Improving project-level code generation using combined relevant context

Dmitriy Fedrushkov[0009−0003−9088−4056],
Denis Tereshchenko[0009−0003−7497−676X],
Sergey Kovalchuk[0000−0001−8828−4615], and Artem Aliev[0000−0001−7984−4721]

Chebyshev Research Center, Saint Petersburg, Russia
fedrushkov.dmitriy1@huawei.com

Abstract. Within this study, we propose and evaluate an approach to structure and improve context provided in RAG-based solutions for code generation. The approach is based on combination of semantically relevant API and code selection and filtering for better context representation in following LLM prompt. The experimental evaluation performed with CodeGen-350M-mono and several popular benchmarks such as RepoCoder, CoderEval, CoIR show good overall performance (even in comparison to bigger LLMs). Also, the experimental evaluation shows improvement with narrower and more focused context representation (project-scope API instead of popular public API).

Keywords: Artificial Intelligence · Natural Language Processing · Code Generation · Retrieval Augmented Generation

1 Introduction

Retrieval Augmented Generation (RAG) has pushed the boundaries of Large Language Models applications in versatile domains, including software development [5]. Involving the relevant information from external sources to generative models might strongly enhance the outputs in such tasks in software engineering as code generation, code search, question answering and many others. It is also applicable to internal sources and could make language models be useful when working with documents that must not be disclosure [7]. Despite the mentioned advantages, RAG has its own limitations. Inconsistent use of context to be provided to generative AI could lead to reduction of performance which is not resolved with larger size of information provided [2]. This emphasizes the necessity of accurate use of information retrieval techniques when working with LLMs. We believe that small but properly structured pieces of the most relevant information might be more efficient than providing the whole information from relevant documents. Thus, our work is focused on development and investigation of the various approaches of improving RAG prompts by relevant context and their combinations in software development (e.g, code generation task).

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_52

2 Context structuring and optimization

The main idea of the study is to investigate possible performance improvement in RAG-based solution with smaller yet more structured and more relevant information given as a context in LLM query. We consider text-to-code problem of code generation, i.e. given a natural language query we want LLM to generate a source code in the target language resolving the stated problem. Our approach is based on combination of two main sources of information for query extension: relevant API search and code search. The extracted API and code can be included into the prompt of an LLM (see Fig. 1).

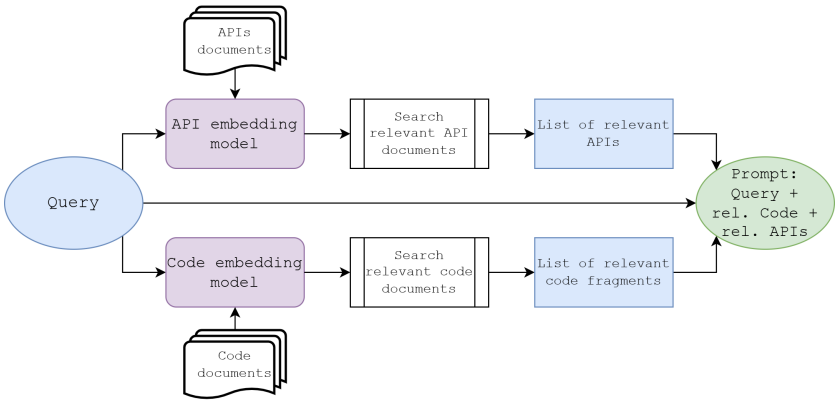

Fig. 1. Prompt creation pipeline

Having the relevant information extracted by retrievers, we've selected the prompt template for code generation with the following structure. Here, `RETRIEVED APIS` and `RELEVANT CODE FRAGMENTS FROM REPO` stays for relevant API and pieces of code extracted by search algorithms respectively. `FUNCTION SIGNATURE` defines target function signature. `DOCSTRING` includes natural language query for code generation task.

```
# Here are some relevant code fragments from other files of the repo:
# --------------------------------------------------
%RELEVANT CODE FRAGMENTS FROM REPO%
# --------------------------------------------------
# Using apis: %RETRIEVED APIS% continue the function:
def %FUNCTION SIGNATURE%:
    %DOCSTRING%
```

Bidirectional decoder for code retrieval. Recent works [1] demonstrated that replacing unidirectional attention masks in decoder-only language models (LLMs) with bidirectional variants enables competitive performance in retrieval tasks. Building upon this, we propose a fine-tuning framework for code retrieval that integrates masked next-token prediction and hard negative con-

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_52

trastive learning, achieving state-of-the-art results on Python code-search benchmarks.

Our approach adapts pre-trained decoder models (e.g., CodeGen 350M-mono) for retrieval tasks through the following key steps:

1. **Structural Tokenization:** Code tokens are augmented with explicit indentation markers to capture syntactic hierarchies.
2. **Bidirectional Training:** Masked next-token prediction with bidirectional attention enables full-sequence context utilization.
3. **Hard Negative Mining:** During training, code snippets are encoded into vectors using the current model checkpoint. The top-k nearest neighbors to each query, excluding the ground truth, are selected as hard negatives for contrastive learning.

Within our experimental study, we fine-tune CodeGen 350M-mono using an `EOS` token pooling strategy on the AdvTest and CodeSearchNet (CSN) datasets, among others. The training procedure is based on a combination of following methods: Masked Language Modeling (MLM) (30% of tokens are masked, and the model reconstructs them using bidirectional attention); Hard Negative Contrastive Learning (a cross-entropy loss is applied to similarity scores between the query and positive/negative code pairs).

Relevant API retrieval. We used a machine learning search algorithm to search for semantically relevant APIs. Receiving natural language query as an input, the search system returns the list of relevant APIs, which could be applicable to solve the programming question. While conventional code search methods return the list of code slices, which can help to tackle the task, our approach suggests the functions' qualified names from standard, publuc and local libraries. For instance, this apparatus returns System.out.println java method when receiving "How to print something" question instead of searching for whole code slice. Our experimental setup involves the following data processing: rather than manually collecting documentation of popular APIs, like Pandas, Torch, etc. [9], we downloaded 26728 open-source Python repositories through GitHub API and parsed them to extract the information about public API calls in functions and form a private API usage database [7]. To avoid the insufficient uses of APIs in real code examples, we collected only the repositories with more than 100 stars. Then we used the input queries (function docstring) to match them with the API calls and develop our ML approach to search semantically relevant APIs.

API database. We used the test part of previously collected public Python repositories as a database to apply searching for the public APIs that would help in developing process according to user necessity. Instructing model to use these APIs seems to improve the quality of generation. To extend the API database, we used an information about project imports and locally installed packages to provide the context about public and/or local APIs that was already used in project. It allowed us to make the context be focused on project scope; this might involve private context to development process and appears to enhance the generated code security owing to use trusted libraries.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_52

Benchmark. We've used two popular repository-level benchmarks for evaluation of our approach, namely RepoCoder [8] and CoderEval [6]. For most of the experiments, we used the Python part of CoderEval benchmark to assess the impact of various contexts in the valid code generation. CoderEval implemented multiple levels of project scopes, which allows us to assess the usefulness of different suggestions from multiple angles. This benchmark also provided the Oracle context that could be used to complete the code, making us able to compare the effectiveness of our retrieved information with the "golden" context. We also used the repositories provided by CoderEval to retrieve the relevant piece of code, which were added to prompts while generating the solutions.

Code generation model. We applied CodeGen-350M-mono [4] model to complete the code generation tasks. Since this model was used in original CoderEval approach, we compared the possibilities of each useful context to improve the efficiency of code generation. The model was run in sampling mode with 256 max new tokens and other parameters equal to default values.

3 Results

Code retrieval. For preliminary evaluation of code retrieval with our training procedure we used CoIR benchmark [3]. For CodeGen 350M-mono, we obtained NDCG@10 metric 0.475 which shows relatively good performance outperforming such models as UniXCoder (0.373), BGE-M3 (0.393), OpenAI-Ada-002 (0.456). The better performance in the benchmark was recorded only by significantly bigger models such as E5-Mistral and Voyage-Code-002.

Next, we evaluated our model on the RepoCoder and CoderEval benchmark for code retrieval and completion tasks for Python code. We were focusing on enhancing bidirectional CodeGen performance but also have fine-tuned UnixCoder-base on AdvTest and CodeSearchNet Python dataset. Exact Match (EM), Edit Similarity (ES), and Edit Tree Distance (ED) were used as quality metrics, with results averaged across repositories (Table 1). The results show that the choice of encoder-decoder architecture significantly impacts performance. For the best results are achieved when the bidirectional CodeGen 350M-mono is used. This configuration outperforms all other combinations, achieving an EM score of 0.321 for API completion and 0.423 for line completion (RepoCoder tasks). The Edit Tree Distance scores for these tasks are 0.690 and 0.769, respectively. In contrast, standalone decoders (without a separate encoder) perform poorly, underscoring the importance of context-aware encoding for code generation tasks.

For CoderEval benchmark we have also tested several decoders together with bidirectional Codegen 350M as well as function completion without RAG (Table 2). The bidirectional CodeGen demonstrated promising results on several benchmarks and demonstrated significant improvement in passing tests for CodeEval function generation benchmark.

The results from CoderEval reveal that bidirectional CodeGen 350M-mono achieves the highest pass@1 score of 0.250, outperforming even the larger 6B

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_52

Table 1. Code generation evaluation with RepoCoder

Encoder	Decoder	Compl. type	EM	ES	ED
-	CodeGen 350M-mono	API	0.204	0.496	0.610
UnixCoder-base	CodeGen 350M-mono	API	0.233	0.482	0.600
UnixCoder (fine-tuned)	CodeGen 350M-mono	API	0.310	0.585	0.679
CodeGen 350M-mono	CodeGen 350M-mono	API	**0.321**	**0.597**	**0.690**
-	CodeGen 350M-mono	Line	0.263	0.520	0.690
UnixCoder-base	CodeGen 350M-mono	Line	0.340	0.540	0.712
UnixCoder (fine-tuned)	CodeGen 350M-mono	Line	0.398	0.625	0.762
CodeGen 350M-mono	CodeGen 350M-mono	Line	**0.423**	**0.638**	**0.769**

Table 2. Code generation evaluation with CoderEval

Encoder	Decoder	EM	ES	ED	pass@1
-	CodeGen 350M-mono	0.200	0.512	0.574	12.3%
UnixCoder-base	CodeGen 350M-mono	0.200	0.534	0.59	19.3%
UnixCoder-(fine-tuned)	CodeGen 350M-mono	0.215	0.559	0.618	21.0%
CodeGen 350M-mono	**CodeGen 350M-mono**	**0.222**	**0.547**	**0.606**	**24.8%**
-	CodeGen 6B-mono	0.210	0.532	0.592	17.1%

model, which achieves a pass@1 score of 0.171. This demonstrates that smaller models with RAG can outperform larger models.

RAG context results. Next, we've evaluated the influence of context given with RAG onto performance of code generation with CoderEval benchmark using `pass@1` metric. Table 3 represents the evaluation results with the use of different prompting strategies. Having versatile prompts and scopes of project, we assessed the influence of each relevant information.

Table 3. Code generation results with different contexts (pass@1)

Scope	Query	Oracle	Pub API	Project API	Project Code	Project API+Code	ChatGPT 3.5
plib runnable	4.76%	19.05%	19.05%	19.05%	28.57%	**38.10%**	21.43%
project runnable	3.91%	8.70%	8.70%	8.70%	4.35%	**34.78%**	9.57%
class runnable	5.82%	10.91%	16.36%	20.00%	**39.62%**	26.42%	8.73%
self contained	22.57%	42.86%	42.86%	28.57%	40.00%	40.00%	**52.29%**
file runnable	7.79%	17.65%	19.12%	19.12%	14.71%	19.12%	**21.03%**
slib runnable	12.50%	35.71%	25.00%	**39.29%**	17.86%	28.57%	15.36%
overall	9.48%	21.30%	21.74%	22.17%	24.78%	**28.26%**	21.04%

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_52

Providing relevant context that would not overwhelm the model resulted in undeniable upgrades of project-level code generation. We observed significant improvement of small model contribution that overcame ChatGPT generation results. As expected, different prompting strategies influenced various aspects of project-level code completion. Interestingly, combined context improved project runnable by approximately 4 times in comparison with Oracle context, while use of only one of context suggestions did not improve this at all. All these might signalize that the quality of context in project scope is more essential than the quantity of model parameters.

Code generation example. We've inspected several randomly chosen tests in the CoderEval experimental evaluation and see that our RAG approach provides reasonable improvement in the code generation process. For example, for the following task (CoderEval 62ece4982e6aefcf4aabbd65) we have the following natural language query:

```
Given an argparse.ArgumentParser instance, return its argument flags in a
    space-separated string.
Args: script: argparse.ArgumentParser instance
Returns: argument flags in a space-separated string
```

Ground-truth solution given by the benchmark was as follows:

```
def parser_flags(parser):
    """
    Given an argparse.ArgumentParser instance, return its argument flags in a
      space-separated string.
    Args:
    script: argparse.ArgumentParser instance

    Returns:
    argument flags in a space-separated string
    """
    return ' '.join(option for action in parser._actions for option in action
    .option_strings)
```

The model without RAG extension gives the following solution:

```
def parser_flags(parser):
    """
    Given an argparse.ArgumentParser instance, return its argument flags in a
      space-separated string.
    Args:
    script: argparse.ArgumentParser instance

    Returns:
    argument flags in a space-separated string
    """
    flags = ""
    for arg in parser.arg_strings:
            flags += " " + arg
    return flags
```

Our approach constructed following RAG-based prompt for code generation:

```
# Here are some relevant code fragments from other files of the repo:
# --------------------------------------------------
```

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_52

```
# the below code fragment can be found in:
# borgmatic-commands-completion-parser_flags5.py
# borgmatic-commands-completion-parser_flags0.py
# --------------------------------------------------
# import argparse
#
# def parser_flags(parser: argparse.ArgumentParser) -> str:
#     return ' '.join(parser.prefix_chars + flag for action in parser.
    _actions for flag in action.option_strings)
#
#
# def test_parser_flags():
# --------------------------------------------------
# the below code fragment can be found in:
# borgmatic-commands-completion-parser_flags1.py
# --------------------------------------------------
# import argparse
#
# def parser_flags(parser: argparse.ArgumentParser) -> str:
#     return ' '.join(parser.prefix_chars + option_strings for action in
    parser._actions for option_strings in action.option_strings)
#
#
# def test_parser_flags():
# --------------------------------------------------
# Using apis: ['borgmatic-commands-completion-parser_flags1.parser_flags', '
    borgmatic-commands-completion-parser_flags2.parser_flags', 'borgmatic-
    commands-completion-parser_flags3.parser_flags', 'borgmatic-commands-
    completion-parser_flags4.parser_flags', 'getattr'] continue the function:
def parser_flags(parser):
    """
    Given an argparse.ArgumentParser instance, return its argument flags in a
     space-separated string.
    Args:
    script: argparse.ArgumentParser instance

    Returns:
    argument flags in a space-separated string
    """
```

Finally, the following code was obtained with much better matching to the original (ground-truth) solution:

```
def parser_flags(parser):
    """
    Given an argparse.ArgumentParser instance, return its argument flags in a
     space-separated string.
    Args:
    script: argparse.ArgumentParser instance

    Returns:
    argument flags in a space-separated string
    """
    return ''.join(flag for action in parser._actions for flag in action.
    option_strings)
```

4 Conclusion

Within the presented study we propose an approach for structuring RAG prompt in code generation with relevant code samples and API retrieved with semantic search in public and project-related scope. Having CodeGen 350M-mono

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_52

as the main experimental model, our experimental evaluation shows that the proposed approach enables achieving high performance with relatively smaller model. Moreover, the approach gives insight that proper (project-related) selection of RAG context show better performance comparing to direct usage of all public information (e.g., with comparing of public API vs. project-related API). We see this approach as promising starting point for further development of RAG-based code generation and other software development tasks which can give more consistent, comprehensive, and aligned application of RAG solutions. The promising research directions we consider as further step in this approach development are semantic and knowledge-graph involvement into RAG implementation, application of more advanced metrics in code generation evaluation, extending the study to larger number of modern benchmarks, etc. Future work would also include the comparison with other RAG approaches to see the effectiveness of versatile retrieving methods and contexts for generative models that are used when solving software engineering tasks.

References

1. BehnamGhader, P., Adlakha, V., Mosbach, M., Bahdanau, D., Chapados, N., Reddy, S.: Llm2vec: Large language models are secretly powerful text encoders (2024), https://arxiv.org/abs/2404.05961
2. Leng, Q., Portes, J., Havens, S., Zaharia, M., Carbin, M.: Long context rag performance of large language models (2024), https://arxiv.org/abs/2411.03538
3. Li, X., Dong, K., Lee, Y.Q., Xia, W., Yin, Y., Zhang, H., Liu, Y., Wang, Y., Tang, R.: Coir: A comprehensive benchmark for code information retrieval models (2024), https://arxiv.org/abs/2407.02883
4. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: Codegen: An open large language model for code with multi-turn program synthesis (2023), https://arxiv.org/abs/2203.13474
5. Yang, Z., Chen, S., Gao, C., Li, Z., Hu, X., Liu, K., Xia, X.: An empirical study of retrieval-augmented code generation: Challenges and opportunities. ACM Trans. Softw. Eng. Methodol. (2025). https://doi.org/10.1145/3717061
6. Yu, H., Shen, B., Ran, D., Zhang, J., Zhang, Q., Ma, Y., Liang, G., Li, Y., Wang, Q., Xie, T.: Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3623316
7. Zan, D., Chen, B., Gong, Y., Cao, J., Zhang, F., Wu, B., Guan, B., Yin, Y., Wang, Y.: Private-library-oriented code generation with large language models (2023), https://arxiv.org/abs/2307.15370
8. Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.G., Chen, W.: Repocoder: Repository-level code completion through iterative retrieval and generation (2023), https://arxiv.org/abs/2303.12570
9. Zhang, K., Zhang, H., Li, G., Li, J., Li, Z., Jin, Z.: Toolcoder: Teach code generation models to use api search tools (2023), https://arxiv.org/abs/2305.04032

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_52