-test constructions rather than deployment traces: the 12k-file corpus pools 482 ReccEval repositories, the 52k-file corpus combines ten large open-source projects (airflow, pytorch, tensorflow, among others), and the 117k-file corpus extends that construction. Retrieval overhead is about 0.77 s at 12k files and 1.4–1.8 s at 52k–117k files. LSP-based structural retrieval contributes at most 178 ms, so most of the growth comes from the lexical Jaccard scan. For the largest corpus, one-time indexing takes 124.1 s, approximately 1.1 ms per file. Incremental indexing could reduce update costs by re-indexing only files affected by edits. Future work will explore this optimization alongside end-to-end IDE evaluation.

Table 1: RepoEval results and ablations across Qwen2.5-Coder backbones. **Bold** marks the best score and underlining marks the second-best; the same convention applies to Tables 2 and 3. Text-only and graph-only variants include post-processing.

Method	Qwen2.5-Coder-0.5B						Qwen2.5-Coder-3B					
	Line		API		Function		Line		API		Function	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
REPOCODER	24.58	58.56	16.78	48.78	2.12	32.78	38.03	<u>67.43</u>	<u>28.32</u>	56.23	2.32	34.38
REPOFORMER	<u>28.12</u>	52.34	18.11	55.45	<u>3.83</u>	48.92	34.68	64.27	26.82	62.11	<u>6.14</u>	56.20
RLCODER	8.63	34.99	11.44	31.84	0.00	13.42	10.41	33.12	13.11	28.79	0.66	12.21
CODERAG	10.38	42.31	11.13	49.08	0.00	29.75	14.18	46.19	14.17	54.87	0.31	31.63
DRAcO	18.75	42.38	17.69	45.18	2.22	32.17	34.92	55.95	27.95	57.72	2.64	34.83
GRAPHCODER	26.91	50.88	17.94	53.47	2.86	46.73	32.97	61.48	25.86	61.72	5.29	54.61
CoCoMIC	23.51	44.38	<u>18.16</u>	44.30	1.47	30.04	34.48	53.90	28.16	55.23	2.33	34.23
ACAR (text-only)	24.44	53.64	17.43	<u>58.24</u>	3.65	<u>50.60</u>	37.27	63.84	26.44	<u>67.03</u>	6.11	<u>57.80</u>
ACAR (graph-only)	23.74	51.64	16.93	57.74	3.35	50.30	36.77	62.34	25.94	66.53	5.81	57.30
ACAR (w/o post.)	21.38	44.29	14.46	46.51	2.42	48.89	33.62	54.65	22.84	54.47	4.62	54.99
ACAR	33.62	<u>54.65</u>	19.96	59.57	5.27	51.71	<u>37.50</u>	69.52	29.04	68.43	8.35	59.60

Table 2: ReccEval results and ablations across Qwen2.5-Coder backbones.

Method	Qwen2.5-Coder-0.5B		Qwen2.5-Coder-3B	
	EM	ES	EM	ES
REPOCODER	24.57	62.82	30.10	66.94
REPOFORMER	<u>27.97</u>	63.41	36.25	<u>69.20</u>
RLCODER	7.51	48.75	13.76	55.18
CODERAG	13.99	54.50	23.82	63.33
DRAcO	26.25	60.95	33.27	68.38
GRAPHCODER	25.83	<u>62.94</u>	33.71	68.02
CoCoMIC	14.43	56.36	20.85	62.59
ACAR (text-only)	25.51	51.73	<u>39.24</u>	64.82
ACAR (graph-only)	24.81	50.43	38.49	63.37
ACAR (w/o post.)	19.41	39.83	33.99	54.28
ACAR	30.21	61.03	42.98	72.46

4.4 Generalization and Robustness (RQ3)

Backbone generalization. We evaluate ACAR on five 0.5B–7B backbones from two families: Qwen2.5-Coder-0.5B/3B/7B (Hui et al., 2024) and DeepSeek-Coder-1.3B/6.7B (Guo et al., 2024). Figure 3 reports EM gains over each standalone backbone, using a shared y -axis scale for within-family capacity trends. Setting labels abbreviate *RepoEval* (RE), *CrossCodeEval* (CC), *ReccEval* (RC), and *TypeScript* (TS); the suffix names the completion task or, for CrossCodeEval, the language.

ACAR improves EM on all five settings with each family’s largest evaluated backbone. Gains increase with capacity throughout the DeepSeek settings but are not monotonic across Qwen scales. Improvements hold at every tested scale on CC-py, RC-line, and RE-func; standalone generation performs better on short RE-line and RE-api targets with the smaller Qwen backbones. Against retrieval baselines, ACAR leads EM on 4 of 5 settings with DeepSeek-Coder-6.7B, trailing RE-POFORMER only on RE-api. Both methods ex-

Table 3: CrossCodeEval aggregate results and ablations across Qwen2.5-Coder backbones.

Method	Qwen2.5-Coder-0.5B		Qwen2.5-Coder-3B	
	EM	ES	EM	ES
REPOCODER	12.97	63.93	16.85	67.54
REPOFORMER	<u>19.40</u>	<u>66.10</u>	28.40	<u>71.45</u>
RLCODER	7.07	56.23	12.78	62.74
CODERAG	9.71	59.78	16.66	66.20
DRACo	17.61	64.56	22.97	69.75
GRAPHCODER	16.72	65.26	23.94	69.02
CoCoMIC	9.85	60.45	13.49	63.98
ACAR (text-only)	18.34	60.28	<u>30.73</u>	70.87
ACAR (graph-only)	17.64	58.98	30.43	69.37
ACAR (w/o post.)	14.15	51.59	28.59	65.40
ACAR	22.13	67.38	32.86	74.34

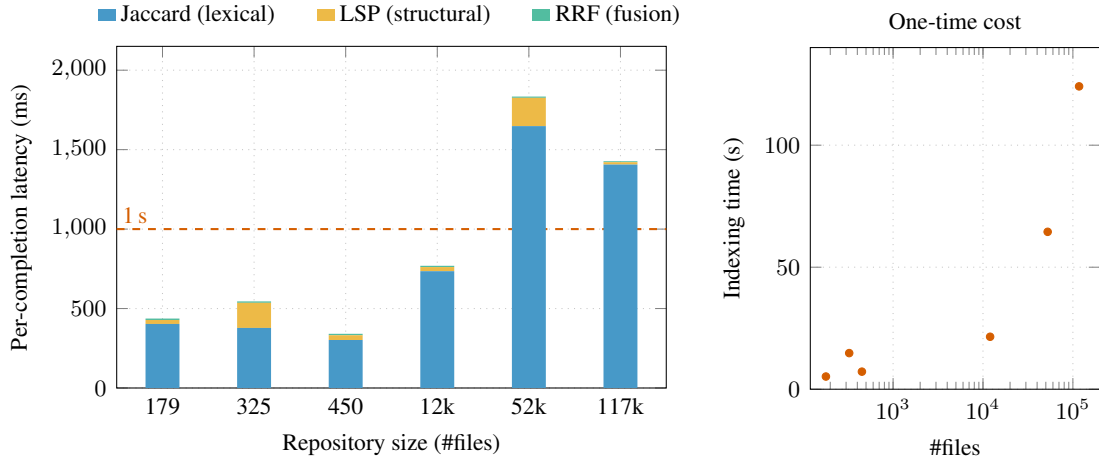


Figure 2: Retrieval latency and indexing cost as repository size grows.

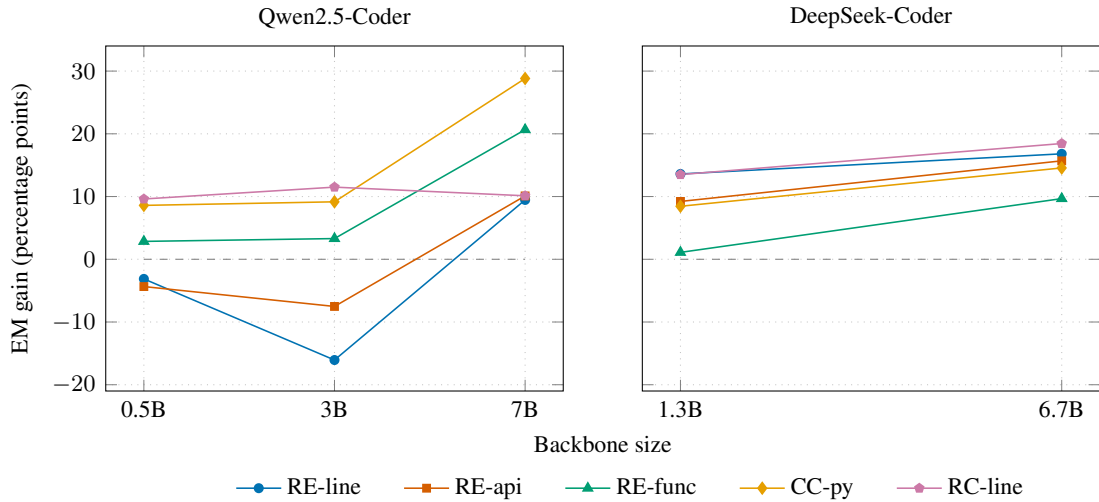


Figure 3: EM gain of ACAR over standalone across model backbones.

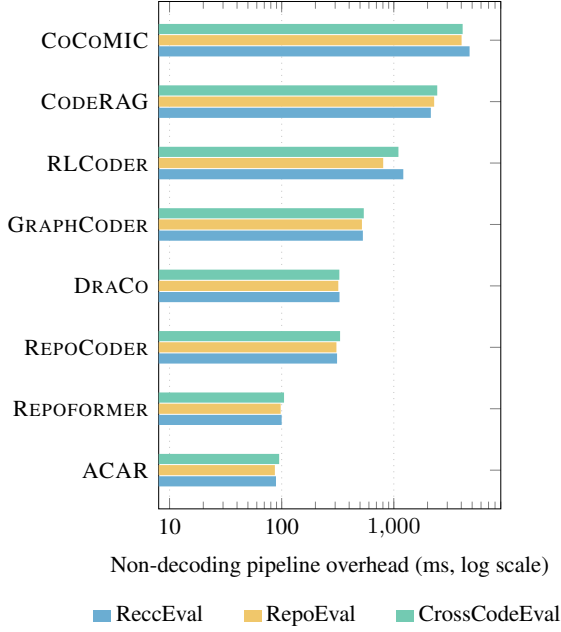


Figure 4: Non-decoding pipeline overhead across three benchmarks (ms, log scale).

ceed standalone EM on all ten DeepSeek settings. Appendix E provides EM/ES comparisons; Appendix H examines retrieved-answer overlap.

Multilingual generalization. In Table 4, ACAR achieves the highest EM and ES across all four CrossCodeEval languages with Qwen2.5-Coder-3B. Its macro-average scores of 32.86 EM and 74.34 ES exceed the strongest retrieval baseline by 4.46 and 2.89 points, respectively. Under Qwen2.5-Coder-0.5B, ACAR leads EM in every language and achieves the best average ES (67.38), although REPOFORMER and REPOCODER lead ES on TypeScript and C#, respectively. The macro-average gains over the strongest retrieval baseline are larger at 3B. ACAR is evaluated at full coverage, while parser-dependent baseline scores use the method-specific subsets reported in Appendix D.

Performance without detected overlap. We examine performance on records with no detected overlap between the reference answer and ACAR’s retrieved context, using the heuristic described in Appendix H. With Qwen2.5-Coder-7B, ACAR’s subset EM scores are 43.89 on ReccEval, 61.56 on RepoEval-line, and 54.55 on CrossCodeEval-Python, versus full-set scores of 44.33, 66.90, and 63.08, respectively. Function-level EM shows the largest gap, from 28.57 to 10.66, suggesting that reusable repository evidence is particularly important in this setting. ACAR’s retrieval output defines the subset, which is not used to compare methods.

4.5 Component Contributions (RQ4)

We ablate three components: text-based retrieval, graph-based retrieval, and post-processing. The ablated variants appear in Tables 1–3. Both text-only and graph-only variants remain below the full system on every benchmark, indicating that lexical and structural signals are complementary. Removing post-processing causes the largest ES decline: from 72.46 to 54.28 on ReccEval, 74.34 to 65.40 on CrossCodeEval, and 69.52 to 54.65 on RepoEval-line under Qwen2.5-Coder-3B. The transfer study in Appendix F finds mixed or negative changes when the same post-processing is applied to REPOFORMER and GRAPHCODER, highlighting its dependence on the generation pipeline. We also examine the structure of the fused context: the fragment analysis in Appendix G shows multi-line spans averaging roughly 20–45 lines, with single-line fragments accounting for only 2.7–6.7%.

5 Conclusion

ACAR integrates lexical and AST/LSP-derived structural retrieval with syntax-aware refinement in a repository-level FIM framework. Line-level fusion combines complementary evidence into contiguous spans, while post-processing refines insertion boundaries. Across RepoEval, ReccEval, and CrossCodeEval, ACAR outperforms retrieval baselines in most settings and improves EM over standalone generation in all five settings with the largest tested Qwen and DeepSeek backbones. Ablations show that both retrieval branches and refinement improve completion quality. With 87–95 ms of warm-cache, non-decoding overhead across benchmarks, ACAR offers a *fast and focused* approach to *interactive* code completion. Smaller Qwen backbones perform better without retrieval on RepoEval line and API tasks, motivating selective retrieval. Further directions include incremental indexing and end-to-end IDE evaluation.

Limitations

Several limitations arise from ACAR’s design for latency-sensitive code completion and the scope of our evaluation with locally deployable backbones.

Parsing and symbol resolution. The coverage of the structural branch depends on the quality of Tree-sitter parsing and LSP symbol resolution. Pre-processor macros, template metaprogramming, and non-standard syntax in auto-generated code may

Table 4: CrossCodeEval results by language under Qwen2.5-Coder-0.5B and 3B. Best scores per metric column are color-highlighted (blue for 0.5B, red for 3B); second-best are underlined in the same color.

Method	Backbone	Python		Java		TypeScript		C#		Avg.	
		EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
REPOCODER	0.5B	12.97	63.93	11.24	62.47	11.89	63.12	15.78	66.20	12.97	63.93
	3B	16.85	67.54	14.93	66.18	15.42	66.83	20.20	69.61	16.85	67.54
REPOFORMER	0.5B	<u>24.83</u>	<u>69.37</u>	<u>15.94</u>	64.08	<u>18.27</u>	<u>65.42</u>	18.56	65.53	<u>19.40</u>	<u>66.10</u>
	3B	<u>34.17</u>	<u>73.08</u>	<u>25.36</u>	<u>70.19</u>	<u>26.08</u>	<u>70.61</u>	27.98	<u>71.93</u>	28.40	<u>71.45</u>
RLCODER	0.5B	3.22	49.01	13.60	61.20	6.63	55.09	4.81	59.60	7.07	56.23
	3B	5.87	56.65	23.66	69.00	11.39	61.11	10.20	64.21	12.78	62.74
CODERAG	0.5B	14.13	61.36	13.85	62.78	5.81	53.87	5.06	61.10	9.71	59.78
	3B	22.90	69.70	22.54	69.99	11.42	60.60	9.79	64.51	16.66	66.20
DRACo	0.5B	20.15	66.42	15.87	63.18	16.23	63.95	18.19	64.69	17.61	64.56
	3B	26.38	71.54	20.65	68.43	21.12	69.28	23.73	69.75	22.97	69.75
GRAPHCODER	0.5B	17.86	66.02	14.91	<u>64.21</u>	16.04	<u>64.88</u>	18.07	<u>65.94</u>	16.72	65.26
	3B	28.71	71.02	22.68	68.94	23.41	69.21	20.95	66.91	23.94	69.02
CoCoMIC	0.5B	11.86	62.34	8.47	59.28	8.92	59.67	10.15	60.51	9.85	60.45
	3B	16.24	65.87	11.58	62.76	12.03	63.14	14.11	64.15	13.49	63.98
ACAR (w/o post.)	0.5B	13.43	44.14	12.16	52.14	11.59	53.44	<u>19.40</u>	56.62	14.15	51.59
	3B	29.76	59.33	25.20	66.23	25.00	67.65	<u>34.39</u>	68.39	<u>28.59</u>	65.40
ACAR	0.5B	<u>28.74</u>	<u>75.56</u>	<u>17.63</u>	<u>65.99</u>	<u>20.67</u>	63.37	<u>21.49</u>	64.61	<u>22.13</u>	<u>67.38</u>
	3B	<u>39.89</u>	<u>75.56</u>	<u>29.31</u>	<u>73.82</u>	<u>26.43</u>	<u>73.33</u>	<u>35.80</u>	<u>74.66</u>	<u>32.86</u>	<u>74.34</u>

yield incomplete ASTs, while limited language-server support may produce sparser symbol tables. Dependencies beyond the depth-2 traversal bound are not reached by this branch. When structural evidence is incomplete, RRF combines it with lexical evidence; when it is unavailable, retrieval falls back to lexical context.

Backbone capacity range. We evaluate locally deployable backbones from 0.5B to 7B; extending the evaluation to frontier-scale code models is left for future work.

Benchmark scope. Our experiments cover three benchmarks, four languages, and over 3,000 repositories. Broader deployment scenarios, including mixed-language projects, complex build systems, and continuous multi-file editing, require further evaluation beyond the current benchmark coverage.

Function-level completion. Function-level EM is lower on the subset without detected answer overlap than on the full set. Both reusable implementation evidence and differences in task difficulty

may contribute to this gap; the comparison does not isolate their effects. Appendix H details the subset construction and results.

Societal Impact

Repository-level completion systems such as ACAR may reduce repetitive work, help developers navigate cross-file dependencies, and improve consistency across large codebases. By combining repository-wide evidence with low retrieval overhead, ACAR aims to make this assistance practical for interactive development. Realizing these benefits requires attention to the risks of propagating vulnerable patterns, exposing sensitive context, and reusing third-party code without adequate attribution. Even plausible completions may contain logical or security defects. Responsible deployment should therefore restrict retrieval to authorized files, avoid retaining sensitive context, and subject generated code to testing, security scanning, provenance and license checks, and human review before production use.

Supplementary Materials Availability

To support reproducibility, the implementation of ACAR, including source code, evaluation scripts, and links to all benchmark datasets used in this study, is publicly available at <https://github.com/longstnguyen/ACAR>.

AI Use Statement

We used OpenAI’s GPT models¹ to improve the manuscript’s language and organization and to check numerical and logical consistency. These tools also assisted with code debugging during experiment reproduction. For all baselines, we used the authors’ official implementations, with multilingual and other evaluation adaptations described in Appendices C and D. We reviewed all AI-assisted writing and code changes against our methods and results and revised them as needed. We take full responsibility for the final manuscript and accompanying code, including their accuracy, originality, and appropriate attribution of prior work.

References

- Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. [Unified Pre-training for Program Understanding and Generation](#). In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.
- Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. [Grounded Copilot: How Programmers Interact with Code-Generating Models](#). *Proc. ACM Program. Lang.*, 7(OOPSLA1).
- Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, Usvsn Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. [GPT-NeoX-20B: An Open-Source Autoregressive Language Model](#). In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 95–136, virtual+Dublin. Association for Computational Linguistics.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 34 others. 2021. [Evaluating Large Language Models Trained on Code](#).
- Wei Cheng, Yuhan Wu, and Wei Hu. 2024. [Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7957–7977, Bangkok, Thailand. Association for Computational Linguistics.
- Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. [Towards Learning \(Dis-\)Similarity of Source Code from Program Contrasts](#). In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6300–6312, Dublin, Ireland. Association for Computational Linguistics.
- Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. [CoCoMIC: Code Completion by Jointly Modeling In-file and Cross-file Context](#). In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3433–3445, Torino, Italia. ELRA and ICCL.
- 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 *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS ’23*, Red Hook, NY, USA. Curran Associates Inc.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. [CodeBERT: A Pre-Trained Model for Programming and Natural Languages](#). In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. [InCoder: A Generative Model for Code Infilling and Synthesis](#). In *The Eleventh International Conference on Learning Representations*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu 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](#).
- Scott Henninger. 1991. [Retrieving software objects in an example-based programming environment](#). In *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR ’91*, page 251–260,

¹<https://openai.com/>

- New York, NY, USA. Association for Computing Machinery.
- Rosco Hill and Joe Rideout. 2004. [Automatic Method Completion](#). In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, page 228–235, USA. IEEE Computer Society.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jia-Jun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Shanghaoran Qian, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report.
- 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 '24, page 570–581, New York, NY, USA. Association for Computing Machinery.
- Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. [ReACC: A Retrieval-Augmented Code Completion Framework](#). In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland. Association for Computational Linguistics.
- 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*.
- Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. [RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph](#). In *The Thirteenth International Conference on Learning Representations*.
- Xingliang Wang, Baoyi Wang, Chen Zhi, Junxiao Han, Xinkui Zhao, Jianwei Yin, and Shuiguang Deng. 2025a. GRACE: Graph-Guided Repository-Aware Code Completion through Hierarchical Code Fusion.
- Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025b. [RLCoder: Reinforcement Learning for Repository-Level Code Completion](#). In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 1140–1152. IEEE Press.
- Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. REPO-FORMER: Selective Retrieval for Repository-Level Code Completion. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. [RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.
- 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 *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 23278–23288, Suzhou, China. Association for Computational Linguistics.

A Implementation Details

We summarize the implementation settings used in all experiments, including inference hyperparameters, prompt construction, hardware environment, and backbone configurations.

ACAR configuration. Table 5 summarizes the decoding and retrieval settings. For FIM completion, decoding stops at predefined special tokens or the maximum generation length. All experiments use unquantized models to avoid introducing differences from low-bit inference.

Table 5: Inference hyperparameters for ACAR.

Decoding	
Input token budget	8,000
Maximum generated tokens	48
Temperature	0.2
Top- k	5
Quantization	None
Retrieval	
Query window S_t (lines)	50
Candidate window S_m (lines)	50
Sliding-window step (lines)	10
Retrieved-context budget (tokens)	2,048
Text retrieval top- k	10
Graph retrieval top- k	10
DFS traversal depth s	2
RRF constant c	60

Prompt templates. Following the FIM formulation in Section 3, we serialize retrieved snippets in ranked order within the token budget and insert them into backbone-specific prompts. Fig-

```

## Qwen2.5-Coder
{intro}
<|file_sep|>{filename}
<|fim_prefix|>{prefix}<|fim_suffix|>{suffix}<|fim_middle|>

## DeepSeek-Coder
{intro}
#{filename}
<|fim_begin|>{prefix}<|fim_hole|>{suffix}<|fim_end|>

```

Figure 5: Backbone-specific FIM prompt templates; intro denotes serialized retrieved context.

ure 5 shows the templates used in our implementation. ACAR uses the Qwen2.5-Coder format in the main experiments and DeepSeek-Coder’s native template for cross-backbone runs.

Hardware and software. All experiments were conducted on a server equipped with an AMD EPYC 7V13 64-core processor (24 CPU cores allocated, 1 thread per core, base clock 2.5 GHz), an NVIDIA A100 GPU with 80 GB memory, and 512 GB of system RAM. The software environment was Ubuntu 20.04.6 (64-bit).

Backbone models. Our main experiments evaluate ACAR and all baselines using two Qwen-based backbones: [Qwen2.5-Coder-0.5B](#) and [Qwen2.5-Coder-3B](#). For cross-scale experiments, we include [Qwen2.5-Coder-7B](#), [DeepSeek-Coder-1.3B-Instruct](#), and [DeepSeek-Coder-6.7B-Instruct](#). All backbones are used in their publicly released form without additional fine-tuning.

B Benchmark Statistics

Table 6 reports the numbers of repositories, files, and completion instances in each evaluation benchmark, along with average prompt and reference lengths where available. We count lines using `splitlines()` and tokens using the Qwen2.5-Coder tokenizer. The eight repositories shared by the three RepoEval tasks are counted only once in the repository total.

ReccEval. This dataset focuses on large-scale Python repositories with dense inter-file dependencies and relatively long prompts ([Cheng et al., 2024](#)), making retrieval quality particularly important when relevant evidence is scattered across repository files.

RepoEval. RepoEval evaluates repository-level completion at three granularities: line-level insertion, API invocation, and function-body generation ([Zhang et al., 2023](#)). The three tasks share eight

repositories but differ in their evaluated files and completion targets.

CrossCodeEval. This benchmark tests generalization across Python, Java, TypeScript, and C#, whose syntax and symbol organization differ ([Ding et al., 2023](#)). Its completion targets are selected to require cross-file evidence.

C Baseline Methods and Protocols

We group the seven retrieval baselines by their main design emphasis, then describe their evaluation settings. The groups can overlap, as some methods combine several retrieval strategies. We also compare against standalone backbones, which generate completions without retrieved repository context. The comparison focuses on systems designed for code completion; REPOGRAPH ([Ouyang et al., 2025](#)) and GRACE ([Wang et al., 2025a](#)) instead target broader repository engineering objectives or require substantially heavier graph construction.

Iterative and selective retrieval. REPOCODER ([Zhang et al., 2023](#)) alternates similarity-based retrieval of repository snippets with code generation. We use its default of one retrieval-generation iteration; Figure 4 reports the corresponding non-decoding overhead. REPOFORMER ([Wu et al., 2024](#)) uses selective retrieval, allowing the model to decide whether external context is likely to improve completion before retrieving it.

Learned and hybrid retrieval. In RLCODER ([Wang et al., 2025b](#)), the retriever is trained with reinforcement learning to select relevant candidates, with a stop signal allowing it to skip retrieval. CODERAG ([Zhang et al., 2025](#)) combines log-probability-guided query construction with multi-path retrieval over lexical, dense, and dataflow signals, followed by a preference-aligned setwise reranker. Like CODERAG, ACAR combines lexical and structural evidence, but uses line-level rank fusion without a learned reranker.

Table 6: Statistics of the evaluation benchmarks.

Dataset	#Repos	#Files	#Examples	Prompt (Avg.)		Reference (Avg.)	
				#Lines	#Tokens	#Lines	#Tokens
ReccEval (Python)	2,635	6,461	6,461	135.9	1,138.0	1.0	6.7
RepoEval (Line)	8	761	1,600	116.0	1,000.6	1.0	11.0
RepoEval (API)	8	456	1,600	130.9	1,139.5	2.1	20.5
RepoEval (Function Body)	8	114	455	121.4	964.0	8.9	83.1
CrossCodeEval (Python)	471	1,368	2,665	90.6	802.2	1.0	10.5
CrossCodeEval (Java)	239	745	2,139	106.7	871.5	1.1	12.5
CrossCodeEval (TypeScript)	193	779	3,356	115.9	864.1	1.7	15.9
CrossCodeEval (C#)	99	642	1,768	71.1	528.2	1.7	12.1
Total	3,645	11,326	20,044	–	–	–	–

Structural retrieval. In DRACO (Cheng et al., 2024), dataflow analysis guides context retrieval over a graph of code entities and their dependencies. Similarly, GRAPHCODER (Liu et al., 2024) performs coarse-to-fine retrieval over its code context graph.

Joint context modeling. CoCoMIC (Ding et al., 2024) uses CCFinder to retrieve related entities from a project context graph. It then jointly attends to retrieved cross-file context and in-file context.

Distinction from graph-fusion methods. Both GRAPHCODER (Liu et al., 2024) and GRACE (Wang et al., 2025a) use whole nodes or snippets as atomic retrieval units. GRAPHCODER ranks and concatenates retrieved snippets, whereas GRACE fuses retrieved subgraphs with the query graph before serialization. ACAR differs in two ways. First, it fuses lexical and structural evidence through *line-level* RRF and reconstructs contiguous spans, supporting subspan selection across multiple symbols rather than treating each retrieved node as indivisible. Second, it combines retrieval with suffix-aware FIM generation and AST-guided post-processing, a combination absent from both methods.

Method-specific settings. Appendix A reports the experimental configuration, and Appendix D details multilingual adaptations and parsing coverage. CODERAG’s published evaluation covers ReccEval and CrossCodeEval. We add a RepoEval loader while retaining its left-context-only conditioning, extending the comparison beyond its published evaluation scope. RLCODER uses its released retriever checkpoint without retraining on

our backbones; its published scope covers line and API completion but not function-level completion.

D Multilingual Baseline Adaptation

DRACO and GRAPHCODER were designed primarily for Python (GRAPHCODER additionally supports Java). For CrossCodeEval, we implemented language-specific adapters for DRACO on Java, TypeScript, and C#, and for GRAPHCODER on TypeScript and C#. Each method’s retrieval procedure, ranking, and prompt template are preserved; the adapters replace the parsing and entity-extraction components, with DRACO’s dataflow modification detailed below. To meet GRAPHCODER’s upstream *tree-sitter*==0.20.4 requirement, its grammars are pinned to *Application Binary Interface* (ABI) version 14. Components not listed in Tables 7 and 8 remain unmodified.

GraphCoder adaptation. We extended GRAPHCODER’s *Code Context Graph* (CCG) construction with new *Control Dependence Graph* (CDG), *Control Flow Graph* (CFG), and *Data Dependence Graph* (DDG) builder functions, as summarized in Tables 7 and 8. These builders share the same node schema across languages. Narrowing the CDG catch-all rule was particularly important for TypeScript, where the original rule would otherwise subsume all nested control-flow nodes.

DraCo adaptation. Table 7 lists each language’s pinned grammar and the constructs the adapter resolves; Table 8 lists modified components. At line and API level, the adapter simplifies intra-method dataflow to focus on import resolution, which dom-

Table 7: Language adapters for the parser-dependent baselines.

Method	Language	Tree-sitter grammar	Language-specific handling
DRACO	Python	unchanged	original parsers re-exported; legacy path preserved bit-identically
	Java	tree-sitter-java 0.23.5	import_declaration; class, interface, enum, record and annotation declarations
	C#	tree-sitter-c-sharp 0.23.5	using_directive; struct, record, delegate
	TypeScript	tree-sitter-typescript 0.23.2	import_statement; index.ts mapped to its directory
GRAPHCODER	C#	tree-sitter-c-sharp @ 6563c3a	–
	TypeScript	tree-sitter-typescript v0.21.2	statement_block excluded from the CDG catch-all rule; a 22-line smoke test yields 7 CDG nodes before the fix and 14 after

Table 8: Implementation touch-points in the two adapted baselines.

Method	Component modified	Change
DRACO	MetadataParser, DataflowParser	replaced with language-specific modules; graph traversal and prompt assembly untouched
	intra-method dataflow	simplified to one-step Hint/Rhint/Assign/Inherit edges
GRAPHCODER	CDG/CFG/DDG builders	new for C# and TypeScript, ≈ 170 lines of code each, using the Java implementation as a template
	CCG node schema	startRow, endRow, sourceLines, defSet, useSet; identical across languages
	driver	file extension table; comment prefix // for C# and TypeScript; --language argument choices

inates the retrieval signal in these settings.

Validation and coverage. We verify valid CDG intervals, non-zero CFG out-degree, and at least 95% file-level parse success; failures concentrate in auto-generated sources (e.g., Migration*.cs, *.d.ts). Retained instance coverage across parser-dependent runs is 75–99%. This differs from file-level parse success because one file may supply multiple completion targets. On CrossCodeEval, DRACO evaluates 2,086/2,665 Python, 2,005/2,139 Java, 1,662/1,768 C#, and 3,323/3,356 TypeScript examples; GRAPHCODER evaluates 2,448/2,665, 2,005/2,139, 1,662/1,768, and 3,323/3,356, respectively. Failed instances are excluded from both numerator and denominator rather than scored as incorrect, so these scores use method-specific subsets. Because coverage and parser adaptations vary by method and language, we do not interpret cross-language score gaps causally.

E Backbone Scaling Results

This appendix presents detailed results for the backbone comparisons summarized in Section 4.4 and Figure 3. We evaluate five backbones from two model families, ranging from 0.5B to 7B parameters. For DeepSeek, we also compare ACAR with

five retrieval baselines. Scores are reported as EM / ES on a 0–100 scale. Bold marks the best score per metric in Tables 10, 11, and 13, the better score within each comparison pair in Table 15, and the highest total in Table 12. Tied best scores receive the same highlighting.

Gains over standalone backbones. Table 9 compares ACAR with standalone generation across all five backbones. ACAR improves EM in all five evaluation settings for Qwen2.5-Coder-7B, DeepSeek-Coder-1.3B, and DeepSeek-Coder-6.7B, covering all 15 comparisons. For Qwen2.5-Coder-7B, these gains range from 9.5 to 28.8 EM points.

DeepSeek baseline comparisons. We compare ACAR with five retrieval baselines on DeepSeek-Coder-1.3B and DeepSeek-Coder-6.7B in Tables 10 and 11. At 6.7B, ACAR ranks first in EM on 4 of 5 settings, with its largest margin over REPOFORMER on RC-line at 11.79 points; REPOFORMER leads only on RE-api by 0.81 points. Results at 1.3B are more mixed: ACAR leads RE-line by 6.04 points and ranks second on CC-py, while DRACO leads CC-py and RC-line but falls below standalone on RE-line and RE-api. Function-level completion remains challenging at this scale, where the best EM is 8.35. ACAR’s RE-func gain

Table 9: Standalone and ACAR performance across five backbones. All entries report EM / ES (%).

Backbone	Method	RE-line	RE-api	RE-func	CC-py	RC-line
Qwen2.5-Coder-0.5B	Standalone	36.75 / 66.75	24.31 / 56.62	2.42 / 23.26	20.15 / 63.37	20.60 / 56.45
	ACAR	33.62 / 54.65	19.96 / 59.57	5.27 / 51.71	28.74 / 75.56	30.21 / 61.03
Qwen2.5-Coder-3B	Standalone	53.56 / 77.88	36.56 / 66.90	5.05 / 31.72	30.73 / 71.70	31.48 / 66.12
	ACAR	37.50 / 69.52	29.04 / 68.43	8.35 / 59.60	39.89 / 75.56	42.98 / 72.46
Qwen2.5-Coder-7B	Standalone	57.44 / 80.18	40.06 / 67.36	7.91 / 36.15	34.26 / 73.94	34.21 / 68.78
	ACAR	66.90 / 78.03	50.16 / 68.92	28.57 / 56.40	63.08 / 78.41	44.33 / 63.80
DeepSeek-Coder-1.3B	Standalone	38.88 / 64.76	26.25 / 54.37	1.76 / 19.43	18.99 / 59.71	21.17 / 55.14
	ACAR	52.48 / 69.84	35.47 / 60.22	2.86 / 42.48	27.44 / 53.47	34.67 / 64.47
DeepSeek-Coder-6.7B	Standalone	45.38 / 71.67	29.06 / 56.79	2.86 / 26.08	27.17 / 68.55	27.24 / 62.37
	ACAR	62.19 / 76.49	44.75 / 66.73	12.53 / 53.10	41.73 / 62.31	45.69 / 73.86

over standalone increases from 1.10 points at 1.3B to 9.67 at 6.7B, indicating greater benefit from retrieval with the larger DeepSeek backbone.

Robustness across DeepSeek scales. Table 12 counts the settings in which each method improves EM over its standalone backbone. Among the evaluated methods, only ACAR and REPOFORMER improve on all five settings at both tested DeepSeek scales, covering all ten comparisons. GRAPHCODER and DRACO fall below standalone in multiple settings, including RE-line and CC-py. At 6.7B, DRACO also trails standalone slightly on RE-api, with 29.00 versus 29.06 EM.

Qwen2.5-Coder-7B baseline comparisons. Table 13 reports 7B results with RLCODER and CODERAG as retrieval baselines. Both baselines use left context without suffix conditioning, limiting direct comparability on function-level FIM. ACAR achieves the highest EM in all five settings and the highest ES in three, while standalone generation achieves higher ES on RE-line and RC-line. ACAR’s largest EM gains over standalone occur on RepoEval-function and CrossCodeEval-Python.

F Post-Processing Transfer

We apply ACAR’s post-processing unchanged to saved raw generations from REPOFORMER and GRAPHCODER, then re-score the outputs without repeating retrieval or inference. These dedicated transfer runs use the DeepSeek family, with raw EM scores within one point of the corresponding baselines on shared settings in Appendix E. Tables 14 and 15 report EM / ES on a 0–100 scale, with deltas expressed in score points.

Transfer effects. Transferring post-processing yields mixed results, as shown in Table 14. Figure 6 plots EM changes on a shared scale, with negative values indicating degradation. Averaged over eight settings, REPOFORMER loses 2.84 EM and 2.18 ES points at 1.3B, and 3.35 EM and 2.34 ES points at 6.7B. For this method, post-processing improves both metrics only on ReccEval, with declines across all RepoEval tasks and CrossCodeEval languages. For GRAPHCODER, average gains are small: 0.08 EM and 0.05 ES points at 1.3B, and 0.19 EM and 0.38 ES points at 6.7B. Both metrics improve on RepoEval-line and the Java and C# subsets of CrossCodeEval but decline elsewhere, indicating that transfer effects depend on the method and evaluation setting.

Comparison with ACAR. Table 15 compares ACAR with GRAPHCODER augmented by the same post-processing step. ACAR achieves higher EM in 4 of 5 settings at 1.3B and all 5 at 6.7B. The exception is function-level completion at 1.3B, where GRAPHCODER with post-processing reaches 8.13 EM versus ACAR’s 2.86.

Interaction with FIM generation. Despite suffix conditioning, a model may repeat suffix code beyond the intended insertion, making an otherwise useful completion fail exact match. ACAR’s post-processing uses AST-guided truncation and overlap trimming to realign the completion with surrounding syntax. Its effect therefore depends on the generated output, which may help explain the mixed transfer results. For comparison, the Qwen2.5-Coder-3B ablations report 28.59 EM on CrossCodeEval and 33.99 on ReccEval for ACAR

Table 10: Baseline comparison on DeepSeek-Coder-1.3B.

Method	RE-line	RE-api	RE-func	CC-py	RC-line
Standalone	38.88 / 64.76	26.25 / 54.37	1.76 / 19.43	18.99 / 59.71	21.17 / 55.14
REPOFORMER	46.44 / 75.36	37.94 / 72.39	8.35 / 54.49	23.64 / 68.07	28.39 / 63.05
RLCODER	14.09 / 44.83	24.79 / 57.75	0.99 / 19.05	18.19 / 66.49	18.05 / 59.35
CODERAG	11.92 / 45.75	11.03 / 51.59	1.24 / 29.45	24.06 / 69.67	24.17 / 63.80
DRACO	16.12 / 29.97	8.12 / 40.83	1.98 / 48.73	29.77 / 72.36	38.55 / 71.87
GRAPHCODER	19.62 / 29.41	31.88 / 59.16	8.35 / 41.35	13.19 / 54.34	28.08 / 57.71
ACAR	52.48 / 69.84	35.47 / 60.22	2.86 / 42.48	27.44 / 53.47	34.67 / 64.47

Table 11: Baseline comparison on DeepSeek-Coder-6.7B.

Method	RE-line	RE-api	RE-func	CC-py	RC-line
Standalone	45.38 / 71.67	29.06 / 56.79	2.86 / 26.08	27.17 / 68.55	27.24 / 62.37
REPOFORMER	55.56 / 80.24	45.56 / 78.24	10.11 / 57.47	32.12 / 73.50	33.90 / 67.31
RLCODER	15.86 / 45.06	26.94 / 59.35	0.99 / 22.07	23.06 / 70.56	23.72 / 63.61
CODERAG	13.16 / 48.50	13.26 / 55.34	1.55 / 30.46	29.49 / 73.02	29.07 / 67.36
DRACO	39.50 / 75.13	29.00 / 71.82	3.96 / 49.48	9.62 / 61.16	10.90 / 55.83
GRAPHCODER	17.88 / 26.77	35.38 / 59.38	9.23 / 43.57	17.16 / 58.24	32.34 / 61.63
ACAR	62.19 / 76.49	44.75 / 66.73	12.53 / 53.10	41.73 / 62.31	45.69 / 73.86

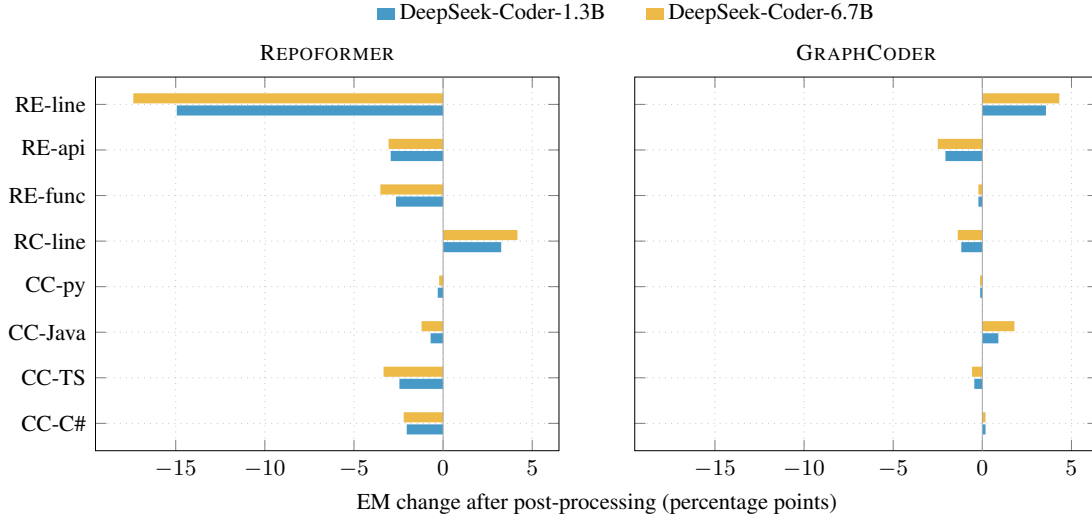


Figure 6: EM change after transferring ACAR’s post-processing to each baseline.

without post-processing, versus REPOFORMER’s 28.40 and 36.25, respectively.

G Retrieved-Context Structure

We analyze retrieved-context structure across all five evaluation settings to check whether line-level fusion yields coherent multi-line spans. For ev-

ery ACAR prompt, we measure the number of spans, their average length in lines, the proportion of single-line spans, and the number of contributing files. We also measure the percentage reduction in span count from an additional merge of neighboring spans from the same file. Only prompts analyzed end to end are counted, so record counts

Table 12: Number of evaluation settings on which each method improves EM over standalone.

Method	DeepSeek-Coder		Total
	1.3B	6.7B	
REPOFORMER	5/5	5/5	10/10
RLCODER	0/5	0/5	0/10
CODERAG	2/5	2/5	4/10
DRACO	3/5	1/5	4/10
GRAPHCODER	3/5	3/5	6/10
ACAR	5/5	5/5	10/10

match or fall slightly below the benchmark sizes in Table 6. They also differ marginally from those in Table 17, where the overlap analysis requires a corresponding completion record.

Table 16 shows that retrieved context consists predominantly of multi-line spans, averaging roughly 20–45 lines across benchmarks. Single-line fragments account for only 2.7–6.7% of spans. An additional merge of same-file spans separated by at most one line would collapse fewer than 2% of spans, indicating little residual fragmentation after the grouping step in Section 3.2. This pattern is consistent with the retrieval design: ACAR decomposes outputs into line-level identities for RRF scoring, but both sliding-window Jaccard retrieval and whole-symbol LSP retrieval supply multi-line context. Their agreement allows RRF to reinforce contiguous line clusters.

H Retrieved-Answer Overlap

We examine how accuracy varies with retrieved-answer overlap by auditing ACAR prompts across all five evaluation settings. We use three overlap signals: (1) exact-substring containment of the reference after whitespace normalization; (2) exact-line match between the reference and a retrieved line for line and API tasks; and (3) high token-level containment, where at least 80% of reference identifiers appear in the retrieved context. A record is *overlap-positive* if any signal triggers and *no-detected-overlap* otherwise. No detected overlap does not rule out analogous implementations elsewhere in the repository. We recompute EM on the no-detected-overlap subset using the main evaluation pipeline. Table 17 and Figure 7 report the results under Qwen2.5-Coder-7B.

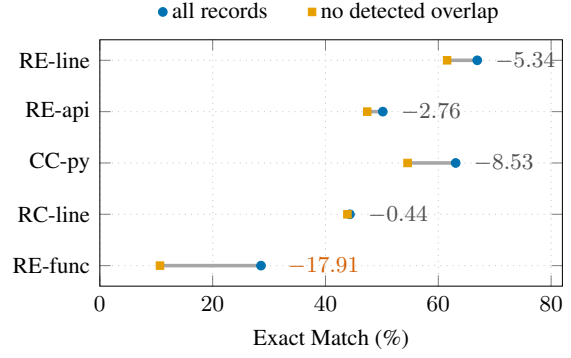


Figure 7: EM on all records and on the subset with no detected retrieved-answer overlap.

Line and API completion. On RepoEval-line, RepoEval-API, ReccEval, and CrossCodeEval-Python, EM on the no-detected-overlap subset differs from full-set EM by 0.4–8.5 points. ReccEval changes little, with 43.89 versus 44.33 EM; subset scores remain at 61.56 on RepoEval-line and 54.55 on CrossCodeEval-Python.

Function-level completion. ACAR scores 10.66 EM on the no-detected-overlap subset of RepoEval-function, versus 28.57 on the full set. This larger gap suggests that reusable repository evidence is particularly important for multi-line synthesis, but differences in task difficulty between subsets may also contribute. ACAR’s retrieval output defines the subset, so comparing methods on it requires re-scoring every method on the same subset.

Overlap and reusable evidence. Overlap rates range from 30% to 57%. In repository-level completion, matching content may come from test files, sibling implementations, or documentation examples that can support code reuse. Reporting overlap rates alongside subset EM shows both how often such evidence is retrieved and how ACAR performs when no overlap is detected.

I Evaluation Metrics

We evaluate repository-level code completion using EM and ES. For each method and setting, N denotes the number of evaluated instances, while $\hat{Y}_i$ and Y_i denote the predicted and reference completions for instance i , respectively.

EM. EM measures the percentage of predictions that exactly match the reference:

$$\text{EM} = \frac{100}{N} \sum_{i=1}^N \mathbf{1}[\hat{Y}_i = Y_i],$$

Table 13: Baseline comparison under Qwen2.5-Coder-7B.

Method	RE-line	RE-api	RE-func	CC-py	RC-line
Standalone	57.44 / 80.18	40.06 / 67.36	7.91 / 36.15	34.26 / 73.94	34.21 / 68.78
RLCODER	16.88 / 46.09	25.27 / 53.95	0.66 / 18.13	6.63 / 57.46	15.35 / 56.88
CODERAG	15.11 / 48.44	17.31 / 59.56	0.31 / 31.91	27.17 / 72.21	28.43 / 66.33
ACAR	66.90 / 78.03	50.16 / 68.92	28.57 / 56.40	63.08 / 78.41	44.33 / 63.80

Table 14: Post-processing transfer results for REPOFORMER and GRAPHCODER.

Setting	DeepSeek-Coder-1.3B			DeepSeek-Coder-6.7B		
	Raw	+ post.	Δ	Raw	+ post.	Δ
REPOFORMER						
RE-line	45.75 / 74.45	30.81 / 70.36	−14.94 / −4.09	55.38 / 79.44	38.00 / 74.79	−17.38 / −4.65
RE-api	38.19 / 71.58	35.25 / 68.71	−2.94 / −2.87	45.81 / 77.91	42.75 / 74.57	−3.06 / −3.34
RE-func	8.79 / 58.50	6.15 / 56.33	−2.64 / −2.17	10.99 / 61.43	7.47 / 58.47	−3.52 / −2.96
RC-line	28.39 / 63.05	31.65 / 66.62	+3.26 / +3.57	33.90 / 67.31	38.07 / 71.75	+4.17 / +4.44
CC-py	23.71 / 69.32	23.41 / 68.18	−0.30 / −1.14	32.42 / 74.87	32.20 / 73.61	−0.22 / −1.26
CC-Java	24.12 / 70.99	23.42 / 67.28	−0.70 / −3.71	34.17 / 77.07	32.96 / 73.55	−1.21 / −3.52
CC-TS	18.33 / 63.97	15.88 / 60.20	−2.45 / −3.77	26.16 / 70.24	22.82 / 66.13	−3.34 / −4.11
CC-C#	30.60 / 75.30	28.56 / 72.05	−2.04 / −3.25	41.69 / 80.30	39.48 / 76.96	−2.21 / −3.34
GRAPHCODER						
RE-line	19.62 / 29.41	23.19 / 34.83	+3.57 / +5.42	18.06 / 26.92	22.38 / 32.83	+4.32 / +5.91
RE-api	31.88 / 59.16	29.81 / 53.50	−2.07 / −5.66	35.69 / 60.10	33.19 / 53.84	−2.50 / −6.26
RE-func	8.35 / 41.35	8.13 / 38.79	−0.22 / −2.56	8.79 / 43.22	8.57 / 41.25	−0.22 / −1.97
RC-line	28.08 / 57.71	26.90 / 57.38	−1.18 / −0.33	32.17 / 61.58	30.79 / 61.22	−1.38 / −0.36
CC-py	13.19 / 54.34	13.07 / 53.93	−0.12 / −0.41	17.16 / 58.10	17.03 / 57.65	−0.13 / −0.45
CC-Java	8.28 / 35.78	9.18 / 39.45	+0.90 / +3.67	10.12 / 35.65	11.92 / 41.00	+1.80 / +5.35
CC-TS	6.32 / 43.13	5.87 / 42.40	−0.45 / −0.73	8.58 / 47.19	8.00 / 46.36	−0.58 / −0.83
CC-C#	3.67 / 39.80	3.85 / 40.76	+0.18 / +0.96	3.49 / 36.37	3.67 / 37.99	+0.18 / +1.62

Table 15: *GraphCoder* (GC) with ACAR post-processing versus ACAR.

Setting	DeepSeek-Coder-1.3B		DeepSeek-Coder-6.7B	
	GC + post.	ACAR	GC + post.	ACAR
RE-line	23.19 / 34.83	52.48 / 69.84	22.38 / 32.83	62.19 / 76.49
RE-api	29.81 / 53.50	35.47 / 60.22	33.19 / 53.84	44.75 / 66.73
RE-func	8.13 / 38.79	2.86 / 42.48	8.57 / 41.25	12.53 / 53.10
CC-py	13.07 / 53.93	27.44 / 53.47	17.03 / 57.65	41.73 / 62.31
RC-line	26.90 / 57.38	34.67 / 64.47	30.79 / 61.22	45.69 / 73.86

where $\mathbf{1}[\cdot]$ is the indicator function.

distance. For each instance,

ES. ES measures normalized similarity between prediction and reference based on Levenshtein edit

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

Table 16: Span coherence of retrieved context across five evaluation settings.

Dataset	#Records	Spans/prompt	Lines/span	Single-line %	Files/prompt	Merge saves
RE-line	1,595	3.87	26.31	2.7%	~3.6	0.4%
RE-api	1,591	3.76	19.74	3.2%	~3.5	0.2%
RE-func	455	4.38	27.86	4.9%	~4.0	0.5%
CC-py	2,660	2.43	44.98	3.6%	~2.3	1.6%
RC-line	6,368	4.25	30.77	6.7%	~4.0	0.2%

Table 17: Retrieved-answer overlap under Qwen2.5-Coder-7B.

Dataset	n	overlap %	EM (all)	EM (no overlap)	EM Δ
RE-line	1,595	43.0	66.90	61.56	-5.34
RE-api	1,591	30.0	50.16	47.40	-2.76
CC-py	2,660	56.8	63.08	54.55	-8.53
RC-line	6,366	50.0	44.33	43.89	-0.44
RE-func	455	40.2	28.57	10.66	-17.91

where $\text{Lev}(\cdot, \cdot)$ denotes Levenshtein distance and $|\cdot|$ denotes string length. The reported score is the mean similarity scaled by 100:

$$\text{ES} = \frac{100}{N} \sum_{i=1}^N \text{ES}_i.$$

Aggregation and score differences. Both metrics are reported on a 0–100 scale, and differences are absolute score points rather than relative percentage changes. Aggregate CrossCodeEval scores are macro-averages across the four languages, with each language weighted equally.