

# 000 SARA CODER: ORCHESTRATING SEMANTIC AND 001 STRUCTURAL CUES FOR RESOURCE-OPTIMIZED 002 REPOSITORY-LEVEL CODE COMPLETION 003 004

006 **Anonymous authors**

007 Paper under double-blind review

## 011 ABSTRACT

013 Despite Retrieval-Augmented Generation improving code completion, traditional  
014 retrieval methods struggle with information redundancy and a lack of diversity  
015 within limited context windows. To solve this, we propose a resource-optimized  
016 retrieval augmentation method, SARA CODER. It maximizes information diver-  
017 sity and representativeness in a limited context window, significantly boosting  
018 the accuracy and reliability of repository-level code completion. Its core Hier-  
019 archical Feature Optimization module systematically refines candidates by dis-  
020 tilling deep semantic relationships, pruning exact duplicates, assessing structural  
021 similarity with a novel graph-based metric that weighs edits by their topological  
022 importance, and reranking results to maximize both relevance and diversity. Fur-  
023 thermore, an External-Aware Identifier Disambiguator module accurately resolves  
024 cross-file symbol ambiguity via dependency analysis. Extensive experiments on  
025 the challenging CrossCodeEval and RepoEval-Updated benchmarks demonstrate  
026 that SARA CODER outperforms existing baselines across multiple programming  
027 languages and models. Our work proves that systematically refining retrieval re-  
028 sults across multiple dimensions provides a new paradigm for building more ac-  
029 curate and resource-optimized repository-level code completion systems.

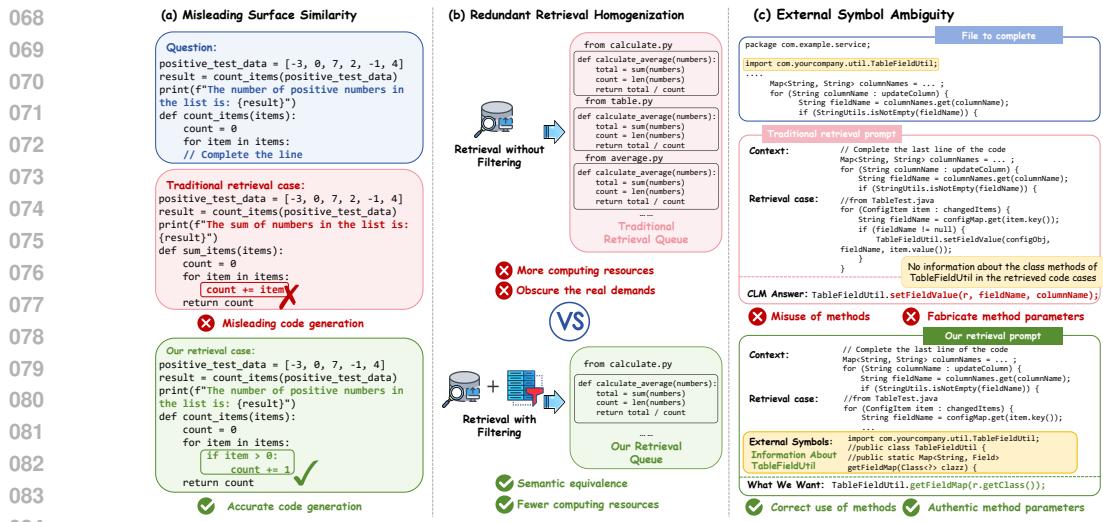
## 030 1 INTRODUCTION

032 Code Large Language Models (Code LLMs) Izadi et al. (2022); Li et al. (2022b); Allal et al. (2023),  
033 built on the Transformer architecture and trained on massive code corpora. These models compress  
034 vast programming knowledge into hundreds of millions of parameters and have been successfully  
035 applied in numerous real-world development scenarios, and have significantly advanced the intelli-  
036 gence of modern software development. However, these models often only process local information  
037 within their context window. As codebases grow and development tasks become increasingly com-  
038 plex, this limitation becomes more pronounced. Tasks like understanding functionality and fixing  
039 bugs often require integrating a wide range of context, such as related API definitions, dependent  
040 modules, and type constraints. Since the models cannot perceive information outside their window,  
041 the suggestions they generate tend to be localized, often insufficient, and inaccurate.

042 Retrieval-Augmented Generation (RAG) Tang et al. (2023); Zan et al. (2022); Zhang et al. (2023)  
043 overcomes the limitations of traditional models by introducing an external knowledge retrieval  
044 mechanism. In this framework, an efficient retriever can dynamically fetch relevant code snippets,  
045 API documentation, or type definitions from an external codebase in real time, thereby expanding  
046 the model’s contextual awareness. The generative model then integrates these retrieval results with  
047 the current context to produce syntactically correct, semantically consistent, and standard-compliant  
048 code suggestions. Retrieval-Augmented Generation is gradually becoming a foundational technol-  
049 ogy for achieving accurate and trustworthy repository-level intelligent code completion.

050 However, current Retrieval-Augmented Generation (RAG) methods for code completion often have  
051 a single-aspect information problem. Methods like GraphCoder Liu et al. (2024b) focus on retriev-  
052 ing similar code snippets, while others like DraCo Cheng et al. (2024) prioritize cross-file context.  
053 This narrow focus overlooks the need to combine different types of information, even though devel-  
opers in real-world settings require both cross-file context for project-wide consistency and similar

054 code snippets for structural reference and efficiency. Limited resources, such as a finite context window,  
 055 make the effective fusion of these different information types a critical challenge. To make  
 056 matters worse, existing similarity-based retrieval methods have three major practical flaws. **(a) Mis-**  
 057 **leading Superficial Similarity:** They often retrieve irrelevant code snippets that share surface-level  
 058 similarities, which can mislead the model and result in incorrect or ineffective suggestions. **(b) Re-**  
 059 **dundant Retrieval Homogenization:** Relying solely on similarity ranking frequently yields redun-  
 060 dant or duplicate snippets. This wastes valuable context window space and provides a homogeneous,  
 061 narrow viewpoint, hindering the model’s ability to generate innovative or superior solutions. **(c) Ex-**  
 062 **ternal Symbol Ambiguity:** These methods fail to capture essential dependencies like unreferenced  
 063 classes or architectural rules. This ambiguity can trigger a cascade of errors, including failed type  
 064 inference and mismatched signatures, compromising the correctness and reliability of the generated  
 065 code. Ultimately, these flaws not only degrade the quality of retrieved information but also exacerbate  
 066 the challenge of a finite context window by filling the limited space with irrelevant, repetitive,  
 067 or ambiguous data, thereby compromising the generation of correct and innovative code.



085 Figure 1: The pitfalls of pure similarity retrieval and the highlights of SARACODER. Pink boxes il-  
 086 lustrate traditional retrieval results based purely on surface similarity, while green boxes demon-  
 087 strate results from our method SARACODER.

088 To address these challenges, we introduce SARACODER, a Semantic-Aware Code Retrieval frame-  
 089 work with **Redundancy Reduction** and **Ambiguity Suppression** for Repository-Level Code Comple-  
 090 tion, that leverages hierarchical resource-optimization to enhance repository-level code completion  
 091 under resource-constrained conditions. By deeply mining relationships in code snippets, remov-  
 092 ing redundancy and optimized reranking, we provide LLMs with richer, higher-quality reference  
 093 cases while minimizing context window consumption. To address misleading superficial simi-  
 094 larity, we utilize semantic alignment distillation to capture deep semantic relationships and a graph-  
 095 based structural similarity metric, which weighs editing operations by topological importance to  
 096 assess the structural proximity of candidates to the target context. To combat redundant retrieval  
 097 homogenization, we integrate MD5-based deduplication pruning and diversity-aware reranking, en-  
 098 suring relevance while maximizing diversity. Additionally, to resolve external symbol ambiguity  
 099 in repository-level code completion, we introduce an external-aware identifier disambiguator that  
 100 analyzes project-level dependencies for LLMs. Our key contributions are:

- 102 • We propose SARACODER, a hierarchical and resource-optimized retrieval-augmented code com-  
 103 pletion framework. SARACODER has redefined its code completion goal from retrieving a large  
 104 volume of relevant code to providing the most valuable information within a limited context.
- 105 • As an intelligent information filter, SARACODER upgrades code completion by moving from  
 106 surface-level code matching to intelligent decision-making based on deep semantics and project  
 107 structure. This provides LLMs with high-quality, precise information, leading to more efficient  
 108 repository-level code completion within limited contexts.

108

- 109 SARACoder’s design allows it to maintain suggestion quality even with a nearly full context  
110 window. This ensures that when complementing other methods that provide different information,  
111 it minimizes negative interference and maximizes the preservation of their content. This synergy  
112 enables it to complement orthogonally other methods that provide cross-file context, providing a  
113 cooperative improvements when used in combination.

114

## 2 RELATED WORK

116 Current Retrieval-Augmented Generation methods for repo-level code completion mainly rely on  
117 code similarity or cross-file dependencies.

### 2.1 SIMILAR CODE SNIPPET RETRIEVAL FOR RAG

121 This approach enhances the quality of code LLM generation by retrieving semantically similar  
122 code snippets and integrating them into prompts, mimicking the reference behaviors of program-  
123 mers. CodeSearchNet Husain et al. (2020) pioneers large-scale code corpora construction, providing  
124 retrieval-based completion references; CodeRetriever Li et al. (2022a) integrates pretrained models  
125 like CodeBERT Feng et al. (2020) to enhance complex scenario handling; ReACC Lu et al. (2022)  
126 combines vector and string-based retrieval to significantly optimize long-code processing; Graph-  
127 Coder Liu et al. (2024b) improves code completion by using program dependencies for structured  
128 representations, allowing coarse-to-fine retrieval for Python and Java. However, like many other  
129 methods, its dependency analysis does not fully grasp deep semantic relationships in code. Addi-  
130 tionally, most approaches rely too much on surface-level textual similarity. This often results in  
131 redundant retrieved content, wasting resources.

### 2.2 CROSS-FILE DEPENDENCY RETRIEVAL AUGMENTATION

132 This method approaches code completion in complex repositories by leveraging cross-file code con-  
133 text (e.g., dependencies, dataflow, and subsequent similar code). Inspired by Ding et al.’s Ding et al.  
134 (2023) observation that subsequent content of high-similarity snippets effectively informs comple-  
135 tion, it injects these snippets into prompts. COCOMIC Ding et al. (2024) dynamically fuses the con-  
136 text of this file with the cross-file entities retrieved by CCFinder (compressed into [SUM] vectors)  
137 through a joint attention mechanism to achieve location-aware code completion. DraCo Cheng et al.  
138 (2024) extends this paradigm through dataflow-guided retrieval, parsing private repositories into  
139 code entities and constructing a repository-specific context graph reflecting dependencies. DraCo re-  
140 trievals precise contextual knowledge from this graph to generate well-structured prompts, overcom-  
141 ing cross-file information barriers and repository-specific accuracy gaps. Current limitations include  
142 Python-exclusive implementation with type-sensitive dependencies lacking multilingual support.

### 2.3 REPOSITORY-LEVEL CODE COMPLETION EVALUATION

145 Traditional code completion benchmarks like Chen et al. (2021); Austin et al. (2021) focus on iso-  
146 lated snippets, but modern software development’s complexity demands a better evaluation. To  
147 address this, specialized benchmarks such as RepoEval Zhang et al. (2023), CrossCodeEval Ding  
148 et al. (2023), RepoBench Liu et al. (2024a), and ReccEval Cheng et al. (2024) have emerged. They  
149 provide standardized, rigorous tests across various languages and project scales. These benchmarks  
150 divide repository-level code completion into two scenarios. **(1) In-File Completion:** A high-  
151 frequency task that uses only the current file’s context (e.g., RepoEval). **(2) Cross-File Comple-  
152 tion:** A more complex task that requires understanding and completing code with dependencies on  
153 symbols from other files (e.g., CrossCodeEval and ReccEval). This shift highlights the move from  
154 simple, isolated tests to comprehensive evaluations that reflect real-world coding environments.

## 3 METHOD

157 As shown in Figure 2, SARACoder is a **hierarchical feature-optimized retrieval-enhanced code**  
158 **completion framework**. Formally, given a code context  $C_{context} = \{x_1, x_2, \dots, x_n\}$  and its con-  
159 taining file path  $F$ , the task aims to predict the next statement  $\hat{y}$ .

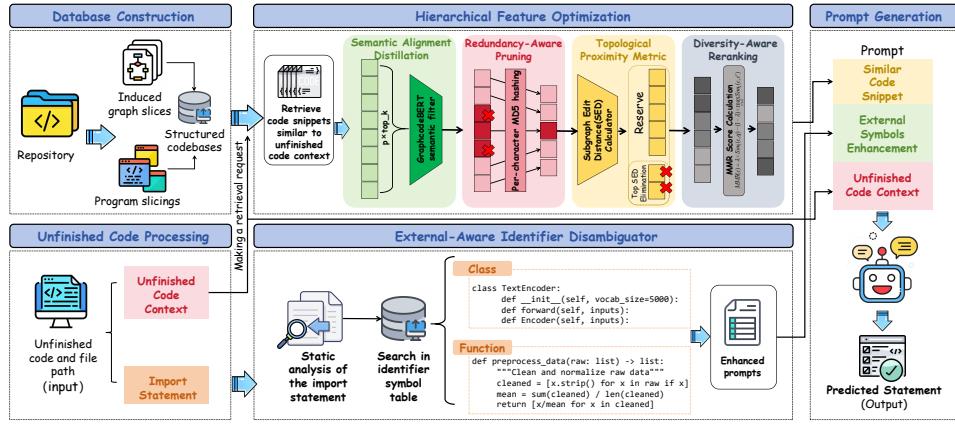


Figure 2: An illustration of SARACODER framework. **(1) Database Construction.** This phase constructs a key-value codebase. This involves using a slicing algorithm to create induced graph slices, which are then precisely mapped to source code snippets. **(2) Code Retrieval.** This phase takes code context as input and retrieves similar code, then refines suggestions via Hierarchical Feature Optimization. Concurrently, an External-Aware Identifier Disambiguator clarifies external symbols via dependency analysis, delivering highly accurate candidates. **(3) Code Generation.** This phase generates prompts by integrating outputs from code retrieval with the code completion context. These prompts are then fed into an LLM to predict completion statements.

### 3.1 DATABASE AND INITIAL CANDIDATE CONSTRUCTION

To better represent code logic, we introduce a multi-level code context graph model that integrates control flow, data dependency, and control dependency Liu et al. (2024b). This structured representation offers enhanced generalization capabilities compared to serialization methods, enabling more effective capture of task-relevant context and facilitating easier adaptation to other languages. We utilize program slicing to generate precisely mapped, task-relevant subgraphs from source code on-demand, constructing structured codebases tailored to support specific analysis tasks. When a code completion task request occurs, we extract the Unfinished Code Context  $C_{\text{context}}$  and the Import Statements  $I$  from the code file.  $C_{\text{context}}$  is then used to retrieve an initial candidate set of  $\text{top\_}k \times p$ <sup>1</sup> code snippets  $C$  via text similarity from structured codebases.

### 3.2 HIERARCHICAL FEATURE OPTIMIZATION (HF-OP)

#### 3.2.1 SEMANTIC ALIGNMENT DISTILLATION

Semantic alignment distillation addresses Superficial Similarity Misguidance by leveraging the GraphcodeBERT Guo et al. (2021), a pretrained model specialized in code understanding, to capture deep semantic relationships between code snippets. First, the query code  $Q$  and candidate set  $C$  are tokenized into subword sequences and uniformly padded or truncated to a fixed length  $L = 512$ . Subsequently, during the feature encoding phase, a 768-dimensional semantic vector  $v_s$  is extracted for each code unit  $s \in Q \cup C$ , with vector space standardized through L2 normalization. When code repositories lack sufficient repetitive or relevant code, standard filtering methods are too strict, often leading to **zero-candidate scenarios**. This scarcity of reference material then hurts the accuracy of large language models. To fix this “one-size-fits-all” problem, we introduce a new dynamic quantile threshold mechanism. During the dynamic filtering phase, the cosine similarity set  $S = \{\cos(v_Q, v_c) \mid c \in C\}$  is computed between the query vector  $v_Q$  and all candidate vectors  $v_c$ . An adaptive threshold  $\tau = \text{quantile}(S, 0.75)$  is set at the 75th percentile, outputting filtered results  $C_{SAD} = \{c \mid \cos(v_Q, v_c) \geq \tau\}$ . To reduce redundant computation overhead and improve efficiency, a caching mechanism stores encoding results for high-frequency code.

<sup>1</sup>To ensure a high-quality final candidate set of  $\text{top\_}k$  results, we expand the initial candidate pool to  $\text{top\_}k \times p$ , allowing more candidates to participate in the Hierarchical Feature Optimization.

216 3.2.2 REDUNDANCY-AWARE PRUNING  
217

218 This module implements lightweight hash-based deduplication via exact text matching. Using the  
219 MD5 algorithm Rivest (1992), it generates 128-bit hash fingerprints (*single computation*  $\approx 0.02\text{ms}$ ,  
220 *memory footprint* 32 bytes/hash) to eliminate verbatim duplicates from candidate set  $C_{SAD}$  with  
221 minimal computational cost, significantly reducing downstream overhead. The module maintains  
222 a global hash set  $H_{\text{seen}}$  to dynamically track processed sample fingerprints: for each candidate  
223  $c \in C_{SAD}$ , if its MD5 hash  $h_c \notin H_{\text{seen}}$ ,  $c$  is added to deduplicated result set  $C_{RAP}$  and  $H_{\text{seen}}$   
224 is updated. This achieves real-time processing with  $O(N)$  time complexity. After semantic align-  
225 ment distillation processing, the number of code snippets requiring MD5 hashing is limited and their  
226 structure is fixed by syntactic and semantic constraints. The MD5 collision resistance (theoretical  
227 probability  $\approx 1.47 \times 10^{-18}$ ) is sufficient for strict sensitivity. Additionally, MD5’s superior speed  
228 and lower memory footprint provide optimal cost-performance.

229 3.2.3 TOPOLOGICAL PROXIMITY METRIC  
230

231 At this layer, the decaying subgraph edit distance (D-SED) is introduced to measure the graph simi-  
232 larity between the query graph  $G_q$  and the candidate graph  $G_c$  (Ranjan et al.; Zeng et al.). A higher  
233 D-SED value indicates less similarity. We calculate D-SED for code snippets to quantify their struc-  
234 tural similarity and retain those with the closest match.

$$D - \text{SED}(G_q, G_c) = \sum_{op \in O} \gamma^{l(op)} \cdot c(op) \quad (1)$$

235 Editing operations  $O$  are the set of operations to transform  $G_c$  to  $G_q$ , include adding, deleting, and  
236 modifying nodes and edges. Each operation  $op \in O$  has a cost  $c(op)$  and a hop count  $l(op)$  from its  
237 “core node”. For simplicity, we choose the node with the largest ID as core node. Operations closer  
238 to the core exert greater structural influence.  $\gamma \in (0, 1)$  is an attenuation factor that reduces the cost  
239 weight for operations farther from the core node. After computing D-SED scores for each candidate  
240  $c \in C_{RAP}$ , we compute a composite score  $s$  as a weighted sum of text similarity (calculated during  
241 initial candidate generation) and structural similarity (D-SED scores). Subsequently, we generate  
242  $Q_{TPM} = [(c, s), \dots]$ , ordered in descending score  $s$ .

243 3.2.4 DIVERSITY-AWARE RERANKING  
244

245 This module implements a variability-aware ranking model based on the Maximal Marginal Rele-  
246 vance (MMR) Carbonell & Goldstein (2017) algorithm to maximize result diversity while preserving  
247 relevance. It addresses homogeneity in traditional rankings through adversarial similarity calcula-  
248 tion and dynamic weight adjustment.  $S$  contains items  $(c, s) \in Q_{TPM}$  that have not been selected  
249 into  $C_{final}$  yet.  $\text{Sim}_1$  represents the relevance  $(s_i = \pi_2 \circ \iota_{c_i}(S))$  of item  $c_i$  to query  $q$ .  $\text{Sim}_2$  denotes  
250 the maximum cosine similarity between  $c_i$  and any item  $c_j$  in the selected set  $C_{final}$ .  $\lambda$  is a trade-off  
251 parameter that balances the emphasis between relevance ( $\text{Sim}_1$ ) and diversity ( $\text{Sim}_2$ ).  
252

$$\text{MMR} = \arg \max_{c_i \in S} \left[ \lambda \cdot \text{Sim}_1(c_i, q) - (1 - \lambda) \cdot \max_{c_j \in C_{final}} \text{Sim}_2(c_i, c_j) \right] \quad (2)$$

253 3.3 EXTERNAL-AWARE IDENTIFIER DISAMBIGUATOR (EAID)  
254

255 This module enhances knowledge through external identifier augmentation. Firstly, file-level entity  
256 modeling parses code per file  $F$ , extracting method entities  $E_{\text{method}}$  (functions/class methods with  
257 identifier, alias, line range  $[l_{\text{start}}, l_{\text{end}}]$ , parameter signature, scope) and class entities  $E_{\text{class}}$  (class def-  
258 initions with identifier, alias, line range  $[l_{\text{start}}, l_{\text{end}}]$ , member mappings) that built in the  $F$ . After  
259 that, it generates a structured identifier symbol table  $ST_{\text{lib}} = \{\text{identifier} \mapsto \text{syntax features}\}$ , where  
260  $\text{identifier}$  corresponds to either: (1) the unique identifier of a method entity  $\forall e \in E_{\text{method}}$ , or (2) the  
261 unique identifier of a class entity  $\forall c \in E_{\text{class}}$ , with the mapped *syntax features* containing all as-  
262 sociated attributes for that entity. Subsequently, the dependency resolution mechanism processes all  
263 import statements ( $I$ ) within the unfinished file. For Intra-project Cross-module Reference, this phase  
264 retrieves complete entities ( $E_{\text{lib}}$ ) from the pre-built entity library ( $ST_{\text{lib}}$ ) by determining their corre-  
265 sponding file paths ( $p$ ). These paths are constructed through decomposition of module components  
266 derived from either dotted names (e.g., `my.module.MyClass`) or relative imports (e.g., `from`

270 .sub\_module import MyClass), which are subsequently joined using directory separators  
 271 (/) and appended with the .py file extension. For standard and third-party libraries, the system  
 272 constructs a lightweight reference table  $T_{\text{ext}} = \{\text{canonical name} \mapsto \text{alias}\}$  to efficiently manage  
 273 external dependencies without full entity resolution. The enhanced prompts  $P_E = I \oplus E_{\text{lib}} \oplus T_{\text{ext}}$ .  
 274

### 275 3.4 PROMPT GENERATION

276  
 277 Following code retrieval and external link resolution,  
 278 SARACODER employs an external LLM to generate  
 279 subsequent statements. The final prompt  $P_{\text{final}}$   
 280 is constructed by concatenating three components:  
 281 the **external symbols enhancement**  $P_E$  where  
 282 entities are ordered by file import sequence reflecting  
 283 call probability decay—with function entities popu-  
 284 lated with complete function bodies and class entities  
 285 containing variable tables and method definitions;  
 286 the **similar code snippets**  $C_{\text{final}}$  containing  
 287 code snippets strictly sorted in ascending order of  
 288 similarity and annotated with source paths; the **un-  
 289 finished code context**  $C_{\text{context}}$ . This architecture fol-  
 290 lows  $P_{\text{final}} = C_{\text{final}} \oplus P_E \oplus C_{\text{context}}$ . (Figure 3).  
 291

## 292 4 EXPERIMENTS

### 293 4.1 EXPERIMENTAL SETTINGS

#### 294 4.1.1 DATASETS

295 We primarily utilize two datasets here: CrossCodeEval and RepoEval-Updated. (1) **CrossCodeEval Ding et al. (2023):** This benchmark evaluates code completion in complex Cross-File scenarios like type inference and dependency analysis. It is ideal for assessing performance that requires a deep understanding of code across multiple files. (2) **RepoEval-Updated Liu et al. (2024b):** Expanded from RepoEval Zhang et al. (2023), this new version, includes repositories of varying scales, offering a better way to evaluate In-File completion performance. We use CrossCodeEval to test models on code completion tasks required complex cross-file dependencies. RepoEval-Updated assesses basic syntax, common API usage, and local context understanding. Table 1 shows details.

#### 309 4.1.2 EVALUATION INDICATORS SETTING

310 In this study, the following several evaluation indicators are used to assess the effect of code com-  
 311 pletion Lu et al. (2021); Ding et al. (2023).

- 312 • **Code Exact Match (EM):** Proportion of generated code exactly matching the ground truth. EM  
 313 is given only for a perfect semantic and syntactic match.
- 314 • **Identifier Exact Match (ID\_EM):** The percentage of identifiers (variables, functions, etc.) per-  
 315 fectly matching the ground code. A high ID\_EM score indicates the model’s strong contextual  
 316 understanding, enabling it to accurately predict and generate contextually appropriate identifiers.
- 317 • **Identifier F1 Score (ID\_F1):** A more nuanced evaluation of identifier matching by combining  
 318 precision and recall. It offers a more comprehensive assessment of identifier completion quality,  
 319 particularly beneficial in scenarios where models might generate partial but correct identifier sets.
- 320 • **Edit Similarity (ES):** Similarity metric between generated and ground-truth code based on edit  
 321 distance. It tolerates slight variations, requiring the completed code to be highly similar in struc-  
 322 ture, syntax, and token order to the target.

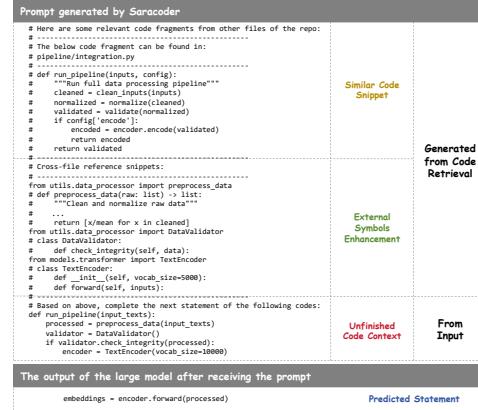


Figure 3: Prompt template.

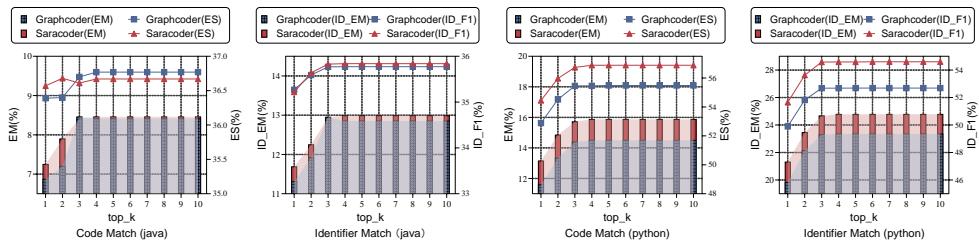
Table 1: CrossCodeEval vs. RepoEval-Updated comparison.

	CrossCodeEval		RepoEval-Updated	
	Python	Java	Python	Java
Total Repositories	471	239	10	8
Total Files	14348	5868	3258	8260
Total Task cases	2665	2139	2000	1600
Applicable Scenarios	Cross-File completion	In-File completion		

324 4.1.3 BASELINE SETTING  
325326 We employ the following five methods as controls to evaluate the effectiveness of retrieval-  
327 augmented generation (RAG) in code completion: No RAG (Zero-shot-only baseline), Shifted RAG  
328 (Target-context dynamic retrieval), Vanilla RAG (Exemplar-similarity fixed retrieval), Repocoder  
329 (Iterative-fragment integration Zhang et al. (2023)), Graphcoder (Structure-modeling CCG utiliza-  
330 tion Liu et al. (2024b)).331 4.1.4 MODEL SELECTION  
332333 In this experiment, we select Codegen2-7b, Codegen25-7b Nijkamp  
334 et al. (2023b;a), CodeLlama-7b-Instruct Rozière et al. (2024) and  
335 deepseek-coder-6.7b-instruct Guo et al. (2024) for code completion task inference.  
336337 4.2 MAIN RESULTS  
338339 To evaluate the performance of SARACODER on repository-level code completion, we have formu-  
340 lated the following four research questions (RQs):341 **RQ1 Effectiveness in Cross-File Scenarios:** How does SARACODER perform when cross-context  
342 understanding is required, compared to other methods?  
343344 **RQ2 Cost Analysis in Cross-File Scenarios:** How does SARACODER’s resource consumption  
345 compare to GraphCoder in Cross-File scenarios?  
346347 **RQ3 Synergistic Gain Property:** How does SARACODER perform when integrated orthogonally  
348 with other methods that provide cross-file context?  
349350 **RQ4 Advantage in In-File Scenarios:** How does SARACODER perform on tasks without cross-  
351 context requirements and what are its advantages?  
352353 4.2.1 FOR RQ1: DOMINANT CROSS-FILE CODE ACCURACY.  
354355 Table 2 illustrates that SARACODER surpasses the top-performing Repocoder on the CrossCodeEval  
356 dataset, achieving an average improvement of 1.50 in EM, 0.77 in ES, 1.11 in ID\_EM, and 0.61 in  
357 ID\_F1. This indicates SARACODER provides more effective information and generates code with  
358 higher semantic accuracy, better capturing intended functionality. The enhanced ID\_EM further  
359 shows SARACODER’s superior ability to interpret context and select appropriate identifiers. These  
360 advancements effectively mitigate misleading superficial similarity and external symbol ambiguity,  
361 leading to more reliable and contextually relevant code. For Java code completion, SARACODER  
362 shows better EM and ID\_EM, with slightly lower ES and F1 scores. This discrepancy is attributed to  
363 the inherent characteristics of Java’s static typing system and complex code structure. These features  
364 lead to the generation of code that is logically correct but contains numerous textual variations and  
365 boilerplate redundancies. Consequently, small structural deviations (e.g., misplaced brackets) are  
366 more readily penalized by metrics such as ES and F1.  
367368 4.2.2 FOR RQ2: COST-OPTIMIZED ACCURACY ADVANTAGE IN CROSS-FILE.  
369370 We experiment with code completion efficiency using codegen25-7b and Graphcoder. Our goal is  
371 to see how retrieving more similar cases ( $top\_k$ ) impacts accuracy. Since using fewer  $top\_k$  cases  
372 saves input tokens<sup>2</sup>, this study shows the balance between resources and accuracy. Our experiments  
373 demonstrate significant performance saturation for both retrieval methods when  $top\_k$  reaches 3-4,  
374 with no observable fluctuations upon increasing to  $top\_k = 10$ . SARACODER achieves compre-  
375 hensive superiority in Python tasks (e.g., 9.4% EM improvement) while maintaining advantages in  
376 Java tasks despite a marginal 0.1 decrease in ES. Crucially, under resource-constrained  $top\_k = 1$   
377 conditions: all Python metrics outperform the baseline; three Java metrics (EM/ES/ID\_EM) show  
378 improvements; and Java ID\_F1 initially trails (35.22 vs. 35.27) but ultimately surpasses the baseline  
379 at saturation (35.84 vs. 35.77). Our method achieves performance breakthroughs at lower computa-  
380 tional cost (stable at  $top\_k \approx 4$ ) by reducing redundant and homogeneous cases (Figure 4).  
381382 <sup>2</sup>For relevant explanations, please refer to the Appendix A.6.2  
383

378 Table 2: Performance comparison on the CrossCodeEval dataset. Numbers are shown in percentage  
379 (%) . The top results are **bolded**, and the second best are underlined.

381 Language	382 Methods	Codegen2-7b				Codegen25-7b				deepseek-coder-6.7b-instruct				CodeLlama-7b-Instruct			
		383 Code Match	384 Identifier Match	385 Code Match	386 Identifier Match	387 Code Match	388 Identifier Match	389 Code Match	390 Identifier Match	391 Code Match	392 Identifier Match	393 Code Match	394 Identifier Match	395 Code Match	396 Identifier Match		
397 Python	No Rag	0.00	13.38	0.00	2.24	0.00	13.26	0.00	2.10	0.00	4.51	0.00	0.57	0.00	13.27	0.00	2.22
	Shifted Rag	4.84	46.67	11.48	42.72	7.40	48.88	14.09	44.62	8.19	50.18	14.77	46.69	6.91	49.12	13.60	45.12
	Vanilla Rag	9.48	50.97	17.15	47.81	12.39	53.92	23.61	53.14	13.00	54.00	26.63	55.77	11.45	52.93	19.23	50.49
	Repocoder	12.47	54.08	21.57	51.89	16.62	56.80	25.73	<b>57.85</b>	17.11	58.11	26.71	56.46	15.14	56.28	24.56	54.22
	Graphcoder	10.88	52.36	19.68	49.73	14.54	55.29	23.38	52.61	15.53	57.05	24.29	55.01	13.30	55.41	22.63	52.91
	SARACODER	<b>15.04</b>	<u>56.03</u>	<b>24.44</b>	<u>54.68</u>	<b>18.36</b>	<u>58.30</u>	<b>27.28</b>	56.22	<b>19.72</b>	<u>59.93</u>	<b>28.52</b>	<u>58.26</u>	<b>17.91</b>	<u>58.37</u>	<b>27.77</b>	<u>56.82</u>
398 Java	No Rag	1.03	21.79	0.64	16.86	1.50	21.77	24.78	24.78	6.40	35.79	10.42	32.90	0.93	20.83	1.96	16.17
	Shifted Rag	6.08	46.09	12.11	43.76	5.89	38.00	10.23	36.44	5.84	<b>36.19</b>	11.64	<b>35.23</b>	6.73	43.75	12.71	41.68
	Vanilla Rag	9.30	<b>47.42</b>	15.71	<u>45.69</u>	10.38	<u>40.76</u>	15.29	<u>39.91</u>	8.88	33.59	15.01	33.51	10.93	<u>45.01</u>	17.81	<u>44.08</u>
	Repocoder	10.71	41.83	16.18	41.51	<b>12.16</b>	<u>42.38</u>	<b>17.63</b>	<u>41.69</u>	9.58	34.25	<u>15.76</u>	34.13	<b>13.23</b>	<u>46.01</u>	<b>19.87</b>	<u>45.14</u>
	Graphcoder	8.13	45.18	14.35	43.32	8.42	36.77	12.85	35.84	7.39	32.34	12.76	32.05	8.51	40.57	14.96	39.57
	SARACODER	<b>11.73</b>	<u>46.69</u>	<b>18.37</b>	<u>45.47</u>	<u>11.40</u>	39.33	16.22	38.97	<b>11.92</b>	34.55	<b>17.72</b>	34.99	12.95	42.71	<u>19.54</u>	42.33



399 Figure 4: Impact of  $top\_k$  on CrossCodeEval. (The two on the left are Java tasks, and the two on the  
400 right are Python tasks.)

#### 4.2.3 FOR RQ3: SYNERGISTIC INTEGRATION OF SARACODER ACHIEVES ENHANCED COMPLETION.

404 We examine two prominent methods that demonstrate exceptional performance in Cross-File scenarios. **(1) Repocoder** Zhang et al. (2023), distinct from the original, assumes that if code snippets are 405 similar, their subsequent content is also likely relevant. In the next search round, it specifically gets 406 the code following those similar snippets (hereafter referred to as Repocoder). **(2) Draco** Cheng et al. 407 (2024), analyzes code to create entity dependency graphs, allowing detailed background knowledge 408 retrieval. It then uses this information to create structured prompts. Currently, Draco only works 409 with Python. As shown in Table 3, adding our method significantly boosts all four Python metrics 410 (by 3.42 to 4.52) compared to using Repocoder or Draco alone. For Java, our method improves EM 411 by 0.45 and ID\_EM by 0.33 over Repocoder, showing SARACODER exhibits significant synergistic 412 gain property with existing cross-file methods.<sup>3</sup>

#### 4.2.4 FOR RQ4: ENHANCED IN-FILE ACCURACY AND RESOURCE EFFICIENCY.

416 On the RepoEval-Updated dataset (Table 4), SARACODER shows superior semantic and identifier 417 accuracy (surpassing the top-performing Graphcoder: +0.547 EM, +0.737 ES, +0.125 ID\_EM, and 418 +0.667 F1) for both Python and Java code completion. The cost analysis (Appendix A.6.1) further 419 indicates SARACODER generally performs better and exhibits higher stability across most Python 420 metrics (excluding EM) and all Java metrics. This makes it particularly effective for resource- 421 constrained environments, especially at lower  $top\_k$  values. However, SARACODER’s gains over 422 Graphcoder in code and identifier matching are smaller here than on CrossCodeEval. This is pri- 423 marily because RepoEval-Updated projects contain a higher prevalence of similar code snippets, 424 resulting in reduced code diversity within the repository. Overall, the conclusions align with those 425 from the CrossCodeEval dataset.

#### 4.3 ABLATION STUDY

426 To understand the importance of each part of SARACODER, we conduct ablation tests on the Cross- 427 CodeEval dataset (Figure 5). “-EAID” indicates disabling **External-Aware Identifier Disambiguator**, 428 resulting in the loss of external dependency integration capabilities; “-HF\_OP” denotes remov- 429 430

<sup>3</sup>You can find the causes of synergistic gains in Appendix A.6.3.

432 Table 3: Performance benefits of SARACODER when integrated orthogonally with other cross-file  
 433 approaches (%). The CrossCodeEval dataset is used in this part.

434

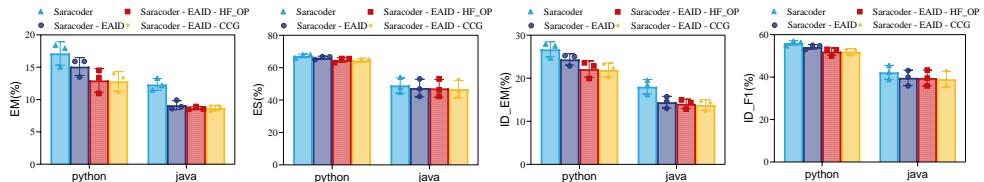
435 Language	436 Methods	Codegen2-7b				Codegen25-7b				CodeLlama				
		437 Code Match	438 Identifier Match	439 Code Match	440 Identifier Match	441 Code Match	442 Identifier Match	443 Code Match	444 Identifier Match	445 Code Match	446 Identifier Match	447 Code Match	448 Identifier Match	
445	446	EM	447 ES	EM	448 F1	EM	449 ES	450 F1	EM	451 ES	452 F1	EM	453 ES	454 F1
455 Python	No Rag	5.44	57.85	11.71	42.22	7.77	60.52	14.45	45.40	9.49	61.97	16.44	47.36	
	Shift Rag	4.87	58.36	11.94	42.91	7.44	60.20	14.17	44.78	6.95	60.35	13.75	45.36	
	Vanilla Rag	9.52	61.87	17.42	48.01	12.43	63.81	20.74	51.00	11.48	63.66	19.42	50.72	
	SARACODER	12.16	54.16	18.37	45.47	11.50	44.90	16.22	38.95	13.32	48.04	19.54	42.34	
	Repocoder	12.50	64.48	21.87	52.09	16.66	66.67	25.99	55.06	15.19	66.24	24.74	54.39	
	Repocoder + SARACODER	15.94 <small>+3.44</small>	66.43 <small>+1.95</small>	25.73 <small>+3.86</small>	54.80 <small>+2.71</small>	19.49 <small>+2.83</small>	68.53 <small>+1.86</small>	28.90 <small>+2.91</small>	57.49 <small>+2.43</small>	19.19 <small>+4.00</small>	68.45 <small>+2.21</small>	29.05 <small>+4.31</small>	57.47 <small>+3.08</small>	
455 Java	Draco	20.06	66.33	29.13	56.53	22.93	68.70	32.45	59.34	23.50	68.56	32.57	59.49	
	Draco + SARACODER	24.06 <small>+4.00</small>	69.40 <small>+3.07</small>	34.00 <small>+4.87</small>	61.12 <small>+4.59</small>	27.05 <small>+4.12</small>	71.86 <small>+3.16</small>	36.80 <small>+4.35</small>	63.48 <small>+4.14</small>	27.20 <small>+3.7</small>	72.01 <small>+3.45</small>	37.29 <small>+1.72</small>	64.35 <small>+4.85</small>	
	No Rag	0.00	25.92	0.05	17.48	0.00	25.46	0.05	17.61	0.00	25.17	0.00	17.23	
	Shift Rag	6.45	54.84	12.11	43.75	6.08	44.73	10.27	42.95	7.11	50.96	12.72	41.68	
	Vanilla Rag	9.86	55.71	15.71	45.11	10.47	47.09	15.29	39.93	11.31	51.48	17.81	44.09	
	SARACODER	12.16	54.16	18.37	45.47	11.50	44.90	16.22	38.95	13.32	48.04	19.54	42.34	
	Repocoder	11.22	56.89	17.72	47.41	10.85	47.93	16.18	41.50	13.60	52.17	19.87	45.14	
	Repocoder + SARACODER	11.50 <small>+0.28</small>	56.09 <small>-0.80</small>	17.72 <small>0.00</small>	46.96 <small>-0.45</small>	11.27 <small>+0.42</small>	46.53 <small>-1.40</small>	16.41 <small>+0.23</small>	40.47 <small>-1.03</small>	14.26 <small>+0.66</small>	50.79 <small>-1.38</small>	20.62 <small>+0.75</small>	44.38 <small>-0.76</small>	

443 Table 4: Performance comparison on the RepoEval-Updated dataset. Numbers are shown in per-  
 444 centage (%). The top results are **bolded**, and the second best are underlined.

445

446 Language	447 Methods	Codegen2-7b			Codegen25-7b			deepseek-coder-6.7b-instruct			CodeLlama-7b-Instruct		
		448 Code Match	449 Identifier Match	450	451 Code Match	452 Identifier Match	453	454 Code Match	455 Identifier Match	456	457 Code Match	458 Identifier Match	459
460	461	EM	462 ES	463 F1	464	EM	465 ES	466 F1	467	EM	468 ES	469 F1	470
471 Python	No Rag	17.40	32.54	23.75	30.21	19.55	34.48	25.75	32.16	11.50	30.33	15.30	22.39
	Shifted Rag	32.70	59.22	40.10	55.66	36.45	61.96	43.20	58.31	20.90	42.95	26.50	38.88
	Vanilla Rag	38.70	63.58	46.45	60.43	42.25	66.26	48.75	62.79	22.20	41.48	27.85	37.58
	Repocoder	37.60	61.98	45.10	58.47	40.55	64.48	46.85	60.71	21.35	40.18	26.65	35.93
	Graphcoder	42.40	65.73	49.45	62.07	44.65	67.59	51.00	63.82	28.50	44.63	33.35	42.63
	SARACODER	<b>42.60</b>	<b>65.92</b>	<b>50.15</b>	<b>62.61</b>	<u>44.50</u>	<u>67.79</u>	<b>51.10</b>	<b>63.84</b>	<b>28.25</b>	<b>46.91</b>	<b>33.45</b>	<b>42.95</b>
471 Java	No Rag	6.55	16.84	9.15	8.84	5.35	16.21	9.05	8.65	6.40	20.61	7.75	8.73
	Shifted Rag	30.87	62.52	43.94	61.01	26.63	58.46	37.75	56.57	28.00	55.12	36.81	53.39
	Vanilla Rag	33.50	63.82	45.44	62.08	32.00	61.52	41.56	59.48	21.13	46.51	31.19	45.15
	Repocoder	30.13	60.01	42.31	57.10	28.75	57.73	37.88	54.46	22.19	46.93	32.06	44.26
	Graphcoder	37.75	66.19	50.68	64.77	36.63	64.74	46.13	62.62	28.81	55.75	40.06	53.61
	SARACODER	<b>37.93</b>	<b>67.06</b>	<b>50.93</b>	<b>65.48</b>	<u>36.75</u>	<u>65.54</u>	<b>46.44</b>	<b>62.61</b>	<b>29.75</b>	<b>56.38</b>	<b>40.75</b>	<b>54.18</b>

455 ing **Hierarchical Feature Optimization**, canceling the similar fragment screening mechanism; “-  
 456 CCG” indicates disabling the **code context graph**, so it lost the understanding of code structure. The  
 457 ablation experiments demonstrate that the complete SARACODER achieves optimal performance,  
 458 with all components positively contributing to repository-level completion. Notably, even without  
 459 EAID, SARACODER still outperforms Shift RAG and Vanilla RAG, and even surpasses Repocoder  
 460 in Python tasks<sup>4</sup>, proving that HF\_OP screening substantially enhances case quality.



468 Figure 5: Ablation study. (Each three-data-point group represents CodeGen2-7B, CodeGen2.5-7B,  
 469 and CodeLlama-7B-Instruct models. Bar lengths show their average performance, with I-shaped  
 470 error bars indicating standard deviation)

## 473 5 CONCLUSION AND OUTLOOK

474 In this paper, we present SARACODER, a resource-optimized repository-level code completion  
 475 method. It solves the problems of superficial similarity dispersion, retrieval redundancy and rigidity,  
 476 and external symbol ambiguity by combining semantic topology with disambiguation. SARACODER  
 477 uniquely addresses superficial similarity dispersion, retrieval redundancy and rigidity, and external  
 478 symbol ambiguity, reducing unnecessary context window length consumption, and providing more  
 479 diverse and higher-quality completion reference information content under resource-constrained  
 480 conditions. This method improves code completion quality and can positively complement other  
 481 cross-file methods, providing synergistic improvements when used in combination. However, while  
 482 both Java and Python are prominent and widely used languages, the generalizability of this method  
 483 to other programming languages has not yet been achieved. Future work will pursue two key direc-  
 484 tions: expanding language coverage and exploring cross-language code completion.  
 485

<sup>4</sup>Detailed data can be found in Table 8 in appendix.

486 REPRODUCIBILITY STATEMENT  
487488 We have submitted the relevant code in the supplementary materials. The names of the experimental  
489 benchmarks, the prompt templates used, and the model’s hyperparameter settings can all be found  
490 in Section 3.4 and A.5. The Appendix A.5.1 and A.5.2 provides a detailed description of the  
491 experimental setup for the mechanism experiments.  
492493 REFERENCES  
494495 Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz  
496 Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi,  
497 Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin,  
498 Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García  
499 del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas,  
500 Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia  
501 Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries,  
502 and Leandro von Werra. Santacoder: don’t reach for the stars! Workingpaper, arXiv, January  
503 2023.504 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan,  
505 Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large  
506 language models, 2021. URL <https://arxiv.org/abs/2108.07732>.507 Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering  
508 documents and producing summaries. *SIGIR Forum*, 51(2):209–210, August 2017. ISSN 0163-  
509 5840. doi: 10.1145/3130348.3130369. URL [https://doi.org/10.1145/3130348.  
510 3130369](https://doi.org/10.1145/3130348.3130369).511 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared  
512 Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri,  
513 Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan,  
514 Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian,  
515 Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios  
516 Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex  
517 Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders,  
518 Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec  
519 Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob Mc-  
520 Grew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large  
521 language models trained on code, 2021. URL <https://arxiv.org/abs/2107.03374>.522 Wei Cheng, Yuhang Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level  
523 code completion. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of  
524 the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long  
525 Papers)*, pp. 7957–7977, Bangkok, Thailand, August 2024. Association for Computational Lin-  
526 guistics. doi: 10.18653/v1/2024.acl-long.431. URL [https://aclanthology.org/2024.  
527 acl-long.431/](https://aclanthology.org/2024.acl-long.431/).528 Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Kr-  
529 ishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodee-  
530 val: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Con-  
531 ference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL  
532 <https://openreview.net/forum?id=wgDcbBMSfh>.533 Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Par-  
534 minder Bhatia, Dan Roth, and Bing Xiang. CoCoMIC: Code completion by jointly modeling  
535 in-file and cross-file context. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessan-  
536 dro Lenci, Sakriani Sakti, and Nianwen Xue (eds.), *Proceedings of the 2024 Joint Interna-  
537 tional Conference on Computational Linguistics, Language Resources and Evaluation (LREC-  
538 COLING 2024)*, pp. 3433–3445, Torino, Italia, May 2024. ELRA and ICCL. URL <https://aclanthology.org/2024.lrec-main.305/>.

540 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing  
 541 Qin, Ting Liu, Dixin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming  
 542 and natural languages. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Asso-  
 543 ciation for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020.  
 544 Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL  
 545 <https://aclanthology.org/2020.findings-emnlp.139/>.

546 Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan,  
 547 Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn  
 548 Drain, Neel Sundaresan, Jian Yin, Dixin Jiang, and Ming Zhou. Graphcodebert: Pre-training  
 549 code representations with data flow. In *International Conference on Learning Representations*,  
 550 2021. URL <https://openreview.net/forum?id=jLoC4ez43PZ>.

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

554 Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Code-  
 555 searchnet challenge: Evaluating the state of semantic code search, 2020. URL <https://arxiv.org/abs/1909.09436>.

556 Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: multi-token code completion by  
 557 jointly learning from structure and naming sequences. In *Proceedings of the 44th International  
 558 Conference on Software Engineering*, ICSE ’22, pp. 401–412, New York, NY, USA, 2022. Asso-  
 559 ciation for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510172. URL  
 560 <https://doi.org/10.1145/3510003.3510172>.

561 Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Dixin  
 562 Jiang, Weizhu Chen, and Nan Duan. CodeRetriever: A large scale contrastive pre-training method  
 563 for code search. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of  
 564 the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 2898–2910,  
 565 Abu Dhabi, United Arab Emirates, December 2022a. Association for Computational Linguis-  
 566 tics. doi: 10.18653/v1/2022.emnlp-main.187. URL [https://aclanthology.org/2022.emnlp-main.187/](https://aclanthology.org/2022.emnlp-main.187).

567 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom  
 568 Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cy-  
 569 prien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl,  
 570 Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Rob-  
 571 son, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-  
 572 level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022b. doi: 10.1126/  
 573 science.abq1158. URL <https://www.science.org/doi/abs/10.1126/science.abq1158>.

574 Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code  
 575 auto-completion systems. In *The Twelfth International Conference on Learning Representations*,  
 576 2024a. URL <https://openreview.net/forum?id=pPjZIOuQuF>.

577 Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang  
 578 Wang. Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based  
 579 on code context graph. In *Proceedings of the 39th IEEE/ACM International Conference on Au-  
 580 tomated Software Engineering*, ASE ’24, pp. 570–581, New York, NY, USA, 2024b. Asso-  
 581 ciation for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695054. URL  
 582 <https://doi.org/10.1145/3691620.3695054>.

583 Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin  
 584 Clement, Dawn Drain, Dixin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou,  
 585 Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng,  
 586 Shengyu Fu, and Shujie LIU. Codexglue: A machine learning benchmark dataset for code under-  
 587 standing and generation. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Infor-  
 588 mation Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL <https://>

594        //datasets-benchmarks-proceedings.neurips.cc/paper\_files/paper/  
 595        2021/file/c16a5320fa475530d9583c34fd356ef5-Paper-round1.pdf.  
 596

597        Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. ReACC:  
 598        A retrieval-augmented code completion framework. In Smaranda Muresan, Preslav Nakov,  
 599        and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for  
 600        Computational Linguistics (Volume 1: Long Papers)*, pp. 6227–6240, Dublin, Ireland, May  
 601        2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.431. URL  
 602        <https://aclanthology.org/2022.acl-long.431/>.

603        Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Code-  
 604        gen2: Lessons for training llms on programming and natural languages, 2023a. URL <https://arxiv.org/abs/2305.02309>.

605        Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese,  
 606        and Caiming Xiong. Codegen: An open large language model for code with multi-turn program  
 607        synthesis, 2023b. URL <https://arxiv.org/abs/2203.13474>.

608        Rishabh Ranjan, Siddharth Grover, Sourav Medya, Venkatesan Chakaravarthy, Yogish Sabharwal,  
 609        and Sayan Ranu. GREED: A neural framework for learning graph distance functions. In Al-  
 610        ice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neu-  
 611        ral Information Processing Systems*, 2022. URL <https://openreview.net/forum?id=3LBxVcnsEkV>.

612        R. Rivest. Rfc1321: The md5 message-digest algorithm, 1992.

613        Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi  
 614        Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémie Rapin, Artyom Kozhevnikov, Ivan Ev-  
 615        timov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong,  
 616        Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier,  
 617        Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.  
 618        URL <https://arxiv.org/abs/2308.12950>.

619        Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. Do-  
 620        main adaptive code completion via language models and decoupled domain databases, 2023. URL  
 621        <https://arxiv.org/abs/2308.09313>.

622        Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu  
 623        Chen, and Jian-Guang Lou. Cert: Continual pre-training on sketches for library-oriented code  
 624        generation. In Lud De Raedt (ed.), *Proceedings of the Thirty-First International Joint Conference  
 625        on Artificial Intelligence, IJCAI-22*, pp. 2369–2375. International Joint Conferences on Artificial  
 626        Intelligence Organization, 7 2022. doi: 10.24963/ijcai.2022/329. URL <https://doi.org/10.24963/ijcai.2022/329>. Main Track.

627        Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing  
 628        stars: on approximating graph edit distance. *Proc. VLDB Endow.*, 2(1):25–36, August 2009.  
 629        ISSN 2150-8097. doi: 10.14778/1687627.1687631. URL <https://doi.org/10.14778/1687627.1687631>.

630        Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang  
 631        Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval  
 632        and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023  
 633        Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore,  
 634        December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.  
 635        151. URL <https://aclanthology.org/2023.emnlp-main.151/>.

## 642        A APPENDIX

### 643        A.1 CONTEXT GRAPH CONSTRUCTION

644        Code parsing transforms source code into an intermediate representation that is easier to analyze and  
 645        process, and is a fundamental step to understand the semantics and structure of a program. Abstract

Syntax Tree (AST) is one of the most commonly used and effective intermediate representations in code parsing. It can map source code to a tree topology structure, and accurately represent the syntax features and context relationships of code elements. By traversing and manipulating the abstract syntax tree, the relationship graph in the code can be constructed efficiently. Tree-sitter is a CFG-based parser generator that can support a variety of programming languages including Python, Java, C++. The core advantage of Tree-sitter is its efficient parsing performance and wide support for multiple languages, which makes the system have the ability to parse the code of multiple programming languages uniformly, and provides the possibility of multi-language code analysis. In code, the following relationships play a key role in semantic analysis, refactoring, debugging, and maintenance. We use tree-sitter to model the following relationships. See Table 5 for details

Table 5: Semantic Relationships in Code Analysis

Relationship	Definition	Syntax Examples		Type-Sensitive Characteristics
		Python	Java	
Assignment	Variable obtains type identity through assignment	count: int = 10	String s = new String()	Type inference and propagation
Contextual Binding	Creates temporary type bindings in specific syntactic structures	with open(file) as f:	try (BufferedReader br = ...)	Context-dependent type lifecycle management
Reference	Access to existing variables or properties	obj.calculate()	this.value	Late-bound type resolution
Type Declaration	Explicit annotation of variable/return types	def func() -> list[str]:	List<Integer> list = new ArrayList<>()	Basis for static type checking
Parameter Constraint	Type constraints on function parameters	def sort(items: Sequence[T]):	void sort(List<? extends Comparable> l)	Input type validation
Return Constraint	Type constraints on function return values	@return:type(float)	public int getValue() { ... }	Output type consistency guarantee
Inheritance	Subclasses automatically acquire parent class members	class Child(Parent):	class Child extends Parent { ... }	Type hierarchy inheritance
Implementation	Class fulfillment of interface contracts	class MyList(ABC):	class ArrayList implements List { ... }	Foundation for polymorphic behavior
Override	Subclass overriding of parent class methods	def method(self): ...	@Override void method() { ... }	Dynamic method dispatch
Import Dependency	Cross-module import dependencies	import pandas as pd	import java.util.List;	Type visibility control
Invocation	Execution dependencies between methods/functions	math.sqrt(x)	Collections.sort(list)	Type compatibility verification

## A.2 METHOD SUPPLEMENT

## A.3 IMPLEMENTATION OF CONTEXT GRAPH SLICING

We begin by initializing three empty sets:  $V_{cf}$  for control flow,  $V_{dd}$  for data dependencies, and  $V_{cd}$  for control dependencies, along with an empty queue  $Q$ . The process starts by adding  $v_{target}$  to  $Q$ . We then enter a loop, continuing as long as  $Q$  is not empty. In each iteration, a vertex  $v$  is dequeued. We apply two critical checks: first, a **hop count check** stops processing if  $v$  is more than  $h$  hops from  $v_{target}$ ; second, a **size check** halts if the combined size of  $V_{cf} \cup V_{dd} \cup V_{cd}$  reaches  $k$  statements. If  $v$  passes these checks, we update the sets:  $v$  goes into  $V_{cf}$ , its data dependency predecessors go into  $V_{dd}$ , and its control dependency predecessors go into  $V_{cd}$ . Following this, all unvisited control flow predecessors of  $v$  are enqueued. The loop concludes when  $Q$  is empty or a size/hop limit is hit. Finally, using the union  $V_{cf} \cup V_{dd} \cup V_{cd}$  as the vertex set, we generate the **induced subgraph**  $G_h(v_{target})$ , which represents our final **context graph slice**.

## A.4 RETHINKING ON THE RETRIEVAL RANGE

Based on preliminary research, we observed that several retrieval-augmented methods for finding similar code snippets employ a zero-filtering strategy for subsequent code within the same file. Specifically, this strategy assigns a similarity score of zero—for both textual and graph-structural similarity—to any code segment located after the current line requiring completion within the same file, relative to the context of the current completion point. This approach stems from the assumption that code segments appearing after the completion point in the same file hold no semantic relevance to the current completion task. The rationale behind this assumption likely lies in a developer mindset: “During code completion, neither the code being completed nor the subsequent code exists yet; therefore, later code offers no reference value.”

We contend that this perspective may not universally hold. Firstly, in practical development, due to modular programming logic, most programmers’ cognitive context is not continuous. Development does not strictly follow a top-down sequence based on physical line numbers; rather, it often involves non-linear thought processes. Consequently, there is no inherent correspondence between physical line numbers and the semantic boundaries of code segments. Secondly, within the same file, different functions may share similar implementations. For instance, Different classes might exhibit identical initialization logic; operations on distinct variables may follow similar patterns. Therefore, code segments following the completion point are not entirely devoid of reference value. In fact, due to the typically homogeneous nature of tasks handled within a single file, these subsequent segments might

702 be more contextually relevant to the actual scenario requiring completion compared to snippets  
 703 retrieved from entirely different files.  
 704

705 **A.4.1 HE\_OP'S PREVIOUS PREPARATION**  
 706

707 After the database is established, when we receive a code completion request, we first conduct a  
 708 global search for similar code snippet in the context of the location to be completed, and obtain a  
 709 larger candidate pool by relaxing the location constraints. Use Jaccard similarity to calculate the  
 710 text similarity with the query code, and select the  $top_k \times p$  samples with the highest similarity  
 711 as the candidate set to participate in the subsequent process. Among them, Jaccard similarity is a  
 712 measurement method used to measure the similarity between two sets. It measures the similarity  
 713 between the two sets by calculating the ratio of their intersection to their union.  
 714

715 **A.4.2 THE ALGORITHM OF SEMANTIC ALIGNMENT DISTILLATION**  
 716

717 For details, please refer to Algorithm 1.

---

718 **Algorithm 1** Code Similarity Analysis

719 **Require:** Query  $Q$ , Candidates  $C$ , Max length  $L = 512$

720 **Ensure:** Filtered results  $R$

721 **Preprocess:**

722 Tokenize and pad  $Q$  and  $C$  to length  $L$

723 **Encode:**

724 **for**  $s \in \{Q\} \cup C$  **do**

725     Extract features  $v_s \in \mathbb{R}^{768}$

726     Normalize  $v_s$

727 **end for**

728 **Filter:**

729     Compute similarities  $S = \{\cos(v_Q, v_c) | c \in C\}$

730     Set threshold  $\tau = \text{quantile}(S, 0.75)$

731      $R \leftarrow \{c | \cos(v_Q, v_c) \geq \tau\}$

732 **Cache:**

733     **if**  $s \in \text{Cache}$  **then**

734         Retrieve cached  $v_s$

735     **else**

736         Compute and store  $v_s$

737 **end if**

---

738 **A.4.3 D-SED CALCULATION**

739 For details, please refer to Algorithm 2. The following is the symbol explanation.

740

- 741 •  $X_{\mathcal{A}}$ : Set of aligned vertices (mapped to target graph)
- 742 •  $X_h^l(\hat{y}) \setminus X_{\mathcal{A}}$ : Set of unaligned vertices (to be inserted/deleted)
- 743 •  $E_{\mathcal{A}}$ : Set of aligned edges
- 744 •  $E_h^l(\hat{y}) \setminus E_{\mathcal{A}}$ : Set of unaligned edges
- 745 •  $h(v, \tilde{y})$ : Distance from vertex  $v$  to reference point  $\tilde{y}$
- 746 •  $c(v, \mathcal{A}(v))$ : Vertex substitution cost (for aligned vertices)
- 747 •  $c(v)$ : Vertex insertion/deletion cost (for unaligned vertices)
- 748 •  $c(e, \mathcal{A}(e))$ : Edge substitution cost (for aligned edges)
- 749 •  $c(e)$ : Edge insertion/deletion cost (for unaligned edges)

750 **Key Features:**

751

- 752 • **Distance-based decay**:  $\gamma^{h(v, \tilde{y})}$  weights edit costs based on proximity to reference

---

756 **Algorithm 2** Decay Subgraph Edit Distance (D-SED)

---

757 **Require:** Graphs  $G_h^l(\hat{y})$  and  $G_h^l(x)$   
 758    Decay factor  $\gamma \in (0, 1)$   
 759 **Ensure:** SED between  $G_h^l(\hat{y})$  and  $G_h^l(x)$   
 760    1: SED  $\leftarrow 0$   
 761    2: **for** each vertex  $v \in X_{\mathcal{A}}$  **do**  
 762       3:    SED  $\leftarrow$  SED +  $\gamma^{h(v, \hat{y})} \cdot c(v, \mathcal{A}(v))$   
 763       4: **end for**  
 764    5: **for** each vertex  $v \in X_h^l(\hat{y}) \setminus X_{\mathcal{A}}$  **do**  
 765       6:    SED  $\leftarrow$  SED +  $\gamma^{h(v, \hat{y})} \cdot c(v)$   
 766       7: **end for**  
 767    8: **for** each edge  $e = (v, t, u) \in E_{\mathcal{A}}$  **do**  
 768       9:    SED  $\leftarrow$  SED +  $\gamma^{h(v, \hat{y})} \cdot c(e, \mathcal{A}(e))$   
 769       10: **end for**  
 770    11: **for** each edge  $e = (v, t, u) \in E_h^l(\hat{y}) \setminus E_{\mathcal{A}}$  **do**  
 771       12:    SED  $\leftarrow$  SED +  $\gamma^{h(v, \hat{y})} \cdot c(e)$   
 772       13: **end for**  
 773    14: **return** SED

---

774  
 775    • **Four cost categories:** Separates vertex/edge and aligned/unaligned cases  
 776    • **Reference point:**  $\hat{y}$  serves as the anchor for distance calculations  
 777    • **Asymmetric treatment:** Focuses on edits in  $G_h^l(\hat{y})$  relative to  $G_h^l(x)$

---

## 780 A.5 DETAILS OF EXPERIMENT SETUP

## 781 A.5.1 PARAMETER SETTING

782 In this study, we mainly adopt the Greedy Decoding strategy for text generation. Its core configu-  
 783 ration is: disable sampling (`do_sample=False`) and set the temperature value (`temperature`)  
 784 to 0. This combination ensures that the model necessarily selects the token with the highest logical  
 785 probability at each step, thereby eliminating randomness in the generation process and facilitating  
 786 a strict and reproducible evaluation of the model’s performance. To simplify and observe the per-  
 787 formance of various methods within the limited length of the context window, the maximum length  
 788 limit for text is 2048 new tokens (`max_num_tokens`).  
 789

## 790 A.5.2 RESOURCE CONTROL AND ALLOCATION IN THE EXPERIMENT

791 In practice, the `max_num_tokens` parameter controls the total length of the context window. The  
 792 system sets a maximum length limit for the “unfinished code context,” which is no more than half  
 793 of the total window. The remaining context window is dynamically used to accommodate other  
 794 information, including retrieved similar code snippets. Since the output of the External Symbols  
 795 Enhancement module is highly compressed, we primarily use the `top_k` parameter to control the  
 796 number of similar code snippets that can be included in the prompt. This effectively constrains the  
 797 use of this portion of the resources. The value of `top_k` directly determines the number of similar  
 798 examples that can be introduced; a smaller value allocates less context space to similar code snippets.  
 799 Through this mechanism, we can achieve flexible and precise control over the different components  
 800 of the information within a limited total resource budget.  
 801

## 802 A.5.3 DETAILS OF DATASET

803 The detailed information of dataset is as follows in the table 6 and table 7.

## 804 A.5.4 CALCULATION OF EVALUATION INDICATORS

805 (1) Code Exact Match(EM): The code exact match measures the proportion of the generated code  
 806    completions that are exactly the same as the ground truth.

810  
811  
812  
813  
814  
815  
816  
817  
818  
Table 6: The dataset of CrossCodeEval

	<b>Python</b>	<b>Java</b>
Total number of repositories	471	239
Total number of documents	14348	5868
Total number of task cases	2665	2139

819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
Table 7: The dataset of RepoEval-Updated

<b>Language</b>	<b>Project Name</b>	<b>Creation time</b>	<b>The number of files</b>	<b>Total project file size (MB)</b>
Python	devchat	2023-04-17	40	0.5
	nemo.aligner	2023-09-01	54	1.6
	awslabs.fortuna	2022-11-17	168	1.9
	task.weaver	2023-09-11	113	3.0
	huggingface_diffusers	2022-05-30	305	6.2
	opendilab_ACE	2022-11-23	425	6.8
	metagpt	2023-06-30	374	17.9
	apple	2023-02-25	265	23.8
	QingruZhang	2023-05-31	1357	32.6
	nerfstudio-project_nerfstudio	2022-05-31	157	54.5
Java	FloatingPoint-MC_MIN	2023-07-10	2628	269.5
	itlemon.chatgpt4j	2023-04-04	67	0.4
	mybatis-flex_mybatis-flex	2023-02-27	487	8.8
	Guiqu1aixi_rocketmq	2023-04-25	988	10.6
	SimonHalvdansson_Harmonic-HN	2023-05-23	51	16.8
	Open-DBT_open-dbt	2023-02-27	366	20.0
	QuasiStellar_custom-pixel-dungeon	2023-05-08	1093	51.3
	gentics.cms-oss	2023-05-08	2580	130.5

833  
834  
835 (2) Identifier Exact Match(ID\_EM): The identifier exact match measures the proportion of the generated code completions that are exactly the same as the ground truth.  
836  
837 (3) Identifier F1 Score: The Identifier F1 score measures the degree of match between the identifiers (variable names, function names, etc.) in the generated code and the actual identifiers. It  
838 combines Precision (correctness) and Recall (completeness).  
839

$$840 \quad F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \quad (3)$$

840 (4) Edit Distance Similarity (ES) : Edit Distance similarity is calculated based on the edit distance  
841 and measures the degree of similarity between the generated code string and the real code string.  
842

$$843 \quad ES = 1 - \frac{ED(S_1, S_2)}{\max(\text{len}(S_1), \text{len}(S_2))} \quad (4)$$

844 Among them,  $\text{len}(S_1)$  and  $\text{len}(S_2)$  are the lengths of  $S_1$  and  $S_2$  respectively.  $ED(S_1, S_2)$  is the  
845 edit Distance between  $S_1$  and  $S_2$ , also known as the Levenshtein Distance(Algorithm 3), which  
846 is usually calculated through Dynamic Programming.  
847

### 848 A.5.5 DETAILS OF BASELINES

849 • **No RAG:** As a basic control experiment, this method only relies on the pre-trained knowledge  
850 base of large language models (LLMs), and directly inputs the current code context into the model  
851 for autoregressive generation. The characteristic of this method lies in completely ignoring the  
852 context information of the code base, and it can be used to evaluate the native reasoning ability of  
853 LLMs in zero-shot scenarios.  
854 • **Shifted RAG:** The core of this method is the sliding window offset mechanism. This mechanism  
855 dynamically adjusts window positions during retrieval, prioritizing code segments likely to  
856 contain target call chains. Through temporal probability prediction, it enhances temporal relevance  
857 between retrieval results and completion targets. The approach demonstrates distinct advantages  
858 in scenarios like API invocation sequences and control flow continuation.  
859 • **Vanilla Rag:** Given the context, retrieve a set of similar code snippets from the repository through  
860 a fixed-size sliding window and call the LLM to obtain the predicted next statement.  
861

---

864                   **Algorithm 3** Levenshtein Distance Calculation

865

866                   **Require:** Two strings:  $str1$  (length  $m$ ),  $str2$  (length  $n$ )

867                   **Ensure:** Edit distance between  $str1$  and  $str2$

868                   Initialize  $dp$  as 2D array of size  $(m + 1) \times (n + 1)$

869                   **for**  $i = 0$  **to**  $m$  **do**

870                     $dp[i][0] \leftarrow i$  {Deletion operations}

871                   **end for**

872                   **for**  $j = 0$  **to**  $n$  **do**

873                     $dp[0][j] \leftarrow j$  {Insertion operations}

874                   **end for**

875                   **for**  $i = 1$  **to**  $m$  **do**

876                    **for**  $j = 1$  **to**  $n$  **do**

877                    **if**  $str1[i - 1] == str2[j - 1]$  **then**

878                     $dp[i][j] \leftarrow dp[i - 1][j - 1]$  {Characters match}

879                    **else**

880                     $dp[i][j] \leftarrow 1 + \min \begin{cases} dp[i][j - 1] & \text{(Insertion)} \\ dp[i - 1][j] & \text{(Deletion)} \\ dp[i - 1][j - 1] & \text{(Substitution)} \end{cases}$

881                    **end if**

882                   **end for**

883                   **end for**

884                   **return**  $dp[m][n]$  {Final edit distance}

---

886

887

- **Repocoder**<sup>5</sup>: This is an iterative retrieval-augmented framework for repository-level code completion. It addresses the challenge of leveraging fragmented repository information by integrating similarity-based retrievers with pretrained code LLMs, enabling precise cross-file completion of unfinished code.
- **Graphcoder**: This is a structured retrieval-augmented code completion framework. Its core innovation lies in employing a graph-based retrieval-generation process, which utilizes Code Context Graphs (CCG) to accurately model code dependencies, replacing traditional sequence-based context representations.

896

897                   A.6 ADDITIONAL RESULTS

898

900                   Table 8: Detailed data of the ablation experiment

901

Language	Methods	Codegen2-7b				Codegen25-7b				CodeLlama			
		Code Match		Identifier Match		Code Match		Identifier Match		Code Match		Identifier Match	
		EM	ES	EM	F1	EM	ES	EM	F1	EM	ES	EM	F1
Python	No Rag	5.44	57.85	11.71	42.22	7.77	60.52	14.45	45.40	9.49	61.97	16.44	47.36
	Shift Rag	4.87	58.36	11.64	42.91	7.44	60.20	14.17	44.78	6.95	60.35	13.75	45.36
	Vanilla Rag	9.52	61.87	17.42	48.01	12.43	63.81	20.74	51.00	11.48	63.66	19.42	50.72
	SARACODER	15.07	66.04	24.71	54.86	18.40	67.95	27.50	56.38	17.94	67.99	27.96	57.00
	- EAID	13.49	65.02	22.86	52.95	15.90	66.80	25.01	54.74	15.90	66.63	25.27	54.98
	- HF_OP	10.96	63.05	19.95	49.90	14.54	65.66	23.57	52.88	13.37	65.63	22.86	53.11
	- CCG	11.11	63.23	20.14	50.19	13.98	65.27	23.27	52.78	13.37	65.28	22.36	52.86
Java	No Rag	0.00	25.92	0.05	17.48	0.00	25.46	0.05	17.61	0.00	25.17	0.00	17.23
	Shift Rag	6.45	54.84	12.11	43.75	6.08	44.73	10.27	36.46	7.11	50.96	12.72	41.68
	Vanilla Rag	9.68	55.71	15.71	45.71	10.47	47.09	15.29	39.93	11.31	51.48	17.81	44.09
	SARACODER	12.16	54.16	18.37	45.47	11.50	44.90	16.22	38.95	13.32	48.04	19.54	42.34
	- EAID	8.88	53.12	14.63	43.12	8.60	42.29	13.00	35.84	9.91	46.89	15.71	40.03
	- HF_OP	8.56	53.08	14.35	43.33	8.51	42.29	12.86	35.84	8.93	46.36	14.96	39.57
	- CCG	8.88	52.67	14.31	42.92	8.18	42.10	12.34	35.57	8.98	45.44	14.68	38.51

915

916

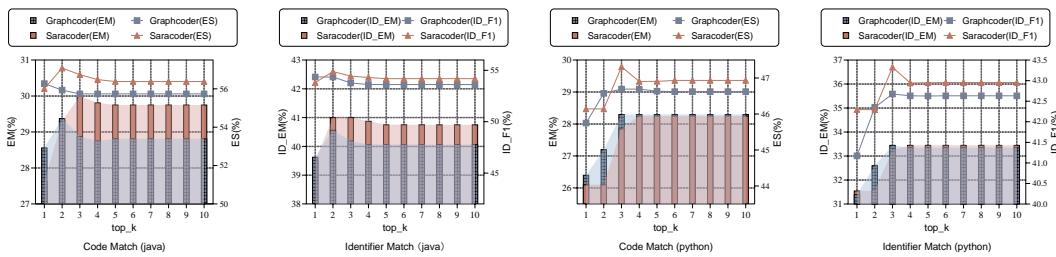
917                   <sup>5</sup>We attempt to run the code publicly released by Zhang et al., but fail to execute them with the provided instructions. For the specific implementation here, please refer to Ding et al.

918  
919

## A.6.1 COST ANALYSIS IN IN-FILE SCENARIOS

920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934

In this Token cost analysis experiment on the efficiency of code completion inference, we used the RepoEval\_Updated dataset, the Deepseeking -coder model for code inference, and Graphcoder as the control group for the experiment. The main comparison is to show the changes of each accuracy index as the number of similar cases retrieved by the code in the prompt word (top-k) increases. As can be seen from Figure 6, whether it is a Graphcoder or SARACODER, with the increase of top\_k, they basically show a trend of first rising, then falling, and finally gradually stabilizing. This indicates that as top-k increases, the noise cases that may be introduced may lead to a decrease in accuracy. SARACODER demonstrates advantages in the vast majority of scenarios: in the python code completion task, the stable values of the three metrics except EM are higher than those of the original retrieval method, among which the ES value increases by 0.31, the ID\_EM value increases by 0.1, and the ID\_F1 value increases by 0.32. In the java code completion task, the stable values of the four indicators have all seen relatively significant improvements, increasing by 0.937, 0.641, 0.687, and 0.568 respectively. In addition, it can be seen from the figure that when the top\_k is between 2 and 4, the performance of the improved retrieval method is significantly better than that of the original retrieval method. This indicates that our method is more suitable for scenarios with scarce computing resources.

935  
936  
937  
938  
939  
940  
941  
942  
943

944

Figure 6: Impact of top\_k on RepoEval-Updated. (The two on the left are Java tasks, and the two on the right are Python tasks.

945

946  
947

## A.6.2 QUANTITATIVE ANALYSIS OF TOP\_K AND TOKEN.

948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959

In our experiments, we focused on minimizing the influence of external factors. To do this, we used the smallest runnable (executable and producing valid output) module, which included both unfinished code context and HE\_OP (Hierarchical Feature Optimization). The unfinished code context provided the input content, while HE\_OP offered code completion reference cases. Importantly, HE\_OP is directly influenced by the top\_k parameter. We conducted our tests using two key top\_k values(4 and 10) and set max\_token\_num to 2048. As shown in the table 9, clearly demonstrate that a top\_k of 4 significantly reduces input token consumption across all three models compared to a top\_k of 10. On average, each task saved approximately 22.38 input tokens when top\_k was set to 4, confirming that a smaller top\_k value leads to lower token consumption. Furthermore, we observed no significant drop in output token count when top\_k was reduced. This, coupled with the results in Figure 6 showing no decline in accuracy, indicates that our method effectively reduces resource consumption while maintaining output quality.

960  
961  
962

Table 9: Quantitative analysis of top\_k and token. The comparison of the average input and output tokens on each task when the dataset is Repo\_Updated and max\_num\_tokens is 2048.

963  
964  
965  
966  
967

Method	Codegen2-7b		deepseek-coder-6.7b-instruct		CodeLlama-7b-Instruct	
	#In	#Out	#In	#Out	#In	#Out
context	821.627	95.15	778.52	93.48	763.93	96.86
context+HF_OP (top_k=10)	1665.50	88.15	1621.92	90.86	1611.37	96.99
context+HF_OP (top_k=4)	<b>1644.26</b>	89.20	<b>1596.83</b>	88.57	<b>1590.55</b>	98.06

968

969

## A.6.3 CASE STUDY IN SYNERGISTIC GAIN PROPERTY

970  
971

To better understand the causes of synergistic gains, we analyzed our experimental cases. SARACODER’s External-Aware Identifier Disambiguator effectively resolves “type inference chain break-

age” in cross-file dependencies by injecting essential symbolic relationships. Still, it occasionally introduces irrelevant information, which can lead to misinterpretations. Repocoder, when used independently, offers prompts that closely align with common coding patterns. However, it faces challenges with the adaptability of external information, often clashing with local APIs or project constraints, thereby limiting its accuracy. Draco stands out for its deep semantic modeling, which generates detailed data and control flow graphs to pinpoint highly relevant cross-file context; nonetheless, it encounters difficulties when the code’s intent is unclear. SARACODER significantly contributes by providing semantically aligned code examples. These examples offer crucial “intent hinting” and “structure references,” compensating for Draco’s limitations in ambiguous code scenarios. As a result, the “Draco + SARACODER” combination synergistically boosts performance: Draco delivers precise cross-file context, while SARACODER guides intent and structure. Moreover, the external disambiguation module within SARACODER clarifies identifiers, effectively alleviating Repocoder’s issues with external information adaptability and conflicts, making the “Repocoder + SARACODER” combination a more effective choice than using Repocoder in isolation.

#### 986 A.7 THE USE OF LARGE LANGUAGE MODELS (LLM)

988 In order to enhance the language quality and clarity of this academic paper, the author utilized AI-  
989 powered tools for text refinement during the writing process. The specific details are as follows:

990 **Purpose of Use:** The primary purposes for using AI tools were to:

- 992 • Check grammar and spelling for certain sentences.
- 993 • Optimize vocabulary choices for more precise and academic expression.
- 994 • Adjust sentence structures to improve logical coherence and readability between paragraphs.

996 **Method of Use:** The author input original paragraphs written by themselves into the AI tools and  
997 then manually judged, filtered, and revised the text based on the refinement suggestions provided.  
998 All adopted changes were carefully considered by the author to ensure they fully align with the  
999 original intent and academic rigor of the paper.

1000 **Disclaimer of Responsibility:** All academic content in this paper, including core arguments, re-  
1001 search data, result analysis, argumentation process, and final conclusions, was independently created  
1002 and is the sole responsibility of the author. The AI tools were used purely as an auxiliary aid and did  
1003 not generate any critical academic viewpoints, research data, or conclusions. The author assumes  
1004 full responsibility for the final content of the paper.

1005 **Tools Used:** The AI tools used in this process were: Gemini-2.5 Flash, deepseek-V3.1.