GraphCoder: Enhancing Repository-Level Code Completion via Code Context Graph-based Retrieval and Language Model

Anonymous ACL submission

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 through *code context graph* (CCG) that consists of control-flow and data/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 show that: compared with the state-of-the-art method RepoCoder, GraphCoder improves the *exact match* metric by 5.93% on average.

1 Introduction

Code LLMs (large language models), such as Codex (Chen et al., 2021), StarCoder (Li et al., 2023a) and Code Llama (Roziere et al., 2023), have demonstrated impressive capability in general code completion tasks (Zheng et al., 2023; Zan et al., 2023; Zhang et al., 2023b). Some of them 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.

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

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 (Zan et al., 2022; Tang et al., 2023; Zhang et al., 2023a). 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 leading to degraded completion performance (Yoran et al., 2023; Shi et al., 2023).

To mitigate the knowledge-lack problem mentioned above, several methods have been proposed following the RAG pattern of *retrieval-augmented generation* (Parvez et al., 2021; Lu et al., 2022; Zhang et al., 2023a). 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 lexical 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 eight real-world repositories demonstrate the effectiveness of GraphCoder: GraphCoder more accurately locates code snippets with a higher completion target hit rate (+5.22% on average) and a higher exact match (+5.93% on average) for code completion compared to the state-of-the-art method RepoCoder.

To summarize, our main contributions are:

- A structured representation of source code CCG (code context graph) to capture relevant long-distance context for predicting the semantics of code completion target;
- 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;
- Extensive experiments[3] demonstrate that GraphCoder outperforms existing RAG frameworks with higher hit rate and exact match value.

2 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]The code and dataset are available at https://anonymous.4open.science/r/GraphCoder-627E.

2.1 Code Context Graph

A code context graph is the superimposition of three kinds of graphs about source 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 (Ferrante et al., 1987).

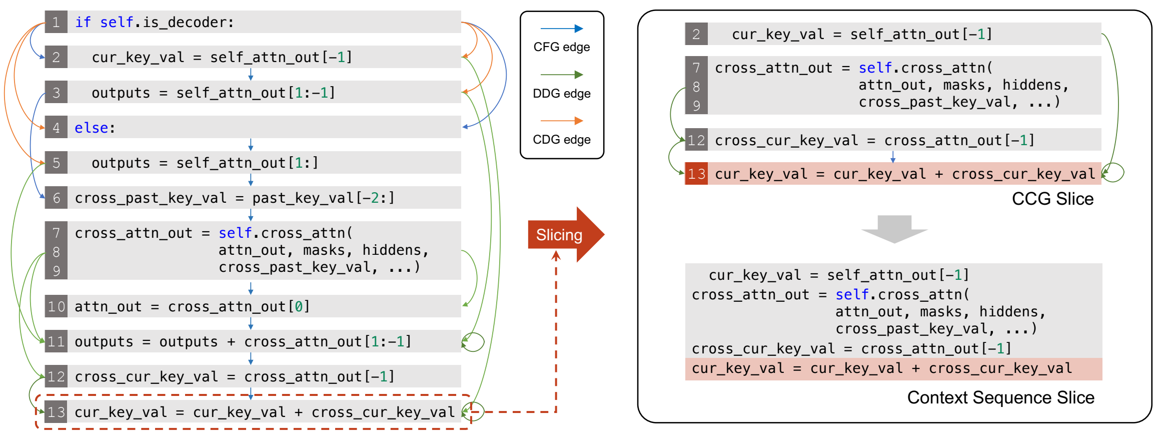
Definition 1 (Code Context Graph) *A code context graph $G = (X, E, T, \lambda)$ is a directed multi-graph, where*

- *$X = \{x_1, \cdots, x_n\}$ is the vertex set, each of which represents a code statement;*
- *$E = \{e_1, \cdots, e_m\}$ 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 (Allen, 1970; Gold, 2010; Long et al., 2022). 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.

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 (Ferrante et al., 1987; Natour, 1988; Cytron et al., 1991). Specifically, an edge exists between two statements if one directly affects whether the other will be executed, distinguishing it from the CFG.

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 (Ferrante et al., 1987; Harrold et al., 1993).



```
1  if self.is_decoder:
2    cur_key_val = self_attn_out[-1]
3    outputs = self_attn_out[1:-1]
4  else:
5    outputs = self_attn_out[1:]
6  cross_past_key_val = past_key_val[-2:]
7  cross_attn_out = self.cross_attn(
8                   attn_out, masks, hiddens,
9                   cross_past_key_val, ...)
10 attn_out = cross_attn_out[0]
11 outputs = outputs + cross_attn_out[1:-1]
12 cross_cur_key_val = cross_attn_out[-1]
13 cur_key_val = cur_key_val + cross_cur_key_val
```

```
2    cur_key_val = self_attn_out[-1]
7  cross_attn_out = self.cross_attn(
8                   attn_out, masks, hiddens,
9                   cross_past_key_val, ...)
12 cross_cur_key_val = cross_attn_out[-1]
13 cur_key_val = cur_key_val + cross_cur_key_val
```

```
  cur_key_val = self_attn_out[-1]
cross_attn_out = self.cross_attn(
                 attn_out, masks, hiddens,
                 cross_past_key_val, ...)
cross_cur_key_val = cross_attn_out[-1]
cur_key_val = cur_key_val + cross_cur_key_val
```

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

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

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/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 the 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 $X_{CF} \cup X_{CD} \cup X_{DD}$.

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/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/control dependence (lines 7-9). The context sequence slice consists of all statements in the CCG slice, ordered by line number.

3 GraphCoder

3.1 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 code repository, GraphCoder completes the given context through three steps: *database construction*, *code retrieval*, and *code generation*.

- In the *database construction* step (Section 3.2), 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 3.3), GraphCoder takes a code completion context as in-

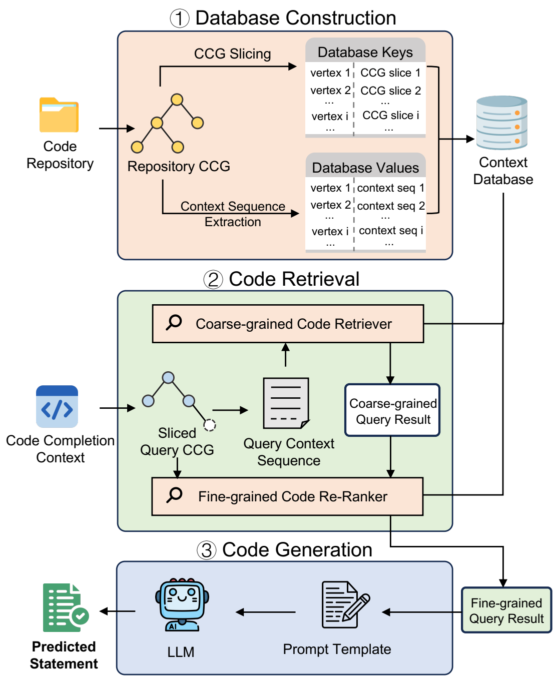


Figure 2: An illustration of GraphCoder framework.

put 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 3.4), 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.

3.2 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 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.

3.3 Code Retrieval

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

Query CCG Construction. GraphCoder initially extracts the sliced query CCG of the completion target. Specifically, GraphCoder 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 Code 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 measure applicable to code sequences, including sparse retriever BM25 (Robertson et al., 2009), Jaccard index (Jaccard, 1912) based on the bag-of-words model, as well as dense retrievers like similarity of embeddings from CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020).

Fine-Grained Code 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 (Zeng et al., 2009; Ranjan et al., 2022). 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 (Zeng et al., 2009; He and Singh, 2006), we calculate it by extending the quadratic-time greedy assignment (GA) algorithm (Riesen et al., 2015a,b) with a decay-with-distance factor. Specifically, we first obtain an alignment $\mathcal{A}$ between the vertices in $G^l_h(\hat{y})$ and $G^l_h(x)$ by the GA algorithm (Riesen et al., 2015a). The aligned vertex pairs in $\mathcal{A}$ reflects the vertex substitution relationship between $X^l_h(\hat{y})$ and $X^l_h(x)$. For a vertex v in $G^l_h(\hat{y})$, we denote the $\mathcal{A}(v)$ as its aligned vertex in $G^l_h(x)$. Let $X_\mathcal{A}$ be $\{v \mid v \in X^l_h(\hat{y}), (v,u) \in \mathcal{A}\}$, $E_\mathcal{A}$ be $\{e \mid e = (v,t,u) \in G^l_h(\tilde{y}), (\mathcal{A}(v), t', \mathcal{A}(u)) \in G^l_h(x)\}$, 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 Algorithm 2.

Algorithm 2: Decay-with-distance SED

```
   Input  : Graphs $G^l_h(\hat{y})$ and $G^l_h(x)$ as well
            as a decay-with-distance factor $\gamma$.
   Output : Decay-with-distance SED between
            $G^l_h(\hat{y})$ and $G^l_h(x)$.
 1 $SED \leftarrow 0$;
 2 for $v \in X_\mathcal{A}$ do
 3     $SED \leftarrow SED + \gamma^{h(v,\tilde{y})} c(v, \mathcal{A}(v))$;
 4 end
 5 for $v \in X^l_h(\hat{y}) \setminus X_\mathcal{A}$ do
 6     $SED \leftarrow SED + \gamma^{h(v,\tilde{y})} c(v)$;
 7 end
 8 for $e = (v,t,u) \in E_\mathcal{A}$ do
 9     $SED \leftarrow SED + \gamma^{h(v,\tilde{y})} c(e, \mathcal{A}(e))$;
10 end
11 for $e = (v,t,u) \in E^l_h(\hat{y}) \setminus E_\mathcal{A}$ do
12     $SED \leftarrow SED + \gamma^{h(v,\tilde{y})} c(e)$;
13 end
14 return $SED$;
```

3.4 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 (Zhang et al., 2023a) of retrieval-augmented prompt formatting (Appendix C), we arrange the retrieval code snippets in ascending similarity order, each of which is accompanied with its original path file; then these arranged code snippets are concatenated by the code completion context X as the final prompt of the LLM.

4 Experimental Setup

4.1 Dataset: RepoEval-Updated

The dataset RepoEval-Updated is used in our experiments for repository-level code completion evaluation. In particular, RepoEval-Updated is derived from another dataset RepoEval (Zhang et al., 2023a), 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 removing those repositories created before 2022-03-31 and adding more recent repositories created after 2023-01-01, in order to avoid data leakage for most existing code LLMs whose training data is released before 2023. Those newly-added repositories are selected following the same criteria as RepoEval; the details are shown in Table 4 in Appendix B.

RepoEval-Updated includes two kinds of completion tasks, namely *API-level* and *line-level* tasks; each of them consists of 1600 test cases. A line-level task is generated by randomly removing a code line from repository and encapsulating its forward code snippet as a completion task. 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.

4.2 Evaluation Metrics

Metrics for the retrieval. Following the established practice for RAG (Gao et al., 2023), we employ *hit@k* to assess the retrieval performance. In addition, we also employ two rank-related metrics: *Mean Average Precision* (MAP) (Cormack and Lynam, 2006; Hirsch and Hofer, 2023) and *Area Under the Curve* (AUC) (Zuva and Zuva, 2012).

Metrics for the completion. Following previous studies (Lu et al., 2022; Liu et al., 2023), we evaluate the completion performance using two metrics: *Exact Match* (EM) and *Edit Similarity* (ES).

4.3 Methods for Comparison

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 completion 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 (Lu et al., 2022).

RepoCoder (Zhang et al., 2023a). 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.

4.4 Implementation Details

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 (Lu et al., 2022; Zhang et al., 2023a), a model which transforms code snippets into sets of tokens and calculates similarity using the Jaccard Index (Jaccard, 1912).
Code Generation. To avoid data leakage, we exclude in our consideration those LLMs without a 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 (Li et al., 2023a), and CodeGen2 models (1B, 7B, and 16B) (Nijkamp et al., 2023).

The detailed hyper-parameter settings related to the code retrieval and code generation are described in the Appendix C.

5 Experimental Results

5.1 Retrieval Effectiveness

Overall Performance. Table 1 shows the retrieval results across different retrieval frameworks. GraphCoder framework with the coarse-to-fine retrieval process outperforms other baselines in the majority of cases 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. By comparing Shifted RAG with Vanilla RAG, we can conclude that there is a position gap between the most similar code snippets and the intended completion target. RepoCoder shows the highest hit@1 except for GraphCoder. However, the retrieval performance of RepoCoder depends largely on the generation capabilities of LLMs. The RepoCoder results in Table 1 is the average results of the five LLMs used. Based on the generation result of CodeGen2-1B, RepoCoder achieves a hit@1 of 18.93%, while StarCoder results in a higher value of 25.06%.
Ablation Study. To further understand how the coarse-grained and the fine-grained retrieval contribute to GraphCoder performance, we conduct an ablation experiment. The results are shown in Table 1, where GraphCoder-C and GraphCoder-F represents the GraphCoder variants with only the coarse-grained and the fine-grained retrieval, respectively. As seen from Table 1, the coarse-grained retrieval plays a significant role in GraphCoder. Moreover, adding the fine-grained retrieval process proves beneficial after the coarse-grained process filters out several candidate snippets. However, when relying solely on fine-grained similarity, GraphCoder-F yields a lower hit rate.
Hyper-Parameter Sensitivity. In Fig. 3, we demonstrate the sensitivity of GraphCoder to its

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

	hit@1	hit@5	MAP	AUC
API-level				
Vanilla RAG	6.81	14.88	10.18	10.04
Shifted RAG	12.19	18.75	15.29	13.34
RepoCoder	21.48	34.50	26.88	26.72
GraphCoder-C	26.63	32.56	29.19	26.14
GraphCoder-F	25.44	30.06	27.44	26.05
GraphCoder	**29.00**	**34.75**	**31.41**	**26.79**
Line-level				
Vanilla RAG	9.44	18.38	13.34	13.21
Shifted RAG	15.95	23.63	19.37	17.47
RepoCoder	24.53	32.25	28.24	**25.92**
GraphCoder-C	26.81	31.98	29.23	24.12
GraphCoder-F	17.88	21.06	19.33	17.69
GraphCoder	**27.44**	**32.56**	**29.77**	25.10

Table 1: Experimental results on retrieval effectiveness, which are formatted as percentages (%). GraphCoder-C and GraphCoder-F are variants of GraphCoder, where GraphCoder-C includes only coarse-grained retrieval, and GraphCoder-F includes only fine-grained retrieval.

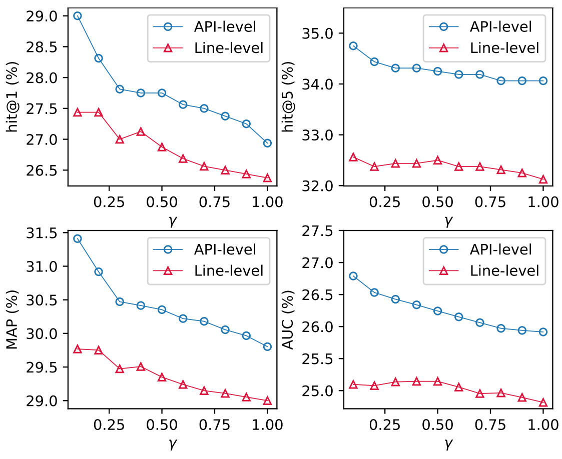


Figure 3: GraphCoder retrieval effectiveness with the variation of hyper-parameter γ, where γ is the decay-with-distance factor in the fine-grained re-ranking step.

	GPT3.5		StarCoder15B		CodeGen2-1B		CodeGen2-7B		CodeGen2-16B	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
API-level										
No RAG	24.62	53.21	11.81	35.77	20.75	49.93	24.00	52.97	24.50	53.23
Vanilla RAG	31.31	51.98	33.50	51.89	23.69	50.06	26.62	52.25	26.62	52.37
Shifted RAG	36.69	60.98	10.25	27.29	22.69	47.28	24.81	48.26	24.25	47.07
RepoCoder	33.12	53.48	34.38	57.09	29.38	54.64	30.88	55.86	31.25	56.87
GraphCoder-C	40.62	62.23	34.75	58.42	33.94	58.96	37.19	62.14	38.56	62.79
GraphCoder-F	38.50	61.08	33.94	58.12	33.12	59.04	37.06	62.12	37.44	62.60
GraphCoder	**40.94**	**62.74**	**35.56**	**58.60**	**35.44**	**60.45**	**38.69**	**63.79**	**40.12**	**64.68**
Line-level										
No RAG	30.13	56.39	15.12	37.93	24.62	51.92	31.25	57.50	31.69	57.45
Vanilla RAG	33.31	52.90	33.00	52.91	31.56	54.51	36.06	58.47	36.06	57.48
Shifted RAG	43.63	65.32	14.62	30.88	29.56	51.45	33.12	52.83	31.75	50.44
RepoCoder	34.19	54.19	34.25	52.62	33.62	56.51	38.50	60.15	38.19	59.37
GraphCoder-C	43.63	65.44	**34.81**	53.45	38.94	62.65	43.75	66.57	44.69	66.76
GraphCoder-F	39.12	62.32	29.63	51.26	33.94	59.04	39.19	63.32	40.06	64.15
GraphCoder	**44.37**	**65.63**	34.12	**53.52**	**39.25**	**62.72**	**43.78**	**66.84**	**44.81**	**67.06**

Table 2: Experimental results on the code completion effectiveness. The values presented are formatted as percentages (%). GPT3.5 refers to GPT3.5-Turbo-Instruct. GraphCoder-C and GraphCoder-F are variants, with GraphCoder-C including only coarse-grained retrieval, and GraphCoder-F including only fine-grained retrieval.

hyperparameter γ, which is the dependence distance shrink factor in the fine-grained retrieval. A lower γ places more emphasis on the local structure of the completion target. From Fig. 3, we can conclude that the effectiveness of fine-grained re-ranking tends to increase as γ decreases.

5.2 Code Completion

Overall Performance. Table 2 shows the code completion results of the five methods. It can be observed that GraphCoder exhibits a significant improvement on both the API-level and line-level code completion tasks compared to No RAG completion and baseline RAG methods. The improvement is more significant on GPT3.5-Turbo-Instruct and CodenGen2 series models compared to StarCoder, since StarCoder often wrongly encapsulates the predicted next statement into the code comment, thus influencing the EM metric. Compared to the vanilla RAG, GraphCoder increases the EM values on API-level and line-level tasks by 9.80% and 7.27% 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 EC and ES. 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 the code LLMs.

Ablation Study. In Table 2, we also give the ablation study results of the two components in GraphCoder by evaluating the code completion performance separately with only the coarse-grained (GraphCoder-C) and fine-grained (GraphCoder-F) retrieval steps. Similar to the results in Table 1, the superior performance of GraphCoder mainly benefits from the coarse-grained retrieval step. While the fine-grained re-ranking process does contribute to a slight improvement in completion results, its significance becomes more evident when the context window is smaller. This is illustrated by the larger gap between GraphCoder-C and GraphCoder, particularly in the CodeGen2 models.

5.3 Efficiency

To investigate the retrieval efficiency of GraphCoder and sliding window-based methods, we compare their end-to-end retrieval running time per completion task in Table 3. Specifically, the end-to-end retrieval running time comprises the time

	API-level runtime (sec)	Line-level runtime (sec)
Sliding window-based RAG		
- Stride = 1	1.3138	1.2059
- Stride = 5	0.2623	0.2617
- Stride = 10	0.1344	0.1296
GraphCoder		
- Coarse	0.7822	0.8039
- Fine	0.0247	0.0235
- Overall	0.8069	0.8274

Table 3: The end-to-end retrieval time per completion task in seconds for GraphCoder and sliding window-based methods (i.e., Vanilla RAG, Shifted RAG, and RepoCoder) with varied strides.

needed for converting code sequences into bag-of-words embedding (via a local tokenizer) and searching for the top-k code snippets based on their corresponding similarity measure. Since Vanilla RAG, Shifted RAG, and RepoCoder are all sliding window-based methods with the same embedding and searching similarity measure, we present their average retrieval time in Table 3.

It can be observed that GraphCoder outperforms sliding window-based methods in terms of time efficiency when the stride is set to 1. As the sliding window's stride increases, the number of code snippets in the database decreases, leading to less time spent by sliding window-based methods for larger strides. Therefore, GraphCoder exhibits lower efficiency compared to sliding window-based methods with larger strides (5 and 10), mainly because of GraphCoder's larger statement-level database. Despite this, from a cost-effective perspective, GraphCoder remains an affordable option.

6 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 (Liao et al., 2023; Ding et al., 2022; Shrivastava et al., 2023a,b; Zhang et al., 2023a). Various training or fine-tuning based methods, including n-grams (Tu et al., 2014) and Transformers (Svyatkovskiy et al., 2020; Ding et al., 2022), have been proposed to integrate repository context into language models (LMs). However, challenges persist due to the dynamic nature of repository-level features driven by continuous project development. To address this limitation, retrieval-augmented LMs have been proposed (Tang et al., 2023; Khandelwal et al., 2019; Lu et al., 2022; Zhang et al., 2023a). Khandelwal et al. (Khandelwal et al., 2019) and Tang et al. (Tang et al., 2023) 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 (Lu et al., 2022; Zhang et al., 2023a; Liao et al., 2023) have adopted a pre-processing strategy, that retrieves relevant snippets and adds them into LLMs' prompt. However, existing works only consider the context as a code sequence format without considering structural dependencies among statements.

Retrieval-Augmented Generation for LLMs. Large language models (LLMs) have demonstrated impressive capabilities in understanding both natural language and code, such as the GPT (Brown et al., 2020; Achiam et al., 2023), the LLama (Touvron et al., 2023), and the GLM series (Zeng et al., 2022; Du et al., 2022). However, LLMs exhibit limited performance in handling domain-specific queries that are beyond the knowledge of its training data (Kandpal et al., 2023). To address this problem, one of the effective practices is retrieval-augmented generation (RAG), which is a framework first introduced by Lewis et al. (Lewis et al., 2020). To better locate the relevant information to the the target answer, several approaches have been proposed, such as introducing a hypothetical questions (Li et al., 2023b) and adopting language model to rewrite the query (Ma et al., 2023).

7 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/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. Experimental results demonstrate GraphCoder's effectiveness, significantly improving the completion target hit rate for retrieval and achieving higher exact match values in the code completion.

Limitations

While this paper boasts numerous merits, it also bears some limitations.

Limited Effectiveness for Repositories With Few Code Duplication. Although we have proved the effectiveness of GraphCoder based on extensive experiments, 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 Appendix A.

Limited Time Efficiency for Larger Repository. GraphCoder has demonstrated exceptional superiority in various repo-level code completion tasks compared to its baselines. Yet, as repository size increases, the graph-based traits in GraphCoder would result in comparatively lower retrieval efficiency, as detailed in Section 5.3. So, we plan to expand GraphCoder in the future, with the objective of enhancing its effectiveness without compromising on efficiency.

Limited Exploration for Various Languages Models. Indeed, we are keen to use more recent newly released code LLMs, such as CodeLlama (Roziere et al., 2023) and DeepSeek-Coder (Guo et al., 2024), to verify the effectiveness of GraphCoder. Regrettably, these recent models pose a data leakage risk to RepoEval (Zhang et al., 2023a) and even our newly constructed RepoEval-Updated. Therefore, we meticulously select five suitable code LLMs without data leakage risk to validate GraphCoder's effectiveness, including OpenAI's GPT-3.5, StarCoder 15B, CodeGen2 1B, 7B and 16B, for the fairness of experiments.

References

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*.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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*.

Gordon V Cormack and Thomas R Lynam. 2006. Statistical precision of information retrieval evaluation. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 533–540.

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):451–490.

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*.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 320–335.

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*, pages 1536–1547.

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):319–349.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

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

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*.

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.

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

Huahai He and Ambuj K Singh. 2006. Closure-tree: An index structure for graph queries. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 38–38. IEEE.

Thomas Hirsch and Birgit Hofer. 2023. The map metric in information retrieval fault localization. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1480–1491. IEEE.

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

Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. 2023. Large language models struggle to learn long-tail knowledge. In *International Conference on Machine Learning*, pages 15696–15707. PMLR.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2019. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172*.

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*, 33:9459–9474.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Xinze Li, Zhenghao Liu, Chenyan Xiong, Shi Yu, Yu Gu, Zhiyuan Liu, and Ge Yu. 2023b. Structure-aware language model pretraining improves dense retrieval on structured data. *arXiv preprint arXiv:2305.19912*.

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*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

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*, volume 36, pages 5792–5799.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240.

Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting for retrieval-augmented large language models. *arXiv preprint arXiv:2305.14283*.

IA Natour. 1988. On the control dependence in the program dependence graph. In *Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, pages 510–519.

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*.

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*, pages 2719–2734.

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*, 35:22518–22530.

Kaspar Riesen, Miquel Ferrer, and Horst Bunke. 2015a. Approximate graph edit distance in quadratic time. *IEEE/ACM transactions on computational biology and bioinformatics*, 17(2):483–494.

Kaspar Riesen, Miquel Ferrer, Andreas Fischer, and Horst Bunke. 2015b. 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*, pages 3–12. Springer.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

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*.

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*, pages 31210–31227. PMLR.

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

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023b. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443.

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)*, pages 421–433. IEEE.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280.

Ori Yoran, Tomer Wolfson, Ori Ram, and Jonathan Berant. 2023. Making retrieval-augmented language models robust to irrelevant context. *arXiv preprint arXiv:2310.01558*.

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*, pages 2369–2375. ijcai.org.

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)*, pages 7443–7464, Toronto, Canada. Association for Computational Linguistics.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*.

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):25–36.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023b. Unifying the perspectives of nlp and software engineering: A survey on language models for code.

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.

Keneilwe Zuva and Tranos Zuva. 2012. Evaluation of information retrieval systems. *International journal of computer science & information technology*, 4(3):35.

A Influence of Code Duplication on GraphCoder Effectiveness

To analyze the impact of code duplication in a repository on the performance of GraphCoder, we present in Fig. 4 the correlation between the repository's duplication ratio and the improvement in Exact Match (EM) achieved by GraphCoder when compared to No RAG. The repository's duplication ratio represents the proportion of duplicated code lines to the total number of code lines in a repository. The EM improvement value represents the difference between GraphCoder and in-file completion (No RAG) based on GPT-3.5-Turbo-Instruct.

The results in Fig. 4 indicate that as the duplication ratio increases, GraphCoder's effectiveness becomes more significant, particularly in line-level code completion tasks. In the diffusers repository which has the highest duplication ratio, GraphCoder exhibits the most significant EM improvement on both the API-level and line-level task. In

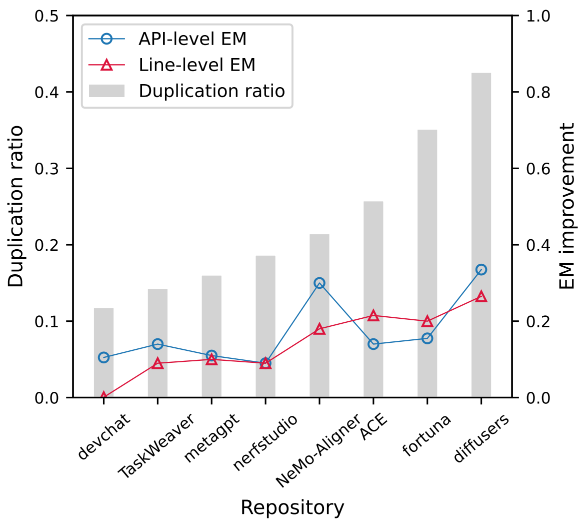


Figure 4: Correlation between the repository's duplication ratio and the improvement in EM achieved by GraphCoder when compared to No RAG.

the devchat repository with few code duplication, the effectiveness of GraphCoder is limited. However, this tendency is not consistently observed; for example, compared with fortuna, GraphCoder achieves higher EM on API-level task despite having a lower code duplication ratio in the NeMo-Aligner repository.

B Repositories in RepoEval-Updated

The dataset used in our experiments, RepoEval-Updated, includes eight real-world open-source GitHub repositories[5]. This dataset is derived from the RepoEval benchmark (Zhang et al., 2023a) by removing the repositories created before March 31, 2022 and add more recently repositories created after January 1, 2023 to ensure no overlap with the training data of advanced code LLMs released in 2023. The newly added repositories are selected based on the same criteria as RepoEval: open-source license, non-fork original repositories, over 100 stars, over 80% of files written in Python, and explicit unit tests. The details of the selected repositories are shown in Table 4.

To generate the line-level completion test tasks, we randomly select 200 code lines from each repository, adhering to criteria that the target completion lines are non-repetitive, not code comments, and contain at least 5 tokens (Zhang et al., 2023a). Recognizing a shortage of intra-repository APIs invocations in the devchat repository, we construct 100 test samples specifically for devchat. For the larger repository metagpt, we randomly select 300 test samples, while other repositories each provide 200 test samples, thus forming 1600 api-level completion test samples in total.

[5]https://github.com

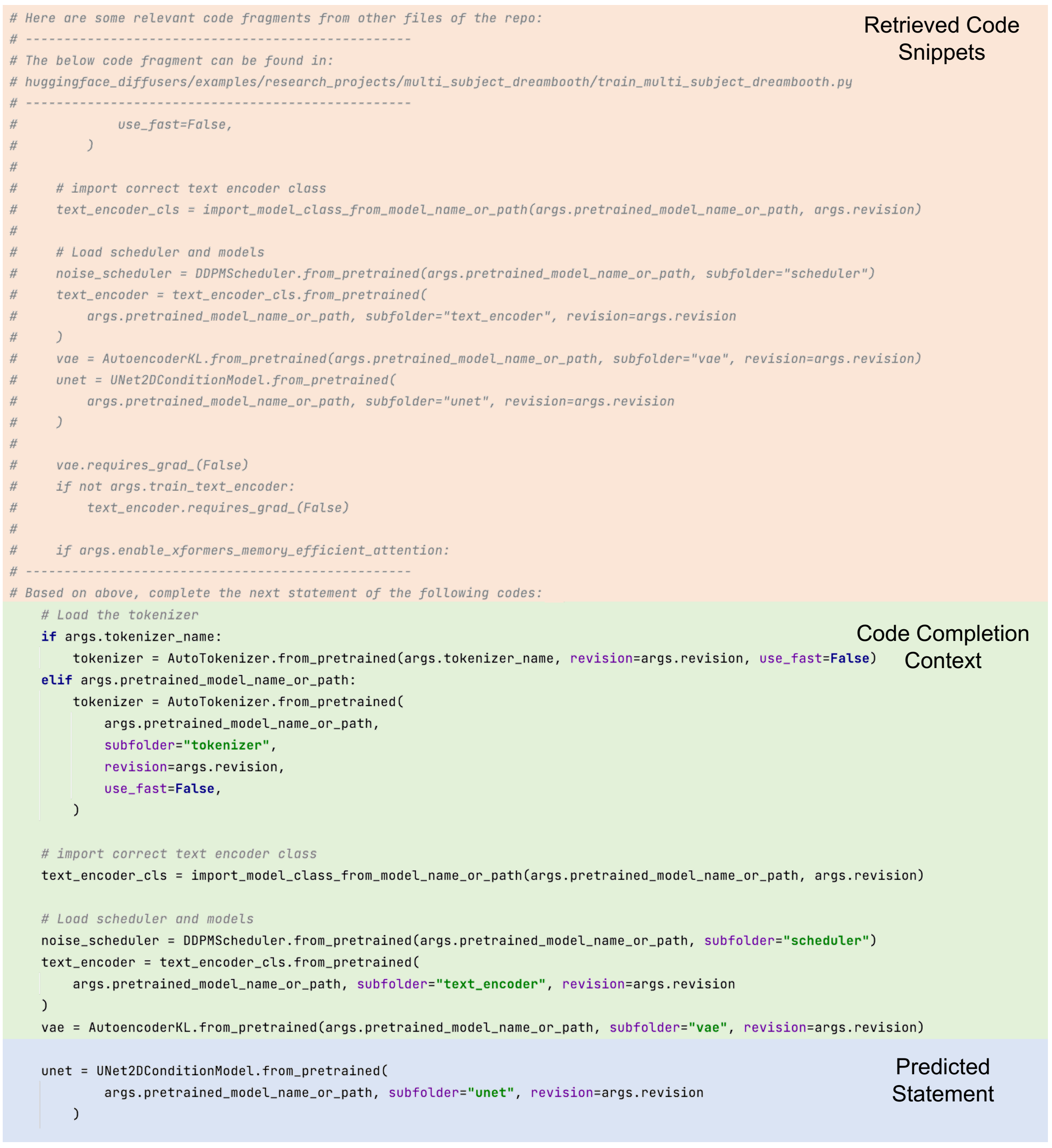
C Hyper-Parameter Settings

Following established practice in code completion (Zhang et al., 2023a), we fill the LLMs' context window by two parts: the retrieved code snippets, and the completion context. Each part occupies half of the context window. The maximum number of retrieved code snippets is 10. The maximum number of tokens in the generated completion is set to 100. For sliding window-based approaches (Vanilla RAG, Shifted RAG and RepoCoder), we fix the window size as 20 lines and a default sliding stride of 1. For GraphCoder, its graph 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.

An example of the prompt template employed in GraphCoder is shown in Fig. 5.

Repo name	GitHub Link	Created at	Stars	#Files	Size
devchat*	devchat-ai/devchat	2023-04-17	270	40	0.5MB
NeMo-Aligner*	NVIDIA/NeMo-Aligner	2023-09-01	119	54	1.6MB
fortuna	awslabs/fortuna	2022-11-17	826	168	1.9MB
TaskWeaver*	microsoft/TaskWeaver	2023-09-11	4035	113	3.0MB
diffusers	huggingface/diffusers	2022-05-30	20854	305	6.2MB
ACE	opendilab/ACE	2022-11-23	230	425	6.8MB
metagpt*	geekan/MetaGPT	2023-06-30	34140	374	17.9MB
nerfstudio	nerfstudio-project/nerfstudio	2022-05-31	7959	157	54.5MB

Table 4: Statistics of repositories in RepoEval-Updated. * corresponds to the newly added repositories to the original benchmark. All the newly added repositories are archived on 2024-01-05. #Files indicates the number of Python files in the repository. Statistics are accurate as of February 2024.



```
# Here are some relevant code fragments from other files of the repo:
# --------------------------------------------------
# The below code fragment can be found in:
# huggingface_diffusers/examples/research_projects/multi_subject_dreambooth/train_multi_subject_dreambooth.py
# --------------------------------------------------
#             use_fast=False,
#         )
#
#     # import correct text encoder class
#     text_encoder_cls = import_model_class_from_model_name_or_path(args.pretrained_model_name_or_path, args.revision)
#
#     # Load scheduler and models
#     noise_scheduler = DDPMScheduler.from_pretrained(args.pretrained_model_name_or_path, subfolder="scheduler")
#     text_encoder = text_encoder_cls.from_pretrained(
#         args.pretrained_model_name_or_path, subfolder="text_encoder", revision=args.revision
#     )
#     vae = AutoencoderKL.from_pretrained(args.pretrained_model_name_or_path, subfolder="vae", revision=args.revision)
#     unet = UNet2DConditionModel.from_pretrained(
#         args.pretrained_model_name_or_path, subfolder="unet", revision=args.revision
#     )
#
#     vae.requires_grad_(False)
#     if not args.train_text_encoder:
#         text_encoder.requires_grad_(False)
#
#     if args.enable_xformers_memory_efficient_attention:
# --------------------------------------------------
# Based on above, complete the next statement of the following codes:
    # Load the tokenizer
    if args.tokenizer_name:
        tokenizer = AutoTokenizer.from_pretrained(args.tokenizer_name, revision=args.revision, use_fast=False)
    elif args.pretrained_model_name_or_path:
        tokenizer = AutoTokenizer.from_pretrained(
            args.pretrained_model_name_or_path,
            subfolder="tokenizer",
            revision=args.revision,
            use_fast=False,
        )

    # import correct text encoder class
    text_encoder_cls = import_model_class_from_model_name_or_path(args.pretrained_model_name_or_path, args.revision)

    # Load scheduler and models
    noise_scheduler = DDPMScheduler.from_pretrained(args.pretrained_model_name_or_path, subfolder="scheduler")
    text_encoder = text_encoder_cls.from_pretrained(
        args.pretrained_model_name_or_path, subfolder="text_encoder", revision=args.revision
    )
    vae = AutoencoderKL.from_pretrained(args.pretrained_model_name_or_path, subfolder="vae", revision=args.revision)

    unet = UNet2DConditionModel.from_pretrained(
            args.pretrained_model_name_or_path, subfolder="unet", revision=args.revision
        )
```

Figure 5: An example showing the prompt format employed in GraphCoder.