GraphCoder: Enhancing Repository-Level Code Completion via Coarse-to-fine Retrieval Based on Code Context Graph

Wei Liu*
weiliu@stu.pku.edu.cn
Key Lab of High Confidence Software Technologies (PKU), MoE, China
School of Computer Science, PKU
Beijing, China

Ailun Yu*
yuailun@pku.edu.cn
Key Lab of High Confidence Software Technologies (PKU), MoE, China
School of Computer Science, PKU
Beijing, China

Daoguang Zan*
daoguang@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Bo Shen
shenbo21@huawei.com
Huawei Cloud Computing Technologies Co., Ltd.
Beijing, China

Wei Zhang†
zhangw.sei@pku.edu.cn
Key Lab of High Confidence Software Technologies (PKU), MoE, China
School of Computer Science, PKU
Beijing, China

Haiyan Zhao
zhhy.sei@pku.edu.cn
Key Lab of High Confidence Software Technologies (PKU), MoE, China
School of Computer Science, PKU
Beijing, China

Zhi Jin†
zhijin@pku.edu.cn
Key Lab of High Confidence Software Technologies (PKU), MoE, China
School of Computer Science, PKU
Beijing, China

Qianxiang Wang
wangqianxiang@huawei.com
Huawei Cloud Computing Technologies Co., Ltd.
Beijing, China

Abstract

The performance of repository-level code completion depends upon the effective leverage of both *general* and *repository-specific* knowledge. Despite the impressive capability of code LLMs in general code completion tasks, they often exhibit less satisfactory performance on repository-level completion due to the lack of repository-specific knowledge in these LLMs. To address this problem, we propose GraphCoder, a retrieval-augmented code completion framework that leverages LLMs' general code knowledge and the repository-specific knowledge via a *graph-based retrieval-generation* process. In particular, GraphCoder captures the context of completion target more accurately through *code context graph* (CCG) that consists of control-flow, data- and control-dependence between code statements, a more structured way to capture the completion target context than the sequence-based context used in existing retrieval-augmented approaches; based on CCG, GraphCoder further employs a *coarse-to-fine* retrieval process to locate context-similar code snippets with the completion target from the current repository. Experimental results demonstrate both the effectiveness and efficiency of GraphCoder: Compared to baseline retrieval-augmented methods, GraphCoder achieves higher exact match (EM) on average, with increases of +6.06 in code match and +6.23 in identifier match, while using less time and space.

CCS Concepts

• **Software and its engineering** → **Search-based software engineering**; • **Information systems** → **Language models**; **Query representation**; • **Mathematics of computing** → **Graph algorithms**.

Keywords

Code completion, Large language model, Retrieval augmented generation, Code graphs

ACM Reference Format:
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 *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695054

*The first three authors contributed equally to this work.
†Corresponding authors.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695054

1 Introduction

Code Large Language Models (LLMs), such as Codex [3], StarCoder [19] and Code Llama [33], have demonstrated impressive capability in general code completion tasks [42, 45, 46]. These transformer-based [39] large language models encode and compress extensive code knowledge into billions or even trillions of

parameters through training on vast code corpora. Some of these LLMs have been deployed as auto-completion plugins (e.g., GitHub Copilot[1], CodeGeeX[2]) in modern Integrated Development Environments (IDEs), and successfully streamline the real-world software development activities to a certain degree.

However, compared with their performance in general scenarios, code LLMs exhibit less satisfactory performance in repository-level code completion tasks, due to the lack of repository-specific knowledge in these LLMs [38, 41, 44]. Specifically, the repository-specific knowledge (including code style and intra-repository API usage) cannot be well learned by or even inaccessible to code LLMs during their pre-training and fine-tuning phases, particularly for those newly created, personal privately owned, or confidential business repositories. One superficial remedy to this knowledge-lack problem is to concatenate all the code files in the repository as the prompt to LLMs in the situation that the size of LLMs' context window is continuously growing. However, this kind of remedy puts too much irrelevant information into the prompt, bringing unnecessary confusion to LLMs and thus leading to degraded completion performance [34, 40].

To mitigate the knowledge-lack problem mentioned above, several methods have been proposed following the RAG pattern of *retrieval-augmented generation* [24, 27, 44]. For each completion task, RAG first retrieves a set of context-similar code snippets from the current repository, and then injects these snippets into the prompt, with the hope of improving the generation results of code LLMs; these retrieved snippets play the role of augmenting code LLMs with the repository-specific knowledge related to a completion task. As a result, the effectiveness of RAG largely depends on how to define the relevance between a code snippet and a completion task. Most existing RAG methods follow the classical NLP style and locate a set of related code snippets of a completion task by considering sequence-based context similarity.

In this paper, we follow the RAG pattern for repository-level code completion, but explore a more structured style to locate relevant code snippets of a completion task. Specifically, we propose GraphCoder, a graph-based RAG code completion framework. The key idea of GraphCoder is to capture the context of a completion task by leveraging the structural information in the source code via an artifact called *code context graph (CCG)*. In particular, a CCG is a statement-level multi-graph that consists of a set of statements as vertices, as well as three kinds of edges between statements, namely *control flow*, and *data/control dependence*. The CCG contributes to improving retrieval effectiveness from three aspects: (1) Replacing sequence representation of code with structured representation to capture more relevant statements of the completion task; (2) Augmenting the sequence-based similarity between the context of two statements with structure-based similarity to identify deeply matched statements of the completion target from the repository; (3) Adopting a *decay-with-distance* structural similarity to weight the different importance of context statements to the completion target. Experiments based on 8000 real-world repository-level code completion tasks demonstrate the effectiveness of GraphCoder. GraphCoder more accurately retrieves relevant code snippets with increases of +6.06 in code exact match and +6.23 in identifier exact match on average compared to RAG baseline methods while using less retrieval time and database storage space.

To summarize, our main contributions are:

- An approach GraphCoder to enhance the effectiveness of retrieval by a coarse-to-fine process, which considers both structural and lexical context, as well as the dependence distance between the completion target and the context;
- A graph-based representation CCG (code context graph) of source code to capture relevant long-distance context for predicting the semantics of code completion target instead of the widely adopted sequence-based one;
- Extensive experiments upon 5 LLMs and across 8000 code completion tasks from 20 repositories demonstrate that GraphCoder achieves higher exact match values with reduced retrieval time and overhead in database storage space.

2 Related Work

Repository-level Code Completion. The task of repository-level code completion is gaining significant attention for intelligent software development in real-world scenarios [6, 20, 35, 36, 44]. Recently, a growing number of large language models (LLMs) have been emerged and demonstrated superior performance in general code completion tasks [1, 11, 23, 33]. However, they demonstrate limited performance on repository-level code completion tasks due to a lack of knowledge [5, 21, 38, 41, 44]. To address this issue, existing methods inject repository-level knowledge into LLMs either by fine-tuning them [6, 35] or by directly employing pre-trained models [17, 24, 37, 38, 44]. Representative fine-tuning methods, such as CoCoMIC [6] and RepoFusion [35], train the language model using both in-file and relevant cross-file contexts to inject knowledge into LLMs. However, challenges persist due to the infeasibility of applying these methods to closed-source LLMs and the dynamic nature of repository-level features driven by continuous project development. To mitigate this problem, a series of methods that directly utilize pre-trained models have been proposed [17, 24, 38, 44]. Khandelwal et al. [17] and Tang et al. [38] propose a post-processing framework that adjusts the probability for the next token output by LMs with repository-level token frequency. Nevertheless, these methods are sensitive to manually selected interpolated weights. With the emergence of code LLMs demonstrating remarkable code comprehension capabilities, several approaches [20, 24, 37, 44] have adopted a pre-processing strategy, that retrieves relevant snippets and adds them into LLMs' prompt. Additionally, benchmarks like RepoEval [44], RepoBench [21], and CrossCodeEval [5] have been introduced to advance the study in this field by systematically evaluating the performance of repository-level completion methods.

Retrieval-augmented Code Completion. Retrieval-augmented code completion is a technique that aims to integrate domain-specific knowledge into LLMs. This technique typically first extracts the context of the completion target, then retrieves relevant code snippets, and finally concatenates the retrieved code snippets with the original context to guide the generation of LLM. To model the context of completion target, most existing methods follow the basic idea of natural language processing, which directly extracts the last few lines of the completion target as its context and then uses it

[1]https://github.com/features/copilot
[2]https://codegeex.cn

for retrieval [17, 24, 37, 38, 44]. However, these methods ignore the intrinsic structure underlying the code. Inspired by this, both the CoCoMIC [6] and the RepoHyper [28] construct the method-level graph to facilitate the retrieval step. Nonetheless, they still overlook the statement-level structure, which is crucial for understanding the semantics of the completion context. For the generation step, it includes two distinct modes [18, 37]: per-token and per-output generation. In the per-token generation, a retrieval process is initiated for each generated token, so each token is associated each a unique set of retrieval code snippets, such as the methods kNN-LM [17], kNM-LM [38], and FT2Ra [12]. Consequently, the number of retrievals grows as the length of the generated tokens increases, leading to a significant increase in retrieval time. Additionally, the requirement to access each stage of token generation limits the method's compatibility with closed-source LLMs. In the per-output generation, a single set of retrieval code snippets is used to produce a whole sequence at once [24, 28, 44], thus improving the retrieval efficiency and facilitating compatibility with closed-source LLMs.

3 Basic Concepts

In this section, we introduce two concepts used in GraphCoder, namely *code context graph* (CCG) and *CCG slicing*. The former is employed to transform a code snippet into a structured representation (i.e., a set of statements as well as a set of structural relationships between them). Given a statement x in a CCG G, the latter is used to extract a G's subgraph that consists of x and x's h-hop depended elements as well as relationships between them.

3.1 Code Context Graph

A code context graph is the superimposition of three kinds of graphs about code: control flow graph (CFG), control dependence graph (CDG), and data dependence graph (DDG). The latter two graphs together are commonly identified as program dependence graph [8].

Definition 1 (Code Context Graph). *A code context graph $G = (X, E, T, \lambda)$ is a directed multi-graph, where*

- *$X = \{x_1, \cdots, x_n\}$ is the vertex set, each of which represents a code statement or a predicate;*
- *$E = \{e_1, \cdots, e_m\}$ is the edge set; each edge is a triple (x_i, t, x_j) where $x_i, x_j \in X$, and $t \in T$ denoting the edge type;*
- *$T = \{CF, CD, DD\}$ is the edge type set, where CF denotes the* control-flow *edge, CD the* control dependence*, and DD the* data dependence*;*
- *λ is a function that maps each edge in E to its type in T, i.e., for $e = (x_i, t, x_j)$, $\lambda(e) = t$.*

Control flow graphs (CFG) provide a detailed representation of the order in which statements are executed [2, 9, 22]. The vertices of CFG represent statements and predicates. The edges indicate the transitions of control between statements, including the sequential executions, jumps, and iterative loops. The construction of a control flow graph is based on the abstract syntax tree (AST): Initially, statements and predicates are identified, and the sequential execution order is extracted from the AST. Subsequently, control transfer edges are added by analyzing conditional statements (e.g., if, for), iterative statements (e.g., for, while), and jump statements (e.g., continue, break).

Algorithm 1: CCG Slicing

Input : CCG graph $G = (X, E, T, \lambda)$, statement of interest $\tilde{x} \in X$, maximum hops h, and maximum number of statements l.
Output : A CCG slicing graph $G_h^l(\tilde{x})$.

1 Initialize sets X_{CD}, X_{CF} and X_{DD} as $\emptyset$;
2 Push $\tilde{x}$ into an empty queue q;
3 **while** *q is not empty* **do**
4 $\quad x \leftarrow q.pop()$;
5 $\quad$ **if** *x exceeds h hops from $\tilde{x}$* **then** break ;
6 $\quad X_{CF} \leftarrow X_{CF} \cup \{x\}$;
7 $\quad X_{DD} \leftarrow X_{DD} \cup \{z \mid (z, DD, x) \in E\}$;
8 $\quad X_{CD} \leftarrow X_{CD} \cup \{z \mid (z, CD, x) \in E\}$;
9 $\quad$ **if** $|X_{CF} \cup X_{CD} \cup X_{DD}| \geq l$ **then** break ;
10 $\quad$ **for** $z \in \{z \mid (z, CF, x) \in E, z \notin X_{CF}\}$ **do**
11 $\quad\quad$ **if** *z has not been visited by q* **then** $q.push(z)$;
12 $\quad$ **end**
13 **end**
14 $G_h^l(\tilde{x}) \leftarrow G[X_{CF} \cup X_{DD} \cup X_{CD}]$;
15 **return** $G_h^l(\tilde{x})$

Control dependence graphs (CDG) focus on identifying the control dependencies between statements, with edges emphasizing the direct influence of one statement on the execution of another [4, 8, 25]. Specifically, an edge exists between two statements if one directly affects the execution of another, distinguishing it from the CFG. Based on the CFG, CDG can be constructed by analyzing the statement reachability.

Data dependence graphs (DDG) reflect the dependencies arising from variable assignments and references, where edges represent that there is a variable defined in one statement is used by another [8, 13]. The DDG can be generated through a two-step process: First, we identify the set of variables defined and used by each statement, respectively. Second, for a variable v, a DDG edge is established between two statements if there exists a CFG path from the statement defining v to the statement using v, without intervening definitions of v.

3.2 CCG Slicing

Definition 2 (CCG Slice). *Given a code context graph $G = (X, E, T, \lambda)$ and a statement of interest $\tilde{x} \in X$, the h-hop CCG slice of $\tilde{x}$ in G with maximum l statements, denoted as $G_h^l(\tilde{x})$, is defined by the output of Algorithm 1.*

Algorithm 1 outlines the CCG slicing process to capture the context of a given statement $\tilde{x}$ in graph G. The key idea is to extract an induced subgraph of G with vertices within h hops of control-flow neighbors of $\tilde{x}$, along with the vertices they have data and control dependence on, limited to a maximum of l vertices. Starting from $\tilde{x}$ (lines 2, 3, and 5), Algorithm 1 first updates current visited control-flow neighbors set X_{CF} (line 7), and then adds its data dependence (DD) in-neighbors to X_{DD} (line 8) and its control dependence(CD) in-neighbors to X_{CD} (line 9). After that, Algorithm 1 pushes its control-flow (CF) in-neighbors to queue for the next traversing step (lines 11-13). The final output of Algorithm 1 is the induced subgraph of G whose vertex set is $X_{CF} \cup X_{CD} \cup X_{DD}$.

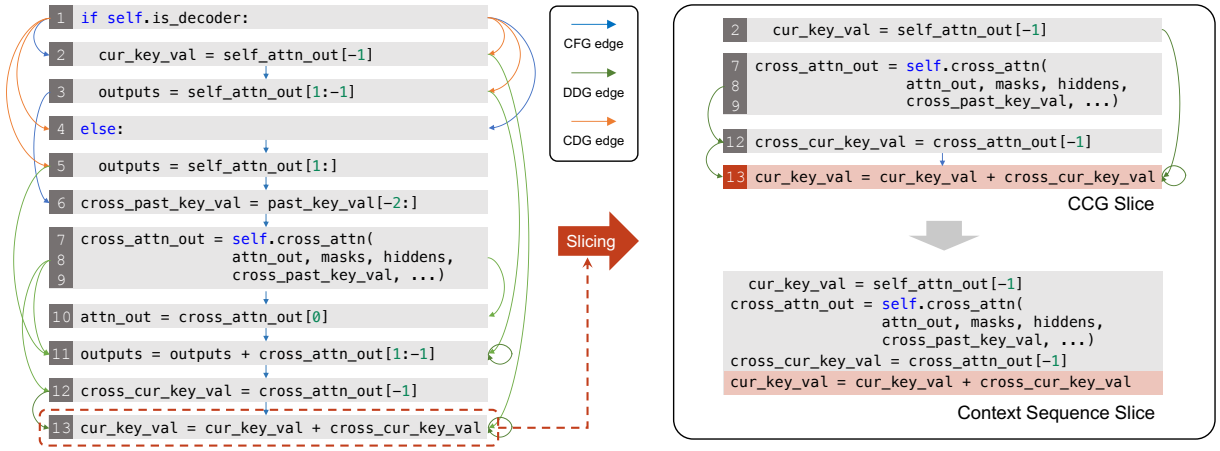

Figure 1: An example of the code context graph (CCG) and its CCG slice with statement of interest $\tilde{x} = 13$.

Fig. 1 provides an example of a code snippet along with its corresponding CCG and a CCG slice. The code snippet, comprising 13 lines, contains a total of 11 statements, 11 *CF* edges, 9 *DD* edges, and 4 *CD* edges. Focusing on a statement of interest (line 13), its one-hop CCG slice includes all statements it has data and control dependence on (lines 2, 12, and 13), as well as its one-hop control-flow in-neighbor (line 12) and its in-neighbor's data and control dependence (lines 7-9). The context sequence slice consists of all statements in the CCG slice, ordered by line number.

4 GraphCoder

4.1 Overview

GraphCoder is a graph-based framework for repository-level code completion tasks. In general, a code completion task aims to predict the next statement $\tilde{y}$ for a given context $X = \{x_1, x_2, \cdots, x_n\}$. Fig. 2 gives an overview of GraphCoder's workflow. Given a context in a code repository, GraphCoder completes the code through three steps: *database construction*, *code retrieval*, and *code generation*.

- In the *database construction* step (Section 4.2), GraphCoder constructs a key-value database that maps each statement's CCG slice to the statement's forward and backward l lines of code.
- In the *code retrieval* step (Section 4.3), GraphCoder takes a code completion context as input and retrieves a set of similar code snippets through a *coarse-to-fine* grained process. In the coarse-grained sub-process, GraphCoder filters out top-k candidate code snippets based on the similarity of context sequence slice; in the fine-grained sub-process, the candidate snippets are re-ranked by a *decay-with-distance* structural similarity measure.
- In the *code generation* step (Section 4.4), GraphCoder generates a prompt by concatenating the fine-grained query result and the code completion context, and then feeds the prompt into an LLM, waiting for the LLM to return a predicted statement $\tilde{y}$ of the code completion context.

4.2 Database Construction

Given a code repository, we establish a key-value database $\mathcal{D}$. For each statement x_i in the code repository, a key-value is generated

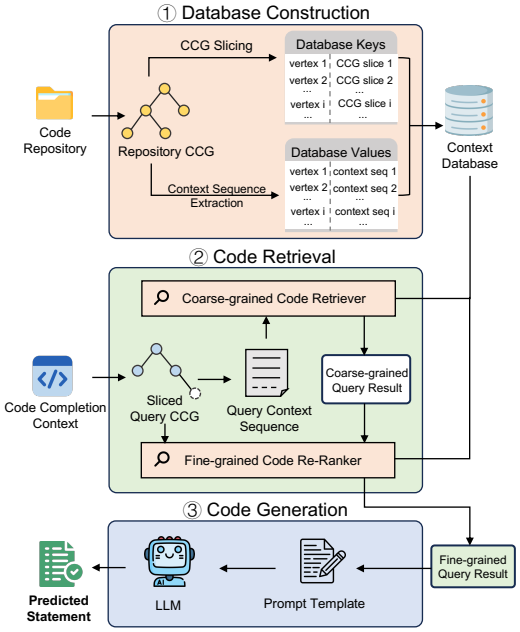

Figure 2: An illustration of GraphCoder framework.

and stored in $\mathcal{D}$: the *key* is x_i's CCG slice $G_h^l(x_i)$, and the *value* is x_i's forward and backward l lines of code, i.e., $\{x_{i-l/2}, \cdots, x_i, x_{i+l/2}\}$ centered around x_i.

4.3 Code Retrieval

The code retrieval step takes a completion context X as input, and outputs a set of code snippets, through three sub-steps: query CCG construction, coarse-grained retrieval, and fine-grained re-ranking.

Query CCG construction. GraphCoder initially extracts the sliced query CCG of the completion target. Specifically, it converts the given context X to its CCG representation G. A dummy vertex $\tilde{y}$ is then added to G to represent the statement to be predicted. An assumption is made that there exists a control-flow edge from the

last statement x_n in X to the statement to be predicted $\tilde{y}$. The sliced query CCG is then obtained by slicing from $\tilde{y}$, denoted as $G_h^l(\tilde{y})$.

Coarse-grained retrieval. Given a sliced query CCG $G_h^l(\tilde{y})$, the coarse-grained retrieval step outputs the top-k most similar results in $\mathcal{D}$ based on coarse-grained similarity. The coarse-grained similarity($CSim$) between $G_h^l(\tilde{y})$ and a key $G_h^l(x)$ in $\mathcal{D}$ is calculated as follows:

$$CSim(G_h^l(\tilde{y}), G_h^l(x)) = sim(X_h^l(\tilde{y}), X_h^l(x))$$

where $X_h^l(\hat{y})$ and $X_h^l(x_i)$ denotes the context sequence slice based on $G_h^l(\hat{y})$ and $G_h^l(x_i)$, respectively. sim denotes any similarity applicable to code sequences, including sparse retriever BM25 [32], Jaccard index [15] based on the bag-of-words model, as well as dense retrievers like similarity of embeddings from CodeBERT [7] and GraphCodeBERT [10].

Fine-grained re-ranking. In this step, GraphCoder re-ranks the coarse-grained query result based on the decay-with-distance subgraph edit distance. The subgraph edit distance (SED) is the minimum cost of transforming one graph into a subgraph of another one through a series of edit operations [29, 43]. The subgraph edit operations include the deletion and the substitution of vertex or edges. For a vertex v and an edge e in $G_h^l(\hat{y})$, the edit cost function $c(\cdot)$ is defined as follows:

- Vertex deletion cost $c(v) = 1$;
- Vertex substitution cost $c(v,u) = 1 - sim(v,u)$;
- Edge deletion cost $c(e) = 1$;
- Edge substitution cost $c(e, e') = \mathbf{1}_{\lambda(e)\neq\lambda(e')}$.

where sim denotes any similarity measure for code sequences, and the substitution cost of the dummy vertex $\tilde{y}$ for any other vertex is assumed to be 0.

Since the subgraph edit distance problem is NP-hard [14, 43], we calculate it by extending the quadratic-time greedy assignment (GA) algorithm [30, 31] with a decay-with-distance factor. Specifically, we first obtain an alignment $\mathcal{A}$ between the vertices in $G_h^l(\hat{y})$ and $G_h^l(x)$ by the GA algorithm [30]. Subsequently, we accumulate the edit costs as indicated by $\mathcal{A}$, as described in Algorithm 2. The aligned vertex pairs in $\mathcal{A}$ reflects the vertex substitution relationship between $X_h^l(\hat{y})$ and $X_h^l(x)$. For a vertex v in $G_h^l(\hat{y})$, we denote the $\mathcal{A}(v)$ as its aligned vertex in $G_h^l(x)$. Let $X_{\mathcal{A}}$ be $\{v \mid v \in X_h^l(\hat{y}), (v,u) \in \mathcal{A}\}$, $E_{\mathcal{A}}$ be $\{e \mid e = (v,t,u) \in G_h^l(\tilde{y}), (\mathcal{A}(v), t', \mathcal{A}(u)) \in G_h^l(x)\}$, and $h(v,\tilde{y})$ be the number of hops from $\tilde{y}$ to v, the decay-with-distance SED determined by $\mathcal{A}$ is calculated in Alg. 2.

4.4 Code Generation

After obtaining a set of retrieved code snippets, GraphCoder employs an external LLM as a black box to generate the next statement of the given code completion context X. Following the commonly-used practice [44] of retrieval-augmented prompt formatting, we arrange the retrieval code snippets in ascending similarity order, each of which is accompanied by its original path file; then these arranged code snippets are concatenated by the code completion context X as the final prompt of the LLM as shown in Fig. 3.

Algorithm 2: Decay-with-distance SED

Input : Graphs $G_h^l(\hat{y})$ and $G_h^l(x)$ as well as a decay-with-distance factor γ.
Output : Decay-with-distance SED between $G_h^l(\hat{y})$ and $G_h^l(x)$.

$SED \leftarrow 0$;
for $v \in X_{\mathcal{A}}$ **do**
  $SED \leftarrow SED + \gamma^{h(v,\tilde{y})} c(v, \mathcal{A}(v))$;
end
for $v \in X_h^l(\hat{y}) \setminus X_{\mathcal{A}}$ **do**
  $SED \leftarrow SED + \gamma^{h(v,\tilde{y})} c(v)$;
end
for $e = (v,t,u) \in E_{\mathcal{A}}$ **do**
  $SED \leftarrow SED + \gamma^{h(v,\tilde{y})} c(e, \mathcal{A}(e))$;
end
for $e = (v,t,u) \in E_h^l(\hat{y}) \setminus E_{\mathcal{A}}$ **do**
  $SED \leftarrow SED + \gamma^{h(v,\tilde{y})} c(e)$;
end
return SED;

```
# Here are some relevant code fragments from other files of the repo:
# --------------------------------------------------
# The below code fragment can be found in:
# huggingface_diffusers/tests/test_pipelines_common.py
# --------------------------------------------------
#  model = self.model_class(**init_dict)
#  model.to(torch_device)                                   Retrieved Code
#  model.eval()                                             Snippets
#
#  with tempfile.TemporaryDirectory() as tmpdirname:
#      model.save_pretrained(tmpdirname)
#      new_model = self.model_class.from_pretrained(tmpdirname)
#      new_model.to(torch_device)
# --------------------------------------------------
# Based on above, complete the next statement of the following codes:

  # Warmup pass when using mps (see #372)
  if torch_device == "mps":
      _ = pipe(**self.get_dummy_inputs(torch_device))

      inputs = self.get_dummy_inputs(torch_device)          Code Completion
                                                            Context
      output = pipe(**inputs)[0]

      with tempfile.TemporaryDirectory() as tmpdir:
         pipe.save_pretrained(tmpdir)

         pipe_loaded = self.pipeline_class.from_pretrained(tmpdir)   Predicted
                                                                     Statement
```

Figure 3: Prompt template used in GraphCoder.

5 Experimental Setup

To evaluate the performance of GraphCoder, we have formulated the following four research questions (RQs):

- **RQ1 (Effectiveness):** How does GraphCoder perform compared with other methods for repository-level code completion tasks?
- **RQ2 (Generalizability):** How does GraphCoder perform with different base model sizes and across various repositories?
- **RQ3 (Ablation):** How does each internal component of GraphCoder influence its performance?
- **RQ4 (Cost):** What is the resource consumption of GraphCoder compared with other methods?

5.1 An Updated Dataset: RepoEval-Updated

The dataset RepoEval-Updated is used for repository-level code completion evaluation. In particular, RepoEval-Updated is derived from the benchmark RepoEval [44], which consists of a set of repository-level code completion tasks constructed from a collection of GitHub Python repositories created between 2022-01-01 and

2023-01-01. RepoEval-Updated refreshes RepoEval by making two key changes: (1) It removes repositories created before March 31, 2022, and adds more recent repositories created after January 1, 2023. This update helps prevent data leakage for most existing code LLMs, whose training data was released before 2023. (2) Following previous work [21, 37], RepoEval-Updated adds open-source Java repositories of GitHub to include a variety of programming languages. The details of repositories are shown in Table 1.

Following established work [44], we divide the repository-level code completion tasks into two categories based on their complexity, namely *line-level* and *API-level* tasks:

- **(Easy) Line-level tasks**: A line-level task is generated by randomly removing a code line and adhering to criteria that the target completion lines are not code comments, contain at least 5 tokens, and then encapsulating its forward code snippet as a completion task.
- **(Hard) API-level tasks**: An API-level task is generated in a similar way except that the removed code line includes at least one intra-repository defined API invocation.

For each task level and programming language, we randomly sample 2000 repository-level completion tasks, thus forming 8000 (2000 × 2 task levels × 2 programming languages) tasks in total.

5.2 Evaluation Metrics

Following the established practice [5, 6], we evaluate the performance of RAG using the following metrics:

- **Code match**: To evaluate the level of code matching, we use two string-based metrics: exact match (EM) and edit similarity (ES). The EM is a binary metric that takes the value of 1 if the predicted code equals to y, and 0 otherwise. The ES is a more fine-grained evaluation and is calculated as $ES = 1 - Lev(y, \tilde{y})/\max(|y|, |\hat{\tilde{y}}|)$, where Lev represents the Levenshtein distance.
- **Identifier match**: We evaluate identifier matching, such as API and variable names, with two metrics: EM and F1 score. To calculate these two metrics, we first extract the identifiers from $\tilde{y}$ and y, and then directly compare their identifiers to obtain the EM and F1 scores.

5.3 Methods for Comparison

As GraphCoder focuses on improving retrieval results by incorporating statement-level structural information into code context instead of the widely adopted sequence-based one, we select the following four methods to evaluate its effectiveness:

- **No RAG.** This method simply feeds the code completion context into an LLM and takes the output of the LLM as the predicted next statement.
- **Vanilla RAG.** Given a context, this method retrieves a set of similar code snippets from a repository via a fixed-size sliding window and invokes an LLM to obtain a predicted next statement.
- **Shifted RAG.** This method is similar to vanilla RAG, except that it returns the code snippet in the subsequent window that is more likely to include the invocation example of target code. This method is also mentioned in ReAcc [24].

Table 1: Statistics of repositories in RepoEval-Updated. The former 10 code repositories are in Python, and the latter 10 are in Java. * corresponds to the repositories of original benchmark. All the newly added repositories are archived on 2024-05-16. #Files indicates the number of Python/Java files in the repository. Statistics are accurate as of May 2024.

Repo name	Created at	#Files	Size (MB)
devchat-ai/devchat	2023-04-17	40	0.5
NVIDIA/NeMo-Aligner	2023-09-01	54	1.6
awslabs/fortuna*	2022-11-17	168	1.9
microsoft/TaskWeaver	2023-09-11	113	3.0
huggingface/diffusers*	2022-05-30	305	6.2
opendilab/ACE*	2022-11-23	425	6.8
geekan/MetaGPT	2023-06-30	374	17.9
apple/axlearn	2023-02-25	265	23.8
QingruZhang/AdaLoRA	2023-05-31	1357	32.6
nerfstudio-project/nerfstudio*	2022-05-31	157	54.5
itlemon/chatgpt4j	2023-04-04	67	0.4
Aelysium-Group/rusty-connector	2023-02-25	133	2.6
neoforged/NeoGradle	2023-07-08	129	3.3
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	2023-05-12	1093	51.3
gentics/cms-oss	2023-05-08	2580	130.5
FloatingPoint-MC/MIN	2023-07-10	2628	269.5

- **RepoCoder** [44]. A sliding window-based method that locates the completion target through an iterative retrieval and generation process. In each iteration, RepoCoder retrieves the most similar code snippets based on the code LLMs' generation results from the last iteration.

5.4 Implementation Details

5.4.1 Code Retrieval. To ensure a fair comparison, we use the same measure to compute the similarity between code sequences across different methods for comparison. Specifically, we employ a sparse bag-of-words model, known for its effectiveness in retrieving similar code snippets [24, 44], a model that transforms code snippets into sets of tokens and calculates similarity using the Jaccard Index [15]. For sliding window-based methods (Vanilla RAG, Shifted RAG, and RepoCoder), we fix the window size as 20 lines and a default sliding stride of 1. For GraphCoder, its maximum hop h is set to 5, the maximum number of statements l is set to 20, and the decay-with-distance factor is set to 0.1. To construct CCG, we first build the abstract syntax tree (AST) of a code snippet by utilizing tree-sitter[3], and then identify the statements within the code snippet and performing control-flow/dependencies analysis techniques.

5.4.2 Code Generation. To avoid data leakage, we exclude in our consideration those LLMs without an explicit training data timestamp or a timestamp after 2023-01-01. Among the remaining LLMs, we select 5 LLMs with diverse code understanding capabilities: GPT-3.5-Turbo-Instruct [4], StarCoder 15B [19], and CodeGen2 models (1B, 3.7B, 7B, and 16B) [26]. Following established practice in code completion [44], we fill the LLMs' context window with two parts:

[3]https://tree-sitter.github.io/tree-sitter/
[4]https://platform.openai.com/docs/models/gpt-3-5-turbo

the retrieved code snippets, and the completion context. Each part occupies half of the context window. The maximum number of retrieved code snippets k is 10. The maximum number of tokens in the generated completion is set to 100. The temperature of LLMs is set to 0 to ensure reproducibility.

Notice that CCG is language-agnostic, GraphCoder can be migrated to other programming languages by following the same procedure. In our experiments, we focus on Python and Java as proof-of-concept languages to showcase GraphCoder's performance. All experiments are conducted on a cluster equipped with 14 Xeon Gold 6330 CPUs and NVIDIA A100-80GB GPU.

6 Experimental Results

6.1 RQ1: Effectiveness

In this subsection, we study the effectiveness of GraphCoder compared with baseline methods both quantitatively and qualitatively.

6.1.1 Quantitative analysis of effectiveness. Table 2 shows the completion results on line-level and API-level tasks across different methods. Across all LLMs, GraphCoder outperforms other baselines for both API-level and line-level code completion tasks. This result demonstrates the benefits of utilizing the structural context extracted based on CCG for locating relevant code snippets to the completion target. Compared to the vanilla RAG, GraphCoder increases the code match EM values on API-level and line-level tasks by +4.58 and +7.90 on average, respectively. This observation emphasizes the effectiveness of GraphCoder's retrieval in repository-level code completion scenarios. Furthermore, compared with other sliding window-based RAG methods (Vanilla RAG, Shifted RAG, and RepoCoder), GraphCoder exhibits superior performance with higher code match scores and identifier match scores. Notably, an observation from Table 2 indicates that Shifted RAG's shifting approach does not necessarily enhance No RAG completion performance. However, shifting all retrieved code snippets without considering their content may lead to the retrieval of totally irrelevant code snippets, introducing potential confusion for LLMs. Additionally, the effectiveness of RAG methods is more evident on harder API-level tasks, where performance is generally lower than on line-level tasks. This observation emphasizes the necessity of retrieving relevant code snippets from repositories for API-level tasks in real-world scenarios.

6.1.2 Qualitative analysis of effectiveness. To further investigate the differences among various methods, we analyze the number of tasks that they complete correctly by the Venn diagrams shown in Fig. 4. It is observed that GraphCoder completes the highest number of tasks that none of the other methods can correctly complete on Java and API-level Python tasks. By re-examining the experimental results, we observe that GraphCoder's superiority mainly derives from its structural alignment-based similarity (i.e., decay-with-distance subgraph edit distance) to locate structure deeply-matched code snippets. As shown in Fig. 5, the code snippet retrieved by GraphCoder is better aligned with the code completion context compared to RepoCoder. Consequently, GraphCoder's snippet provides more relevant information, enabling the LLM to correctly generate the next statement. Additionally, there is a small proportion of tasks that are correctly completed by all RAG

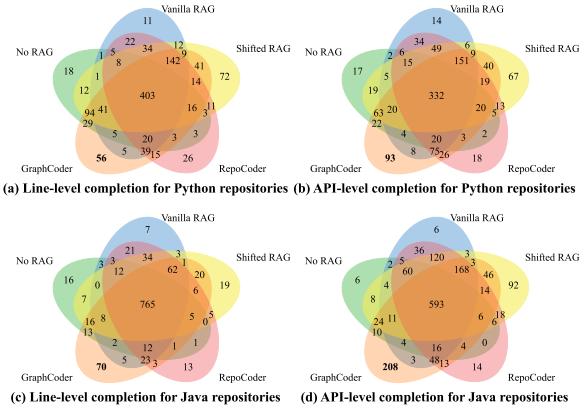

Figure 4: Venn diagram of completion results on GPT3.5-Turbo-Instruct model of different methods. It shows the number of tasks that are completed correctly.

methods except GraphCoder. This occurs because retrieved code snippets can sometimes be misleading. Specifically, when the context of retrieved code snippets is very similar to that of the current code completion task, LLMs tend to directly copy the subsequent statement of the retrieved snippets without adapting to the slightly different completion context.

Answer to RQ1: *On 8000 repository-level code completion tasks, GraphCoder demonstrates superior performance compared to baseline RAG methods, achieving a higher EM in both code match (+6.06) and identifier match (+6.23) on average.*

6.2 RQ2: Generalizability

In this subsection, we explore the generalizability of GraphCoder by examining its performance on base models of different sizes and on repositories with various code duplication ratios.

6.2.1 Generalizability across base models of different sizes. Fig. 6 shows the performance of RAG methods as the model size increases. The results are averaged across line-level and API-level tasks for both Python and Java languages. As model size increases, GraphCoder consistently yields the best performance among the four RAG methods, which indicates that its structural similarity-based retrieval successfully enables the model to better understand and generate code based on the retrieved code snippets with similar structures and patterns. As model size increases, the RAG methods and No RAG tend to perform better. However, on repository-level tasks, the performance of models without RAG does not follow the scaling law [16]; the performance does not scale as a power-law with model size due to the lack of intra-repository knowledge. Additionally, it can be observed that there is a phase transition from CodeGen2-3.7B to CodeGen2-7B; when transitioning from CodeGen2-3.7B to CodeGen2-7B, a substantial change occurs. In contrast, from CodeGen2-7B to CodeGen2-16B, the improvement in performance is limited. Therefore, considering cost-effective balance, an LLM of size 7B may be a good choice for implementing RAG-based completion methods.

Table 2: Experimental results on the code completion effectiveness. The values presented are formatted as percentages (%). GPT3.5 refers to GPT3.5-Turbo-Instruct. The results of RepoCoder are obtained after three iterations.

			GPT3.5				StarCoder-15B				CodeGen2-16B			
			Code Match		Identifier Match		Code Match		Identifier Match		Code Match		Identifier Match	
			EM	ES	EM	F1	EM	ES	EM	F1	EM	ES	EM	F1
Line-level	Python	No RAG	33.10	60.28	40.35	56.15	19.75	42.46	21.10	29.48	33.75	60.71	40.90	56.32
		Vanilla RAG	37.90	58.47	44.40	54.30	32.80	53.28	36.75	42.27	39.00	61.15	45.90	56.58
		Shifted RAG	45.65	68.49	51.70	63.62	18.70	36.34	22.35	25.55	35.65	55.54	41.00	50.80
		RepoCoder	38.20	59.71	44.75	54.59	34.30	53.22	**38.10**	43.14	41.10	63.05	48.25	58.44
		GraphCoder	**46.60**	**69.42**	**53.80**	**65.55**	**34.50**	**54.11**	**38.10**	**43.24**	**46.65**	**69.29**	**53.55**	**65.25**
	Java	No RAG	43.20	76.01	51.60	68.31	23.75	46.07	34.80	37.39	32.80	69.09	43.90	59.56
		Vanilla RAG	48.05	78.04	56.20	70.89	28.10	49.85	32.35	35.02	36.15	69.67	46.95	59.37
		Shifted RAG	48.15	78.09	55.80	70.74	25.15	52.25	30.10	35.33	36.15	70.16	46.80	59.81
		RepoCoder	48.30	78.16	56.50	70.76	30.15	51.73	34.35	36.80	37.70	70.86	48.80	60.72
		GraphCoder	**50.60**	**78.94**	**58.70**	**72.00**	**30.83**	**54.89**	**35.81**	**39.14**	**40.30**	**72.05**	**50.45**	**61.95**
API-level	Python	No RAG	27.75	56.55	30.90	54.33	15.05	39.49	15.35	26.16	27.70	56.61	30.35	53.43
		Vanilla RAG	37.50	57.98	40.05	56.77	35.90	54.08	37.25	51.15	35.80	58.66	38.60	56.82
		Shifted RAG	41.65	65.06	44.25	63.58	17.60	34.88	18.00	24.61	34.80	55.32	37.35	52.82
		RepoCoder	39.40	59.28	42.10	57.88	36.70	58.30	40.15	55.74	41.00	63.07	44.00	61.45
		GraphCoder	**45.25**	**66.81**	**48.80**	**65.70**	**38.90**	**60.37**	**41.59**	**56.15**	**48.75**	**69.97**	**51.75**	**69.03**
	Java	No RAG	37.95	71.91	40.70	63.79	23.95	52.43	24.50	37.01	24.25	60.88	26.45	47.80
		Vanilla RAG	54.10	79.27	57.00	73.59	46.50	63.03	46.70	52.09	52.35	75.89	54.10	70.32
		Shifted RAG	58.80	81.45	61.25	76.16	49.20	67.22	49.40	55.64	54.65	77.87	56.50	73.58
		RepoCoder	56.05	79.80	58.55	74.27	48.45	64.40	48.70	53.76	57.20	78.82	59.05	74.06
		GraphCoder	**61.57**	**82.66**	**63.72**	**77.68**	**54.90**	**69.85**	**55.00**	**59.83**	**60.15**	**80.53**	**61.55**	**76.34**

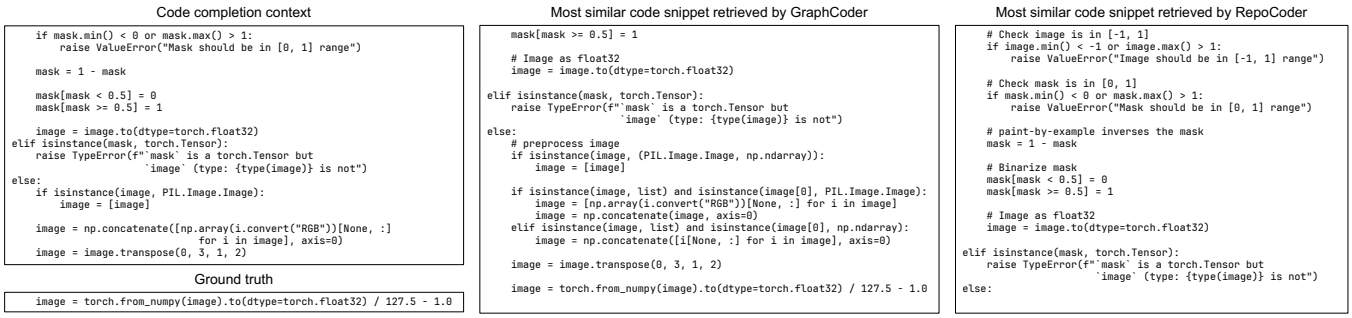

Figure 5: A qualitative example demonstrating the effectiveness of GraphCoder.

6.2.2 Generalizability across repositories of various duplication ratios. Fig. 7 shows the correlation between the repository's duplication ratio and the performance of RAG methods. Intuitively, it is easier to retrieve useful code snippets when a repository has a large amount of duplication. The code duplication ratio [44] measures duplicated code lines in a repository, while the identifier duplication ratio measures lines with repeated identifiers relative to the total code lines. The results of Fig. 7 are based on GPT-3.5-Turbo-Instruct. As shown in Fig. 7, GraphCoder outperforms other RAG methods in 15 out of 20 repositories, indicating its superior performance. The trend depicted by the curve suggests that repositories with lower duplication levels benefit more from GraphCoder. Specifically, when the code duplication ratio is below 40%, GraphCoder surpasses the best result of baseline methods by 9.13%, and by 5.25% when the ratio is above 40%. These results highlight GraphCoder's ability to effectively retrieve relevant code snippets compared to other baseline RAG methods, particularly in more challenging tasks with less superficial code duplication.

Answer to RQ2: *As the model size increases, GraphCoder consistently outperforms other RAG methods and demonstrates greater effectiveness in repositories with lower repetition.*

6.3 RQ3: Ablation Study

In this subsection, we systematically evaluate the effects of the key components in GraphCoder. For simplification, all the experiments are conducted based on GPT3.5-Turbo-Instruct.

6.3.1 Ablation study of components in CCG. The three components in CCG are the control flow graph (CFG), the data dependence graph (DDG), and the control dependence graph (CDG). To study the impact of each component, we separately remove each of them and then evaluate their performance.

As shown in Table 3, the importance of the components follows a clear hierarchy: CFG is most critical, followed by CDG, and then DDG. The removal of CFG results in a significant decline in the performance, with average relative reductions of 19.82% in code match

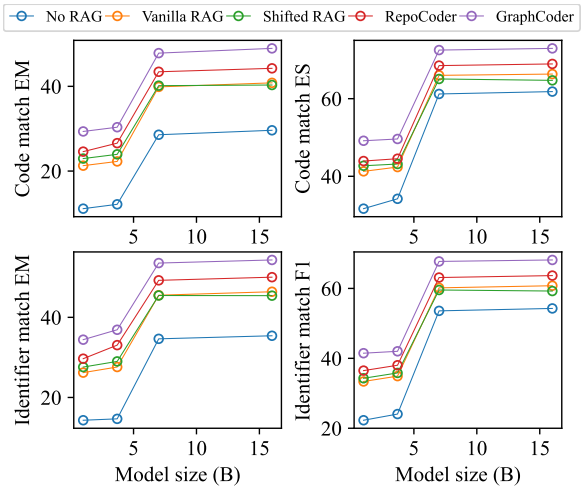

Figure 6: Performance of RAG and non-RAG methods across different base model sizes (CodeGen2 1B, 3.7B, 7B, and 16B).

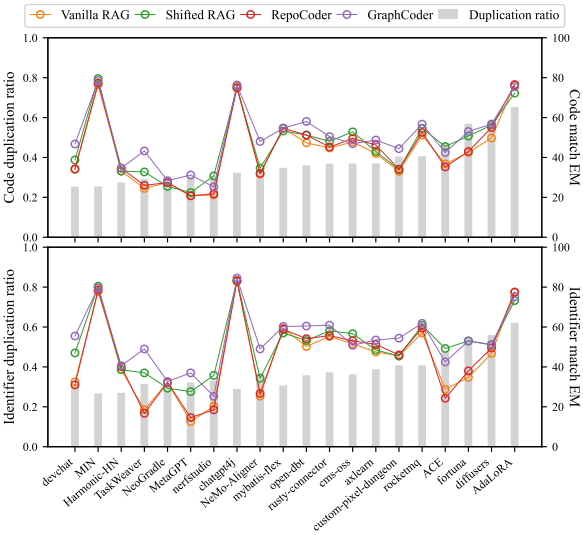

Figure 7: Correlation between the repository's duplication ratio and the performance of RAG methods.

EM and 15.64% in identifier match EM. This underscores the fundamental role of CFG, which is the basis for CCG construction that integrates CDG and DDG nodes through traversal. Without CFG, the CCG slice degenerates to a 1-hop DDG and CDG local subgraph. The most significant drop occurs on the API-level Java tasks. Compared to Python, the more verbose syntax of Java requires a greater need for long-distance relevant information to understand the context of code completion. The impact of removing CDG and DDG is nearly equivalent, with a negligible difference in performance degradation, suggesting that while both are important, predicates in CDG may play a slightly more critical role in determining the semantics of their corresponding statements.

Table 3: Ablation study of components in CCG.

			Code Match		Identifier Match	
			EM	ES	EM	F1
Line level	Python	**GraphCoder**	**46.60**	**69.42**	**53.80**	**65.55**
		- CFG	39.15	62.93	47.10	53.93
		- DDG	42.05	64.96	49.85	56.21
		- CDG	41.70	64.89	49.50	56.16
	Java	**GraphCoder**	**50.60**	**78.94**	**58.70**	**72.00**
		- CFG	45.07	77.10	54.18	69.67
		- DDG	47.62	77.66	56.20	70.27
		- CDG	47.56	77.62	56.09	70.26
API level	Python	**GraphCoder**	**45.25**	**66.81**	**48.80**	**65.70**
		- CFG	35.90	61.01	42.40	51.65
		- DDG	39.80	63.51	46.40	54.66
		- CDG	39.60	63.11	46.20	54.27
	Java	**GraphCoder**	**61.57**	**82.66**	**63.72**	**77.68**
		- CFG	42.06	73.90	45.06	66.19
		- DDG	56.62	79.72	58.77	75.77
		- CDG	56.36	79.61	58.62	75.69

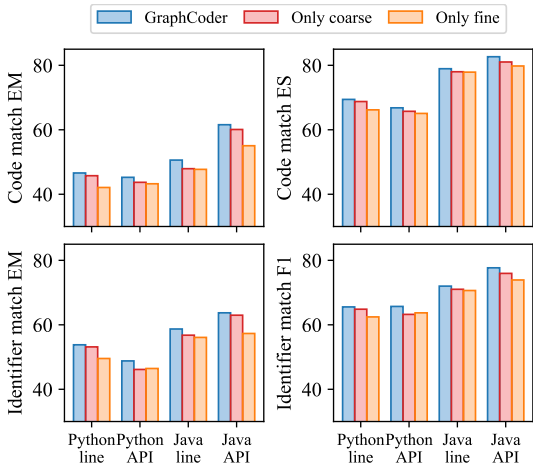

Figure 8: Ablation study for coarse-to-fine steps.

6.3.2 Ablation study for coarse-to-fine steps. Fig. 8 shows the ablation study results of the coarse-to-fine steps in GraphCoder. It evaluates their impact on GraphCoder by comparing the performance of using only the coarse-grained retrieval and only the fine-grained retrieval separately.

As seen from Fig. 8, GraphCoder consistently exhibits better performance than its variants that employ either only coarse-grained or only fine-grained retrieval steps. This observation confirms the benefits of integrating both coarse-grained and fine-grained retrieval steps within GraphCoder. In particular, the coarse-grained retrieval step plays a more significant role than the fine-grained step. When comparing the GraphCoder only coarse variant to the only fine-grained variant, there are notable improvements in the code match EM, ES, and identifier match EM, F1 scores of 2.34, 1.15, 2.41, and 1.08, respectively. A particularly significant drop in performance is observed for the only fine-grained variant on the API-level Java task. This is because, compared with the coarse-grained retrieval, the fine-grained step focuses more on localized context, while Java completion task typically relies more on long-distance relevant context This observation is also consistent with the conclusion drawn from Table 3.

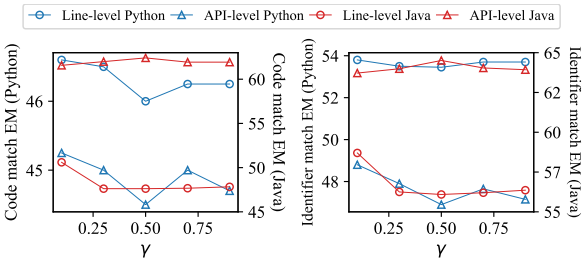

Figure 9: Hyper-parameter sensitivity.

6.3.3 Hyper-parameter sensitivity. To conduct a more granular analysis of the impact of fine-grained step on the performance of GraphCoder, we demonstrate its performance as the hyperparameter γ varies. γ is the dependence distance shrink factor in the fine-grained step. A lower γ places more emphasis on the local structure of the completion target. From Fig. 9, we can observe that the performance of GraphCoder is robust to the hyper-parameter γ. Specifically, the variation of code match EM when γ changes from 0.1 to 0.9 is 0.60, 0.75, 2.98, and 0.84 on line-level, API-level Python, and line-level, API-level Java tasks, respectively. Generally, the optimal γ that yields the best performance depends on the intrinsic feature of tasks. For example, on API-level Python tasks, the best γ is 0.1, while that value on the API-level Java task is 0.5. Although randomly selecting a γ may lead to sub-optimal performance, GraphCoder is still likely superior to other baseline methods. By comparing Fig. 9 with Table 2, GraphCoder's worst performance when varying γ remains better than baseline methods.

Answer to RQ3: *The performance of GraphCoder degrades after the removal of its CCG components or the coarse-to-fine steps. Among its various components, the CFG and the coarse-grained retrieval step are the most critical for its effectiveness.*

6.4 RQ4: Cost

In this subsection, we compare the resource cost of GraphCoder with baseline RAG methods from three aspects: (1) time efficiency of retrieval; (2) database storage; and (3) number of tokens utilized.

6.4.1 Time efficiency of retrieval. To investigate the retrieval efficiency of GraphCoder and sliding window-based methods, we compare their end-to-end retrieval running time on an average of 8000 completion tasks in Table 4. Specifically, the running time comprises the time needed for converting code sequences into bag-of-words embedding (via a local tokenizer) and searching for the top-k code snippets.

Compared to sequence-based methods (Vanilla RAG, Shifted RAG, and RepoCoder), GraphCoder is more time-efficient. This is because the database of GraphCoder is statement-level, whereas that of sliding window-based methods is line-level. The statement-level database significantly reduces the number of entries by not storing blank lines or comments and by consolidating multi-line statements into single entries. This reduction in entries decreases the number of calculations required. For RepoCoder, it is more time-consuming since it requires three iterations, each of which includes a sliding window-based search. Additionally, since the fine-grained step is used for re-ranking, GraphCoder only needs to calculate the similarity between the query CCG and a small subset

Table 4: Details of time efficiency and database storage. The results of RepoCoder are obtained after three iterations.

	Retrieval time (sec)	Database storage #Entries	Size (MB)	Prop.
Vanilla RAG	4.7290	115109	159.4	9.7
Shifted RAG	4.4894	115109	159.4	9.7
RepoCoder	14.0168	115109	159.4	9.7
GraphCoder	1.0753	23560	102.4	6.7
- Coarse	1.0424	-	-	-
- Fine	0.0329	-	-	-

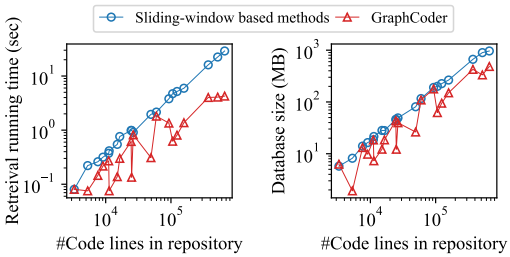

Figure 10: Running time and database size of sliding window based retrieval methods and GraphCoder.

Table 5: Number of input (#In) and output (#Out) tokens used. The results of RepoCoder are obtained after three iterations.

	GPT3.5		StarCoder-15B		CodeGen-16B	
	#In	#Out	#In	#Out	#In	#Out
No RAG	758.18	93.87	803.57	55.87	701.12	70.99
Vanilla RAG	2557.12	67.01	3882.12	77.72	2239.95	64.64
Shifted RAG	2990.92	87.99	3716.51	58.94	2311.48	63.71
RepoCoder	7772.52	207.93	11500.86	234.06	6711.87	195.25
GraphCoder	2666.13	88.14	3715.51	77.04	2201.08	71.29

of entries in the database. This significantly reduces the running time for the fine-grained step. As shown in Table 4, the fine-grained step accounts for only 3.06% of the total retrieval time. To further examine the relationship between repository size and retrieval time, we present the running time for retrieval against the number of code lines in the repository in Fig. 10. It can be observed from Fig. 10 that the retrieval time of GraphCoder increases more slowly compared to sliding window-based methods.

6.4.2 Database storage. Table 4 demonstrates the database size of various RAG methods. Since the Vanilla RAG, Shifted RAG, and RepoCoder are all sliding window-based methods with the same window size and stride, their database are of the same size. As seen in Table 4, GraphCoder reduces the number of entries by 79.5% compared to sliding window-based methods by using a statement-level database instead of a line-level one. Therefore, despite the need to store graph structures, GraphCoder's database is more space-saving. Additionally, Fig. 10 exhibits the relationship between the number of code lines in the repository and the database size. For sliding window-based methods, the database size increases linearly with the number of lines, whereas GraphCoder's database size remains smaller than sliding window-based ones.

6.4.3 Number of tokens. To study the consumption for generation, we compare the number of tokens utilized of GraphCoder with other methods in Table 5. It can be observed that, on average, the computational overhead of GraphCoder, in terms of input/output tokens, is lower than that of the other three retrieval-augmented

methods: Vanilla RAG, Shifted RAG, and RepoCoder. However, the difference in the number of tokens is not significant, as we employ the same prompt template and organization method for retrieved results in the prompt.

Answer to RQ4: *Compared to sliding window-based RAG methods, GraphCoder does not consume more tokens but is more efficient in terms of time and space by virtue of its statement-level database instead of a line-level one.*

7 Threats to Validity

Internal validity. The internal threats to validity lie in the implementation of baseline methods and the selection of code LLMs used in experiments. For the implementation of baseline methods, we directly utilize source code from GitHub provided by RepoCoder [44] and configure it according to their paper to ensure a fair comparison. As for other baseline methods (Vanilla RAG, Shifted RAG), we implement them by ourselves as they have no publicly available implementations. Considering that Vanilla RAG and Shifted RAG are also sliding window methods, similar to RepoCoder, we adopt RepoCoder's implementation to mitigate internal threats brought by potential bugs. For the selection of code LLMs, we are keen to use more recent newly released code LLMs, such as CodeLlama [33] and DeepSeek-Coder [11], to verify the effectiveness of GraphCoder. Regrettably, these recent models pose a data leakage risk to RepoEval [44] and even our newly constructed RepoEval-Updated. Therefore, we meticulously select six suitable code LLMs without data leakage risk to validate GraphCoder's effectiveness, including OpenAI's GPT-3.5, StarCoder 15B, CodeGen2 1B, 3.7B, 7B and 16B, for the fairness of experiments.

External validity. The threat to external validity mainly lies in the generalizability of our method, including its ability to be applied to different programming languages and diverse repositories. Our evaluations focus on open-source repositories in two mainstream programming languages: Python and Java, so our results are limited in this scope. For generalizability to other programming languages, GraphCoder can be migrated with minimal effort by first constructing the code context graph based on the abstract syntax tree produced by tree-sitter, and then directly using GraphCoder to achieve RAG-based code completion. For the generalization to diverse repositories, we try our best to cover a wide range of repositories of different sizes. However, there may exist potential threats to GraphCoder when the downstream evaluation repositories contain relatively low code duplication. This is primarily because low duplication will significantly reduce the recall rate during the retrieval phase of GraphCoder. To clearly delineate the performance boundaries, we offer a more detailed analysis on the impact of code duplication for GraphCoder's efficacy in our experiments to demonstrate its impact.

8 Conclusion

In this paper, we propose GraphCoder, a graph-based code completion framework for repository-level tasks. GraphCoder uses a code context graph (CCG) to capture the completion target's relevant context. The CCG is a statement-level multi-graph with control flow and data and control dependence edges. The retrieval is done through coarse-to-fine steps, involving filtering candidate code snippets and re-ranking them using a decay-with-distance structural similarity measure. After that, GraphCoder employs pre-trained language models to generate the next lines based on the retrieved snippets. To comprehensively evaluate the performance of GraphCoder, we conduct experiments using 5 LLMs and 8000 code completion tasks sourced from 20 repositories. Experimental results demonstrate GraphCoder's effectiveness, significantly improving the accuracy of code completion via more exactly matched code for generation with less retrieval time and space overhead.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant 62192731, 62192730, and 61690200.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[4] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[5] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. In *Advances in Neural Information Processing Systems 36*.

[6] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007* (2022).

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[8] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[9] Robert Gold. 2010. Control flow graphs and code coverage. *International Journal of Applied Mathematics and Computer Science* 20, 4 (2010), 739–749.

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[11] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE]

[12] Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu. 2024. FT2Ra: A Fine-Tuning-Inspired Approach to Retrieval-Augmented Code Completion. *arXiv preprint arXiv:2404.01554* (2024).

[13] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. 1993. Efficient construction of program dependence graphs. *ACM SIGSOFT Software Engineering Notes* 18, 3 (1993), 160–170.

[14] Huahai He and Ambuj K Singh. 2006. Closure-tree: An index structure for graph queries. In *International Conference on Data Engineering*. IEEE, 38–38.

[15] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[16] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[17] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2019. Generalization through memorization: Nearest neighbor language

models. *arXiv preprint arXiv:1911.00172* (2019).
[18] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
[19] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
[20] Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li. 2023. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *arXiv preprint arXiv:2312.05772* (2023).
[21] Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. *arXiv preprint arXiv:2306.03091* (2023).
[22] Ting Long, Yutong Xie, Xianyu Chen, Weinan Zhang, Qinxiang Cao, and Yong Yu. 2022. Multi-View Graph Representation for Programming Language Processing: An Investigation into Algorithm Detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 5792–5799.
[23] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).
[24] 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)*. 6227–6240.
[25] IA Natour. 1988. On the control dependence in the program dependence graph. In *Proceedings of the ACM sixteenth annual conference on Computer science*. 510–519.
[26] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309* (2023).
[27] Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2719–2734.
[28] Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. 2024. RepoHyper: Better Context Retrieval Is All You Need for Repository-Level Code Completion. *arXiv preprint arXiv:2403.06095* (2024).
[29] Rishabh Ranjan, Siddharth Grover, Sourav Medya, Venkatesan Chakaravarthy, Yogish Sabharwal, and Sayan Ranu. 2022. Greed: A neural framework for learning graph distance functions. *Advances in Neural Information Processing Systems* 35 (2022), 22518–22530.
[30] Kaspar Riesen, Miquel Ferrer, and Horst Bunke. 2015. Approximate graph edit distance in quadratic time. *IEEE/ACM transactions on computational biology and bioinformatics* 17, 2 (2015), 483–494.
[31] Kaspar Riesen, Miquel Ferrer, Andreas Fischer, and Horst Bunke. 2015. Approximation of graph edit distance in quadratic time. In *Graph-Based Representations in Pattern Recognition: 10th IAPR-TC-15 International Workshop, GbRPR 2015, Beijing, China, May 13-15, 2015. Proceedings 10*. Springer, 3–12.
[32] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
[33] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[34] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*. PMLR, 31210–31227.
[35] Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. RepoFusion: Training Code Models to Understand Your Repository. *arXiv preprint arXiv:2306.10998* (2023).
[36] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.
[37] Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based Code Completion via Multi-Retrieval Augmented Generation. *arXiv preprint arXiv:2405.07530* (2024).
[38] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 421–433.
[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[40] Ori Yoran, Tomer Wolfson, Ori Ram, and Jonathan Berant. 2023. Making retrieval-augmented language models robust to irrelevant context. *arXiv preprint arXiv:2310.01558* (2023).
[41] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual Pre-training on Sketches for Library-oriented Code Generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, Luc De Raedt (Ed.). ijcai.org, 2369–2375.
[42] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 7443–7464. https://aclanthology.org/2023.acl-long.411
[43] Zhiping Zeng, Anthony KH Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. 2009. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment* 2, 1 (2009), 25–36.
[44] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023*. Association for Computational Linguistics, 2471–2484.
[45] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. arXiv:2311.07989 [cs.CL]
[46] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. arXiv:2311.10372 [cs.SE]