GRACG: Graph Retrieval Augmented Code Generation

Abstract. Modern code generation tools, powered by Large Language Models (LLMs), typically rely on file-level context and often ignore structural information and dependencies distributed across an entire code repository. This limitation leads to inaccurate code suggestions, especially in real-world projects where cross-file relationships are important. To address this gap, we propose a graph-based Retrieval-Augmented Code Generation (RACG) 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 did not observe significant improvements in the metrics, even when using target-relevant functions.

Keywords: graph neural networks, repository-level retrieval, code generation

1. 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 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) 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 semantic 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 resulting. 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.

2. Related Works

2.1 Graph retrieval

Traditional LLM-based code generation systems like GitHub Copilot 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** [1] 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 impose issues when encountering some unseen type of dependency.

Graph Neural Networks (GNNs) is 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 [2] 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 article [3] explores the use of GNNs for knowledge graph retrieval by classifying nodes as either relevant or not relevant to a given query

Main inspiration for this work was paper [4] where **RepoHyper** framework was proposed. This method extracts classes, modules, fields and functions with additional rich set of relations between them in a form of directed graph. Then, the method retrieve context from the graph with help of the both classical approaches and graph neural network (GNN). Firstly, the approach finds relevant nodes by embedding similarity to a prompt they serve as starting points in a graph search. Then, it employs breadth-first search with some limit to a number of node and depth. Then, the method inserts new node to the graph and tries to predict which existing nodes should be connected to it by using GNN. The GNN is trained on a link prediction problem in an unsupervised manner. However, following 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.

2.2 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** [5], **MBPP** [6], and **APPS** [7] 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** [8] and **RepoEval** [9] 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 repos and stress 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** [10] 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. Our work addresses this by proposing a new benchmark that combines all three aspects, enabling holistic evaluation of models in more realistic settings.

Few benchmarks integrate repository-level context, function-level generation, and test-based correctness. **RepoBench** and **RepoEval** address retrieval and generation but offer limited or no correctness testing. **HumanEval** and similar datasets provide correctness signals but lack realistic multi-file context. **DevEval** addresses this gap by aligning with real-world code repositories across multiple dimensions, including code and dependency distributions. It comprises 1,874 testing samples from 117 repositories, annotated by developers, and includes comprehensive annotations and test cases. **DevEval's** design facilitates the evaluation of models in realistic settings, making it a suitable foundation for our benchmark development.

3. Methodology

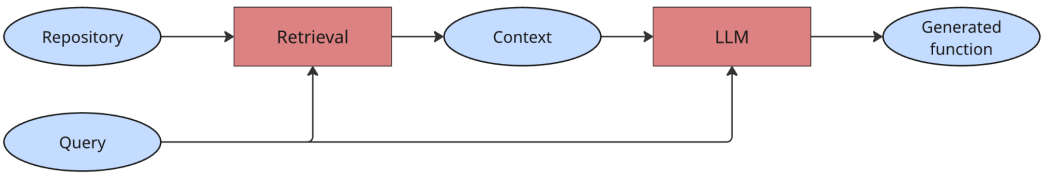

Fig. 1. Overview of pipeline

Repository level code generation problem can be formalized as a task which inputs code repository and user prompt and returns generated code function. The general flow of the proposed pipeline is illustrated in Fig. 1. Here we split the problem into two big parts: retrieval and generation.

3.1 Graph Construction

We can define repository semantic graph as directed heterogeneous graph $G = \left(V, E, T_V, T_E, D\right)$ where:

- V is a set on nodes with node types A = {FILE, CLASS, FUNCTION}
- $E \subseteq V \times V$ is a set of directed edges with relation types R = {IMPORT, OWNER, CALL, INHERITED}
- $T_V: V \to A$ maps each node to its type

- $T_E: E \to R$ maps each edge to its relation type
- $D = \{d_v\}_{v \in V}$ is the set of raw node features

The graph permits only specific typed edge connections between node types,
formally described by the set

$$C = \{(a, r, b) \in A \times R \times A | allowed\ edge\ from\ node\ type\ a\ to\ b\ via\ relation\ r\}$$

The allowed combinations C are:

- **Ownership edges** (OWNER): from FILE, CLASS or FUNCTION to CLASS or FUNCTION
- **Call edges** (CALL): from FILE, CLASS, or FUNCTION to FUNCTION
- **Import Edges** (IMPORT): from FILE to any node type
- **Inheritance Edges** (INHERITED): between CLASS nodes

Each node $v \in V$ in the heterogenous graph G is associated with a tuple of features, where the structure of d_v depends on the node type. All node types contain following features:

- **namespace**: namespace in repository
- **docstring**: if present human labeled documentation or description
- **code**: snippet containing whole class or function from repository

To extract a graph repository, we decided to use the open-source project *SemanticGraphParser* [65] which is based on abstract syntax tree static parser for Python language. We manually checked the correctness of the parser on the 8 random repositories. In addition, signature and docstring extraction was implemented and added to the tool.

3.2. Retrieval

Retrieval problem can be formulated as follows:

$$retrieval: Q \times G \to \{v_1, v_2, \dots, v_k | v_i \in V\}$$

Where Q is a set of all possible user queries, G is a semantic graph and result of retrieval function is a list of graph nodes given in relevancy order.

To solve this problem, we propose assigning each candidate for retrieval a score, where the higher the value the more relevant it is to the query. Then, we simply select top k nodes as our retrieval output.

As our baseline method we will use **KNN** where we encode user query and all graph nodes using the same LLM to project them into the same latent space. In our work **UnixCoder** was used as embedder because it was successfully used in RepoHyper and trained jointly with text and code. Therefore, the embedder should perform better on representing code than general LLM. We assume that relevant code blocks will be similar to user query in the latent space. Score function here can be defined as:

$$score: E \times E \to R score(q, x_v) = cosine\_sim(q, x_v)$$

Where q is an embedding representation of the user query and x_v sis a node embeddings representation based of its code snippet. Then, based on that function we retrieve k nodes with highest score. It is a straightforward and robust method that is used in most of the RAGs. In addition, it does not require any training and has single hyperparameter *k*.

3.2. GNN method

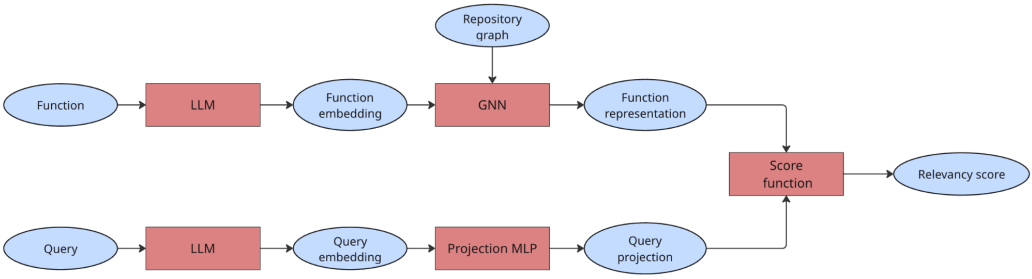

Fig. 2. Overview of GNN relevancy score computation

General flow of the approach can be described as follows. First, we first use a GNN to initialize the node embeddings. Second, for each node we compute the score using user query and node embedding with cosine similarity. Last, we select the nodes with the highest scores. The intuition here is that the GNN incorporates the neighbors information into the node representations, improving their accuracy. For instance, a coordinator function that just calls other functions may achieve a better representation by aggregating the information from its called functions, rather than relying solely on its own code.

In our context, GNN can be formulated as a function that inputs graph and outputs set of embeddings:

$$GNN: G \rightarrow \{h_v \in R^n | \forall v \in V\}$$

Where h_v is node embedding for a node v with a size of n.

However, in this method latent space of the user query may not align to GNN node space after its training. To solve this issue we propose using a learnable projection function implemented via multilayer perceptron (MLP):

$$q' = proj(q),\ q' \in R^n$$
$$score\left(q, h_v\right) = cosine\_sim\left(q', h_v\right)$$

3.2. Training details

3.2.1 Model architecture

We propose using a heterogeneous **GraphSAGE** model as our GNN in a repository semantic graph. This method is well known and often used for graph based tasks. In addition, we decided to align message passing with the principle of encapsulation. Thus, messages should pass only from caller to callee and not from the callee back to the caller. By applying this rule a function representation will not take into account in which ways it is being used.

The model consists of several layers of **GraphSAGE** convolutions. The model contains multiple convolution layers, where each layer corresponds to a different edge type. The convolution layers aggregate node features using the mean aggregation function, and each edge type has its own convolution (**SAGEConv**):

- For file ownership relations, the SAGE convolution updates node features for edges of type:
  - (FILE, OWNER, CLASS)
  - (FILE, OWNER, FUNCTION)
- For function and class ownership relations we assign edges:

    - (CLASS, OWNER, FUNCTION)
    - (CLASS, OWNER, CLASS)
    - (FUNCTION, OWNER, FUNCTION)
    - (FUNCTION, OWNER, CLASS)
- For call relations edges of type
    - (FUNCTION, CALL, FUNCTION)
    - (CLASS, CALL, FUNCTION)
    - (CLASS, INHERITED, CLASS)
- For file import and call relations (IMPORT), the SAGE convolution updates node features for edges of type:
    - (FILE, IMPORT, FILE)
    - (FILE, IMPORT, CLASS)
    - (FILE, IMPORT, FUNCTION),
    - (FILE, CALL, FUNCTION)

3.2.1 Projection layers

For each **SAGEConv**, we use a corresponding **MLP** projection. The projection operates on the query embeddings and transforms them into a new space. The purpose of this transformation is to generate embeddings where higher scores correspond to a higher likelihood of a particular type of edge existing between the query and another node in the graph. **The projection layer is responsible for refining the query embeddings to better align with potential relationships in the graph**. For example, for the edge type (FUNCTION, CALL, FUNCTION), the projection function should produce embeddings where the score is higher for nodes that are likely to be "called" by a given query. Similarly, for (CLASS, OWNER, FUNCTION), the projection should emphasize functions that are likely to be "owned" by a given query.

In summary, the projection layers take the embeddings of query nodes and adjust them in such a way that they better represent the relationships (edges) that might exist between the query and nodes in the graph while having only one embedding for each node.

3.2.1 Dataset

We chose **RepoBench** [REPOBENCH] as the base benchmark for training and evaluating retrieval methods because it has a considerable number of repositories. In addition, it provides line level granularity with labeled cross-file dependencies if present. However, authors provided only labeled samples from repositories and included only GitHub link and its supposed acquisition date. Furthermore, the given dates were actually the date of first git commit in a repository. Thus, it was impossible to acquire the same version codebases.

To mitigate the issue, we decided to use the current date and approximate date of benchmark creation to revert all repositories up to that date. For both attempts there were inconsistencies and some functions were missing.

For this reason, we used only repositories provided by **RepoBench** without labeled annotations. In addition, we have mapped all loaded repositories with its git commit hash for better reproducibility.

To avoid any possibility of data leak in the evaluation dataset was split by the entire graph, meaning it is used either for training, validation or testing. Finally, we splitted dataset into train, validation and test with 0.6, 0.2, 0.2 ratios respectively by number of pairs (FUNCTION, CALL, FUNCTION) with docstring.

To sum up, we collected **1451 python repositories** and saved their commit hash. Then, we parsed them into semantic graphs. Also, split statistics can be viewed in Table 1.

Table 1. Dataset split overview

Dataset split	**N. of Docstring Edge Pairs**	**N. of graphs**
Train	75456	572
Validation	25152	442
Test	25152	437

3.2.1 Training process

Our main assumption is that using docstring with natural language function description is similar to user queries. However, only a relatively small quantity of functions or classes have docstring. Thus, a major part of a graph would not be used for training and insufficient amount of labeled data would not allow for effective training.

For these reasons we propose pretrain task where instead of function or class docstring we use its code snippet as a query. Furthermore, we train a model on an edge prediction problem including not only CALL relations. After that, we *"freeze"* GraphSAGE weights and finetune CALL projections on docstrings.

For the pretrain phase, we randomly choose anchor nodes and sample positive and negative edge pairs to them. Then, the original feature anchor node embeddings (i.e., the raw embeddings from the source code) are projected into the embedding space learned by GraphSAGE. After that, we calculate triplet margin loss:

$$h_v^{proj} = proj\left(x_v\right)$$

$$L(v, v_{pos}, v_{neg}) = max(0,\ d(h_v^{proj}, h_{v_{pos}}) - d(h_v^{proj}, h_{v_{neg}}) + \alpha)$$

Where:

- v are anchor nodes
- v_{pos}, v_{neg} are positive and negative nodes
- x_v is an embedding from UnixCoder using just node code snippet
- d is the cosine distance
- h_v^{proj} are projected anchor nodes embeddings
- $h_{v_{pos}}$ and $h_{v_{neg}}$ are the GraphSAGE node embeddings of the positive and negative nodes respectively.

The finetune or “docstring” phase process is very similar but we sample only from (**FUNCTION, CALL, FUNCTION)** relation and for anchor nodes we use docstring embeddings instead of code.

3.2.1 Evaluation process

Evaluation process can be described in these steps:

- **Initial Candidate Selection**: For pretrain phase we choose functions that are not being called by any other function but are themselves calling another function while for

finetune we choose functions with non-empty docstring. These functions serve as our initial evaluation candidates.

- **Graph Reduction**: From the selected functions, we recursively remove all dependent on candidate nodes from the graph. The rules for node reduction are as follows:
    - Nodes that own candidates or removed nodes including all of their owned nodes.
    - Nodes that call removed nodes or candidate nodes.
    - FILE nodes that import removed or candidate nodes.
    - Nodes that inherit removed or candidate nodes.
- **Evaluation Candidates**: The remaining in the graph candidates, after the reduction process, are considered the evaluation candidates. These nodes are used to assess the model's performance and also removed from the graph.
- **Retrieval:** for each candidate we perform retrieval. Then, using nodes that were connected to the candidate as target we calculate *recall@k, mrr@k*

In our evaluation setup we made **graph reduction** in order to simulate a real life scenario when a user wants to program a new function and tries to find relevant snippets of code. Thus, the graph should not contain any code that is dependent on the current evaluated function or class.

3.2.x End-to-end benchmark construction

To construct a high-quality dataset for benchmarking, we prioritize leveraging existing datasets and benchmarks. We select **DevEval** as a reference baseline for assessing the generation capabilities of our framework due to its repository-level scope, rich task annotations, and inclusion of functional tests. However, upon closer inspection, we identify several critical shortcomings that limit its reliability and reproducibility: missing or broken evaluation scripts, lack of environment setup tools, and inconsistencies in baseline metrics. In particular, the provided *pass@k* scripts fail to validate correct solutions, raising concerns about the trustworthiness of prior results.

To address these issues, we build upon EvoCodeBench—an earlier version of DevEval—and introduce **EvoCodeBenchPlus**, a refined benchmark that reimplements the evaluation pipeline with improved robustness and transparency. Our key contributions include accurate oracle validation, detailed test execution logs, automated virtual environment (venv) setup, and filtering of corrupted repositories. As a result, we produce a cleaned and reproducible benchmark consisting of **1262** tasks, with validated ground-truth completions that achieve *pass@1* = 1.0.

Despite these improvements, EvoCodeBenchPlus inherits certain limitations. It remains monolingual, restricted to Python code and English requirements. This constraint extends to the retrieval-augmented generation (RAG) pipeline, which is built on a Python-specific GNN and parser. Additionally, environment setup remains partially platform-dependent, and the benchmark currently supports only models that adhere to a specific repository-level prompt template.

Nonetheless, EvoCodeBenchPlus provides a more reliable foundation for evaluating repository-level code generation. It enables reproducible, functionally grounded assessment of code LLMs under realistic development scenarios, paving the way for future work on multilingual support, broader language coverage, and more diverse model compatibility.

4. Evaluation and Discussion

4.1 Retrieval results

Retrieval metrics on the test split of RepoBench dataset for KNN and different number of layers GraphSAGE are presented in Table 2.

Finetune metrics better correspond with the goal of a retrieval because docstring is closer to the user query than code. Also, code itself contains which function it calls so it is possible that model will overfit on that data. Thus, finetune metrics are more important to us.

Results show significant boost in recall when compared to baseline KNN which is present on both pretrain and finetune. However, on the finetune phase MRR metric is higher for KNN while recall

is lower. It may indicate that GraphSAGE better retrieves target nodes but it assigns them lower rank.

Table 2. Retrieval metrics

	Pretrain phase		Finetune phase	
Model	Recall@5	MRR@5	Recall@5	MRR@5
KNN	0.199304	0.168072	0.190836	**0.158406**
GraphSAGE with 3 layers	0.675642	0.227222	**0.403165**	0.116296
GraphSAGE with 5 layers	0.692699	0.236872	0.372606	0.103692
GraphSAGE with 7 layers	**0.696718**	**0.253803**	0.357146	0.118612

4.2.2 End-to-end Evaluation

Table 3. Results for different diode insertion strategies.

Generation	**DeepSeek Coder 6.7b**	**StarCoder 2 7b**	**CodeLLaMa 7b**
w/o context	26.65	24.81	27.07
gold cross context	TODO ~30	TODO ~30	TODO ~30
GNN local+cross context	27.60	27.34	25.77

Evaluation Without Context. Table 3 compares the DevEval baseline completions with our model outputs using the cleaned EvoCodeBenchPlus dataset. In the "no context" setting—where no surrounding code is provided—the DevEval baseline achieves a pass@1 score of 26.55% on our filtered dataset, substantially higher than the originally reported 12.54%. This discrepancy suggests that the original DevEval dataset likely included corrupted test cases or suffered from evaluation flaws. Our model's performance closely matches the revised baseline, with minor differences attributable to prompt formatting or generation parameters. These results validate both our dataset cleaning process and the improved evaluation pipeline.

Limited Gains from Gold Context. When evaluating with gold context—where the model has access to the ground truth set of relevant code snippets—the improvement in pass@1 was modest, not exceeding 30% across tasks. This limited gain suggests that even perfect retrieval does not drastically enhance generation quality. One possible explanation is that many benchmark tasks emphasize in-file context, which models can often infer or approximate without external information. Additionally, tasks requiring cross-file reasoning are less prevalent and may not be well-represented or carefully designed by benchmark authors, leading to an underutilization of retrieved context in these evaluations.

6 Conclusion

In this paper, we introduced graph-based Retrieval-Augmented Code Generation (RACG) framework that leverages repository-level context. We proved that utilizing GNN over repository semantic graph encoded with pretrained LLM produce more meaningful and rich representations of functions with incorporation of related code. Thus, leading to more efficient retrieval of code snippets comparing to using only LLM.

In contrast to [RepoHyper] we managed to not associate query to some node in graph which is often troublesome for user. Also, our method runs GNN only once to produce more rich representations of repository function and classes which are later used in retrieval.

However, we did not achieve meaningful end-to-end metrics boost. The possible reason is problems with evaluation benchmark because it authors did not focus on cross repository context.

6 Limitation and Future Work

One of the main limitation of this approach is dependency on the language parser. Different parsers transform repository into graph in a different manner. For example, EvoCodeBench labeled cross references are often include some global variables that are unfortunately ignored by our parser. Moreover, the lack of types and dynamic nature of Python language often produce not accurate graphs.

Regarding GNN retrieval it is worth to research incorporation of a query in message passing process. One of the possible ways is to use attention mechanism to the query when aggregating neighbors. Moreover, using GAT in this context may be for beneficial because called function are not equal when it comes to representing function meaning.

In addition, low MRR may indicate that there is strong false positive node presence which might be solved by using hard sampling. Instead of choosing negative nodes randomly and uniformly we can select nodes with high score. Thus, all false positives nodes with high score will be sampled for loss and diminished by it.

References

[1]. D. Shrivastava, D. Kocetkov, H. de Vries, D. Bahdanau, and T. Scholak, “Repofusion: Training code models to understand your repository,” arXiv preprint arXiv:2306.10998, 2023.

[2]. He, X., Tian, Y., Sun, Y., Chawla, N., Laurent, T., LeCun, Y., Bresson, X. and Hooi, B., 2024. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *Advances in Neural Information Processing Systems*, 37, pp.132876-132907.

[3]. Mavromatis, C. and Karypis, G., 2024. Gnn-rag: Graph neural retrieval for large language model reasoning. *arXiv preprint arXiv:2405.20139*.

[4]. Phan, H. N., Phan, H. N., Nguyen, T. N., & Bui, N. D. (2024). Repohyper: Better context retrieval is all you need for repository-level code completion. *CoRR*.

[5]. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[6]. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... & Sutton, C. (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

[7]. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., ... & Steinhardt, J. (2021). Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

[8]. Liu, T., Xu, C., & McAuley, J. (2023). Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

[9]. Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., ... & Chen, W. (2023). Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

[10]. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... & Vinyals, O. (2022). Competition-level code generation with alphacode. Science, 378(6624), 1092-1097.