GRACG: Graph Retrieval Augmented Code Generation

Konstantin Fedorov
AI Talent Hub
ITMO University
Saint Petersburg, Russia
k.fedorov@niuitmo.ru
0000-0001-9853-8543

Boris Zarubin
Central University
Moscow, Russia
0009-0008-7324-4176

Vladimir Ivanov
Research Center of the Artificial Intelligence Institute
Innopolis University
Innopolis, Russia
v.ivanov@innopolis.ru
0000-0003-3289-8188

***Abstract*—While file-level context is an effective basis for many modern code generation tools powered by Large Language Models (LLMs), it may be insufficient to fully capture the structural and semantic dependencies that extend across entire repositories. To address this issue, we propose a graph-based Retrieval-Augmented Code Generation (GRACG) framework that leverages repository-level context. Our approach models the repository as a heterogeneous graph of files, classes, and functions. A Graph Neural Network (GNN) is used to generate node embeddings that incorporate information from connected nodes. These embeddings are precomputed and serve as an efficient index, allowing us to retrieve context for user queries without re-running the GNN. Retrieved nodes are then used to construct prompts for LLMs to perform repository-aware function generation. We evaluate retrieval performance on a benchmark where the goal is to identify functions likely to be called based on a natural language description. For end-to-end evaluation, we use pass@k, which measures the percentage of tasks where at least one of the top-k generated solutions passes the associated test cases. Our results show that graph-based retrieval outperforms classical methods, highlighting it as a promising direction for future research. However, in terms of end-to-end code generation, we observe non-significant improvements in the metrics, even when oracle functions are used. Importantly, the modular nature of our framework also makes it suitable as a building block in multi-agent settings, where specialized components for retrieval, generation, and verification can collaborate around repository-level knowledge.**

***Index Terms*—code generation, graph neural networks, code retrieval, software repositories, large language models, machine learning for software engineering**

I. Introduction

The development and maintenance of large codebases are common challenges in software engineering. These challenges extend beyond code repetition to include issues such as maintaining consistency across modules, integrating updates across multiple files, and handling incomplete or poorly documented code. For instance, inconsistent naming conventions, lack of clear documentation, and the rapid evolution of software often lead to difficulties when developers try to understand or extend existing code. Recent advancements in Natural Language Processing (NLP), particularly through Large Language Models (LLMs), have had a significant impact on the software development process. Tools like GitHub Copilot [1] use LLMs to autocomplete code based on user comments and the context of a file, which can reduce the occurrence of repetitive patterns within individual files. However, these models are constrained to file-level context and do not consider the entirety of a codebase, making them less effective at addressing cross-file issues such as inconsistencies or unintentional duplication across different parts of a project.

Another potential approach is to feed the entire codebase to an LLM. However, this becomes impractical, as LLMs have limitations in their context size, typically constrained by token limits. When working with large codebases, such solution quickly exceeds these limits. Retrieval Augmented Generation (RAG) [2] can be applied to avoid the problem. RAG is a technique that combines generative models with retrieval systems, allowing for the generation of more accurate and contextually aware outputs by retrieving relevant external information. This allows for the incorporation of up-to-date, domain-specific knowledge at inference time, addressing the problem of outdated or incomplete knowledge within the model. By only retrieving the most relevant information, RAG reduces the need for large context windows, making it particularly useful for code-related tasks. In contrast to natural language, code in a repository has a strict structure and can be represented as a graph that contains functions, classes or files and relations representing function calls, inheritances and etc. These relations may help represent code block better by incorporating its neighbors meanings. For instance, it is hard to understand purpose of an orchestrator function, which only calls other functions, without a proper documentation. Thus, utilization of a semantic graph of a code repository might open the door to more meaningful and efficient retrieval of code snippets.

Beyond improving single-model code generation, such graph-enhanced retrieval frameworks can also serve as components for multi-agent systems. Recent studies show that retrieval-augmented generation can be decomposed into specialized agents that collaborate around filtering, retrieval, and generation [3]. In this sense, our approach provides reusable infrastructure that could be integrated into multi-agent pipelines for software engineering tasks.

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00060

II. Related Work

A. Graph Retrieval

Traditional LLM-based code generation systems like GitHub Copilot [1] often operate with only intra-file context, neglecting cross-file dependencies and repository structure. To overcome this, several repository-aware approaches have emerged. For example, RepoFusion [4] utilize repository structure using the best prompt template from several manually created templates, where the best is selected using a classification model or other algorithm. However, manual nature of templates may cause issues when encountering some unseen type of dependency.

Another notable system is RepoCoder [5] that addresses repository-level code completion with an iterative retrieval–generation loop, combining a similarity-based retriever (embedding search over repository snippets) with a pretrained code LLM. It retrieves relevant code snippets (lines, APIs, functions) to iteratively augment context—yet, it treats the repository as a flat collection of snippets and does not utilize repository structure explicitly.

RAMBO [6] likewise follows an iterative RAG approach for repository-level method body completion. It begins by generating a sketch of the target method, then uses AST parsing to extract repository elements such as classes, methods, and fields, along with their usages across the repository. These elements are iteratively retrieved and integrated into the generation process. While more structurally aware than RepoCoder, RAMBO still does not exploit call graphs or other relations between elements of repository, leaving opportunities for richer graph-based retrieval approaches.

Graph Neural Networks (GNNs) are a promising method, which can integrate the structural information of a graph into its embeddings. GNNs are particularly effective when combined with Large Language Models (LLMs), as they can enhance the embeddings by incorporating graph structures, providing richer context for natural language tasks. For instance, the paper [7] utilizes a Graph Attention Network to encode knowledge graph, with the nodes and edges of the subgraph being encoded by an LLM beforehand. Similarly, the paper [8] explores the use of GNNs for knowledge graph retrieval by classifying nodes as either relevant or not relevant to a given query. The main inspiration for this work was paper [9] in which the RepoHyper framework was proposed. This method extracts classes, modules, fields, and functions with an additional rich set of relations between them in the form of a directed graph. Then, the method retrieves context from the graph with the help of both classical approaches and graph neural network (GNN). Firstly, the approach finds relevant nodes by embedding similarity to a prompt that serve as starting points in a graph search. Secondly, it employs breadth-first search with some limit to a number of nodes and depth. Lastly, the method inserts a new node into the graph and tries to predict which existing nodes should be connected to it using GNN. The GNN is trained on a link prediction problem in an unsupervised manner. However, we can point following limitations: need to assign query to some node in the graph is not always feasible and GNN run for every query which may impose performance overhead.

B. End-to-end Benchmarks

Evaluating retrieval-augmented code generation systems requires benchmarks that reflect the complexities of using external code context and verifying output correctness. We categorize existing benchmarks into three types and discuss their limitations in the context of repository-level code generation. Benchmarks like HumanEval [10], MBPP [11], and APPS [12] assess synthesis at the function level. HumanEval and MBPP focus on writing Python functions from docstrings, while APPS includes a broader range of problem complexity. Compared to repo-level benchmarks, they allow less reasoning over existing codebases, making them less ideal for evaluating retrieval-augmented models. Repository level benchmarks evaluate a model's ability to use context from the entire repository and generate consistent code. RepoBench [13] and RepoEval introduced in [5] are two prominent examples. RepoBench focuses on next-line prediction across repo-wide contexts, including retrieval and generation sub-tasks, while RepoEval evaluates multi-granularity tasks like API usage and function completion. Both benchmarks work with real-world Python repositories and cross-file retrieval. Compared to function-level tasks, these require richer representations and test longer-range dependencies, but they lack comprehensive functional correctness checks. HumanEval and MBPP include small test suites per task, while CodeContests [14] and APPS feature more complex inputs/output pairs. These benchmarks provide rigorous validation of code behavior, but they do not evaluate how well a model retrieves or uses broader context. Functional correctness is necessary but not sufficient for assessing retrieval-augmented systems.

Despite the abundance of benchmarks, few combine repository-level context, function-level generation, and test-based correctness in a single task. RepoBench and RepoEval address retrieval and generation but offer limited or no correctness testing, while HumanEval and similar datasets provide correctness signals but lack realistic multi-file context. DevEval [15] helps bridge this gap by aligning with real-world code repositories across multiple dimensions, including code and dependency distributions. With 1,874 testing samples from 117 repositories, annotated and paired with comprehensive test cases, DevEval enables evaluation in realistic settings and serves as a suitable foundation for our benchmark development.

III. Methodology

We introduce **GRACG** (Graph Retrieval Augmented Code Generation), a repository-aware pipeline that builds a heterogeneous code graph, learns GNN-based node embeddings, retrieves task-relevant context via a learned query–node alignment, and conditions an LLM on the retrieved snippets for function synthesis. An overview is shown in Fig. 1; we provide details below.

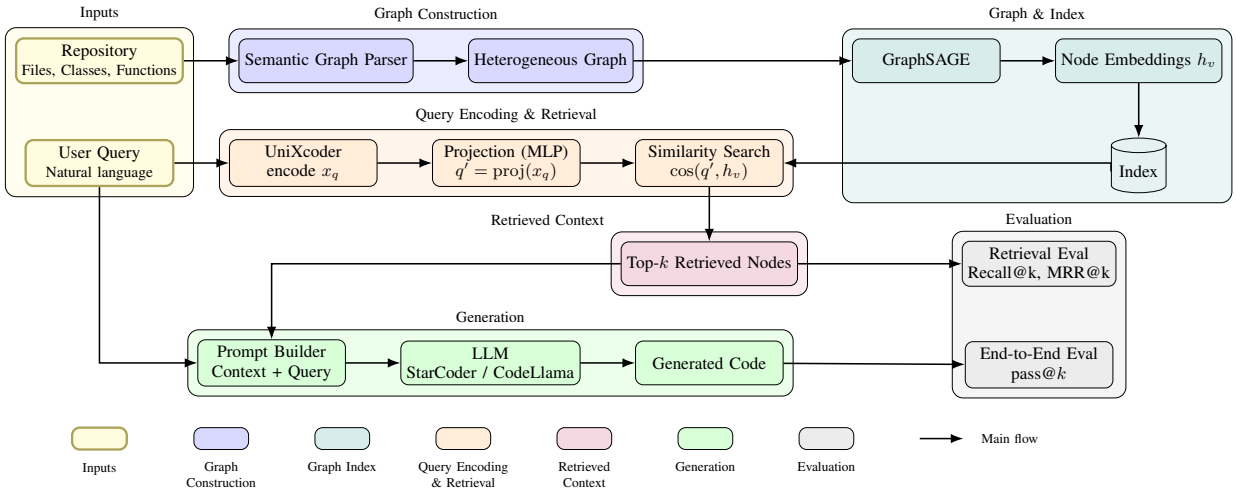

Fig. 1. GRACG pipeline overview. Each block of the pipeline is labeled both by color and captions.

A. Problem Definition

We address the task of *repository-level code generation*, where the goal is to synthesize a function or method implementation using information that may be distributed across multiple files of a repository.

In this setting, the input is a source-code repository together with a natural language query describing the desired functionality. The output is a code snippet that both satisfies the user query and remains consistent with the surrounding repository context.

Because repositories typically contain structural dependencies across files, simply providing a query and a single file as input is insufficient. To overcome this, we retrieve a set of relevant code snippets from the repository to serve as additional context for generation. These snippets are selected using retrieval methods that operate over a semantic graph representation of the repository.

B. Graph Construction

We represent a source code repository as a heterogeneous directed graph

$$G = (\mathcal{V}, \mathcal{E}, \mathcal{T}_V, \mathcal{T}_E, \mathcal{D}), \tag{1}$$

where:

- $\mathcal{V}$ is the set of nodes, with possible node types
$$\mathcal{A} = \{\text{FILE}, \text{CLASS}, \text{FUNCTION}\};$$
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of directed edges, with relation types
$$\mathcal{R} = \{\text{IMPORT}, \text{OWNER}, \text{CALL}, \text{INHERITED}\};$$
- $\mathcal{T}_V : \mathcal{V} \to \mathcal{A}$ maps each node to its type;
- $\mathcal{T}_E : \mathcal{E} \to \mathcal{R}$ maps each edge to its relation type;
- $\mathcal{D} = \{d_v\}_{v \in \mathcal{V}}$ is the set of raw node features, including namespace, docstring, and full code snippet.

Allowed typed edges are defined by the set

$$\mathcal{C} = \{(a, r, b) \mid a, b \in \mathcal{A},\ r \in \mathcal{R},\ \text{edge } (a, r, b) \text{ allowed}\}. \tag{2}$$

We employ the open-source `SemanticGraphParser`[1] to extract G from Python repositories via abstract syntax tree (AST) parsing. This parser was extended to extract function signatures and docstrings. Manual inspection on a random subset of eight repositories confirmed extraction accuracy. A very simple example of the resulting graph structure is illustrated in Fig. 2.

C. Retrieval Approaches

Let $\mathcal{V}$ be the set of candidate nodes in G. A retrieval function

$$\text{retrieve} : \mathcal{Q} \times G \to \mathcal{S}_k \subseteq \mathcal{V} \tag{3}$$

returns the top-k nodes ranked by a relevance score $s(q, v)$.

We use KNN as our baseline method, where both query q and node v are encoded using the `UniXcoder` [16] model to obtain embeddings $x_q, x_v \in \mathbb{R}^d$. Relevance is measured by cosine similarity:

$$s_{\text{KNN}}(q, v) = \frac{x_q \cdot x_v}{\|x_q\| \, \|x_v\|}, \tag{4}$$

where $s_{\text{KNN}}(q, v) \in [-1, 1]$ quantifies the semantic similarity between query q and node v.

In the proposed method, node embeddings h_v are initialized using a heterogeneous GraphSAGE [17] model over G:

$$\{h_v \in \mathbb{R}^n \mid v \in \mathcal{V}\} = \text{GNN}(G), \tag{5}$$

where message passing is restricted to respect encapsulation (e.g., caller-to-callee edges only). We adopt GraphSAGE as it is a well-established GNN architecture with inductive learning capabilities and has been used in prior repository-level retrieval work [9].

[1] https://github.com/Andrchest/SemanticGraphParser

Since query embeddings x_q from `UniXcoder` may not align with h_v, we introduce a learnable projection $\mathsf{proj}(\cdot)$, implemented as a multilayer perceptron (MLP):

$$q' = \mathsf{proj}(x_q), \quad s_{\text{GNN}}(q, v) = \frac{q' \cdot h_v}{\|q'\| \, \|h_v\|}. \tag{6}$$

D. Training Strategy

Our retrieval model is trained in two stages: *pretraining* on code snippets and *finetuning* on docstrings. This approach addresses the scarcity of natural language annotations in repositories while leveraging abundant code structure.

1) Pretraining Phase: The objective of pretraining is to learn repository graph representations by aligning query-like code embeddings with relevant nodes through an edge prediction task.

Let x_v be the `UniXcoder` embedding of the code snippet for node v, and let h_v be its corresponding GNN embedding. For each *anchor* node v, we sample:

- a *positive* node v^+ connected to v by an existing edge of type $r \in \mathcal{R}$;
- a *negative* node v^- chosen uniformly from nodes not connected to v by type r.

We project the anchor embedding via a learnable function $\mathsf{proj}(\cdot)$ and optimize a triplet margin loss:

$$h_v^{\text{proj}} = \mathsf{proj}(x_v), \tag{7}$$

$$\mathcal{L}(v, v^+, v^-) = \max\left(0,\, d(h_v^{\text{proj}}, h_{v^+}) - d(h_v^{\text{proj}}, h_{v^-}) + \alpha\right) \tag{8}$$

where $d(\cdot, \cdot)$ is the cosine distance and α is the margin hyperparameter.

This phase is performed over all relation types, not limited to CALL edges, to capture a broad range of structural dependencies.

2) Finetuning Phase: In the finetuning phase, we adapt the model to natural language queries by replacing code embeddings x_v with docstring embeddings. The training is restricted to the CALL relation, reflecting the primary retrieval goal of locating functions likely to be invoked by a described functionality.

The GNN parameters are frozen, and only the projection layers are updated to align docstring embeddings with the precomputed GNN space.

3) Motivation: This two-stage procedure provides two benefits. Firstly, it allows the GNN to be trained on the entire graph structure without being bottlenecked by limited docstring coverage. Secondly, it produces stable node embeddings that can be reused across queries, enabling efficient retrieval at inference time without re-running the GNN.

E. End-to-End Generation Pipeline

The end-to-end code generation process combines the retrieval component with a large language model (LLM) to synthesize repository-aware code completions. Given a user query q, we retrieve a set of k relevant code snippets from the repository graph G using either the KNN baseline or the proposed GNN-based method. These snippets correspond to full function or class definitions and are accompanied by their docstrings and signatures. The retrieved snippets are concatenated into a structured prompt alongside the user query. The prompt template follows a repository-level format, in which retrieved code context is placed before the query description. The retrieved items are ordered by their relevance scores so that the most relevant snippets appear closest to the query, mitigating truncation when approaching the model's context window limit.

IV. EVALUATION

A. Setup

We evaluate our framework on two complementary tasks:

1) **Retrieval evaluation** on the RepoBench dataset, measuring the ability to retrieve relevant code elements from the repository graph given a natural language description.
2) **End-to-end code generation evaluation** on the EvoCodeBenchPlus benchmark, measuring the functional correctness of generated code when conditioned on retrieved context.

For retrieval, we fix $k = 5$ retrieved items per query. For generation, we use a single completion (*pass@1*) per query.

B. Retrieval Evaluation

We chose RepoBench as the base benchmark for training and evaluating retrieval methods because it contains a large number of repositories and provides line-level granularity with labeled cross-file dependencies. However, we encountered significant reproducibility issues. The authors provide only labeled samples from repositories, along with a GitHub link and an acquisition date, but the date corresponds to the first commit in each repository rather than the actual snapshot used for annotation. Consequently, it was impossible to recover the exact repository versions used in the original benchmark.

To mitigate this, we attempted two strategies: (1) using the current repository state, and (2) reverting to an approximate date of benchmark creation. Both yielded inconsistencies, with missing functions and broken dependencies. Therefore, we opted to use only the repositories provided by RepoBench, discarding the original labeled annotations. For reproducibility, we mapped each loaded repository to its Git commit hash.

We parsed 1451 Python repositories into semantic graphs. The dataset was split by entire graphs into training (60%), validation (20%), and test (20%) partitions, based on the number of (FUNCTION, CALL, FUNCTION) edges with available docstrings. Table I summarizes the split statistics.

To simulate a realistic development scenario where the target code has no existing dependent code in the repository, we:

1) **Initial candidate selection:** For pretraining, select functions not called by any other function but calling at least one function themselves. For finetuning, select functions with non-empty docstrings.

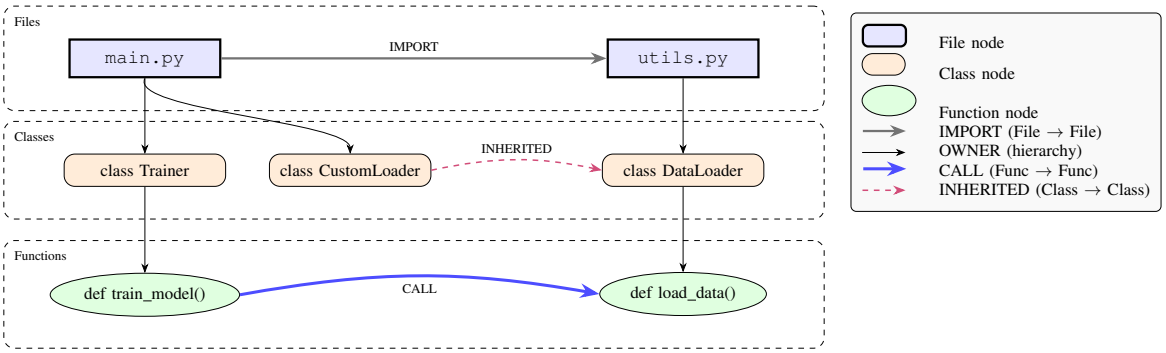

Fig. 2. Example of repository graph.

TABLE I
DATASET SPLIT OVERVIEW FOR RETRIEVAL EVALUATION

Dataset split	Docstring edge pairs	Number of graphs
Train	75456	572
Val	25152	442
Test	25152	437

2) **Graph reduction:** Recursively remove all dependent nodes, including: (i) owners of candidate or removed nodes (and their owned nodes), (ii) callers of candidate or removed nodes, (iii) FILE nodes importing removed or candidate nodes, and (iv) classes inheriting from removed or candidate nodes.
3) **Evaluation candidates:** Remaining candidates in the reduced graph are used for retrieval evaluation.
4) **Retrieval and scoring:** For each candidate, perform retrieval and treat its connected nodes in the original graph as ground truth for computing Recall@k and MRR@k.

1) Metrics: We report Recall@5 and Mean Reciprocal Rank (MRR@5):

2) Results: Table II presents results for the KNN baseline and GraphSAGE models with 3, 5, and 7 layers in both pretraining and finetuning phases.

TABLE II
RETRIEVAL RESULTS ON REPOBENCH TEST SPLIT.

Model	Pretrain Phase		Finetune Phase	
	Recall@5	MRR@5	Recall@5	MRR@5
KNN	0.1993	0.1681	0.1908	**0.1584**
GraphSAGE (3 layers)	0.6756	0.2272	**0.4032**	0.1163
GraphSAGE (5 layers)	0.6927	0.2369	0.3726	0.1037
GraphSAGE (7 layers)	**0.6967**	**0.2538**	0.3571	0.1186

GraphSAGE significantly outperforms KNN in Recall@5, indicating better retrieval coverage. However, in the fine-tuning phase, the MRR metric is higher for KNN while the recall is lower. It may indicate that GraphSAGE better retrieves target nodes but it assigns them lower rank. A possible reason is that the GNN prioritizes structurally central or highly connected nodes (e.g., utility or orchestrator functions, or functions clustered within the same class or file). While these nodes improve coverage, they can displace semantically precise matches from the very top ranks, resulting in lower MRR.

C. End-to-End Code Generation Evaluation

1) Dataset and Baseline Comparison: We conduct end-to-end evaluation on EvoCodeBenchPlus, our curated variant of DevEval [15]. In contrast to the original benchmark, EvoCodeBenchPlus eliminates corrupted repositories, preserves fair evaluation, and provides a reproducible pipeline. As shown in Table III, the DevEval baseline in the *no context* setting achieves 26.55% pass@1 on our dataset, more than double the originally reported 12.54%. This confirms that prior results were affected by flawed test cases and highlights the importance of dataset cleaning.

2) Context Settings: We evaluate three types of context (Table III):

- **No context**: Only the task description and function signature are provided.
- **Gold cross context**: Ground-truth snippets with cross-file dependencies are supplied.
- **GNN context**: Context snippets are retrieved automatically by our GNN-based method.

In the *no context* setting, our models closely match the revised DevEval baseline, validating our evaluation pipeline. Adding *gold cross context* yields only modest gains, with pass@1 not exceeding 30% for any model, despite roughly 30% of tasks containing cross-file dependencies. This suggests that either the benchmark underrepresents true cross-file reasoning or that models struggle to effectively exploit large retrieved code blocks.

The *GNN context* setting produces results similar to *gold cross context*, indicating that our retriever successfully identifies relevant cross-file snippets but does not consistently improve functional correctness. For example, StarCoder 2 improves slightly when using GNN context, suggesting sensitivity to prompt structure and retrieval format. Overall, improvements remain limited, pointing to a mismatch between retrieval quality and the benchmark's ability to reward cross-repository reasoning.

TABLE III
FUNCTIONAL CORRECTNESS ON EVOCODEBENCHPLUS (PASS@1, %)

	pass@1 (%)		
Generation	DeepSeek Coder 6.7B	StarCoder2 7B	CodeLlama 7B
DevEval			
w/o context	26.55	–	26.52
local context (completion)	50.15	47.52	53.23
local context (infilling)	55.29	–	–
EvoCodeBenchPlus			
w/o context	26.65	24.81	27.07
local context (completion)	41.19	40.88	44.21
golden cross context	27.67	24.80	27.12
local context (completion) + gold cross context	**42.64**	**41.17**	**44.35**
GNN context	27.60	25.77	27.34

V. DISCUSSION AND LIMITATIONS

Our results demonstrate that graph-based retrieval with a GNN substantially improves Recall@k compared to a KNN baseline, indicating that structural context from the repository graph enhances identification of relevant code. For example, our retriever achieved Recall@5 close to 40%, a clear improvement over embedding-only methods. This confirms that incorporating repository structure yields richer and more meaningful code representations.

However, these retrieval gains did not consistently translate into higher end-to-end code generation accuracy. Even when supplying *gold context*—the ground-truth set of relevant snippets—pass@1 did not exceed 30% for any model (Table III). This suggests that improvements in retrieval alone are insufficient: many benchmark tasks emphasize in-file reasoning, while cross-file dependencies are underrepresented or loosely defined. Consequently, retrieved snippets, whether gold or GNN-based, often remain underutilized by generation models.

We also observed that local completion—where the model on file-local context—consistently outperforms both gold and retrieved cross-file context. This highlights two important aspects of repository-level generation. First, many benchmark tasks are dominated by in-file dependencies, which can often be resolved from surrounding code such as imports, variable definitions, or sibling functions. Second, large language models appear particularly effective at leveraging file-local continuity, whereas incorporating cross-file snippets introduces longer prompts that may dilute attention or overwhelm the model. Together, these factors explain why local completion establishes such a strong baseline and why gains from cross-file retrieval remain limited in current benchmarks.

Another key contribution of this work is the release of **EvoCodeBenchPlus**, a cleaned and reproducible variant of DevEval. We showed that previously reported results were inflated by corrupted test cases (e.g., baseline pass@1 of 12.54% vs. 26.55% on EvoCodeBenchPlus), underscoring the importance of dataset curation and standardized evaluation.

In contrast to RepoHyper [9], our framework does not require associating a query with a specific node in the graph, a process that can be troublesome and restrictive. Instead, queries and graph nodes are embedded in the same space, enabling flexible retrieval without manual alignment. Moreover, our GNN computes repository-wide embeddings in a single forward pass, after which retrieval reduces to lightweight similarity search. This design is both more efficient and more practical, while also producing richer representations through the combination of pretrained LLM embeddings with structural information.

Beyond code generation, the retrieval component of RAG has potential applications in code search and repository navigation. By surfacing semantically and structurally related functions, the retriever can assist developers in locating examples, identifying dependencies, and exploring project structure. This suggests that graph-based retrieval itself is a valuable outcome, even when end-to-end code generation benefits are modest.

Limitations

Our approach has several limitations:

- **Parser dependency:** The accuracy of the semantic graph depends on the underlying language parser. In Python, dynamic features and missing type annotations may yield incomplete or incorrect graphs.
- **Edge coverage:** Certain relationships, such as global variables or dynamic imports, are not captured, potentially omitting important context.
- **Ranking quality:** Low MRR values indicate that relevant nodes are not consistently ranked at the top. This could be addressed with hard negative sampling or ranking-oriented training objectives.
- **Benchmark scope:** Current evaluation is monolingual (Python) and English-only, limiting generalization to other programming languages and environments.

VI. CONCLUSION

We presented the Graph Retrieval-Augmented Code Generation (GRACG) framework for repository-level code generation. GRACG constructs a semantic graph, encodes nodes with a heterogeneous GraphSAGE model initialized from pretrained LLM embeddings, and retrieves relevant nodes for use in LLM prompts. Compared to a KNN baseline, our approach achieved

significantly higher retrieval recall across a large set of Python repositories.

Compared to RepoHyper, GRACG offers several key improvements: it avoids query-to-node binding, requires only one GNN pass per repository, and produces richer embeddings that integrate structural and semantic repository information. These design choices make GRACG scalable and user-friendly for repository-level reasoning.

We also released EvoCodeBenchPlus, a reproducible benchmark for repository-level code generation. Using this dataset, we validated that earlier DevEval results were unreliable and established a stronger basis for future research. While end-to-end gains remain modest, they are comparable to or exceed those achieved under the gold context setting, providing preliminary evidence for the viability of the proposed method.

Beyond these results, GRACG can also be seen as a reusable component for multi-agent systems. Its modular retrieval and generation capabilities align with recent trends in multi-agent RAG frameworks such as MAIN-RAG [3], highlighting how graph-enhanced retrieval can support collaborative agent workflows in software engineering.

Future Work

Future directions include:

- Incorporating query-aware message passing, e.g., through attention mechanisms or graph transformers [18].
- Using ranking-oriented training and hard negative sampling to improve retrieval ordering.
- Extending parser capabilities to capture additional dependency types such as global variables and dynamic imports.
- Expanding evaluation to multilingual and multi-language repositories to assess generalization.
- Incorporating stronger baselines by comparing against existing GNN-based and repository-aware retrieval systems (e.g., RepoHyper, RAMBO, RepoCoder) to better contextualize performance.
- Designing benchmarks that explicitly require cross-file reasoning, ensuring that retrieval quality directly impacts code generation performance.

In addition to generation, RAG may also serve as a foundation for practical code search and repository exploration tools, where identifying relevant functions and dependencies is itself a valuable outcome for developers.

Code Availability

All code and data used in this work are publicly available:

- GRACG: https://github.com/KonstFed/GRAGC
- EvoCodeBenchPlus: https://github.com/Poseidondon/EvoCodeBenchPlus

Acknowledgments

The study was supported by the Ministry of Economic Development of the Russian Federation (agreement No. 139-10-2025-034 dd. 19.06.2025, IGK 000000C313925P4D0002).

References

[1] GitHub. Github copilot: Your ai pair programmer. https://github.com/features/copilot, n.d. Accessed: 2024-12-03.

[2] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[3] Chia-Yuan Chang, Zhimeng Jiang, Vineeth Rakesh, Menghai Pan, Chin-Chia Michael Yeh, Guanchu Wang, Mingzhi Hu, Zhichao Xu, Yan Zheng, Mahashweta Das, and Na Zou. MAIN-RAG: Multi-agent filtering retrieval-augmented generation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2607–2622, Vienna, Austria, July 2025. Association for Computational Linguistics.

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

[5] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore, December 2023. Association for Computational Linguistics.

[6] Tuan-Dung Bui, Duc-Thieu Luu-Van, Thanh-Phat Nguyen, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. Rambo: Enhancing rag-based repository-level method body completion. *arXiv preprint arXiv:2409.15204*, 2024.

[7] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *Advances in Neural Information Processing Systems*, 37:132876–132907, 2024.

[8] Costas Mavromatis and George Karypis. GNN-RAG: Graph neural retrieval for efficient large language model reasoning on knowledge graphs. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 16682–16699, Vienna, Austria, July 2025. Association for Computational Linguistics.

[9] Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. Repohyper: Better context retrieval is all you need for repository-level code completion. *arXiv preprint arXiv:2403.06095*, 2024.

[10] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[11] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[12] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. In J. Vanschoren and S. Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

[13] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Representation Learning*, volume 2024, pages 47832–47850, 2024.

[14] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[15] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar,

editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[16] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[17] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[18] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.